Advanced topics

1. Aggregate Operations

https://docs.oracle.com/javase/tutorial/collections/streams/index.html

Aggregate operations, pipelines, iterators

A ***pipeline*** is a sequence of aggregate operations. The following example prints the male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter` and `forEach`:

Q: A sequence of aggregate operations is known as a ___ .
A: Pipeline

```
roster
        .stream()
        .filter(e -> e.getGender() == Person.Sex.MALE)
        .forEach(e -> System.out.println(e.getName()));
```

A pipeline contains the following components:

- **A source**: This could be a collection, an array, a generator function, or an I/O channel. In this example, the source is the collection `roster`.

- **Zero or more *intermediate operations***. An intermediate operation, such as `filter`, produces a new stream.

A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline. This example creates a stream from the collection `roster` by invoking the method `stream`.

The `filter` operation returns a new stream that contains elements that match its predicate (this operation's parameter). In this example, the predicate is the lambda expression `e -> e.getGender() == Person.Sex.MALE`. It returns the boolean value `true` if the `gender` field of object `e` has the value `Person.Sex.MALE`. Consequently, the `filter` operation in this example returns a stream that contains all male members in the collection `roster`.

- A *terminal operation*. A terminal operation, such as `forEach`, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of `forEach`, no value at all. In this example, the parameter of the `forEach` operation is the lambda expression `e -> System.out.println(e.getName())`, which invokes the method `getName` on the object `e`. (The Java runtime and compiler infer that the type of the object `e` is `Person`.)

```
double average = roster
    .stream()        //beginning of pipeline
    .filter(p -> p.getGender() == Person.Sex.MALE)  //intermediate operation
    .mapToInt(Person::getAge)    //intermediate operation
    .average()                     //terminal operation
    .getAsDouble();                  //terminal operation
```

Differences Between Aggregate Operations and Iterators

Aggregate operations, like `forEach`, appear to be like iterators. However, they have several fundamental differences:

- **They use internal iteration**: Aggregate operations do not contain a method like `next` to instruct them to process the next element of the collection. With *internal delegation*, your application determines *what* collection it iterates, but the JDK determines *how* to iterate the collection. With *external iteration*, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this

limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. See the section Parallelism for more information.

- **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called *stream operations*.

They support behavior as parameters: You can specify lambda expressions as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

Reduction

`Produces single summary by repeatedly applying a combination operation to the elements.`

The JDK contains many terminal operations (such as `average`, `sum`, `min`, `max`, and `count`) that return one value by combining the contents of a stream. These operations are called *reduction operations*. The JDK also contains reduction operations that return a collection instead of a single value. Many reduction operations perform a specific task, such as finding the average of values or grouping elements into categories.

However, the JDK provides you with the general-purpose reduction operations `reduce` and `collect`, which this section describes in detail.

reduce()

The `reduce` operation always returns a new value.

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

- **`Simple, immutable reduction`**
- `Easy to parallelise`

Example 1:

```
Integer totalAge = roster
    .stream()
    .mapToInt(Person::getAge)
    .sum();
```

Compare this with the following pipeline, which uses the `Stream.reduce` operation to calculate the same value:

```
Integer totalAgeReduce = roster
   .stream()
   .map(Person::getAge)
   .reduce(
       0,
       (a, b) -> a + b);
```

The `reduce` operation in this example takes two arguments:

- `identity`: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is `0`; this is the initial value of the sum of ages and the default value if no members exist in the collection `roster`.
- `accumulator`: The accumulator function takes two parameters: a partial result of the reduction (in this example, the sum of all processed integers so far) and the next element of the stream (in this example, an integer). It returns a new partial result. In this example, the accumulator function is a lambda expression that adds two `Integer` values and returns an `Integer` value:

```
(a, b) -> a + b
```

Example 2:

```
public class Main {
   static int x;
```

```
    static {
        x = 10;
        Stream<Integer> integerStream =     //{20,40,70,110}
                Stream.of(10, 20, 30, 40)
                .map((y) -> {
                    x += y;
                    return x;
                });

        System.out.println(x); //10

        //identity value has not changed. It is still x==10
        // we sum the numbers
        int result = integerStream.reduce(x, (a, b) -> {
            return a + b;
        });

//         java.util.stream.Stream;
//         method: T reduce(T identity, BinaryOperator<T> accumulator);
//         API Note:
//         Sum, min, max, average, and string concatenation are all special cases of
reduction. Summing a stream of numbers can be expressed as:
//         Integer sum = integers.reduce(0, (a, b) -> a+b);

        System.out.println(result); //250
    }

    public static void main(String[] args) {
    }
}
```

collect()

Unlike the `reduce` method, which always creates a new value when it processes an element, the `collect` method modifies, or mutates, an existing value.

```
<R, A> R collect(Collector<? super T, A, R> collector)
```

- **Mutable reduction (can keep state)**
- Allows for more sophisticated operations

Example 1:

Consider how to find the average of values in a stream. You require two pieces of data: the total number of values and the sum of those values

```
class Averager implements IntConsumer
{
    private int total = 0;
    private int count = 0;

    public double average() {
        return count > 0 ? ((double) total)/count : 0;
    }

    public void accept(int i) { total += i; count++; }

    public void combine(Averager other) {
        total += other.total;
```

```
        count += other.count;
    }
}
```

The following pipeline uses the `Averager` class and the `collect` method to calculate the average age of all male members:

```
Averager averageCollect = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(Person::getAge)
    .collect(Averager::new, Averager::accept, Averager::combine);

System.out.println("Average age of male members: " + averageCollect.average());
```

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

The `collect` operation in this example takes three arguments:

- `supplier`: The supplier is a factory function; it constructs new instances. For the `collect` operation, it creates instances of the result container. In this example, it is a new instance of the `Averager` class.
- `accumulator`: The accumulator function incorporates a stream element into a result container. In this example, it modifies the `Averager` result container by incrementing the `count` variable by one and adding to the `total` member variable the value of the stream element, which is an integer representing the age of a male member.
- `combiner`: The combiner function takes two result containers and merges their contents. In this example, it modifies an `Averager` result container by incrementing the `count` variable by the `count` member variable of the other `Averager` instance and adding to the `total` member variable the value of the other `Averager` instance's `total` member variable.

Example 2:

```
public static <T, K, A, D>
Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
                                      Collector<? super T, A, D> downstream)
```

The following example retrieves the names of each member in the collection `roster` and groups them by gender:

```
Map<Person.Sex, List<String>> namesByGender =
        roster
                .stream()
                .collect(
                        Collectors.groupingBy(
                                Person::getGender,
                                Collectors.mapping(
                                        Person::getName,
                                        Collectors.toList())));
```

The `groupingBy` operation in this example takes two parameters, a classification function and an instance of `Collector`. The `Collector` parameter is called a *downstream collector*. This is a collector that the Java runtime applies to the results of another collector. Consequently, this `groupingBy` operation enables you to apply a `collect` method to the `List` values created by the `groupingBy` operator. This example applies the collector `mapping`, which applies the mapping function `Person::getName` to each element of the stream. Consequently, the resulting stream consists of only the names of members. A pipeline that contains one or more downstream collectors, like this example, is called a ***multilevel reduction***.

Example 3:

```
public static <T, K, A, D>
Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
                                      Collector<? super T, A, D> downstream)
```

The following example retrieves the total age of members of each gender:

```
Map<Person.Sex, Integer> totalAgeByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(
                    0,
                    Person::getAge,
                    Integer::sum)));
```

The `reducing` operation takes three parameters:

- `identity`: Like the `Stream.reduce` operation, the identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is `0`; this is the initial value of the sum of ages and the default value if no members exist.
- `mapper`: The `reducing` operation applies this mapper function to all stream elements. In this example, the mapper retrieves the age of each member.

```
public static <T, U>
Collector<T, ?, U> reducing(U identity,
                            Function<? super T, ? extends U> mapper,
                            BinaryOperator<U> op) {
```

- `operation`: The operation function is used to reduce the mapped values. In this example, the operation function adds `Integer` values.

Example 4:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.time.LocalDate;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamsDemo {
   record BatteryDay(LocalDate date, String deviceModel, double percentDrained) {
       BatteryDay(String[] fields) {
           this(LocalDate.parse(fields[0]), fields[1], Double.parseDouble(fields[2]));
       }

       BatteryDay(String line) {
           this(line.split(","));
       }
   }

   public static void main(String[] args) {
       try (Stream<String> lines = new BufferedReader(
               new InputStreamReader(
                       StreamsDemo.class.getResourceAsStream("battery.csv")))
               .lines()
       ) {
```

```
//            lines.forEach(System.out::println);

public static <T, K> Collector<T, ?, Map<K, List<T>>>
groupingBy(Function<? super T, ? extends K> classifier)

            Map<String, List<BatteryDay>> result = lines
                .map(BatteryDay::new)
                .filter(batteryDay -> batteryDay.date.isBefore(LocalDate.now()))
                .collect(Collectors.groupingBy(BatteryDay::deviceModel));//transform  to  a
map


.......

public static <T, K, A, D>
Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
                                      Collector<? super T, A, D> downstream)

Map<String, Double> result = lines
        .map(BatteryDay::new)
        .filter(batteryDay -> batteryDay.date.isBefore(LocalDate.now()))
        .collect(Collectors.groupingBy(
                BatteryDay::deviceModel,
                Collectors.averagingDouble(BatteryDay::percentDrained)
                ));//transform to a map

System.out.println(result);

            System.out.println(result);
        }
    }
}
```

Parallelism

Parallel computing involves dividing a problem into subproblems, solving those problems simultaneously (in parallel, with each subproblem running in a separate thread), and then combining the results of the solutions to the subproblems. Java SE provides the fork/join framework, which enables you to more easily implement parallel computing in your applications. However, with this framework, you must specify how the problems are subdivided (partitioned). With aggregate operations, the Java runtime performs this partitioning and combining of solutions for you.

Note that parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. While aggregate operations enable you to more easily implement parallelism, it is still your responsibility to determine if your application is suitable for parallelism.

Executing Streams in Parallel

You can execute streams **in serial** or **in parallel**. When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results.

To create a parallel stream, invoke the operation `Collection.parallelStream`. Alternatively, invoke the operation `BaseStream.parallel`. For example, the following statement calculates the average age of all male members in parallel:

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Concurrent Reduction

Consider again the following example (which is described in the section Reduction) that groups members by gender. This example invokes the `collect` operation, which reduces the collection `roster` into a `Map`:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The following is the parallel equivalent:

```
ConcurrentMap<Person.Sex, List<Person>> byGender =
    roster
        .parallelStream()
        .collect(
            Collectors.groupingByConcurrent(Person::getGender));
```

Ordering

The order in which a pipeline processes the elements of a stream depends on whether the stream is executed in serial or in parallel, the source of the stream, and intermediate operations.

When running in parallel the results are UNORDERED!

When using parallel and the forEachOrdered, then we may lose the benefits of parallelism

```
listOfIntegers
    .parallelStream()
    .forEachOrdered(e -> System.out.print(e + " "));
```

Side effects

Laziness

All intermediate operations are *lazy*.

Interference

Lambda expressions in stream operations should not *interfere*. **Interference occurs when the source of a stream is modified while a pipeline processes the stream.** For example, the following code attempts to concatenate the strings contained in the `List listOfStrings`. However, it throws a `ConcurrentModificationException`:

```
try {
    List<String> listOfStrings = new ArrayList<>(Arrays.asList("one", "two"));

    // This will fail as the peek operation will attempt to add the
    // string "three" to the source after the terminal operation has
    // commenced/started.
    String concatenatedString = listOfStrings
        .stream()

        // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))

        .reduce((a, b) -> a + " " + b)
        .get();

    System.out.println("Concatenated string: " + concatenatedString);

} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
```

```
}
```

Stateful Lambda Expressions

Avoid using *stateful lambda expressions* as parameters in stream operations. **A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline.** The following example adds elements from the `List listOfIntegers` to a new `List` instance with the **`map` intermediate operation**. It does this twice, first with a serial stream and then with a parallel stream:

```
List<Integer> listOfIntegers = new ArrayList<>(List.of(8, 7, 6, 5, 4, 3, 2, 1));

----------------------------------

System.out.println("Serial stream:");
List<Integer> serialStorage = new ArrayList<>();

listOfIntegers
    .stream()
    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { serialStorage.add(e); return e; })
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

serialStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

-------------------------------------

System.out.println("Parallel stream:");
List<Integer> parallelStorage = Collections.synchronizedList(new ArrayList<>());
listOfIntegers
    .parallelStream()
    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { parallelStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

parallelStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

The lambda expression `e -> { parallelStorage.add(e); return e; }` is a stateful lambda expression. Its result can vary every time the code is run. This example prints the following:

```
Serial stream:
8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
Parallel stream:
8 7 6 5 4 3 2 1
1 3 6 2 4 5 8 7
```

The operation `forEachOrdered` processes elements in the order specified by the stream, regardless of whether the stream is executed in serial or parallel. However, when a stream is executed in parallel, the `map` operation processes elements of the stream specified by the Java runtime and compiler. Consequently, the order in which the lambda expression `e -> { parallelStorage.add(e); return e; }` adds elements to the `List parallelStorage` can vary every time the code is run. For deterministic and predictable results, ensure that lambda expression parameters in stream operations are not stateful.

2. Многонишково програмиране (Multithreading) в Java(SE)

Processes

Processes are often seen as synonymous with programs or applications.
A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a `ProcessBuilder` object.

java.lang.ProcessBuilder - създава цели процеси вместо отделни нишки/Threads - обичайно работим с нишки, а не с цели процеси!!!

Threads

- Threads are also referred to as **lightweight processes**
- Threads are separate execution flows within a single process
- They share the resources of the process including open files and memory

- **Java threads are typically bound to native OS threads that execute them on different OS processors**
- Some JVM implementations of threads also provide so called **green threads** that are scheduled for execution by the JVM itself rather than the OS - от кого се изпълняват
- Threads in the JVM might be
    - user потребителски
    - or daemon threads - отделни независими нишки необвързани с друга нишка от JVM, примерно за scheduling

Which of the following **is not valid** about Java threads?
a.every time a thread is created a new stack is allocated
b.threads have different states
c.threads can have a priority
d. Java threads are not bound to OS threads

What are **lightweight threads** in Java?
a. scheduled only by the JVM and use less resources
b. easier to schedule native threads
c. easier to create than standard Java threads
d. standard Java threads with limited capabilities

Когато пуснем едно стандартно Java приложение:
- A typical Java application contains multiple threads of execution
- The JVM itself once started starts different threads - например нишки за Garbage collector-а, които нишки са native (не минава през предоставените от самата JVM средства за работа с нишки)
- The Java application may also start a number of threads during its execution
- Има готови пакети/класове/библиотеки (като **java.util.concurrent**) - които позволяват по лесен и удобен начин да работим с нишки

Green threads, normal threads, virtual threads

Green Threads(old) had an **N:1 mapping** with OS Threads. All the Green Threads ran on a single OS Thread.
With Virtual Threads, multiple virtual threads can run on multiple native carrier OS threads (**n:m mapping**)
The **old Java green threads** could only use a single core, the new Java virtual threads can use multiple cores.

Like with green threads, virtual threads are managed by the JVM!

The most lightweight version of threads are **fibers(virtual threads)** that are parallel execution flows whiting a single thread (като тези виртуални нишки са в рамките само на JVM, без да се взаимодейства с RAM паметта на операционната система (но реално carrier-а е нишката на ОС). По-лесни са за менъджиране, но нямат пълният набор от опции като стандартната нишка). Currently native support for fibers(virtual threads) in the JVM is being developed under the **project Loom.** От версия **JDK 20/21** може да се използват вече.

In computer programming, a **green thread** (virtual thread) is a thread that is scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system (OS). Green threads emulate multithreaded environments without relying on any native OS abilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support.

Единствената разлика между VirtualTreads и GreenThreads е в това, че VirtualTreads използват **Thread carriers**, а другите GreenThreads не.

Virtual threads are a lightweight implementation of threads that is provided by the JDK rather than the OS. They are a form of user-mode threads, which have been successful in other multithreaded languages (e.g., goroutines in Go and processes in Erlang). User-mode threads even featured as so-called "green threads" in early versions of Java, when OS threads were not yet mature and widespread. However, Java's green threads all shared one OS thread (M:1 scheduling) and were eventually outperformed by platform threads, implemented as wrappers for OS threads (1:1 scheduling). Virtual threads employ M:N scheduling, where a large number (M) of virtual threads is scheduled to run on a smaller number (N) of OS threads.

When Quarkus meets Virtual Threads
https://quarkus.io/blog/virtual-thread-1/

At the beginning of the Java time, Java had *green threads*. Green threads were user-level threads scheduled by the Java virtual machine (JVM) instead of natively by the underlying operating system (OS). They emulated multithreaded environments without relying on native OS abilities. They were managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support. Green threads were briefly available in Java between 1997 and 2000. I used green threads; they did not leave me with a fantastic memory.

In Java 1.3, released in 2000, Java made a big step forward and started integrating **OS threads**. So, the threads are managed by the operating system. It is still the model we are using today. Each time a Java application creates a thread, a platform thread is created, which wraps an OS thread. So, creating a platform thread creates an OS thread, and **blocking a platform thread blocks an OS thread**.

Java 19 introduced a new type of thread: virtual threads. In Java 21, this API became generally available.

…….

The Java Memory Model (JMM)

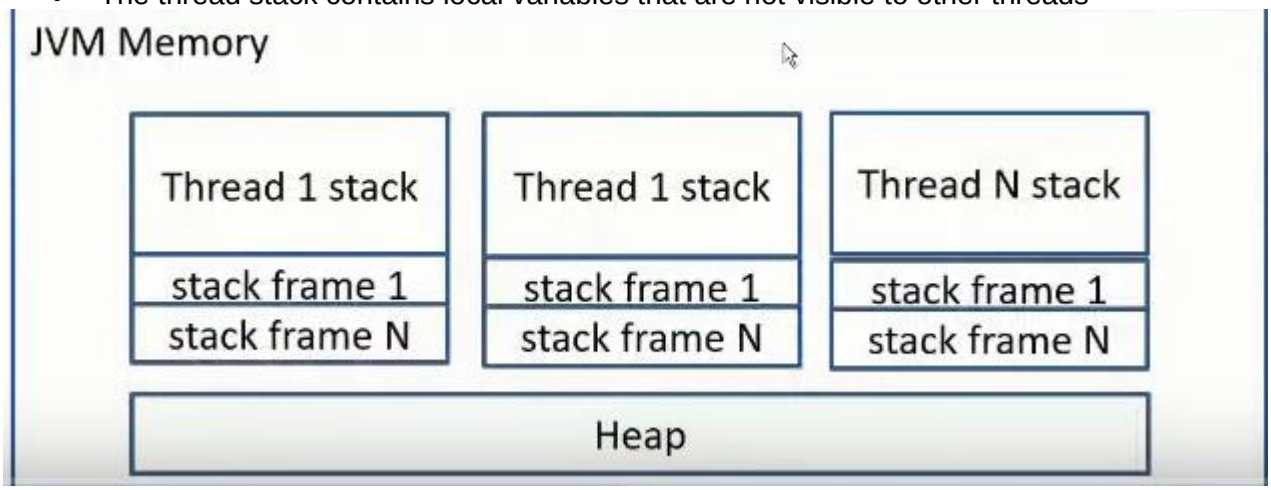
- The Java Memory Model describes how threads interact with the JVM memory
- Different processors typically provide **local caches** that are synchronized with **main memory**
- Since Java threads may run on different processors they may see a different view of the same shared memory due to the caches

Keeping processors caches at all time in sync with main memory is costly and not always feasible (и не винаги възможно) due to performance penalties.

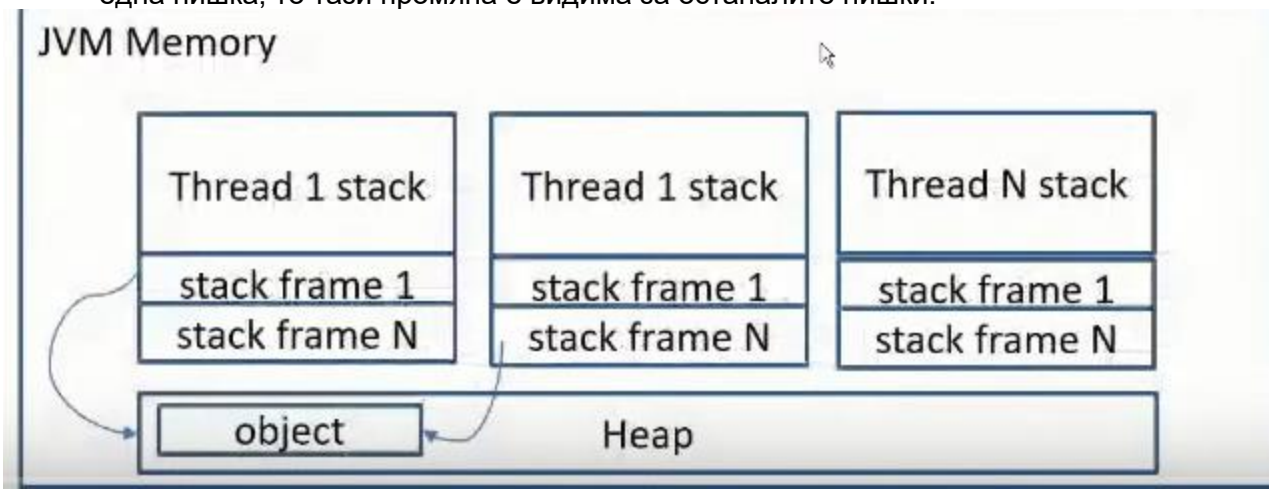
Или да виждат само локалния кеш или всичко да е синхронизирано с основната памет.

- Each thread has its own **thread stack** allocated - стак на изпълнение, който се пълни със стак фреймове

- The thread stack contains local variables that are not visible to other threads



- Since objects are stored on the heap they might be accessed by multiple threads
- Additional mechanisms should be used to ensure proper memory visibility and synchronization among threads - с други думи ако правим някаква промяна на стойност на поле на обект от heap-а от една нишка, то тази промяна е видима за останалите нишки!



The **JVM memory (stack and heap)** is mapped to the RAM on the target machine.

- Synchronization with the main memory can be achieved as follows:
  - Marking a variable as **volatile** makes reading from and writing to the variable directly from main memory - може да сложим тази запазена дума на поле на клас и тогава сме синхронизирани с основната РАМ памет
  - Creating ***synchronized block*** with the **synchronized** keyword may also perform a cache flush (in most cases partial using special instruction like memory barriers) once a thread exits the block - т.е. ако дадена нишка е влязла в блока да изпълнява кода, то други нишки опитващи се да достъпят същия блок (public void …) изчакват.

Synchronized blocks in that regard serve not only as a mechanism to provide order of execution, but also memory visibility.

Всеки обект в JVM има поле **lock**, по което поле дадена нишка може да го достъпи и да го заключи.

volatile - това че сме синхронизирани с основната РАМ памет ни предпазва да не върнем друга стойност на дадената променлива.

Threads and thread pools

Java threads

There are two main ways to create a thread:

- By implementing the **java.lang.Runnable** interface (preferred way)

```
new Thread(new Runnable() {
    @Override
    public void run() {
        //thread logic here
    }
}).start();

new Thread(() -> {
    //thread logic here
}).start();
```

- By extending the **java.lang.Thread** class - изменя поведението на самия `Thread` клас, което не е добре

```
public class CustomThread extends Thread {
    @Override
    public void run() {
        //thread logic here
    }
}

new CustomThread().start();
```

When **is a thread scheduled** for execution by the JVM?
a.when calling the run() method
b.when calling the start() method
c.when the thread is initialized
d.it is not bound to a method invocation

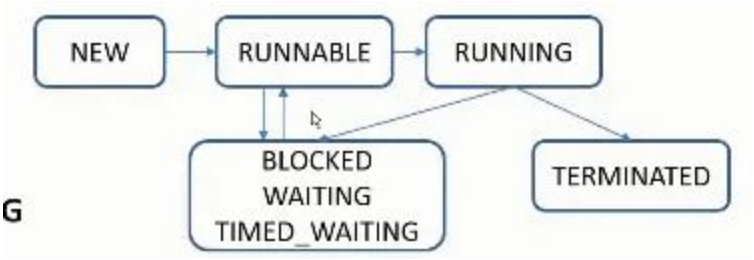
When we call **start()** with an object of **Thread** class that thread goes to the **Runnable** state. So all the threads go to **Runnable** state after calling **start()** by the object of those threads. It is ***JVM thread scheduler***, who picks thread randomly from **Runnable** state to give it in **Running** state. After going to **Running** state, the determined call stack for that specific thread becomes executed.

Again, JVM thread scheduler can stop the execution of a thread by picking that thread from Running state to Runnable state. This time, code execution is paused in the call stack of that thread.

Thread states

A non-running thread might have any of the states defined in the Thread#State enum

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIME_WAITING
- TERMINATED



Thread interruption

- A thread can be interrupted by calling the **Thread.interrupt** method on the thread object
- The **Thread.interrupt** method sets a special flag that indicates the thread is interrupted and that can be checked with the **Thread.isInterrupted** method on the thread object.
- Some methods such as a Thread.sleep throw a **java.lang.InterruptedException** if the calling thread is interrupted.

Thread pools

- Threads might be **reused** since thread creation is an expensive process
- In the JDK different thread pools are provided as an implementation of the **java.util.concurrent.Executor** interface
- The **java.util.concurrent.Executors** class provides static methods for the creation of **different types of thread pools** - например при процесор с 16 ядра, то се създават 16 нишки за примерно 1500 различни задачи. Няма смисъл да се правят повече нишки, защото тогава ще има много **context switching**, който е много скъп процес!

- The **java.util.concurrent.Executor** interface is extended by an **ExecutorService** interface that provides more operations for thread pools
- The ExecutorService interface is further extended by the **ScheduledExecutorService** interface that provides the possibility to schedule threads for execution

Съществуват различни варианти за работа с pools:

- The Executor**s** class and Executor interface
- The ExecutorService
- ScheduledExecutorService – extends ExecutorService
- The **ThreadPoolExecutor**
- The **ForkJoinPool – използва се много за parallel streams/programming**

https://stackify.com/java-thread-pools/

```
// final int numThreads = 10;
int numThreads = Runtime.getRuntime().availableProcessors(); //8
ExecutorService threadPool = Executors.newFixedThreadPool(numThreads);

for (int i = 0; i < 100; i++) {
   Runnable task = new Task(i); //Task класа е имплементация на java.lang.Runnable
   threadPool.execute(task);
}

threadPool.shutdown();
```

```
// waits for all threads to finish
threadPool.awaitTermination(2, TimeUnit.SECONDS);

System.out.println("Finished all threads");
```

Which of the following **is a not** a thread pool characterstic ?
a.Optimizes the creation and scheduling of threads
b.Provide a mechanism to execute a task in a distributed manner
c.Improves resource utilization in the application
d. Elliminates the presence of deadlocks

Which of the following **is not a valid type of thread pool** as defined by the java.util.concurrent.Executors class?
a. cached thread pool - `java.util.concurrent.Executors.newCachedThreadPool();`
b. single thread pool - `java.util.concurrent.Executors.newSingleThreadExecutor();`
c. scheduled thread pool - `java.util.concurrent.Executors.newScheduledThreadPool();`
d. native thread pool

Example with ExecutorService

```
ExecutorService service = Executors.newCachedThreadPool();

Runnable r1 = () -> transferAmount(new PaymentIqTransferDTO(USER_ID, "100.00", EUR,
TX_ONE_ID, "001", "Deposit", "Bambora",
       "101", "EUR", "1.00", "EUR", "11111A1", "A", null));
Runnable r2 = () -> transferAmount(new PaymentIqTransferDTO(USER_ID, "200.00", EUR,
TX_TWO_ID, "001", "Deposit", "Bambora",
       "201", "EUR", "1.00", "EUR", "11111A1", "A", null));
```

// Submits a Runnable task for execution and returns a Future representing that task. The Future's **get** method will return null upon successful completion.

```
Future<?> future = service.submit(r1);
service.submit(r2);
```

// Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

```
service.awaitTermination(1000, TimeUnit.MILLISECONDS);
```

Thread locals

- Thread locals are special types of variables that can only be written and read from a single thread - т.е. паметта на такава нишка не се споделя с други нишки

```
private ThreadLocal threadLocal = new ThreadLocal<>();
threadLocal.set("some value");
String value = (String) threadLocal.get();
```

Futures

- Futures are used to represent the result of a future execution
- In Java futures are represented by the **java.util.concurrent.Future** interface
- Futures provide the possibility to **cancel** a task, **check** if the task is **done** or **get the result** (blocks until computation is done) - позволява ни да върнем и резултат от задачата

```
final int numThreads = 10;
ExecutorService executor = Executors.newFixedThreadPool(numThreads);

Runnable task = new Task();
Future<String> future = executor.submit(task);

// blocks until task is completed.
//In the case of using Runnable, it executes the overrided run() method.
//In the case of using Callable, it executes the overrided call() method.
String result = future.get();
```

Callable vs Runnable

In a **java.util.concurrent.Callable** interface that basically **throws a checked exception and returns some results**.
This is one of the major differences between the upcoming **java.lang.Runnable** interface **where no value is being returned**. In this **Runnable** interface, it simply computes a result else throws an exception if unable to do so.

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

What is a difference between a Callable and a Runnable?

a. **Callable can throw a checked exception while a Runnable cannot**
b. Callable cannot be scheduled by a thread pool while a Runnable can
c. Runnable cannot be scheduled by a thread pool while a Callable can
d. Callable can be canceled while Runnable cannot

Example Fibbonachi with Futures:

```
public class Main {
   public static void main(String[] args) {
   java.util.concurrent.ExecutorService pool =
java.util.concurrent.Executors.newSingleThreadExecutor();

       try {
           java.util.concurrent.Future<java.math.BigDecimal> result = fact(1, pool);
           java.util.concurrent.Future<java.math.BigDecimal> result = fact(5, pool);

           System.out.println(result.get());//the get() methods executes the call() method
which go into recursion and returns the result
       } catch (Exception ex) {
       } finally {
           try {
               pool.awaitTermination(1, java.util.concurrent.TimeUnit.SECONDS);
           } catch (InterruptedException e) {
           }

           pool.shutdown();
       }
   }

   public    static    java.util.concurrent.Future<java.math.BigDecimal>    fact(int    n,
java.util.concurrent.ExecutorService pool) {

       Multiply previousMultiply = new Multiply(null, java.math.BigDecimal.ONE);
```

```
        java.util.concurrent.Future<java.math.BigDecimal> lastTaskResult =
pool.submit(previousMultiply);

        for (int start = 2; start <= n; start++) {
            Multiply current = new Multiply(previousMultiply, new
java.math.BigDecimal(start));
            lastTaskResult = pool.submit(current);
            previousMultiply = current;
        }

        return lastTaskResult;
    }
}


public class Multiply implements java.util.concurrent.Callable<java.math.BigDecimal> {
    private Multiply start;
    private java.math.BigDecimal end;

    public Multiply(Multiply start, java.math.BigDecimal end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public java.math.BigDecimal call() {
        if (this.start == null) {
            return java.math.BigDecimal.ONE;
        }

        java.math.BigDecimal startResult = this.start.call();
        return startResult.multiply(this.end);
    }
}
```

Thread synchronization

Info

- Since different threads can access shared data this may result in data consistency issues also known as **race condition.** Another terminology for the same problem is:
    - **Thread Interference** (намеса)
    - **Memory Consistency Errors** (the need of a **happens-before relationship**)
- To solve issues related to race conditions a mechanism called **thread synchronization** can be used

- Java (JVM) uses object monitors to perform thread synchronization
- A monitor is associated with an object and conducts a region that can be accessed by one thread at a time - благодарение на lock се задава регион
- A monitor as acquired by a thread using a **lock**
- A lock can be either **intrinsic** or **extrinsic**

Intrinsic locks

An **intrinsic** block is created using the **synchronized** keyword that:

- can be applied as a method attribute that makes the method block a synchronized region using the method's object as a monitor
  ```
  public synchronized void someMethod(){}
  ```

- can be specified as a separate block within a method using a target object as a monitor

```
synchronized (object) {
   //block of code
}
```

What is 'synchronized' keyword used for ?
a.To define an implicit lock on a object
b.To define an implicit lock on a thread
c.To define an implicit lock on a method
d. To define an implicit lock on a block

Extrinsic locks

- Extrinsic locks are provided as different implementations of the **java.util.concurrent.locks.Lock** interface:
  - **ReentrantLock** - **това е по подразбиране за intrinsic locks** - ако дадена нишка държи lock към някакъв обект, и влеземе например в друг блок/метод, който държи същия lock, тогава нишката успява да влезе в него
  - **ReadWriteLock**
  - **StampedLock**

- Extrinsic locks provide more flexibility than intrinsic locks such as:
  - The possibility to create the synchronized block across multiple methods
  - The possibility to check if a lock is already acquired with the **tryLock** method

Same as having **synchronized** on the **this** object

```
public void someMethod() {
   reentrantLock.lock();
   try {
       // block of code
   } finally {
       reentrantLock.unlock();
   }
}
```

Joining threads – making order

- The **Thread.join** method provides a mechanism whereby one thread can wait for the completion of another thread - т.е. ни дава възможност да chain-ваме в ПОРЕДНОСТ изпълнението на дадени нишки

```
// called by thread A
public void someMethod() {
   // some logic ...

   // wait for thread B to complete
   threadB.join();
}
```

```
thread1.start();
thread2.start();
thread1.join();
thread2.join();
```

Synchronization issues

- Problems may occur when synchronization is not applied properly such as:

- **deadlock:** two or more threads are locked indefinitely waiting for each other

//нишка 1 локва обект o1, и след като мине 1 секунда, тръгва да вика обект o2. Но обект o2 в това време е локнат от нишка 2!!!

```
public class DeadlockExample {
  public static void main(String[] args) {
    Object o1 = new Object();
    Object o2 = new Object();

    Thread thread1 = new Thread(() -> {
      synchronized (o1) {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        synchronized (o2) {
          System.out.println("Hello from Thread 1");
        }
      }
    });

    Thread thread2 = new Thread(() -> {
      synchronized (o2) {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        synchronized (o1) {
          System.out.println("Hello from Thread 2");
        }
      }
    });

    thread1.start();
    thread2.start();
  }
}
```

- **starvation:** a thread is not able to gain access to shared resource thus not being able to make progress
- **livelock:** similar to deadlock but occurs when two or more threads act on each other as a response and are not blocked but continue indefinitely without being able to make progress

Примери

```
System.out.println(thread1.getState());

thread1.setPriority();
thread1.isDaemon();
thread1.isInterrupted();
thread1.getThreadGroup(); - ако искаме да копираме дадени нишки и да можем да ги менъжираме след това
thread1.setName();
```

```
thread1.sleep(4000);

Thread.currentThread().sleep(4000)
```

Thread safety

Thread safety

- Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly
- Immutable objects are thread safe
- To make a class immutable the following can be done:
  - mark all class fields final or declare the class as **final**
  - ensure **this** reference is not allowed to escape during construction
  - make any fields which refer to **mutable data objects private**
  - don't provide setter method
  - adding **Guarded Blocks**

Java monitor pattern

Само една нишка може да достъпи дадения метод в даден момент. Това може да е проблем ако заключваме целия метод. Вместо това можем да синхронизираме дадена част от въпросния метод където има критични секции – т.н. compound actions – някакъв брояч например за който само е задължително **`synchronized`**.
Или пък да използваме **Atomic**Integer вместо това – дава синхронизация дефакто.
Критични секции в многонишковото програмиране са всичко, което е **mutable**! В света на Java – слагаме **final** думичката правейки го **immutable**, което ни гарантира initialization safety в многонишкова среда!
Важна част в цялото нещо заема и encapsulation-а!

```
public class ValidNamesService {

    private final Object lock = new Object();

    private final Set<String> validNames;

    public ValidNamesService() { this.validNames = new HashSet<>(); }

    public void addName(final String name) {
        synchronized (lock) {
            this.validNames.add(name);
        }
    }

    public boolean hasName(final String name) {
        synchronized (lock) {
            return this.validNames.contains(name);
        }
    }
```

Reordering

При работа с нишки, понякога JVM прави автоматичен re-ordering и може да се получи разместване кое след кое се изпълнява. Тогава се налага да използваме locks!

```java
private static void testMethod() {
    number = 0;
    isFinished = false;
    final Thread firstThread = new Thread(() -> {
        while(!isFinished) {
        }
        System.out.println(number);
    });
    firstThread.start();
    number = 42;
    isFinished = true;
}
```

0
42
42
0
0
42
42
42
.....

Publishing and escaping

Escaping става на въпрос когато работим в стека на дадена нишка, то да не се обръщаме/**да публикуваме**/ към методи извън стека на нишката. Ако се обръщаме, то ни трябва експлицитно да зададем lock! Иначе се получава **escaping.**

```java
private static List<List<Integer>> cache = Collections.synchronizedList(new ArrayList<>());

public static void main(final String... args) {
    // Numbers are confined to stack of the thread
    final Thread thread = new Thread(() -> {
        final int number = 1;
        final List<Integer> numbers = new ArrayList<>();
        numbers.add(number);
        System.out.println(numbers);
        // Numbers are now escaping
        // Avoid doing this
        cache.add(numbers);
    });
    thread.start();
}
```

Confinements

Че използваме методи предназначени само за стека на дадената нишка

Thread confinement

Един thread се занимава само с numbers в случая

```
public class ThreadConfinement {
    private static final List<Integer> numbers = new ArrayList<>();

    public static void main(final String... args) {
        // Numbers are confined to thread
        final Thread thread = new Thread(() -> {
            numbers.add(1);
            System.out.println(numbers);
        });

        final Thread secondThread = new Thread(() -> {
            System.out.println("Second thread");
        });
        thread.start();
    }
```

AdHoc confinement

……

Stack confinement

Да не го публикуваме в cache един вид, защото е извън scope на стека на нишката.

```
private static List<List<Integer>> cache = Collections.synchronizedList(new ArrayList<>());

public static void main(final String... args) {
    // Numbers are confined to stack of the thread
    final Thread thread = new Thread(() -> {
        final int number = 1;
        final List<Integer> numbers = new ArrayList<>();
        numbers.add(number);
        System.out.println(numbers);
        // Numbers are now escaping
        // Avoid doing this
        cache.add(numbers);
    });
    thread.start();
}
```

Lock splitting and log striping

https://www.amazon.com/Java-Concurrency-Practice-Goetz-Bowbeer/dp/9332576521

Трябва да можем да синхронизиране като му зададем върху цялата колекция **synchronized**

```java
final List<Integer> list = Collections.synchronizedList(new ArrayList<>());

synchronized (list) {
  for (final int element : list) {
    System.out.println(element);
  }
}
private static void addItem(final int element, final List<Integer> list) {
  synchronized (list) {
    if (!list.contains(element)) {
      list.add(element);
    }
  }
}
```

Synchronized collections

Всеки един метод на тези колекции е синхронизиран също.

- **StringBuffer** is a thread-safe alternative of **StringBuilder**
- The **java.util.Collections** class provides methods to retrieve thread-safe collections
  - **Collections.synchronizedSet()**)
  - Synchronized list - **Collections.synchronizedList()**
  - Synchronized map - - **Collections.synchronizedMap()**
- **java.util.concurrentHashMap** provides a thread-safe hash map
- **CopyOnWriteArrayList** – връща последно актуално копие на итератора. Update-и през това време и да се правят, то при следващо вземане на копие на итератора, ще се е опреснило с промените. Използва се в случаи когато имаме повече четене на елементи, и малко на брой пъти добавяне на елементи.

```java
private static void fullLockMap() {
    final Map<Integer, Integer> numbers = Collections.synchronizedMap(new HashMap<>());
    synchronized (numbers) {
        for (int number : numbers.keySet()) {
            System.out.println(numbers.get(number));
        }
    }
}

private static void concurrentHashMap() {
    final Map<Integer, Integer> numbers = new ConcurrentHashMap<>();
    for (int number : numbers.keySet()) {
        System.out.println(numbers.get(number));
    }
}
```

```java
private static void fullLock() {
    final List<Integer> numbers = Collections.synchronizedList(new ArrayList<>());
    synchronized (numbers) {
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}

private static void copyOnWrite() {
    final List<Integer> numbers = new CopyOnWriteArrayList<>();
    for (int number : numbers) {
        System.out.println(number);
    }
}
```

Проверка на нишки чрез jps и jstack

jps

```
C:\Users\Martin>jps
6020 XMLServerLauncher
16312 Jps
31596 org.eclipse.equinox.launcher_1.6.400.v20210924-0641.jar
9084 DeadLockExample
```

9084 е id-то на процеса на JVM

jstack като инструмент за разбиране къде точно се случва deadlock например
jstack 9084

```
  java.lang.Thread.State: TIMED_WAITING (on object monitor)
       at java.lang.Object.wait(java.base@18.0.2.1/Native Method)
       - waiting on <0x00000007111018a0> (a java.lang.ref.ReferenceQueue$Lock)
       at java.lang.ref.ReferenceQueue.remove(java.base@18.0.2.1/ReferenceQueue.java:155)
       - locked <0x00000007111018a0> (a java.lang.ref.ReferenceQueue$Lock)
       at jdk.internal.ref.CleanerImpl.run(java.base@18.0.2.1/CleanerImpl.java:140)
       at java.lang.Thread.run(java.base@18.0.2.1/Thread.java:833)
       at jdk.internal.misc.InnocuousThread.run(java.base@18.0.2.1/InnocuousThread.java:1

Thread-0" #15 prio=5 os_prio=0 cpu=0.00ms elapsed=88.82s tid=0x00000230d8966cc0 nid=22388
[0x000000519eaff000]
  java.lang.Thread.State: BLOCKED (on object monitor)
       at bg.jug.academy.concurrency.DeadLockExample.lambda$0(DeadLockExample.java:18)
       - waiting to lock <0x0000000711104980> (a java.lang.Object)
       - locked <0x0000000711104970> (a java.lang.Object)
       at bg.jug.academy.concurrency.DeadLockExample$$Lambda$1/0x0000000800c009f0.run(Unkn
       at java.lang.Thread.run(java.base@18.0.2.1/Thread.java:833)

Thread-1" #16 prio=5 os_prio=0 cpu=0.00ms elapsed=88.82s tid=0x00000230d89689a0 nid=16460
[0x000000519ebff000]
  java.lang.Thread.State: BLOCKED (on object monitor)
       at bg.jug.academy.concurrency.DeadLockExample.lambda$1(DeadLockExample.java:31)
       - waiting to lock <0x0000000711104970> (a java.lang.Object)
       - locked <0x0000000711104980> (a java.lang.Object)
```

3. Advanced concurrent programming

I. Thread communication

Thread communication

- So far we saw how we can use implicit and explicit locks to provide synchronization between threads
- However locking might not be very flexible in coordinating threads
- A supplement mechanism whereby threads can notify ("wake up") each other and wait to be notified is provided by the JVM
    - Provides **a wait-notify mechanism** between the threads in the JVM, thus creating **a publish-subscribe mechanism** at the thread level in the JVM

Wait and notify

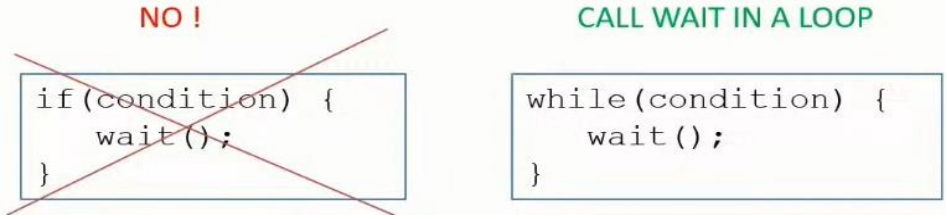
- This is achieved by the **wait**, **notify** and **notifyAll** methods provides by the **java.lang.Object** class
- These methods work over a monitor lock that must be held from threads (i.e. used within a synchronized method or block)
- **notify** wake up only one thread waiting on the monitor lock while **notifyAll** wakes up all threads
    - Be careful when to use notify and notifyAll - in many cases it is more proper to use notifyAll

Exiting waits

- Waking up from the **wait** method can happen in the following situations:
    - When **notify/notifyAll** is called from another thread
    - If timeout expires (in case the overloaded **wait** methods are used)
    - The waiting thread is interrupted by calling the **interrupt** method
    - On rare occasions the OS or the JVM may wake up the thread (also called **spurious wakeup**)

Spurious wake-ups

- To guard against spurious wake ups **wait** must always be called in a loop



Wait/notify example

```
public class PublishSubscribeExample {
   private String message;

   public static void main(String[] args) {
       PublishSubscribeExample example = new PublishSubscribeExample();

       for (int i = 0; i < 100; i++) {
           new Thread(() -> {example.subscribe();}).start();
       }

       try {
           Thread.sleep(3000);
       } catch (InterruptedException e) {
           Thread.currentThread().interrupt(); //interrupt flag is being cleared, so if we want the
layers above to know if the thread was interrupted, we make like this
```

```
        }

        new Thread(() -> {example.publish("Hello threads!");}).start();
    }

    public synchronized void publish(String message){
        this.message = message;
        System.out.println("Notifying all threads ...");
        notifyAll();
    }

    public synchronized void subscribe(){
        while (message == null){
            try {
                wait();
            } catch (InterruptedException e){
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("Message received: " + message);
    }
}
```

Notifying all threads ...
Message received: Hello threads!
Message received: Hello threads!
Message received: Hello threads!

Conditions

- JDK 5 introduced a more flexible (and preferable way) **to specify wait conditions** using the **java.util.concurrent.lock.Condition** interface
- Additional capabilities of the Condition interface include:
    - awaitUntil(Date date) method that waits until a specified date - и ако до това време не се случи, то се събужда нишката
    - **awaitUniterruptibly()** method that awaits until the thread is signalled

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class PublishSubscribeWithConditionExample {
    private String message;
    private ReentrantLock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public static void main(String[] args) {
        PublishSubscribeWithConditionExample example = new PublishSubscribeWithConditionExample();

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {example.subscribe();}).start();
        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); //interrupt flag is being cleared, so if we want the
layers above to know if the thread was interrupted, we make like this
        }

        new Thread(() -> {example.publish("Hello threads!");}).start();
```

```
    }

    public synchronized void subscribe() {
        try {
            lock.lock();
            while (this.message == null){
                try {
                    condition.await();
                } catch (InterruptedException e){
                    Thread.currentThread().interrupt();
                }
            }
            System.out.println("Message received: " + this.message);
        } finally {
            lock.unlock();
        }
    }

    public synchronized void publish(String message) {
        try {
            lock.lock();
            this.message = message;
            System.out.println("Notifying all threads ...");
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
}
```

II. Concurrent (general term) collections

- The standard JDK collections (such as LinkedList and ArrayList) are not thread-safe (except for legacy Vector and Hashtable, които **не се препоръчва да се използват** в нови Java приложения)
- The JDK provides several types of thread-safe collections:
    - **Synchronized collections** (such as ones that can be created with the **Collections.syncronizedXXX** methods) - the following methods from Collections class can be used to create synchronized collections:
        - synchronizedCollection
        - synchronizedSet
        - synchronizedSortedSet
        - synchronizedList
        - synchronizedMap
        - synchronizedSortedMap
        - synchronizedNavigableMap
    - **BlockingQueue** - thread-safe lock-based collections provided by the **java.util.concurrent** package such as the implementations of the **BlockingQueue** interface:
        - ArrayBlockingQueue
        - LinkedBlockingQueue
        - LinkedBlockingQueue
        - DelayQueue
        - PriorityBlockingQueue
        - SyncronousQueue
    - Lock-free thread-safe collections (such as ConcurrentHashMap or Copy-on-write) - **не използва locks и synchronized**(синхронизация), а използва **Compare and Swap** техника на база процесорни инструкции, които ни дават възможност да запазваме нещо в паметта

само ако дадено друго нещо/стойност вече е запазена в тази памет. **По-бързи тъй като lock/unlock коства ресурси/време**. - before JDK 5 ConcurrentHashMap was NOT lock-free.

1. Synchronized collections

```
private static void syncronizedCollectionExample() {
   List<String> items  = new ArrayList<>();
   List<String> synchronizedItems = Collections.synchronizedList(items);
}
```

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
   for (Type e : c)
      foo(e);
}
```

If an explicit iterator is used, the `iterator` method must be called from within the `synchronized` block. Failure to follow this advice may result in nondeterministic behavior. The idiom for iterating over a `Collection` view of a synchronized `Map` is similar. It is imperative that the user synchronize on the synchronized `Map` when iterating over any of its `Collection` views rather than synchronizing on the `Collection` view itself, as shown in the following example.

```
Map<KeyType, ValType> m = Collections.synchronizedMap(new HashMap<KeyType, ValType>());
   ...
Set<KeyType> s = m.keySet();
   ...
// Synchronizing on m, not s!
synchronized(m) {
   while (KeyType k : s)
      foo(k);
}
```

2. BlockingQueue

- BlockingQueue implementations provide the possibility for threads to wait on operations for adding or removing of elements.
- If the blocking queue is full, threads block until space becomes available for adding an element
- If the blocking queue is empty, threads block until an element is inserted in the queue so it can be removed
- BlockingQueues are used to hold tasks submitted to an Executor thread pool. For example, Fixed thread pool uses by default a LinkingBlockingQueue.

```
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueue {
   ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<String>(50);

   public void add() {
       try {
           blockingQueue.add("first");
           blockingQueue.add("second");
           blockingQueue.add("third");
           System.out.println("Adding items completed");
```

```
        }
        catch (Exception e) {
            Thread.currentThread().interrupt();
        }
    }

    public void remove() {
        try {
            System.out.println(blockingQueue.take());
            System.out.println(blockingQueue.take());
            System.out.println(blockingQueue.take());
            System.out.println("Removing items completed");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public static void blockingCollectionExample(){
        BlockingQueue example = new BlockingQueue();
        new Thread(() -> {example.remove();}).start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        new Thread(() -> {example.add();}).start();
    }

    public static void main(String[] args) {
        blockingCollectionExample();
    }
}
```

```
Adding items completed
first
second
third
Removing items completed
```

3. Lock-free collections

- Lock-free collections provided by the JDK come into two flavors:
    - Copy-on-write collections - като добавяме нов елемент, всеки път се прави ново копие, където го записваме този нов елемент. И това изглежда да е скъпа операция.
    - Concurrent collections based on atomic (compare-and-swap) operations

Copy-on-write collections

- Copy-on-write collections create a new collection every time an element is added or removed
- In that regard they are immutable and can be safely accessed from multiple threads
- Copy-on-write collections require extra performance due to the copying of the collection and should be avoided in scenarios where performance is critical
- Copy-on-write collections provided by the JDK include:
    - CopyOnWriteArrayList
    - CopyOnWriteArraySet

```
private static void copyOnWriteExample() {
    CopyOnWriteArrayList<String> copyOnWriteArrayList = new CopyOnWriteArrayList<>();
    copyOnWriteArrayList.add("first");
```

```
    copyOnWriteArrayList.add("second");
    copyOnWriteArrayList.add("third");
}
```

Concurrent (specific term) collections

- Concurrent collections do not use locks, but atomic compare-and-swap (CAS) operations
- The JDK provides support for compare-and-swap instructions provided by modern CPUs
- Avoiding locks (thus context switching) makes concurrent collections more performant than synchronized, blocking and copy-on-write in many scenarios

- Concurrent collections provided by the JDK include:
  - ConcurrentHashMap
  - ConcurrentLinkedQueue
  - ConcurrentLinkedDeque
  - ConcurrentSkipListMap
  - ConcurrentSkipListSet

ConcurrentHashMap

- ConcurrentHashMap provides additional thread-safe atomic operations over a traditional HashMap such as:
  - getOrDefault(key, value)
  - putIfAbsent(key, value)
  - remove(key, value)
  - replace(key, oldValue, newValue)
  - replaceAll(function)
  - computeIfAbsent(key, function)
  - computeIfPresent(key, function)
  - compute(key, function)
  - merge(key, value, function)

```
//Ако използваме обикновен HashMap, не винаги ще ни връща 100 в долния пример със 100 нишки!!!
public class Main {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<Integer, Integer> map = new ConcurrentHashMap<>();
        map.put(1, 0);
        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                map.compute(1, (key, value) -> {
                    return value + 1;
                });
            }).start();
        }

        Thread.sleep(1000);
        System.out.println(map.getOrDefault(1, -1));
    }
}
```

III. Synchronizers

- Synchronizers provide more specific mechanisms for synchronization between a number of threads
- The JDK provides several synchronizers that can be used by applications:
  - CountDownLatch
  - CyclicBarier
  - Semaphore

- Exchanger
- Phaser

Java concurrent animated - визуално описани
https://github.com/vgrazi/java-concurrent-animated

/java-concurrent-animated$ **mvn package -DskipTests**
/java-concurrent-animated/target$ **java -jar javaConcurrentAnimated.jar**

CountDownLatch

- Used when a predefined number of releases should happen before a thread is awakened
- Един път казваме да изчака, и намаляме с countDown()

```
public class CountDownLatch {
   public static void main(String[] args) {
       java.util.concurrent.CountDownLatch latch = new java.util.concurrent.CountDownLatch(5);
       new Thread(() -> {
           try {
               latch.await();
               System.out.println("Workers have finished !");
           } catch (InterruptedException e) {
               Thread.currentThread().interrupt();
           }
       }).start();

       for (int i = 0; i < 5; i++) {
           new Thread(() -> {
               System.out.println("Starting worker: " + Thread.currentThread().getName());
               latch.countDown();
           }).start();
       }
   }
}
```

Starting worker: Thread-1
Starting worker: Thread-2
Starting worker: Thread-4
Starting worker: Thread-3
Starting worker: Thread-5
Workers have finished !

Ако `CountDownLatch(3)`, тогава резултатът е:
Starting worker: Thread-1
Starting worker: Thread-5
Starting worker: Thread-4
Workers have finished !
Starting worker: Thread-3
Starting worker: Thread-2

CyclicBarrier

- Allows a number of threads to await for a predefined number of waits after which they are released
- Извикваме n на брой пъти преди да се освободи изпълнението на нишките

```
public class CyclicBarrier {

   public static void main(String[] args) {
```

```
        java.util.concurrent.CyclicBarrier barrier = new java.util.concurrent.CyclicBarrier(3, () ->
{
            System.out.println("Workers have finished!");
        });

        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                System.out.println("Starting worker: " + Thread.currentThread().getName());
                try {
                    barrier.await();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } catch (BrokenBarrierException e) {
                    System.out.println("Ending worker: " + Thread.currentThread().getName());
                }
            }).start();
        } //barrier can be reset with barrier.reset()
    }
}
```

Starting worker: Thread-2
Starting worker: Thread-0
Starting worker: Thread-1
Workers have finished!
Starting worker: Thread-3
Starting worker: Thread-4

Semaphore

- Each thread is blocked until a permit is available, semaphore is initialized with a number of permits
- Когато искаме в даден или във всеки момент от време да се изпълняват едновременно определен брой нишки

```
public class Semaphore {

    public static void main(String[] args) throws InterruptedException {
        java.util.concurrent.Semaphore semaphore = new java.util.concurrent.Semaphore(3);
        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                try {
                    System.out.println("Starting worker: " + Thread.currentThread().getName());

                    //Acquires a permit from this semaphore, blocking until one is available, or the
thread is interrupted.
                    //Acquires a permit, if one is available and returns immediately, reducing the
number of available permits by one.
                    semaphore.acquire();

                    System.out.println("Ending worker: " + Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }).start();
        }

        Thread.sleep(2000);
        semaphore.release(); //Първите 3 нишки заемат трите части на семафората
        semaphore.release(); //четвърта и пета нишка се блокират докато не се освободи слот от
семафората за тях
    }
}
```

Starting worker: Thread-2

Starting worker: Thread-0
Ending worker: Thread-2
Starting worker: Thread-3
Ending worker: Thread-3
Starting worker: Thread-1
Starting worker: Thread-4
Ending worker: Thread-0
Ending worker: Thread-1
Ending worker: Thread-4

Exchanger

- Provides the possibility to two threads to exchange (swap) objects

```
public class Exchanger {

    public static void main(String[] args) {
        java.util.concurrent.Exchanger exchanger = new java.util.concurrent.Exchanger();

        for (int i = 0; i < 2; i++) {
            new Thread(() -> {
                try{
                    Random random = new Random();
                    Integer value = random.nextInt();
                    Integer exchanged = (Integer) exchanger.exchange(value);
                    System.out.println("Exchanged value " + value + " for " + exchanged);
                } catch (InterruptedException e){
                    Thread.currentThread().interrupt();
                }
            }).start();
        }
    }
}
```

Веднъж се разменят стойностите, като първата нишка блокира, докато не е готова и втората- така че да обменят стойности.
Exchanged value 1878992229 for 2140338631
Exchanged value 2140338631 for 1878992229

Phaser

- Similar to CountDownLatch and CyclicBarrier, but provides the ability **to change dynamically the number of awaiting threads** before they can proceed

```
public class Phaser {

    public static void main(String[] args) {
        java.util.concurrent.Phaser phaser = new java.util.concurrent.Phaser(1);
        phaser.bulkRegister(2);//още два пъти да се извика метода, преди да продължи приложението


        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " arriving at phaser");
                phaser.arriveAndAwaitAdvance();
                System.out.println(Thread.currentThread().getName() + " leaving the phaser");
            }).start();
        }
    }
}
```

```
Thread-0 arriving at phaser
Thread-2 arriving at phaser
Thread-1 arriving at phaser
Thread-0 leaving the phaser
Thread-2 leaving the phaser
Thread-1 leaving the phaser
```

Ако **`phaser.bulkRegister(3), то чакаме за общо четири нишки да влязат, а в нашия случай четвърта никога не идва….`**

```
Thread-1 arriving at phaser
Thread-0 arriving at phaser
Thread-2 arriving at phaser
```

IV. Atomic operations

Compare-and-Swap (CAS)

- Compare-and-Swap is an atomic instruction that compares the location in memory with a given value, and only if they are equals then sets a new value
- CAS is used to implement atomic operations that achieve **optimistic locking** and are thread-safe

```
// represented by the following pseudocode
if (memoryLocation != value) {
        return false;
}
memoryLocation = newValue;
return true;
```

- The JDK provides support for calling CAS instructions through native code as provided by the **jdk.internal.misc.Unsafe** class.
    - Това е клас, който се ползва вътрешно от самата JVM на много места при операции свързани със синхронизация на нишки; Преди се е използвал този клас за приложения, които искат да заделят отделна памет извън HEAP паметта на JVM (unmanaged memory not allocated for the JVM); Не случайно се казва Unsafe - защото ако не се използва внимателно, може да доведе до сериозни проблеми
    - Apart from performing CAS operations the Unsafe class also provides the possibility to access off-heap memory. But since it is an internal class, it is encapsulated as of JDK 9 so cannot be used directly

- An alternative of Unsafe as of JDK 9 that provides support for CAS is provided by the **java.lang.invoke.VarHandle** class

```
import jdk.internal.misc.Unsafe;
import java.lang.invoke.VarHandle;

public class Main {
   public static void main(String[] args) {
       Unsafe.compareAndSetInt();
       VarHandle.compareAndSet();
   }
}
```

Atomic variables

- Maintaining a single variable that is updatable from many threads is a common scalability issue
- Atomic variables are already present in the JDK - they serve as a means to implement updatable variables in a multithreaded environment
- Atomic variables are part of the **java.util.concurrent.atomic** package
- They make use of the CAS support provided by the JDK to provide performant thread-safe operations over shared variables
- The other utilities that we already discussed that make use of CAS are concurrent collections (specific term) like **ConcurrentHashMap**

Atomic variables are provided by the following classes:

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicLong
- AtomicLongArray
- AtomicReference
- AtomicReferenceArray
- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

Пример за AtomicInteger

```
   public static void main(String[] args) {

       AtomicInteger value = new AtomicInteger();
       Thread thread = new Thread(() -> {
           value.getAndIncrement();
       });

       thread.start();
       value.getAndAdd(10);
       try {
           thread.join();
       } catch (InterruptedException e) {
           throw new RuntimeException(e);
       }
       System.out.println(value.get()); //11
   }
```

java.util.concurrent.atomic.AtomicInteger:

```
private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();

public final int getAndIncrement() {
   return U.getAndAddInt(this, VALUE, 1);
}
```

jdk.internal.misc.Unsafe:

```
@HotSpotIntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
   int v;
   do {
       v = getIntVolatile(o, offset);
   } while (!weakCompareAndSetInt(o, offset, v, v + delta));
   return v;
}

@HotSpotIntrinsicCandidate
public final boolean weakCompareAndSetInt(Object o,
                                          long offset, //offset в  паметта
```

```
                                            int expected, //ако тази стойност вече е налична в паметта
                                            int x) { //новата стойност x ако expected е налично
    return compareAndSetInt(o, offset, expected, x);
}



/**
* Atomically updates Java variable to {@code x} if it is currently
* holding {@code expected}.
*
* <p>This operation has memory semantics of a {@code volatile} read
* and write.  Corresponds to C11 atomic_compare_exchange_strong.
*
* @return {@code true} if successful
*/
@HotSpotIntrinsicCandidate
public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);
```

native ще рече, че се извиква отдолу в JVM, вероятно на C++, Compare-and-Swap процесорна инструкция.

Пример за DoubleAccumulator

```
import java.util.concurrent.atomic.DoubleAccumulator;

public class Main {
    public static void main(String[] args) {

        DoubleAccumulator accumulator = new DoubleAccumulator((x, y) -> x + y, 0);
        Thread thread = new Thread(() -> {
            accumulator.accumulate(0.9);
        });

        thread.start();
        accumulator.accumulate(10.1);
        try {
            thread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(accumulator.get()); //11
    }
}
```

Пример за AtomicReference

Използва се за обекти от всякакъв тип, включително и за Стринг.

```
public class Main {
    public static void main(String[] args) {
        AtomicReference<String> reference = new AtomicReference<String>("first");
        Thread thread = new Thread(() -> {
            reference.getAndAccumulate(" some text", (x, y) -> x + y);
        });

        thread.start();
        reference.getAndAccumulate(" otherText", (x, y) -> x + y);
        try {
            thread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(reference.get());
```

```
    }
```

Не се гарантира поредността, но се гарантира thread safety
first otherText some text

StringBuffer

Thread-safe operations.
Но не е реализирано със CAS, а със **copy-on-write - т.е. за повече обем данни може да е бавна операцията**

V. Concurrency utilities

Fork/join framework

- **ForkJoinPool** is an implementation of the **ExecutorService** interface introduced in JDK 7
- The fork/join framework is distinct because it uses a ***work-stealing*** **algorithm**. Worker threads that run out of things to do can steal tasks from other threads that are still busy.
- Provides the possibility to execute tasks that can be organized in a **divide-and-conquer** manner - разбиване на подзадачи например като сортиране с **mergeSort** алгоритъм (see Algorithms with Java notes) при многонишкова среда

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

- Tasks submitted to the **ForkJoinPool** are represented by a **ForkJoinTask** instance

- Typical implementations of parallel tasks do not extend directly **ForkJoinTask**

- **RecursiveTask** instances can be used to execute parallel tasks that return a result

- **RecursiveAction** instances can be used to execute parallel tasks that do NOT return a result

- A **global ForkJoinPool instance** is used for any ForkJoinTasks that are not submitted to a particular ForkJoinPool

- To get a reference to the global ForkJoinPool instance, the static **ForkJoinPool.commonPool()** method can be used

- This is the case with **parallel streams**: they make use of the common ForkJoinPool for task execution

```
import java.util.concurrent.RecursiveAction;

public class NumberFormatAction extends RecursiveAction {
    int start;
    int end;

    public NumberFormatAction(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
```

```
    protected void compute() {
        if (end - start <= 2) {
            System.out.println(start + " " + end);
        } else {
            int mid = start + (end - start) / 2;
            NumberFormatAction left = new NumberFormatAction(start, mid);
            NumberFormatAction right = new NumberFormatAction(mid + 1, end);
            java.util.concurrent.ForkJoinTask.invokeAll(left, right);
        }
    }
}

public static void main(String[] args) {
    NumberFormatAction action = new NumberFormatAction(1, 50);
    ForkJoinPool.commonPool().invoke(action);
}
```

```
Thread.sleep(1000);
System.out.println(Thread.currentThread()); - връща името на текущата нишка
System.out.println(ForkJoinPool.commonPool());
```

```
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
    System.out.println(Runtime.getRuntime().availableProcessors()); - връща 1 ядро по-малко,
защото едно ядро е заето с нишката на main метода
    System.out.println(ForkJoinPool.commonPool()); - - връща състоянието на pool-а
```

Parallel streams / parallel programming

- Parallel streams can be created as regular streams with the **parallelStream** method

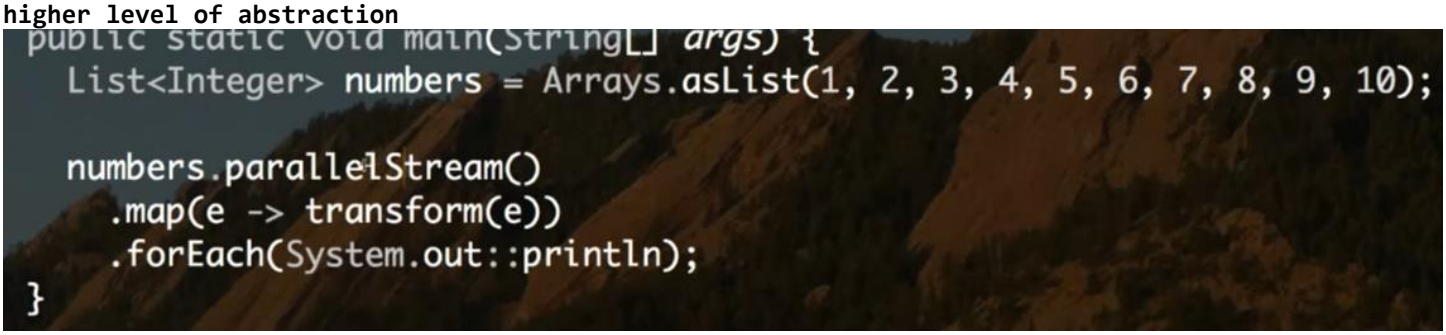
```
List list = new ArrayList();
list.parallelStream();
```

- Parallel streams should be used **only for time-intensive tasks rather than IO**
- In many cases regular streams outperform parallel ones so they need to be used with caution - затова винаги е добре когато мислим да използваме паралелни потоци, то да им сравним бързодействието с нормални потоци

```
threads and fibers – нишки и влакна – ниско ниво код близко до hardware

public class Main implements Runnable {
    public static void main(String[] args) {
        new Main().run();
    }

    @Override
    public void run() {
        int n = 10;

        while (n-- > 0){
            Thread thread = new Thread(this::doSomething); //вземаме нишка от операционната
система / изключително скъпа операция
            thread.start();
        }

    }

    private void doSomething() {
```

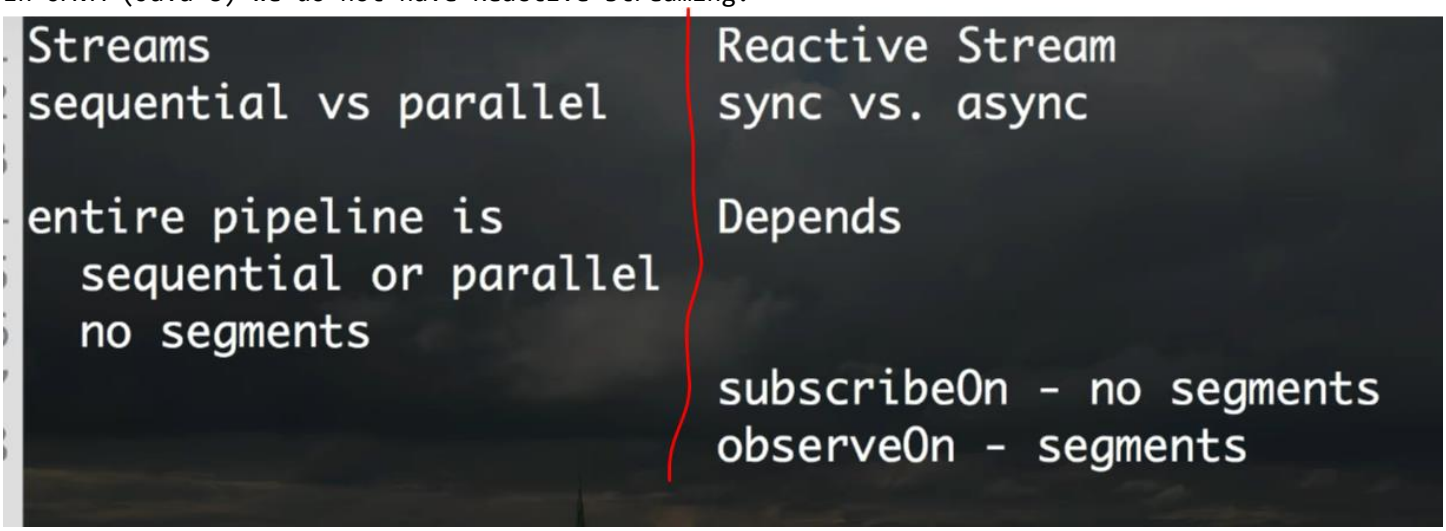
```
        long z = 0;

        while (true){
            z++;
        }
    }
}
```

Imperative style – обикновени цикли, if-else клаузи – как ще се случи
Functional/Declarative style – какво ще се случи (без значение как), не се интересуваме от самото изпълнение, а от логиката - .stream – този стил е много близък до многонишковото програмиране и е higher level of abstraction

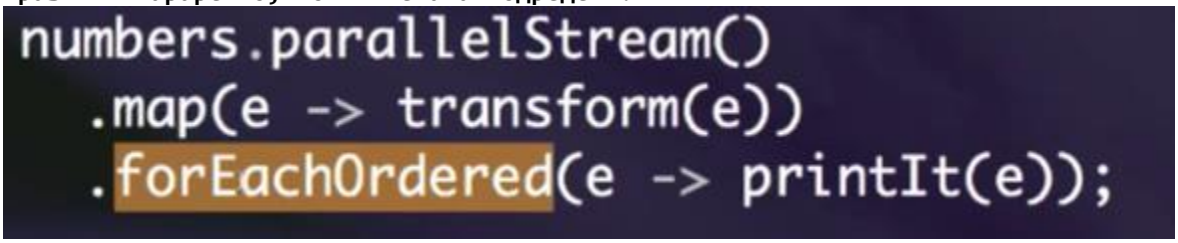



```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
numbers.stream().parallel();  равно на numbers.parallelStream();- произволен ред ги печата

numbers.stream().sequential(); - в последователен ред ги печата.
```

In JAVA (Java 8) we do not have Reactive streaming.



Прави ги парарелно, но ги печата подредени.



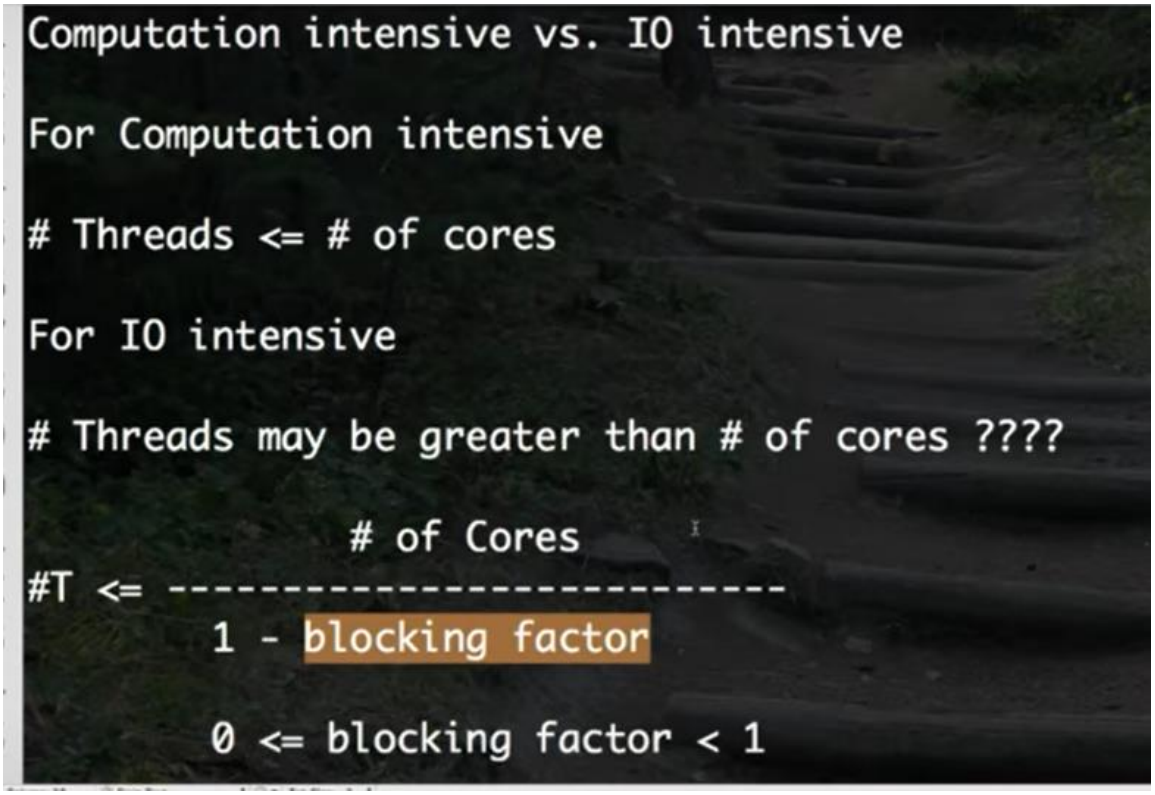
Примери с **.reduce() метода: - като агрегатор**

```java
//Вземи сбора на всички числа
int[] numbers = new int[]{1, 2, 3, 4, 5};
int sum = Arrays.stream(numbers).reduce(0, (val, num) -> val += num);

//вземи най-дългата дума
String[] words = new String[]{"hello", "pesho", "abc", "worlddd"};
String longestWord = Arrays.stream(words).reduce("", (val, w) -> {
    if (w.length() > val.length()) {
        val = w;
    }
    return val;
});

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
numbers.parallelStream().reduce(0, (total,e) -> add(total, e));  //0 е identity value
private static Integer add(Integer total, Integer e) {
    return total+= e;
}

//reduce does not take initial value, it takes identity value – it is the value which will not change the operation if you apply it on
X + identity value 0
Y * identity value 1
```

Да внимаваме – ако превключим от sequential към parallel, identity value трябва да е коректно зададено. Не е ок да напишем 36 години, при паралелното дава друг резултат.

The question is – How many threads should I create? How much food should I eat?
Ако създам прекалено много нишки, даже по-бавно би работило.

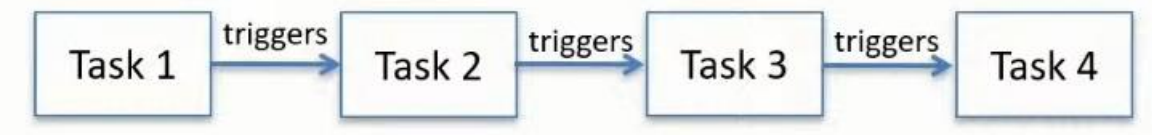
IO intensvive – прави паузи често, тогава може да имаме повече нишки



Configuring number of threads JVM wide:
-Djava.util.concurrent.ForkJoinPool.common.parallelism=100 100 нишки

Completable Future

- Provides a facility to create a chain of dependant non-blocking tasks
- **No divide-and-conquer - it could be any relation of operations - i.e. a smarter way of organization when using CompletableFuture**
- An asynchronous task can be triggered as the result of a completion of another task



- A CompletableFuture may be completed/canceled by a thread prematurely (преждевременно)

- Provides a very flexible API that allows additionally:
  - To combine the result of multiple tasks in a CompletableFuture
  - To provide synchronous/asynchronous callbacks upon completion of a task
  - To provide a CompletableFuture that executes when first task in a group completes
  - To provide a CompletableFuture that executes when all tasks in a group complete

1. Най-простият пример

```
import java.util.concurrent.CompletableFuture;

public class Main {
   public static void main(String[] args) {
       CompletableFuture<Integer> task1 = new CompletableFuture<>();

       // forcing completing of future by specifying result
       boolean complete = task1.complete(10);
   }
}
```

2.

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException

       CompletableFuture<Integer> task1 = CompletableFuture.supplyAsync(() -> {
           //some code logic here
           return 10;
       });

       //executed on completion of the future
       CompletableFuture<Integer> integerCompletableFuture = task1.thenApply((x) -> x * x);
       System.out.println(integerCompletableFuture.get()); //100

       //executed in case of exception or completion of the future
       task1.handle((x, y) -> {
           return ............
       });
       System.err.println(task1.get());

       //can be completed prematurely with a result
       task1.complete(20);
   }
}
```

3. Верига от задачи

```
CompletableFuture<Object> prev = null;
Supplier<Object> supplier = () -> {};

for (int i = 0; i < count; i++) {
   CompletableFuture<Object> task;
   if (prev != null) {
       task = prev.thenCompose(x -> {
           return CompletableFuture.supplyAsync(supplier);
       });
   } else {
       task = CompletableFuture.supplyAsync(supplier);
   }

   prev = task;
}

prev.get(); //изпълнява целия chain
```

A few more things

- There is a **Timer** utility/class that can be used for scheduling tasks, but a **scheduled executor thread pool is more preferable as a more robust alternative**

- A **ThreadLocalRandom** utility is provided since JDK 7 that can be used to generate random numbers for the current thread

- A **Flow** interface introduced in JDK 9 provides a **reactive streams specification for the JDK** that can be used to provide a publish-subscribe mechanism - frameworks like in Spring (implemented by WebFlux reactor) and in Quarkus(implemented by Mutiny)
    - Парадигмата за реактивно програмиране идва от Microsoft като постепенно се имплементира в различни програмни езици. Идеята на реактивното програмиране е да се изпълняват само неблокиращи операции и се касае за IO операции(в това число http протокол, web socket, комуникация с базата данни и т.н.) в не малка част от случаите! Идеята за реактивното програмиране решава въпроса за скалируемост - да може например повече да се бомбардирва със http заявки даден http endpoint

VI. Testing concurrent applications

- Testing concurrent applications for correctness is inherently difficult
- In many cases non-deterministic unit tests are written that try to simulate running a particular piece of code in a multithreaded manner
- Testing frameworks provide certain utilities that can be used to facilitate testing of concurrent applications

- A classical way of running multiple threads against code-under-test in a unit test is to use **CountDownLatch:**

```
@RepeatedTest(10)
public void testLinkedList()
        throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(100);
        for(int i = 0; i < 100; i++) {
                new Thread( () -> {
                        // unit under test …
                        latch.countDown();
                }).start();
        }
        latch.await();
        // perform asserts
}
```

- Frameworks like **ThreadWeaver** provide the possibility to interleave execution threads - interleaving happens using breakpoints (line by line) so that the unit-under-test is tested using different thread ordering

- For performance testing a framework like **JMH**(the ***Java Microbenchmark Harness***) can be used to perform proper application warmup before the test is executed - самото JDK се тества с JMH

VII. Latest concurrency enhancements

- At the time of 2023/2024, JDK 21 is the latest LTS (long-term support) release of JDK 21
- A few but MAJOR additions for writing concurrent applications are introduced:
  - Virtual threads (aka fibers) provided by **Project Loom**
  - Scoped values
  - Structured concurrency

Virtual threads

- Virtual threads are “lightweight” threads scheduled by the JVM
- In comparison with standard platform threads, they are not bound directly to OS threads
- They are still executed by an OS thread
- They are an instance of the **java.lang.Thread** class - същата инстанция както и стандартните нишки
- Virtual threads are very suitable for high-throughput concurrent application
- Typical example are Java server applications where many threads are blocked and waiting
- Not suitable for long running operations

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
   public static void main(String[] args) {

   }

   private static void createSimpleVirtualThread() throws InterruptedException {
       Thread thread = Thread.ofVirtual().start(() -> {...});
       thread.join();
   }
```

```
    private static void createVirtualThreadPool() throws InterruptedException, ExecutionException {
        ExecutorService virtualThreadPool = Executors.newVirtualThreadPerTaskExecutor();
        Future<?> task = virtualThreadPool.submit(() -> {...});
        task.get();
    }
}
```

4. Asynchronous programming

What is Asynchronous Programming?

Let's delve into an analogy of asynchronous programming: Imagine waking up in the morning and starting to boil some eggs for breakfast. Instead of waiting idly for them to cook, you begin another task, such as baking bacon and toast. In this scenario, you're akin to a thread executing I/O operations - you initiate them and let them run independently. Both tasks are non-blocking for you; while the food cooks, you're free to sit at the table, enjoy some YouTube videos, or even take a shower. Once any task is completed (for instance, the bacon is ready), you handle it by placing it on your plate. Similarly, when the eggs are done, you remove them from the water and prepare them to be eaten.

Consider another real-world example of asynchronous operations: a waiter at a restaurant taking orders. Upon receiving an order from a customer, he relays it to the kitchen for preparation. Rather than waiting beside the kitchen counter for the dish to be ready – an approach similar to a Java application operating on a single thread - the waiter moves on to attend to another customer's order. As soon as a meal is prepared, he retrieves it from the kitchen and serves it to the respective customer.

This process mirrors how an application thread behaves during an API call, initiating a time-consuming operation on an external system or executing a complex database query that takes several seconds or more.

Real-life scenarios also require exception handling, akin to software development. For instance, if the waiter takes an order for pasta bolognese but finds out that the kitchen has run out of beef, it poses a resource synchronisation issue typical in asynchronous operations.

Modern web applications, particularly those hosted in cloud environments, need to accommodate thousands of simultaneous users. This scalability is achieved not only through service replication, such as pod replication in EKS, but also by making efficient use of threads within each pod at the application level. Asynchronous operations facilitate non-blocking I/O, leading to more efficient use of resources.

Async vs parallel operations: Asynchronous programming focuses on non-blocking tasks, while parallel programming involves executing multiple computations simultaneously, leveraging multi-core processors. Both approaches improve performance but are used for different purposes.

Async Programming in Other Languages

C# in the .NET framework utilises the Task<T> class and async and await keywords, making asynchronous programming more straightforward and cleaner. An async method returns a Task or Task<T>, which represents ongoing work. C# similar to Java utilises a thread pool for executing asynchronous tasks. When an async method awaits an asynchronous operation, the current thread is returned to the thread pool until the awaited operation completes.

C/C++

```
public async Task<string> GetDataAsync() {
var data = await GetDataFromDatabaseAsync();
return data;
}
```

JavaScript handles asynchronous operations through Promises and async/await syntax, fitting its event-driven nature. A Promise represents an operation that hasn't been completed yet but is expected in the future. JavaScript, particularly in the Node.js environment, operates on a single-threaded event loop for handling asynchronous operations.

JavaScript

```
async function fetchData() {
let response = await fetch('https://api.example.com/data');
let data = await response.json();
console.log(data);
}
```

```
In JAVA it is CompletableFuture
In JS it is Promises
```

Java's CompletableFuture Class

Java's **CompletableFuture** class was introduced in **Java 8. CompletableFuture** is part of Java's **java.util.concurrent** package and provides a way to write asynchronous code by representing a future result that will eventually appear. It lets us perform operations like calculation, transformation, and action on the result without blocking the main thread. This approach helps in writing non-blocking code where the computation can be completed by a different thread at a later time.

CompletableFuture and the broader **Java Concurrency API** make use of thread pools (**like the ForkJoinPool**) for executing asynchronous operations. This allows Java applications to handle multiple asynchronous tasks efficiently by leveraging multiple threads.

```
CompletableFuture.supplyAsync(() -> {
        return "Result of the asynchronous computation";
        })
        .thenAccept(System.out::println);
```

```
public static CompletableFuture<Integer> create() {
    return CompletableFuture.supplyAsync(() -> 2);
}
```

In Java, when a **CompletableFuture** operation is waiting on a dependent future or an asynchronous computation, it doesn't block the waiting thread. Instead, the completion of the operation triggers the execution of dependent stages in the **CompletableFuture** chain, potentially on a different thread from the thread pool.

Example Scenario with CompletableFuture

Let's consider a scenario where we need to perform a series of dependent and independent asynchronous operations:

1. **Fetch User Details**: Given a userID, we first retrieve the user's details asynchronously.
2. **Fetch Credit Score**: Once we have the user's details, we fetch their credit score.
3. **Calculate Account Balance**: Independently, we also calculate the user's account balance from a different source.
4. **Make a Decision**: Finally, we combine the credit score and account balance to make a financial decision.
5. **Handle Potential Errors**

Step 1: Fetching User Details Asynchronously

We start by simulating an asynchronous operation to fetch user details using **supplyAsync**. This returns a **CompletableFuture** that will complete with the user details:

```
CompletableFuture<String>  getUserDetailsAsync(String userId) {
        return CompletableFuture.supplyAsync(() -> "UserDetails for " + userId);
}
```

Step 2: Transforming and Fetching Credit Score

Next, we use **thenApply** to transform the result (e.g., formatting user details) and **thenCompose** to fetch the credit score, demonstrating the chaining of asynchronous operations:

```
CompletableFuture<String>  userDetailsFuture = getUserDetailsAsync("userId123")
        .thenApply(userDetails -> "Transformed " + userDetails);

CompletableFuture<Integer>  creditScoreFuture = userDetailsFuture
        .thenCompose(userDetails -> getCreditScoreAsync(userDetails));
```

thenApply is for synchronous transformations, while **thenCompose** allows for chaining another asynchronous operation that returns a **CompletableFuture**.

Step 3: Calculating Account Balance in Parallel

We calculate the account balance using another asynchronous operation, showcasing how independent futures can run in parallel:

```
CompletableFuture<Double> accountBalanceFuture = calculateAccountBalanceAsync("userId123");
```

Step 4: Combining Results and Making a Decision

With **thenCombine** we merge the results of two independent **CompletableFuture** – credit score and account balance – to make a decision:

```
CompletableFuture<Void> decisionFuture = creditScoreFuture
        .thenCombine(accountBalanceFuture, (creditScore, accountBalance) ->
                        makeDecisionBasedOnCreditAndBalance(creditScore, accountBalance))
        .thenAccept(decision -> System.out.println("Decision: " + decision));
```

Step 5. Error Handling

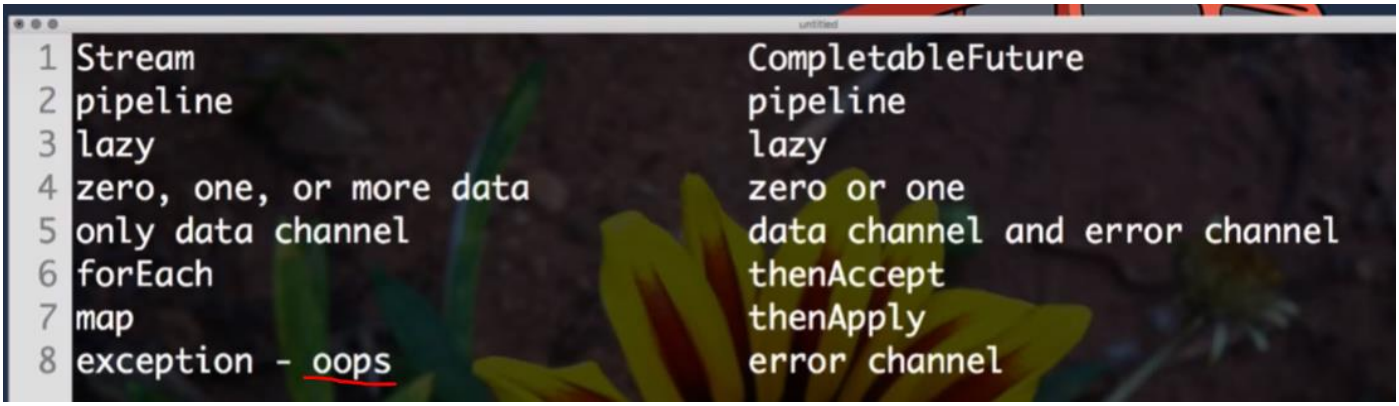
Error handling is crucial in asynchronous programming. We use exceptionally to handle any exceptions that may occur during the asynchronous computations, providing a way to recover or log errors:

```
.exceptionally(ex -> {
        System.err.println("An error occurred: " + ex.getMessage());
        return null;
});
```

CompletableFuture from Indian UK guy

Some Info

```
CompletableFuture<Integer> future = create();
CompletableFuture<Void> future2 = future.thenAccept(data -> System.out.println(data));

create()
        .thenAccept(data -> System.out.println(data))
        .thenApply(d -> d * 10)
        .thenRun(() -> System.out.println("This never dies"));
```




```
.thenAccept(data -> System.out.println(data))
Consumer<String> printer = str -> System.out.println(str);
Arrays.stream(sc.nextLine().split("\\s+")).forEach( e-> printer.accept(e));
```

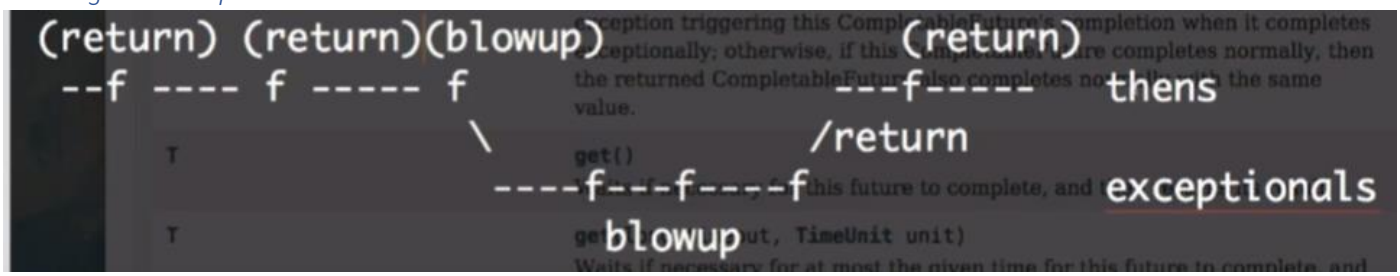
```
.thenApply(d -> d * 10)
Function<Integer, Integer> squared = x -> x * x; - първият е входните данни, втория параметър е типа на изхода
System.out.println(squared.apply(25)); //връща 625
```

```
.thenRun(() -> System.out.println("This never dies"));
Supplier<Integer> genRandomInt = () -> new Random().nextInt(51);
int rnd = genRandomInt.get();
```

Adding data to the pipeline

```
public static void main(String[] args) throws InterruptedException {
    CompletableFuture<Integer> future = new CompletableFuture<Integer>();
    future
            .thenApply(d -> d * 2)
            .thenApply(d -> d + 1)
            .thenAccept(data -> System.out.println(data));
    System.out.println("We built the pipeline");
    System.out.println("Prepared to print");

    sleep(1000);

    future.complete(2);
    sleep(1000);
}
```

Working with exceptions



```java
public static CompletableFuture<Integer> create() {
    return CompletableFuture.supplyAsync(() -> 2);
}

public static void main(String[] args) {
    create()
            .thenApply(data -> data * 2)
            .exceptionally(thr -> handleException(thr))
            .thenAccept(data -> System.out.println(data));
}

private static Integer handleException(Throwable thr) {
    System.out.println("ERROR: " + thr);
    throw new RuntimeException("It is beoynd all hope");
}
```

`future.completeExceptionally(new RuntimeException());` - с две думи нищо не сме свършили

Състояния на CompletableFuture:

Pending (final)
Resolved state (final)
Rejected (final)

`Future.orTimeout();`

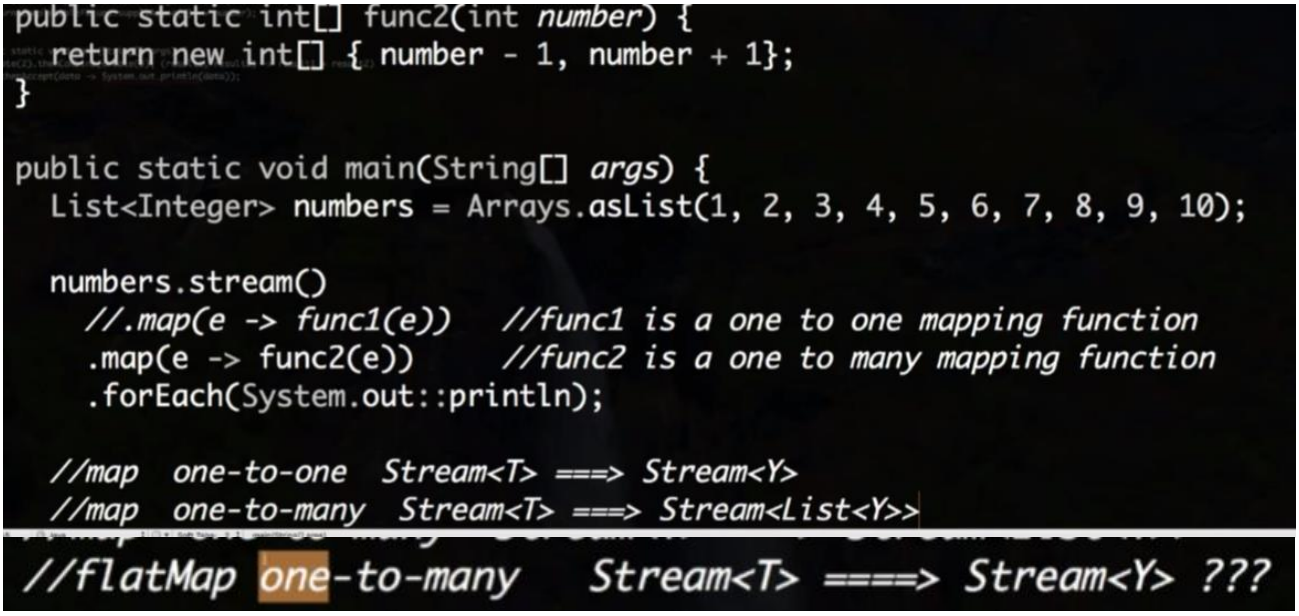
Combine and compose

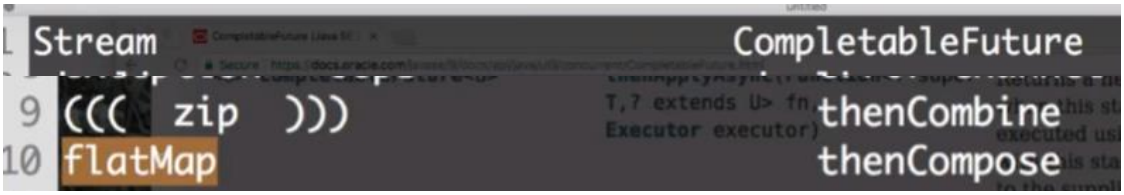
```
//JavaScript
then(e => func(e))  ➔
```

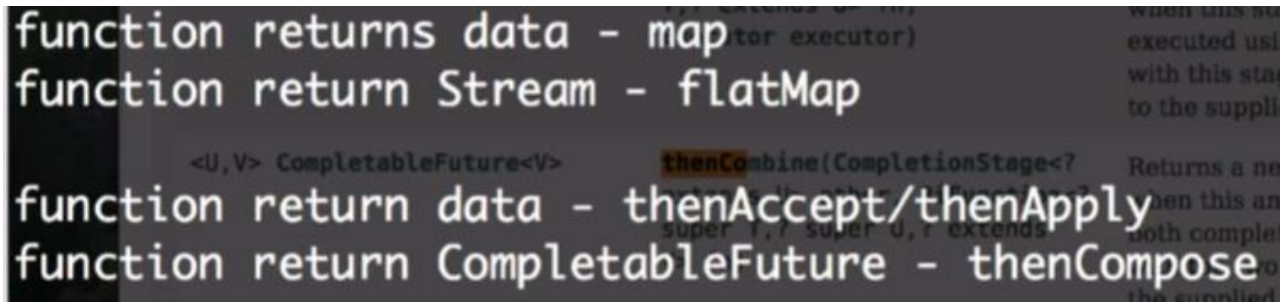
In JavaScript func may return data or return a promise
If date is returned, it is wrapped into a promise
If promise is returned, then that is returned from the then

In JavaScript the return type could be any type, but in JAVA the return type should always be T (what we started with)

```java
public static CompletableFuture<Integer> create(int number) {
    return CompletableFuture.supplyAsync(() -> number);
}
public static void main(String[] args) throws InterruptedException {
    create(2).thenCombine(create(3), (result1, result2) -> result1 + result2)
            .thenAccept(data -> System.out.println(data));
}
```








```java
public static CompletableFuture<Integer> create(int number) {
    return CompletableFuture.supplyAsync(() -> number);
}
public static CompletableFuture<Integer> inc(int number){
    return CompletableFuture.supplyAsync(() -> number + 1);
}

create(2)
//                .thenApply(data -> inc(data))
                .thenCompose(data -> inc(data))
                .thenAccept(result -> System.out.println(result));
```

Using CompletableFutures Effectively

Most Important Methods

1. supplyAsync

The **supplyAsync** method is part of the **CompletableFuture** class introduced in Java 8, residing in the **java.util.concurrent package**. It's designed to run a piece of code asynchronously and return a **CompletableFuture** that will be completed with the **value** obtained from that code. Essentially, it allows you to execute a **Supplier<T>** asynchronously, where **T** is the type of **value** returned by the Supplier.

Syntax and Usage:

```
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

- **U**: The type of value obtained from the **Supplier<U>**
- supplier: A **Supplier<U>** that provides the **value** to be used in the completion of the returned **CompletableFuture<U>**

Simple example:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        // Simulate a long-running operation
        try {
                TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }

        return "Result of the asynchronous operation";
});
```

When you invoke **supplyAsync**, it executes the given **Supplier** asynchronously (usually in a different thread). The method immediately returns a **CompletableFuture** object. This **CompletableFuture** will be completed in the future when the **Supplier** finishes its execution, with the result being the value provided by the **Supplier**.

It allows the main thread to continue its operations without waiting for the task to complete. This is particularly useful in web applications or any I/O-bound applications where you don't want to block the current thread.

By default, tasks submitted via **supplyAsync** without specifying an executor are executed in the common fork-join pool (**ForkJoinPool.commonPool()**). However, you can also specify a custom **Executor** if you need more control over the execution environment:

```
Executor executor = Executors.newCachedThreadPool();
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        // Task here
        return "Result";
          }, executor);
```

2. runAsync

CompletableFuture.runAsync is akin to **CompletableFuture.supplyAsync** but for scenarios where you don't need to return a value from the asynchronous operation. Both methods are intended for executing tasks asynchronously, but they differ in their return types and the type of tasks they're suited for.

runAsync is used to execute a **Runnable** task asynchronously, which does not return a result. Since **Runnable** does not produce a return value, **runAsync** returns a **CompletableFuture<Void>**.

```
static CompletableFuture<Void> runAsync(Runnable runnable)
```

```
static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
```

- **Asynchronous Execution**: Executes the given **Runnable** task in a separate thread, allowing the calling thread to proceed without waiting for the task to complete.
- **No Return Value**: Suitable for asynchronous tasks that perform actions without needing to return a result, such as logging, sending notifications, or other side effects.
- **Custom Executor Support**: Allows specifying a custom **Executor** for more control over task execution, such as using a dedicated thread pool.

Here's an example that demonstrates using **runAsync** to execute a simple asynchronous task:

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        // Simulate a task that takes time but doesn't return a result
        try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println("Task completed");
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }
});

// Do something else while the task executes

future.join(); // Wait for the task to complete if necessary
```

3. get()

The **get()** method blocks the current thread until the **CompletableFuture** completes, either normally or exceptionally. Once the future completes, **get()** returns the result of the computation if it completed normally, or throws an exception if the computation completed exceptionally.

1. **Blocking Behaviour**: Like **join()** ,**get()** is a blocking call. It makes the caller thread wait until the **CompletableFuture**'s task is completed.
2. **Checked Exceptions**: **get()** can throw checked exceptions, including:
    a. **InterruptedException**: If the current thread was interrupted while waiting.
    b. **ExecutionException**: If the computation threw an exception. This exception wraps the actual exception thrown by the computation, which can be obtained by calling **getCause()** on the **ExecutionException**.

3. **Timeout**: The overloaded version of **get(long timeout, TimeUnit unit)** allows specifying a maximum wait time. If the timeout is reached before the future completes, it throws a **TimeoutException**, providing a mechanism to avoid indefinite blocking.

4. **Use Case**: Use **get()** when you need to handle checked exceptions explicitly, or when you need to retrieve the result of the computation within a certain timeframe.

```
Example:
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        // Simulate a long-running operation
        try {
                TimeUnit.SECONDS.sleep(2); // 2-second delay
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }
        return "Result of the asynchronous operation";
});

try {
        // Attempt to retrieve the result, waiting up to 3 seconds
```

```
            String result = future.get(3, TimeUnit.SECONDS);
            System.out.println(result);
} catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("The current thread was interrupted while waiting.");
} catch (ExecutionException e) {
            System.out.println("The computation threw an exception: " + e.getCause());
} catch (TimeoutException e) {
            System.out.println("Timeout reached before the future completed.");
}
```

4. join()

The **join** method on a **CompletableFuture** is a blocking call that causes the current thread to wait until the **CompletableFuture** is completed. During this waiting period, the current thread is inactive, essentially "joining" the completion of the task represented by the **CompletableFuture**.

- **Blocking Behaviour**: **join()** blocks until the **CompletableFuture** upon which it is called completes, either normally or exceptionally. It makes the asynchronous operation synchronous for the calling thread, as the thread will not proceed until the future is completed.
- **Exception Handling**: Unlike **get()**, which throws checked exceptions (such as **InterruptedException** and **ExecutionException**), **join()** is designed to throw an unchecked exception (**CompletionException**) if the **CompletableFuture** completes exceptionally. This can simplify error handling in certain contexts where checked exceptions are undesirable.
- **Usage**: It's typically used when you need to synchronize asynchronous computation at some point, for example, when the result of the asynchronous computation is required for subsequent operations, or at the end of a program to ensure that all asynchronous tasks have completed.

Example scenario:

If you have a main application thread that kicks off an asynchronous task using **CompletableFuture.runAsync()** or **CompletableFuture.supplyAsync()**, and later in the program you need the result of that task or need to ensure that the task has completed before proceeding, you might call **join()**:

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            // Long-running task
            try {
                        TimeUnit.SECONDS.sleep(1); // Simulates a delay
            } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
            }
            System.out.println("Async task finished");
});

System.out.println("Waiting for the async task to complete...");
future.join(); // Blocks here until the above task completes
System.out.println("Main thread can now proceed");
```

5. thenApply(Function<? super T, ? extends U> fn)

- **Purpose**: Applies a synchronous transformation function to the result of the **CompletableFuture** when it completes.
- **Behaviour**: Executes on the same thread that completed the previous stage, or in the thread that calls **get()** or **join()** if the future has already completed.
- **Return Type**: **CompletableFuture<U>** where U is the type returned by the function.

6. thenApplyAsync(Function<? super T, ? extends U> fn)

- **Purpose**: Similar to **thenApply**, but the transformation function is executed asynchronously, typically using the default executor.
- **Behaviour**: Can execute the function in a different thread, providing better responsiveness and throughput for tasks that are CPU-intensive or involve blocking.
- **Return Type**: **CompletableFuture<U>**

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 42);
CompletableFuture<String> applied = future.thenApply(result -> "Result: " + result);
applied.thenAccept(System.out::println); // Prints: Result: 42
CompletableFuture<String> appliedAsync = future.thenApplyAsync(result -> "Async Result: " + result);
appliedAsync.thenAccept(System.out::println); // Prints: Async Result: 42
```

7. thenCombine(CompletionStage<? extends V> other, BiFunction<? super T, ? super V, ? extends U> fn)

- **Purpose**: Combines the result of this **CompletableFuture** with another asynchronously computed value. The combination is done when both futures complete.
- **Behaviour**: The **BiFunction** provided is executed synchronously, using the thread that completes the second future.
- **Return Type**: **CompletableFuture<U>**

8. thenCombineAsync(CompletionStage<? extends V> other, BiFunction<? super T, ? super V, ? extends U> fn)

- **Purpose**: Similar to **thenCombine**, but the **BiFunction** is executed asynchronously.
- **Behaviour**: Useful when the combination function is computationally expensive or involves blocking.
- **Return Type**: **CompletableFuture<U>**

```
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> 40);
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 2);
CompletableFuture<Integer> combined = future1.thenCombine(future2, Integer::sum);
combined.thenAccept(result -> System.out.println("Sum: " + result)); // Prints: Sum: 42

CompletableFuture<Integer> combinedAsync = future1.thenCombineAsync(future2, Integer::sum);
combinedAsync.thenAccept(result -> System.out.println("Async Sum: " + result)); // Prints: Async Sum: 42
```

9. thenAccept and thenAcceptAsync

- **Purpose**: Consumes the result of the **CompletableFuture** without returning a result. **thenAccept** is synchronous, while **thenAcceptAsync** is asynchronous.
- **Use Case**: Useful for executing side-effects, such as logging or updating a user interface, with the result of the **CompletableFuture**.

```
CompletableFuture<Void> accepted = CompletableFuture.supplyAsync(() -> "Hello")
.thenAccept(result -> System.out.println(result + ", World!")); //Prints: Hello, World!

CompletableFuture<Void> acceptedAsync = CompletableFuture.supplyAsync(() -> "Async Hello")
.thenAcceptAsync(result -> System.out.println(result + ", World!")); //Prints: Async Hello, World!
```

10. thenRun and thenRunAsync

- **Purpose**: Executes a **Runnable** action when the **CompletableFuture** completes, without using the result of the future. **thenRun** is synchronous, while **thenRunAsync** is asynchronous.
- **Use Case**: Useful for triggering actions that do not depend on the future's result, such as signalling completion or cleaning up resources.

```
CompletableFuture.supplyAsync(() -> "Task completed")
.thenRun(() -> System.out.println("Running next step...")); // Executed after the supplyAsync task

CompletableFuture.supplyAsync(() -> "Async task completed")
.thenRunAsync(() -> System.out.println("Running async next step...")); // Executed asynchronously after the supplyAsync task
```

11. exceptionally(Function<Throwable, ? extends T> fn)

- **Purpose**: Handles exceptions arising from the **CompletableFuture** computation, allowing for a fallback value to be provided or a new exception to be thrown.
- **Use Case**: Essential for robust error handling in asynchronous programming, allowing for recovery or logging of failures.

```
CompletableFuture<String> exceptionFuture = CompletableFuture.supplyAsync(() ->
        {
                if (true) throw new RuntimeException("Exception!");
                return "No Exception";
        })
.exceptionally(ex -> "Exception Handled: " + ex.getMessage());

exceptionFuture.thenAccept(System.out::println); // Prints: Exception Handled: java.lang.RuntimeException: Exception!
```

12. handle and handleAsync

- **Purpose**: Applies a function to the result or exception of the **CompletableFuture**. The synchronous variant is **handle**, and the asynchronous variant is **handleAsync**.
- **Use Case**: Useful when you need to process the result of a computation regardless of its success or failure, such as optionally transforming the result or providing a default in case of an exception.

```
CompletableFuture<String> handleFuture = CompletableFuture.supplyAsync(() -> {
                if (true) throw new RuntimeException("Handle Exception!");
                return "Handled";
        })
        .handle((result, ex) -> ex != null ? "Recovered: " + ex.getMessage() : result);
handleFuture.thenAccept(System.out::println); // Prints: Recovered: java.lang.RuntimeException: Handle Exception!

CompletableFuture<String> handleAsyncFuture = CompletableFuture.supplyAsync(() -> {
                if (true) throw new RuntimeException("Async Handle Exception!");
                return "Async Handled";
        })
        .handleAsync((result, ex) -> ex != null ? "Async Recovered: " + ex.getMessage() : result);
handleAsyncFuture.thenAccept(System.out::println); // Prints: Async Recovered: java.lang.RuntimeException: Async Handle
Exception!
```

Summary

- **Synchronous vs. Asynchronous**: For each operation(**thenApply**, **thenAccept**, **thenRun**, **thenCombine**) there's an asynchronous variant (**thenApplyAsync**, **thenAcceptAsync**, **thenRunAsync**, **thenCombineAsync**) that can execute its task in a separate thread, making it suitable for longer-running operations.
- **Chaining Computations**: These methods enable chaining multiple stages of computation, transforming results, and combining the outcomes of independent computations in a fluent and readable manner.
- **Error Handling**: Methods like **exceptionally** and **handle** provide mechanisms for dealing with errors that may occur during the asynchronous computations, ensuring resilience and robustness in asynchronous logic.

Counter-intuitive Behaviours and How to Address Them

The **CompletableFuture API** in Java is a powerful mechanism for managing asynchronous operations. However, its flexibility can sometimes lead to counterintuitive behaviours, subtle bugs, and performance issues. Understanding these aspects is crucial for developers to effectively use and debug **CompletableFuture**. Let's dive into each point.

Misuse of CompletableFuture Leading to Subtle Bugs and Performance Issues

- **Blocking Calls Inside CompletableFuture**: Using **get()** or **join()** within a **CompletableFuture**'s chain can block the asynchronous execution, negating the benefits of non-blocking code.

Solution: Replace blocking calls with non-blocking constructs like **thenCompose** for chaining futures or **thenAccept** for handling results.

- **Ignoring Returned Futures**: Not handling the **CompletableFuture** returned by methods like **thenApplyAsync** can lead to unobserved exceptions and behavior that does not execute as expected.

Solution: Always chain subsequent operations or attach error handling (e.g. **exceptionally** or **handle**) to every CompletableFuture.

Debugging Challenges in Asynchronous Code

- **Stack Traces Lack Context**: Exceptions in asynchronous code can have stack traces that don't easily lead back to the point where the async operation was initiated.

Strategies:

  - Use **handle** or **exceptionally** to catch exceptions within the future chain and add logging or breakpoints.
  - Consider wrapping asynchronous operations in higher-level methods that catch and log exceptions, providing more context.

Strategies to Identify and Fix Common Issues

- **Consistent Error Handling**: Attach an **exceptionally** or **handle** stage to each **CompletableFuture** to manage exceptions explicitly.

- **Avoid Common Pitfalls**: For example - executing long-running or blocking operations in **supplyAsync** without specifying a custom executor. This can lead to saturation of the **common fork-join pool**.

Solution: Use a custom executor for CPU-bound tasks to prevent interference with the global common fork-join pool.

- **Debugging Asynchronous Chains**: Break down complex chains of **CompletableFuture** operations into smaller parts. Test each part separately to isolate issues.

Tools and Techniques for Debugging CompletableFuture Chains

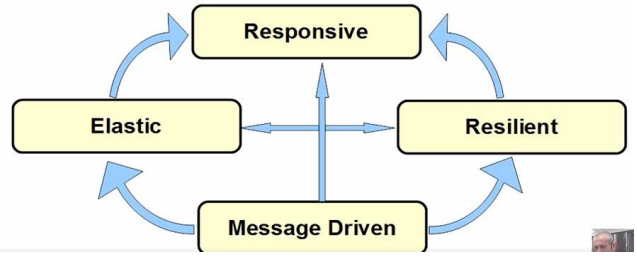
- **Logging**: Insert logging statements within **completion** stages (e.g., after **thenApply**, **thenAccept**) to trace execution flow and data transformation.
- **Visual Debugging Tools**: Some IDEs and tools offer visual representations of **CompletableFuture** chains, which can help in understanding the flow and identifying where the execution might be hanging or failing.
- **Custom Executors for Monitoring**: Use custom executors wrapped with logging or monitoring to track task execution and thread usage. This is particularly useful for identifying tasks that run longer than expected.
- **Async Profiling**: Tools like async-profiler can help identify hotspots and thread activity specific to asynchronous operations.

5. Reactive programming

I. What is Reactive Programming

Reactive Programming

- Is a model of coding where **communication happens** through a **non-blocking stream of data**
- Makes code "reactive", **reacting** to **change** and **not being blocked**, such as performing operations that read/wait for responses from a database or file
- Can react to **events** as data **becomes available**
- Using it we can use **effective resources utilization** (CPU cores) for computing
- It's principles are based on the **Reactive Manifesto - https://reactivemanifesto.org/**

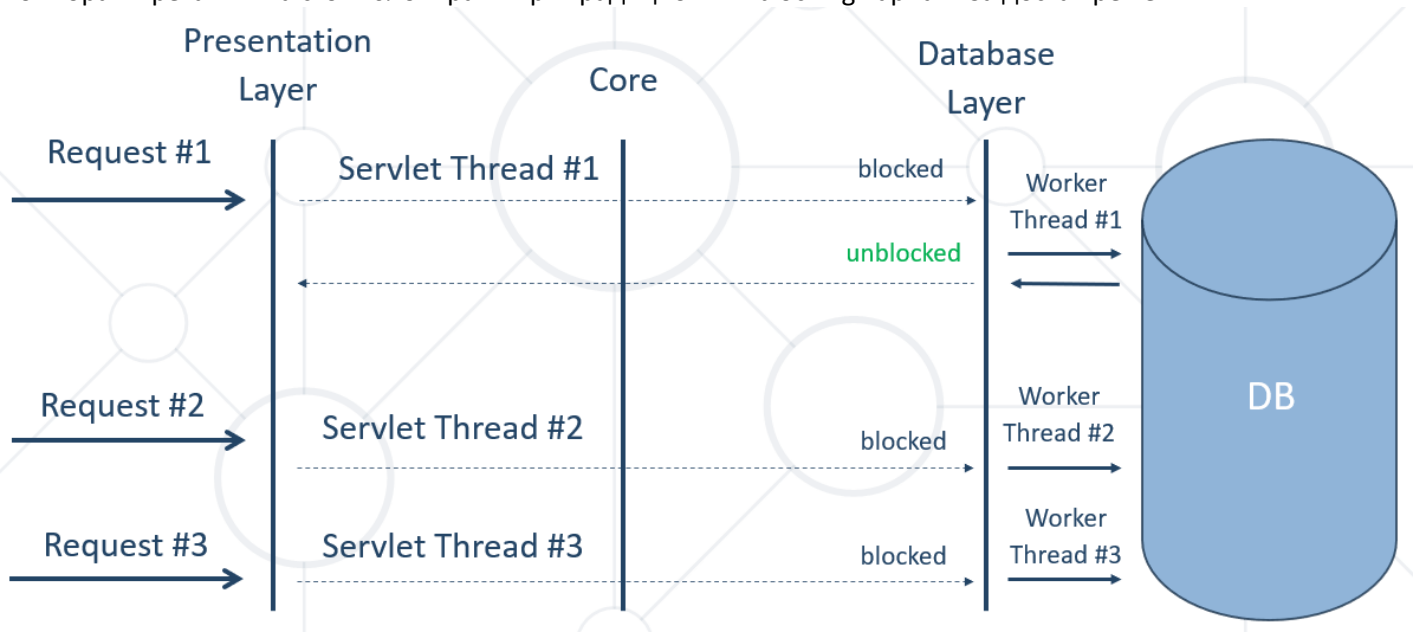


- It is build around **publisher-subscriber pattern** (observer pattern)
- In non-blocking code, the **Back pressure** becomes important to **control the rate of events** so that a fast producer does **not overwhelm** its destination

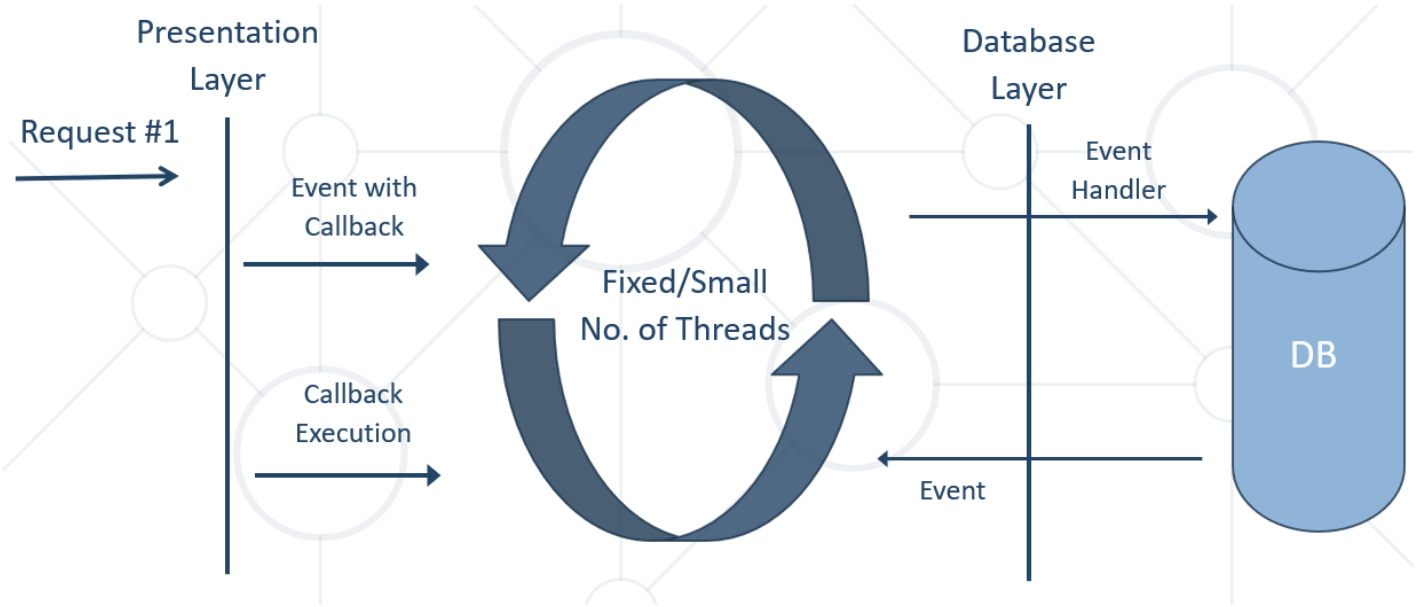
Една щайга ябълка в устата – ми не. Вземам една ябълка, като я изям подавам сигнал, че искам следваща ябълка.
Backpressure е количеството информация, която нашия **consumer** желае да получи.

Traditional MVC is Blocking vs Non-blocking

Първа нишка при традиционния вариант се отблокира навреме/веднага.
Но втора и трета нишка стоят блокирани при традиционния blocking вариант за доста време.



Изпращаме заявка към базата данни, след като нишката изпрати заявката, то спира да служи тук, и прави процесорно нещо друго. След като данните са взети от базата, то тази съща нишка или друга нишка отново се заема със заявката – този път да върне отговора от базата данни.



Reactive Streams and Reactor

- **Reactive Streams**: is a **specification** that defines how an API that implements and follows the Reactive Programming paradigm should work - http://www.reactive-streams.org/
- **Reactor**: is a Java implementation of the **Reactive Streams specification** (Reactor is just the Java implementation)
- **Reactor** project allows building **high-performance (low latency high throughput) non-blocking asynchronous** applications on JVM
- **Reactor** has powerful API for declaring **data transformations** and **functional composition**
- Make use of the concept of **Mechanical Sympathy** built on top of Disruptor / RingBuffer
- **Spring WebFlux** is the "reaction"/implementation of Spring for this paradigm to use on web applications

Projects

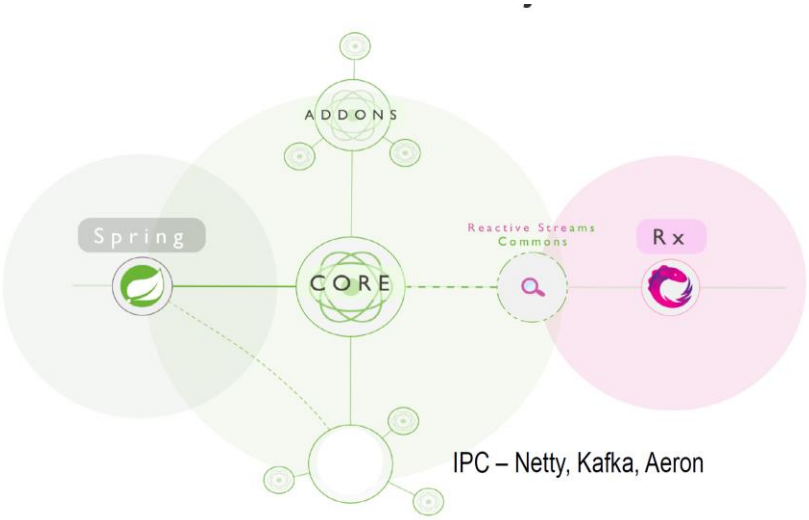
https://reactivex.io/
Rx = Observables + Flow transformations + Schedulers

Go: RxGo , Kotlin: RxKotlin, **Java: RxJava**, JavaScript: RxJS, Python: RxPY, C#: Rx.NET, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby

Spring WebFlux

Kafka

Netty



Hot and Cold Event Streams

PULL based (Cold Event Streams) Cold streams - (e.g. RxJava Observable / Flowable or Reactor Flow / Mono) are streams that run their sequence when and if they are subscribed to. They present the sequence from the start to each subscriber.

PUSH based (Hot Event Streams) - emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not. When subscription is made observer receives current events as they happen.

Hot Stream Example

```
EmitterProcessor<String> emitter =EmitterProcessor.create();
FluxSink<String> sink = emitter.sink();
emitter.publishOn(Schedulers.single())
       .map(String::toUpperCase)
       .filter(s -> s.startsWith("HELLO"))
       .delayElements(Duration.of(1000, MILLIS))
       .subscribe(System.out::println);

sink.next("Hello World!"); // emit -non blocking
sink.next("Goodbye World!");
sink.next("Hello Trayan!");
Thread.sleep(3000);
```

Reactive Streams VS Java 8 Streams

- The core difference is that **Reactive is a hot push model**, whereas the **Java 8 Streams** are a **cold pull model**
  - In reactive events are pushed to the subscribers as they come in
- **Java 8 Streams -** pulling all the data and returning a result
- With **Reactive** we could have an **infinite stream** coming in from an external resource, with **multiple subscribers** attached

Reactive Stream APIs

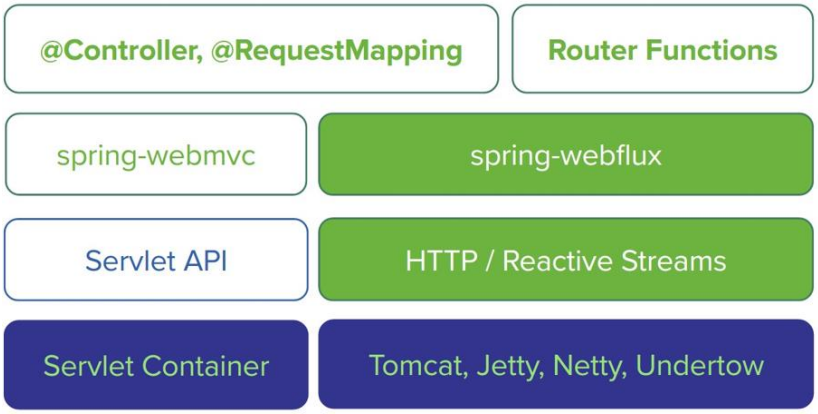
- As per **Java 9 and reactive specification,** below are APIs we need to use for reactive implementation
  - **Publisher** - Emits a sequence of events to subscribers according to the demand received from its subscribers
  - **Subscriber(Consumer)** - Receives&Processes events emitted by a Publisher
  - **Subscription** - Defines a one-to-one relationship between a Publisher and a Subscriber

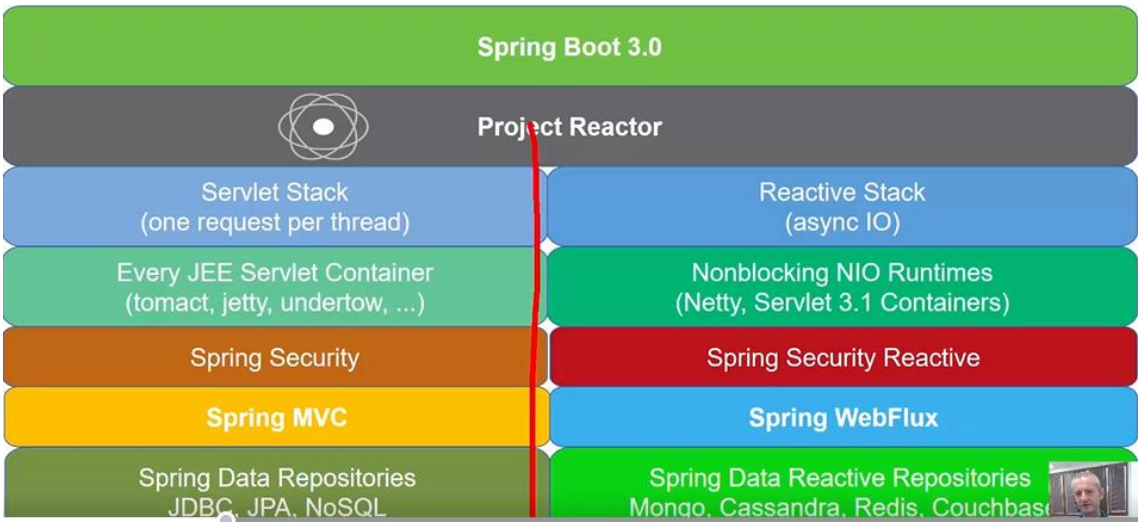
- **Processors** - Represents a processing stage consisting of both a Publisher and a Subscriber (Consumer) and obeys the contracts of both = **a pipe from a Publisher and a Subscriber**

Spring 5 Reactive Building Blocks

- WebFlux
- Spring Data reactive library
- Reactive IO
- Nonblocking Servlet container
- Spring security reactive API

- [Source - https://docs.spring.io/](https://docs.spring.io/)





II. Spring WebFlux

Spring WebFlux

- Spring WebFlux е **алтернатива** на стандартния Spring MVC (макар че могат да се използват и заедно)
- **Web framework** that brings support for the **reactive** programming model
- Implemented **using** the **Project Reactor,** and its publisher implementations — **Flux** and **Mono**, the library chosen by Spring
- WebFlux is **not a replacement for Spring MVC** they can actually complement each other, working together on the same solution

Spring WebFlux Dependencies

- Adding Spring WebFlux Dependency in pom.xml
- The Reactive Web dependency includes Spring WebFlux dependency

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Mono and Flux

Mono – може да emit-ва 0 или 1 парче данни
Flux – може да emit-ва наведнъж повече от едно парче данни

- In WebFlux, the data returned from any operation is packed into a **reactive stream**
- There are **two types** that embody this approach and are the building blocks in WebFlux:
- **Mono** - is a stream which returns zero items or a single item (0..1)
- **Flux -** is a stream which returns zero or more items (0..n)

Working with Mono and Flux

- **Mono/Flux.just()**
  - The easiest way to emit an element is using the **just** method

```
Mono.just(1).subscribe(System.out::println);
Flux.just(1,2,3).subscribe(System.out::println);
```

- Mono/Flux can have more than one **subscribers**

```
Flux<Integer> flux = Flux.just(1,2,3);
flux.subscribe(s->System.out.println("Subscr One-"+ s));
flux.subscribe(s->System.out.println("Subscr Two-"+ s));
```

- Example of **two subscribers** with delay

```
Flux<Integer> flux = Flux.just(1, 2, 3);
flux
        .map(n -> ++n)
        .delayElements(Duration.ofMillis(500))
        .subscribe(System.out::println);     //2, 3, 4

flux
        .delayElements(Duration.ofMillis(1000))
        .subscribe(s -> System.out.println("Subscriber One-" + s));  //1, 2, 3
```

- The **subscribe** method could accept other parameters as well to handle the **error** and **completion** calls

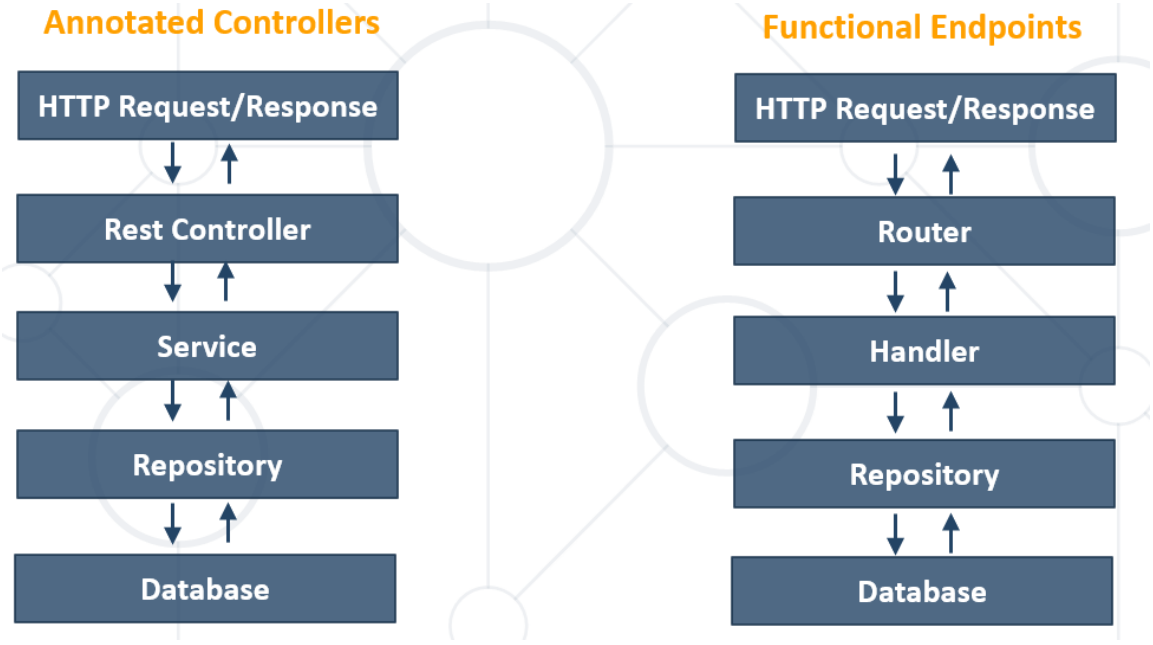
```
Flux.just(1, 2, 3)
        .subscribe(
                i -> System.out.println("Received :: " + i),
                err -> System.out.println("Error :: " + err),
                () -> System.out.println("Successfully completed") //completion – моментът когато
няма повече инфо в нашия flux
        );
```

III. Programming Models

Programming Models

- We can implement it in two ways:
  - **Annotated Controllers**
    - using Spring MVC annotations with minimum modifications
    - The old applications can also be converted to reactive nature and can use its features and benefits
  - **Functional Endpoints**
    - functional lambda style of programming

Annotated Controllers VS Func Endpoints



Annotated Controllers

- **Similar** to how we use controllers in **classic Spring MVC**
- To mark a class as a controller, we use the **@RestController** annotation on a class level.
- Having Spring WebFlux and the Reactor Core dependencies, in the class path, will let Spring know that the **@RestController** is in fact a reactive component and add support for **Mono** and **Flux**

Annotated Controllers Example

- Annotated Controller Example

```
@RestController
@RequestMapping("/students")
public class StudentsControllers {
    //Inject studentsService in constructor

    @GetMapping("/all")
    public Flux<Students> findAll(){
        return studentsService.findAll();
    }
```

Functional Endpoints

- Spring WebFlux includes **WebFlux.fn**, a lightweight functional programming model in which functions are used to **route** and **handle requests** and contracts are designed for immutability
- An HTTP request is handled with a **HandlerFunction** that takes ServerRequest and returns a delayed ServerResponse - **Mono<ServerResponse>**
- Incoming requests are routed to a handler function with a **RouterFunction** - takes ServerRequest and returns a delayed HandlerFunction - **Mono<HandlerFunction>**

Functional Endpoints Example – Handler

- Example of Student Handler

```
public class StudentHandler {
    //...
    public Mono<ServerResponse> getStudent(ServerRequest request) {
        int studentId = Integer.valueOf(request.pathVariable("id"));
        return repository.getStudent(studentId)
                .flatMap(student -> ok().contentType(APPLICATION_JSON).bodyValue(student))
                .switchIfEmpty(ServerResponse.notFound().build());
    } //... }
```

Functional Endpoints Example – Router

- Example of Router Function

```
//... Inject PersonHandler handler in constructor

        RouterFunction<ServerResponse> router = router()
                .GET("/student/{id}", accept(APPLICATION_JSON), handler::getStudent)
                .GET("/student ", accept(APPLICATION_JSON), handler::listStudents)
                .POST("/student ", handler::createStudent)
                .build();
```

Spring WebFlux Configuration – за по-стари Spring версии

- The **@Configuration** and **@EnableWebFlux** annotations mark a class as a configuration class and Spring's bean management will register it
- To use or extend the existing WebFlux configuration API, you can implement **WebFluxConfigurer**

```
@Configuration
@EnableWebFlux
public class Configuration implements WebFluxConfigurer {...}
```

IV. Demos

Demo – the 4 interfaces of reactive streams

```
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.Flow.Subscription;
import java.util.concurrent.Flow.Subscriber;

//I.e. this is the consumer
public class SimpleSubscriber<T> implements Subscriber<T> {
    private List<T> consumedElements = new LinkedList<>();
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
```

```
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        consumedElements.add(item);

        //Applying backpressure - искам вече следващото
        subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("I'm complete!!!");
        consumedElements.forEach(System.out::println);
        System.out.println("--------------");
    }

    public int getConsumedElementsCount(){
        return consumedElements.size();
    }

    public List<T> getConsumedElements() {
        return consumedElements;
    }
}


-----------------------------------------
import java.util.concurrent.Flow.Processor;
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;
import java.util.concurrent.SubmissionPublisher;
import java.util.function.Function;

public class TransformationProcessor<T, R> implements Processor<T, R> {
    private final Function<T, R> transformation;
    private final SubmissionPublisher<R> submissionPublisher;
    private Subscription subscription;

    //нещо като pipe - ще получаваме от нашия publisher едни елементи, и ще ги превръщаме в други
елементи
    public TransformationProcessor(Function<T, R> transformation) {
        this.transformation = transformation;
        this.submissionPublisher = new SubmissionPublisher<R>();
    }

    //publisher
    @Override
    public void subscribe(Subscriber<? super R> subscriber) {
        submissionPublisher.subscribe(subscriber);
    }

    //publisher
    @Override
```

```
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); //the backpressure
    }

    //Subscriber/producer
    @Override
    public void onNext(T item) {
        R transformed = transformation.apply(item);
        submissionPublisher.submit(transformed); //събмитваме надолу по веригата
        this.subscription.request(1); //the backpressure
    }

    //Subscriber/producer
    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    //Subscriber/producer
    @Override
    public void onComplete() {
        System.out.println("Transformation is complete, closing down.");
    }
}

-----------------------------------------
import java.util.List;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;
import java.util.function.Function;

import static org.awaitility.Awaitility.await;
import static org.junit.jupiter.api.Assertions.*;

class ReactiveTests {

    @Test
    void testAllItemsConsumed() {
        //The producer!!!
        SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

        SimpleSubscriber<String> mySubscriber = new SimpleSubscriber<>();

        //set new consumer for producer publisher
        publisher.subscribe(mySubscriber);

        assertEquals(1, publisher.getNumberOfSubscribers()); //Returns the number of current
subscribers.

        List<String> names = List.of("Anna", "John", "Pesho");
        names.forEach(item -> publisher.submit(item));  //sbmit = publishes the given items to
each current subscriber
        publisher.close();

        //Make it synchronous with the awaitility library
        await()
                .atMost(1, TimeUnit.SECONDS)
                .until(
                        () -> mySubscriber.getConsumedElementsCount() == names.size()
                );
```

```
        assertEquals(3, mySubscriber.getConsumedElementsCount());
    }

    @Test
    public void testTransformation() {
        //arrange = given part
        Function<String, String> transfFunc = String::toUpperCase;
        TransformationProcessor<String, String> transformationPipe = new
TransformationProcessor<>(transfFunc);

        SubmissionPublisher<String> startPublisher = new SubmissionPublisher<>();
        SimpleSubscriber<String> finishSubscriber = new SimpleSubscriber<>();

        List<String> items = List.of("SenKo", "Pesho", "lilly"); //от едната страна на pipe-а
        List<String> expectedItems = List.of("SENKO", "PESHO", "LILLY"); //какво излизза от
другата страна на pipe-а


        //act = when
        startPublisher.subscribe(transformationPipe); //началната точка на pipe-а
        transformationPipe.subscribe(finishSubscriber); //крайната точка на Pipe-а

        items.forEach(item -> startPublisher.submit(item));
        startPublisher.close();

        //Make it synchronous with the awaitility library
        await()
                .atMost(1, TimeUnit.SECONDS)
                .until(
                        () -> expectedItems.equals(finishSubscriber.getConsumedElements())
                );

        //assert
        assertEquals(expectedItems, finishSubscriber.getConsumedElements());
    }
}
```

Demo – Mono and Flux

Spring Reactive Web

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-webflux'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
   testImplementation 'io.projectreactor:reactor-test'
}
```

```
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.List;
import java.util.Optional;
import java.util.Random;

public class FluxMonoExplained {
    private User user1 = new User().setFirstName("A").setLastName("B");
    private User user2 = new User().setFirstName("A1").setLastName("B1");

    public User getUser() {
```

```
        if (isAuthenticated()) {
            return user1;
        } else {
            return null;
        }
    }

    public Mono<User> getUserReactive() {
        if (isAuthenticated()) {
            return Mono.just(user1);
        } else {
            return Mono.empty();
        }
    }

    public Optional<User> getUserOpt() {
        if (isAuthenticated()) {
            return Optional.of(user1);
        } else {
            return Optional.empty();
        }
    }

    public List<User> getAllUsers() {
        return List.of(user1, user2);
    }

    public Flux<User> getAllUsersReactive() {
        return Flux.just(user1, user2);
    }


    private boolean isAuthenticated() {
        return new Random().nextBoolean();
    }

}

class User {
    private String firstName, lastName;

    public String getFirstName() {
        return firstName;
    }

    public User setFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    public String getLastName() {
        return lastName;
    }

    public User setLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }
}

------------------------
```

```java
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.function.Consumer;
import java.util.stream.Collectors;

class WebfluxGeneralTest {
    @Test
    public void fluxToStream() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        springProjectsFlux
                .toStream()
                .map(String::toUpperCase)
                .forEach(System.out::println);
    }

    @Test
    public void subscribeToFlux() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        springProjectsFlux.subscribe(
                System.out::println
        );
    }

    @Test
    public void doOnEach() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        springProjectsFlux.doOnEach(
                        signalConsumer -> {
                            if (signalConsumer.isOnNext()) {
                                System.out.println(signalConsumer.get());
                            }
                        }
                )
                .subscribe();
    }

    @Test
    public void mapAndFilter() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        springProjectsFlux
                .map(String::toUpperCase)
                .filter(s -> s.contains("REST"))
                .subscribe(System.out::println);
    }

    @Test
    public void collect() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        springProjectsFlux
                .map(String::length)
                .collect(Collectors.summarizingInt(Integer::intValue))//статистика от дължината на
всички заглавия
                .subscribe(System.out::println);
```

```
//        IntSummaryStatistics{count=7, sum=85, min=10, average=12.142857, max=16}
    }

    @Test
    public void subscribe() {
        Flux<String> springProjectsFlux = Flux.just(getSpringProjects());

        Consumer<String> onNextConsumer = System.out::println;
        Consumer<Throwable> onErrorConsumer = Throwable::printStackTrace;
        Runnable onDone = () -> System.out.println("We are done!");

        springProjectsFlux.subscribe(
                onNextConsumer,
                onErrorConsumer,
                onDone
        );
    }

    @Test
    public void testOnError() {
        Flux<Integer> numbers = Flux.just("1", "two", "3")
                .map(Integer::parseInt);

        Consumer<Integer> onNextConsumer = System.out::println;
        Consumer<Throwable> onErrorConsumer = Throwable::printStackTrace;
        Runnable onDone = () -> System.out.println("We are done!");

        numbers.subscribe(
                onNextConsumer,
                onErrorConsumer,   //throws error this time
                onDone
        );
    }

    @Test
    public void mono(){
        Mono.just("TEST").map(String::toUpperCase).subscribe(System.out::println);
    }

    private String[] getSpringProjects() {
        return new String[]{
                "Spring REST",
                "Spring DATA REST",
                "Spring Batch",  //Java Batching
                "Spring MVC",
                "Spring Webflux",
                "Spring JMS",   //Java Messaging Service
                "Spring Kafka"};
    }
}

-------------


import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;

public class AsyncFlux {

    //синхронно
```

```java
    @Test
    public void syncFlux() {
        MyInteger sum = new MyInteger(0);

        Flux.just(1, 2, 3, 4)
                .reduce(MyInteger::sum)
                .subscribe(sum::set);

        System.out.println(sum);

//        Output
//        Summing 1 and 2
//        Summing 3 and 3
//        Summing 6 and 4
//        MyInteger{initialValue=10}
    }


    //асинхронно
    @Test
    public void asyncFlux() {
        MyInteger sum = new MyInteger(0);

        Flux.just(1, 2, 3, 4)
                .subscribeOn(Schedulers.boundedElastic())
                .reduce(MyInteger::sum)
                .subscribe(sum::set);

        System.out.println(sum);

        //Output
//        MyInteger{initialValue=0}
//        Summing 1 and 2
//        Summing 3 and 3
//        Summing 6 and 4
    }

}


class MyInteger {
    private Integer initialValue;

    public MyInteger(Integer initialValue) {
        this.initialValue = initialValue;
    }

    public static int sum(int a, int b) {
        System.out.println("Summing " + a + " and " + b);
        return a + b;
    }

    public void set(Integer aNumber) {
        this.initialValue = aNumber;
    }

    @Override
    public String toString() {
        return "MyInteger{" +
                "initialValue=" + initialValue +
                '}';
```

```
    }
}
```

Demo – webflux service

//При реактивното програмиране в Spring, за endpoint-и използваме бийнове вместо @Controller, @RequestMapping, и т.н.

See in my repo in GitHub

```
2022-10-19 23:37:43.891  INFO 14744 --- [           main] b.s.w.WebfluxserviceApplication
: No active profile set, falling back to 1 default profile: "default"
2022-10-19 23:37:45.571  INFO 14744 --- [           main] o.s.b.web.embedded.netty.NettyWebServer
: Netty started on port 8080
2022-10-19 23:37:45.584  INFO 14744 --- [           main] b.s.w.WebfluxserviceApplication
: Started WebfluxserviceApplication in 2.201 seconds (JVM running for
```

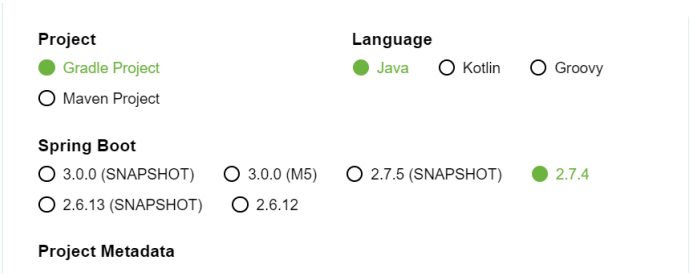
```
public class ExchangeRate {
    private String fromCurrency;
    private String toCurrency;
    private Instant time;
    private BigDecimal rate;

    public ExchangeRate(String fromCurrency, String toCurrency, Instant time, BigDecimal rate) {
        this.fromCurrency = fromCurrency;
        this.toCurrency = toCurrency;
        this.time = time;
        this.rate = rate;
    }
}
```

```
------------------------------
@Service
public class ExchangeRateService {

    private List<ExchangeRate> exchangeRates = new ArrayList<>();

    public ExchangeRateService() {
       exchangeRates.add(new ExchangeRate("EUR", "BGN", Instant.now(), BigDecimal.valueOf(1.96)));
       exchangeRates.add(new ExchangeRate("USD", "BGN", Instant.now(), BigDecimal.valueOf(1.74)));
       exchangeRates.add(new ExchangeRate("GBN", "BGN", Instant.now(), BigDecimal.valueOf(2.16)));
       exchangeRates.add(new ExchangeRate("RSD", "BGN", Instant.now(), BigDecimal.valueOf(0.017)));
    }

    public Flux<ExchangeRate> getExchangeRateStream(int durationInterval) {
        Flux<ExchangeRate> ret = Flux.generate(() -> 0,
                (index, sink) -> {
                    ExchangeRate updateRate = randomize(exchangeRates.get(index));
                    sink.next(updateRate);
                    System.out.println("Generated a new exchange rate...");

                    return (++index) % exchangeRates.size();
                });

        if (durationInterval > 0) {
            return ret.delayElements(Duration.ofSeconds(durationInterval));
        } else {
            return ret;
```

```
        }
    }

-----------------

@Component
public class ExchangeRateHandler {
    private final ExchangeRateService exchangeRateService;

    public ExchangeRateHandler(ExchangeRateService exchangeRateService) {
        this.exchangeRateService = exchangeRateService;
    }

    public Mono<ServerResponse> getExchangeRates(ServerRequest request) {
        int size = Integer.parseInt(request.queryParam("size").orElse("10"));

        //Вземане на моно от отложен server response
        return ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(this.exchangeRateService.getExchangeRateStream(0).take(size),
                        ExchangeRate.class);
    }

    public Mono<ServerResponse> streamExchangeRates(ServerRequest request) {
        int size = Integer.parseInt(request.queryParam("size").orElse("10"));

        //Вземане на моно от отложен server response
        return ok()
                .contentType(MediaType.APPLICATION_STREAM_JSON)
                .body(this.exchangeRateService.getExchangeRateStream(1).take(size),
                        ExchangeRate.class);
    }
}

----------------------
//При реактивното програмиране в Spring, за endpoint-и използваме бийнове вместо @Controller,
@RequestMapping, и т.н.
@Configuration
public class ExchangeRateRouter {

    @Bean
    public RouterFunction<ServerResponse> route(ExchangeRateHandler exchangeRateHandler) {
        return RouterFunctions
                .route(GET("/rates").and(accept(MediaType.APPLICATION_JSON)),
exchangeRateHandler::getExchangeRates)
                .andRoute(GET("/rates").and(accept(MediaType.APPLICATION_STREAM_JSON)), sr ->
exchangeRateHandler.streamExchangeRates(sr));
    }
}
```

Demo – webflux client with Reactive MongoDb driver

```
//При реактивното програмиране в Spring, за endpoint-и използваме бийнове вместо @Controller,
@RequestMapping, и т.н.
```

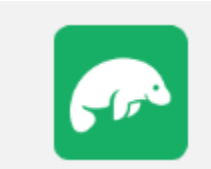


docker-compose.yaml

```yaml
version: 3.1
services:
  mongo:
    image: mongo
    restart: always
    ports:
      - 27017:27017
```

в терминала командата docker-compose up

Mongo graphic user interface GUI
Robo 3T is now Studio 3T Free



```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties
@ConfigurationProperties("softuni.webflux.client")
public class ClientConfig {
    private String schema;
    private String host;
    private String port;
}
```

---

application.yml

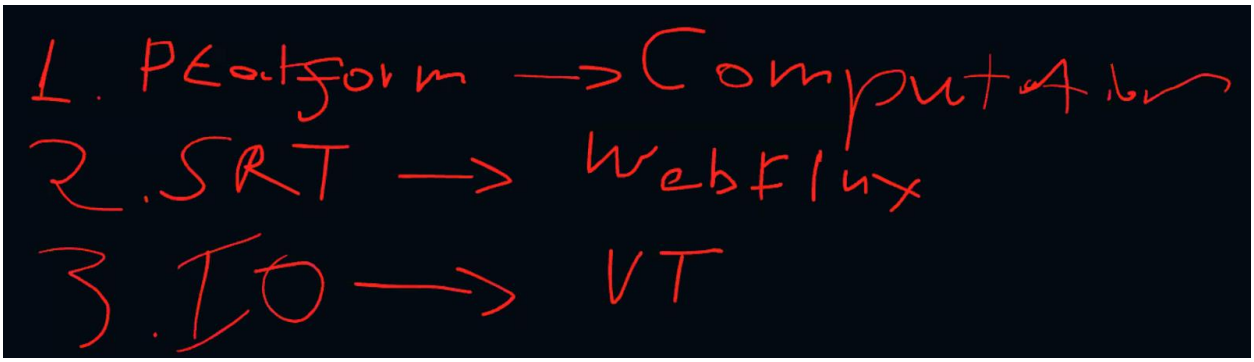
```yaml
softuni:
  webflux:
    client:
      schema: http
      host: localhost
      port: 8000
```

```java
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.math.BigDecimal;
import java.time.Instant;

@Document //като @Entity анотацията за релационните база данни, но за MongoDB
```

```
public class ExchangeRate {
    @Id
    private String id;

    private String fromCurrency;
    private String toCurrency;
    private Instant time;
    private BigDecimal rate;
}

---------------------
@Component
public class ExchangeRateClient {
    private final String SERVICE_URL;
    private final String API_PATH = "/rates";

    public ExchangeRateClient(ClientConfig clientConfig) {
        SERVICE_URL = clientConfig.getSchema() + "://" + clientConfig.getHost() + ":" +
clientConfig.getPort();
    }

    public Flux<ExchangeRate> getRateStream(){
        return WebClient.builder()
                .baseUrl(SERVICE_URL)
                .build()
                .get()
                .uri(API_PATH)
                .accept(MediaType.APPLICATION_STREAM_JSON)
                .retrieve()
                .bodyToFlux(ExchangeRate.class);
    }
}

----------------
@Repository
public interface ExchangeRateRepository extends ReactiveMongoRepository<ExchangeRate, String> {

}


----------------------------
@Service
public class ExchangeRateInit implements ApplicationListener<ContextRefreshedEvent> {

    private final ExchangeRateClient client;
    private final ExchangeRateRepository repository;

    public ExchangeRateInit(ExchangeRateClient client, ExchangeRateRepository repository) {
        this.client = client;
        this.repository = repository;
    }

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        client.getRateStream()
                .subscribe(exchangeRate -> {
                    Mono<ExchangeRate> exchangeRateMono = repository.save(exchangeRate);
                    exchangeRateMono.subscribe(er -> System.out.println("Saved " + er));
                });
    }
}
```

6. Threads comparison

1. Computations -> Platform threads
2. Super Real Time - > Webflux 100%, VirtualThreads 90-95%, CompletableFuture, Future, etc
3. InputOutput -> VirtualThreads (**project Loom and JDK 21**)



Spring Webflux is substituting Spring MVC. You'd better use only one of them in a microservice!

7. Messaging systems

Intro

- Messaging provides a mechanism for loosely-coupled integration of systems
- The central unit of processing in a message is a message which typically contains a **body** and a **header**
- Use cases include:
    - Log aggregation between systems
    - Event propagation between systems - някакви събития се fire-ват
    - Offloading log-running tasks to worker nodes - the result of the task then to be sent to a third systems for example
- Messaging solutions implement different protocols for transferring of messages such as **AMQP** (binary protocol), XMPP, MQTT and many more like XML, JSON, etc.
- The variety of protocols implies vendor lock-in when using a particular messaging solution (also called a messaging broker) - ако е специфичен протокола лошо. Т.е. протокола е добре да е такъв, че да може да се използва от различни message broker systems

- Message brokers
    - ActiveMQ - using JMS (Java Messaging System) Java EE
    - RabbitMQ
    - Qpid
    - TIBCO
    - WebSphere MQ
    - Msmq

- Messaging solutions provide means for:
    - Securing message transfer, authenticating and authorizing messaging endpoints
    - Routing messages between endpoints
    - Subscribing to the broker

- An **enterprise service bus (ESB)** is one layer of abstraction above a messaging solution that further provides:
    - Adapters for different messaging protocols
    - Translation of messages between the different types of protocols

I. RabbitMQ

Info

- An open source message broker written in Erlang
    - Заема малко ранм памет и процесор
    - при Erlang няма context switching като при JVM
    - reliability - дете ако не се изпълни, то родителят му го пуска за изпълнение наново
- **Implements the AMQP protocol** (Advanced Message Queueing Protocol)
- Has a pluggable architecture and provides extension for other protocols such as HTTP, STOMP and MQTT

- AMQP is a binary protocol that aims to standardize middleware communication
- The AMQP protocol derives its origins form the financial industry - processing of large volumes of financial data between different systems is a classic use case of messaging
- The AMQP protocol defines:
    - **Exchanges** - the message broker endpoints that receive messages
    - **Queues** - the message broker endpoints that store messages from exchanges and are used by subscribers for retrieval of messages. The Queue can also be persistent - messages can be saved.
    - **Bindings** - rules that bind exchanges and queues
- The AMQP protocol is programmable - which means that the above entities can be created/ modified/ deleted by applications

- The AMQP protocol defines multiple connection channels inside a single TCP connection in order to remove the overhead of opening a large number of TCP connections to the message broker

- Each message can be published with a **routing key**
- Each binding between an exchange and a queue has a **binding key**
- Routing of messages is determined based on matching between the **routing key** and the **binding key**

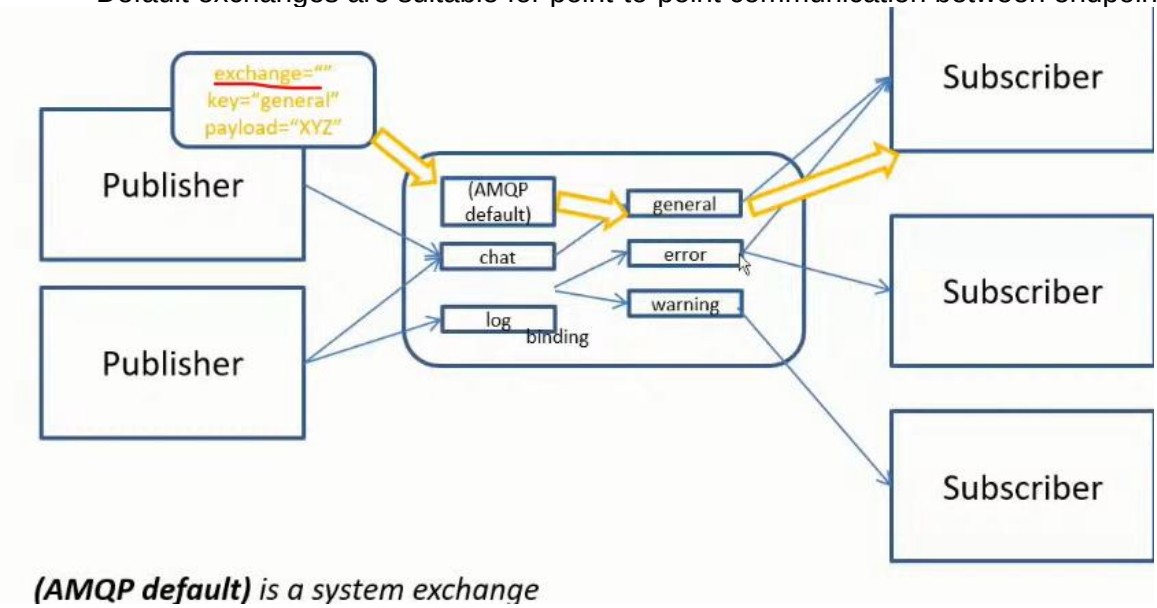
Messaging patterns with RabbitMQ

- Different types of messaging patterns are implemented by means of different types of exchanges
- RabbitMQ provides the following types of exchanges:
    - default - без име като търси съвпадение на **routing key** с **binding key**
    - direct - има име като търси съвпадение на **routing key** с **binding key**
    - fanout
    - topic
    - headers- на база мачинг по хедъри също

Default exchange

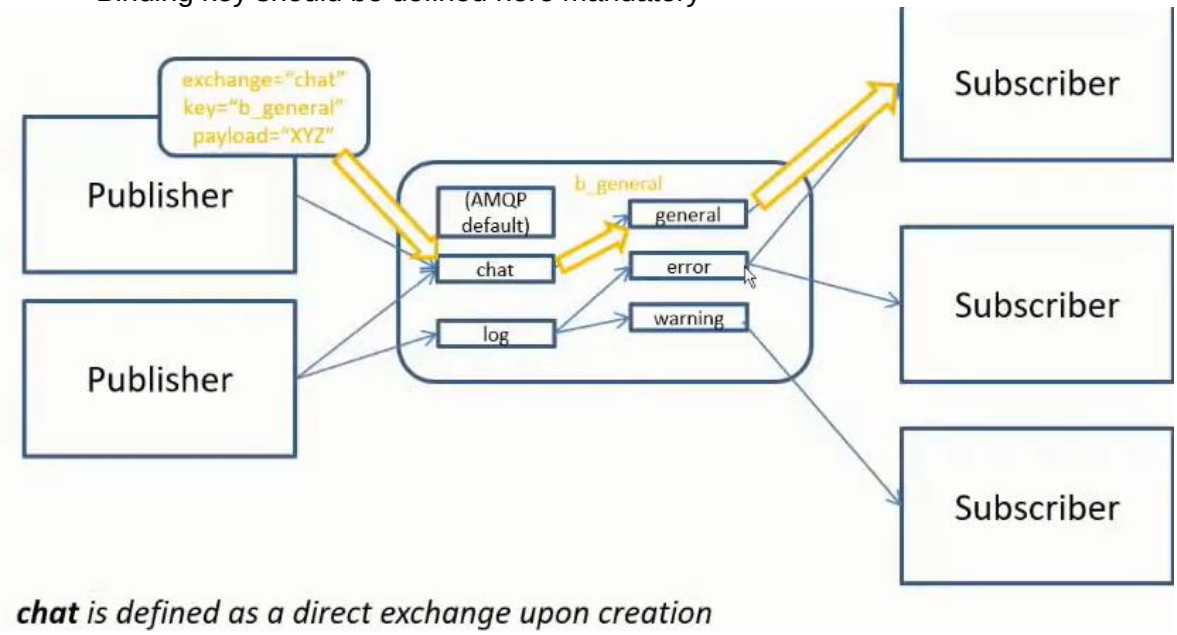
- A default exchange has **the empty string as a name** and routes messages to a queue if the routing key of the message matches the queue name (no binding needs to be declared between a default exchange and a queue)

- Default exchanges are suitable for point-to-point communication between endpoints



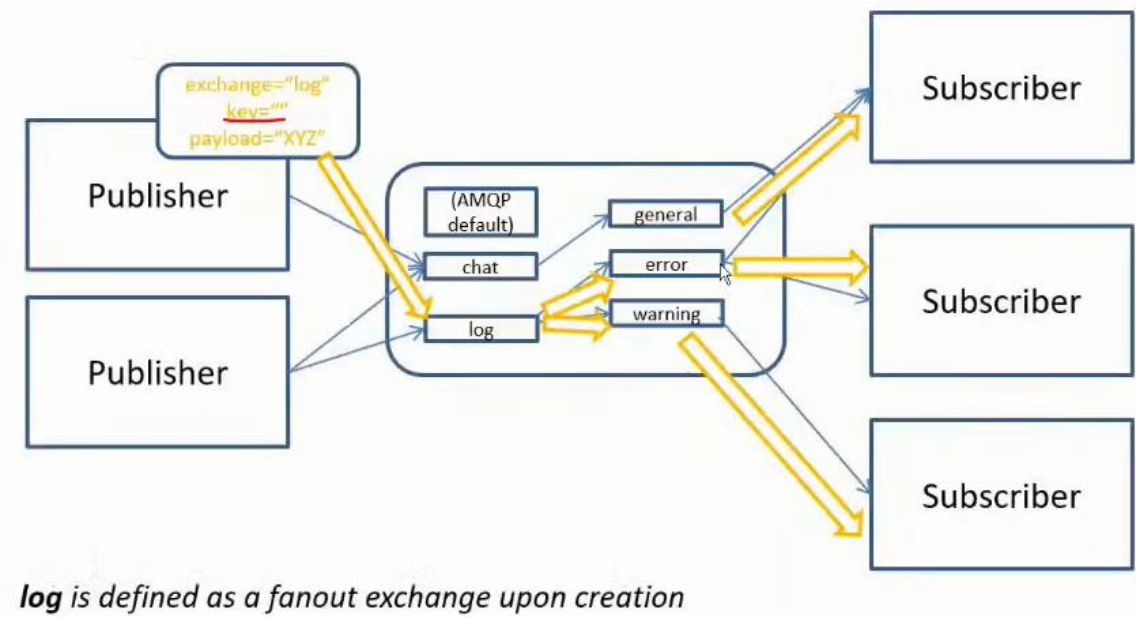
Direct exchange

- A direct exchange routes messages to a queue if the routing key of the message matches the binding key between the direct exchange and the queue
- Direct exchanges are suitable for point-to-point communication between endpoints
- Binding key should be defined here mandatory



Fanout exchange

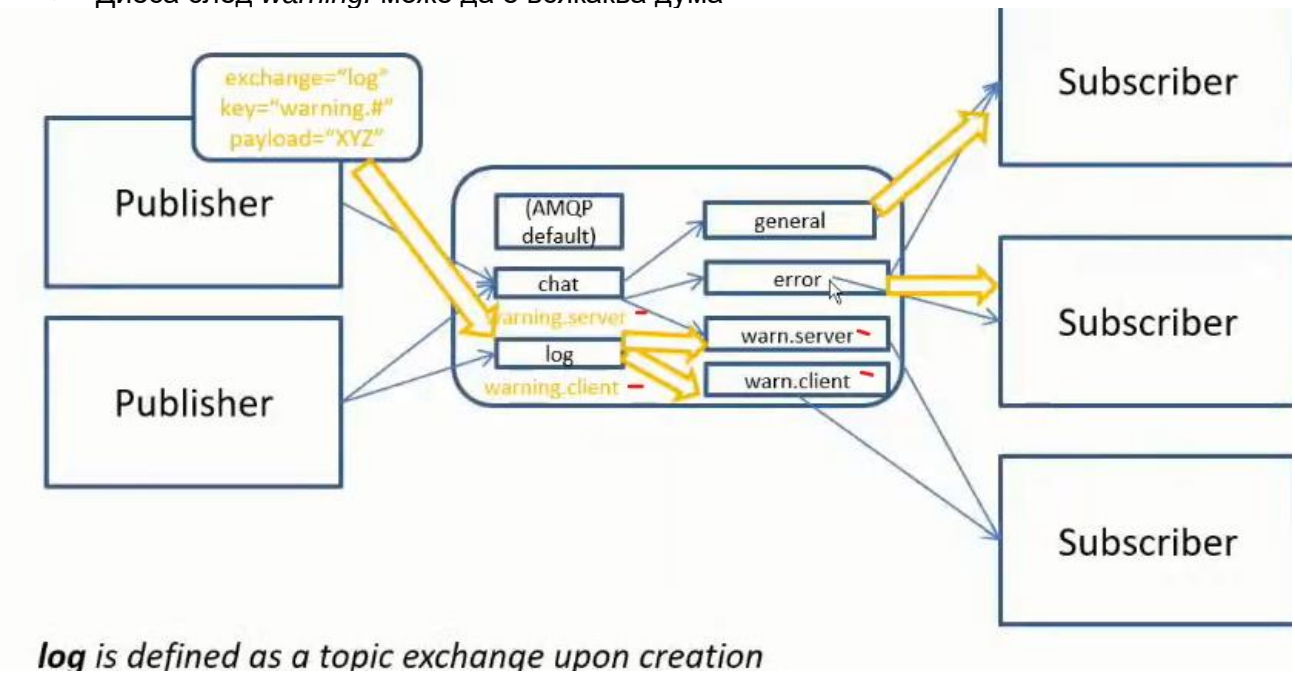
- A fanout exchange routes (broadcasts) messages to all queues that are bound to it (the binding key is not used)
- Fanout exchanges are suitable for publish-subscribe communication between endpoints



***log** is defined as a fanout exchange upon creation*

Topic exchange

- A topic exchange routes (multicasts) messages to all queues that have a binding key (can be a pattern) that matches the routing key of the message
- Topic exchanges are suitable for routing messages to different queues based on the type of message
- Диеса след *warning.* може да е всякаква дума



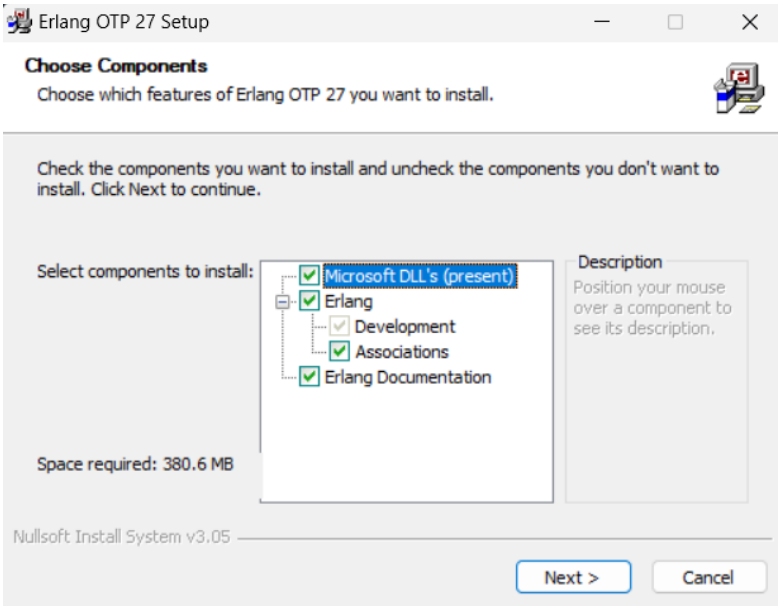
***log** is defined as a topic exchange upon creation*

Headers exchange

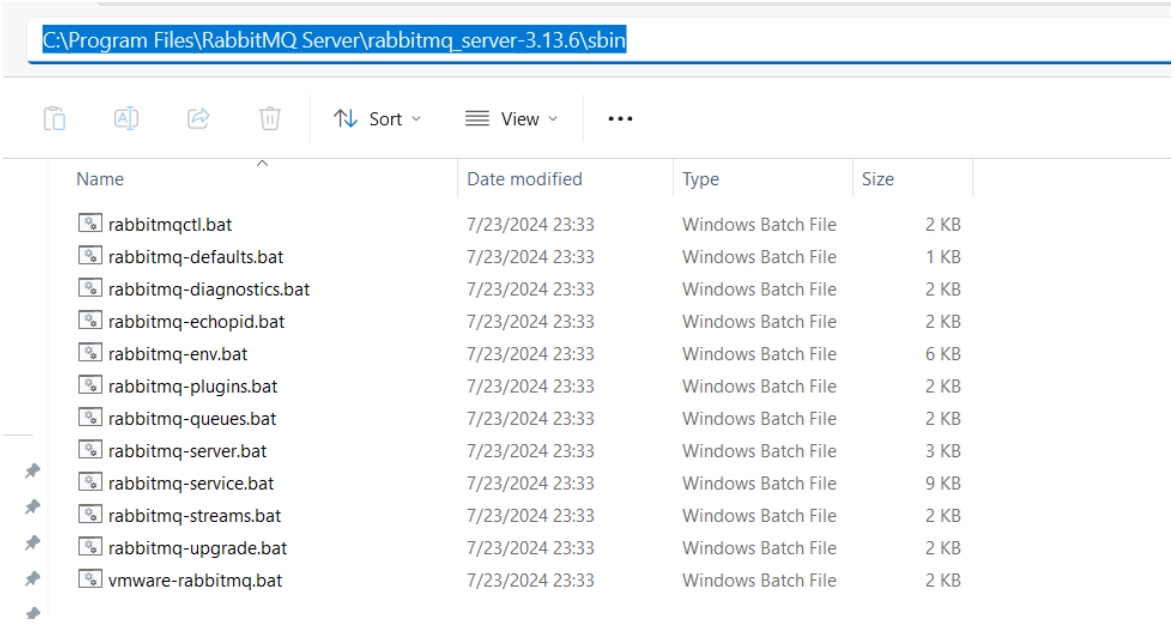
- A headers exchange routes messages based on a custom message header
- Header exchanges are suitable for routing messages to different queues based on more than one attribute

Installation of the RabbitMQ server

First install the Erlang - https://www.erlang.org/downloads



Then install the RabbitMQ server - https://www.rabbitmq.com/docs/download
rabbitmq-service.bat
rabbitmq-service.bat



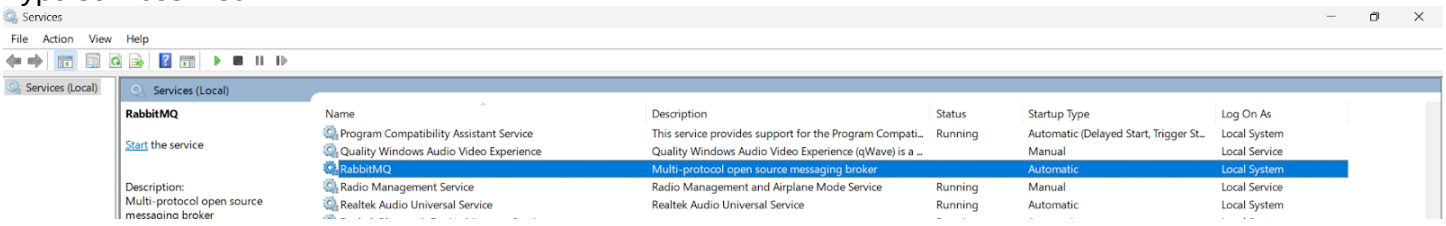
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>**rabbitmq-plugins.bat enable** rabbitmq_management
Enabling plugins on node rabbit@SVILKATA:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@SVILKATA...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.

През CommandPrompt като администратор:

```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>rabbitmq-plugins.bat list
Listing plugins with pattern ".*" ...
 Configured: E = explicitly enabled; e = implicitly enabled
 | Status: [failed to contact rabbit@SVILKATA - status not shown]
 |/
[  ] rabbitmq_amqp1_0                  3.13.6
[  ] rabbitmq_auth_backend_cache       3.13.6
[  ] rabbitmq_auth_backend_http        3.13.6
[  ] rabbitmq_auth_backend_ldap        3.13.6
[  ] rabbitmq_auth_backend_oauth2      3.13.6
[  ] rabbitmq_auth_mechanism_ssl       3.13.6
[  ] rabbitmq_consistent_hash_exchange 3.13.6
[  ] rabbitmq_event_exchange           3.13.6
[  ] rabbitmq_federation               3.13.6
[  ] rabbitmq_federation_management    3.13.6
[  ] rabbitmq_jms_topic_exchange       3.13.6
[*] rabbitmq_management                3.13.6
[*] rabbitmq_management_agent          3.13.6
[  ] rabbitmq_mqtt                     3.13.6
[  ] rabbitmq_peer_discovery_aws       3.13.6
[  ] rabbitmq_peer_discovery_common    3.13.6
[  ] rabbitmq_peer_discovery_consul    3.13.6
[  ] rabbitmq_peer_discovery_etcd      3.13.6
[  ] rabbitmq_peer_discovery_k8s       3.13.6
[  ] rabbitmq_prometheus               3.13.6
[  ] rabbitmq_random_exchange          3.13.6
[  ] rabbitmq_recent_history_exchange  3.13.6
[  ] rabbitmq_sharding                 3.13.6
[  ] rabbitmq_shovel                   3.13.6
[  ] rabbitmq_shovel_management        3.13.6
[  ] rabbitmq_stomp                    3.13.6
[  ] rabbitmq_stream                   3.13.6
[  ] rabbitmq_stream_management        3.13.6
[  ] rabbitmq_top                      3.13.6
[  ] rabbitmq_tracing                  3.13.6
[  ] rabbitmq_trust_store              3.13.6
[*] rabbitmq_web_dispatch              3.13.6
[  ] rabbitmq_web_mqtt                 3.13.6
[  ] rabbitmq_web_mqtt_examples        3.13.6
[  ] rabbitmq_web_stomp                3.13.6
[  ] rabbitmq_web_stomp_examples       3.13.6
```

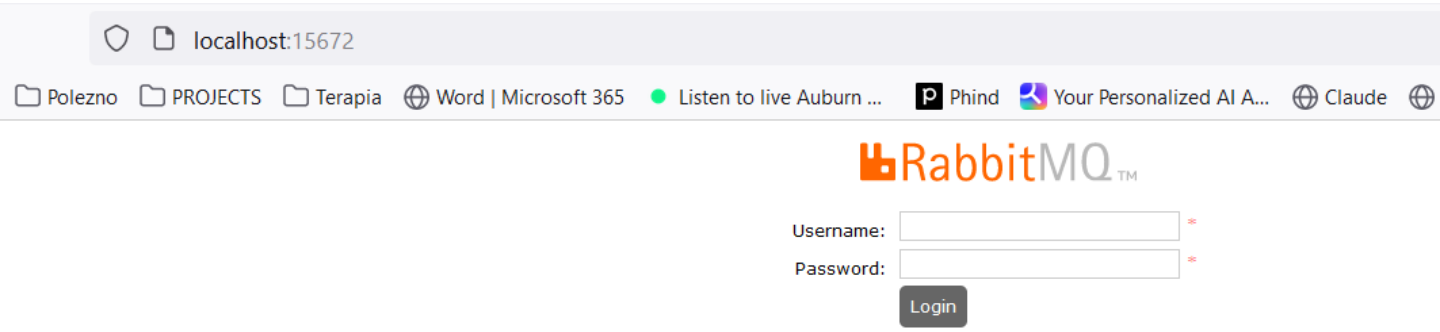
Type services.msc



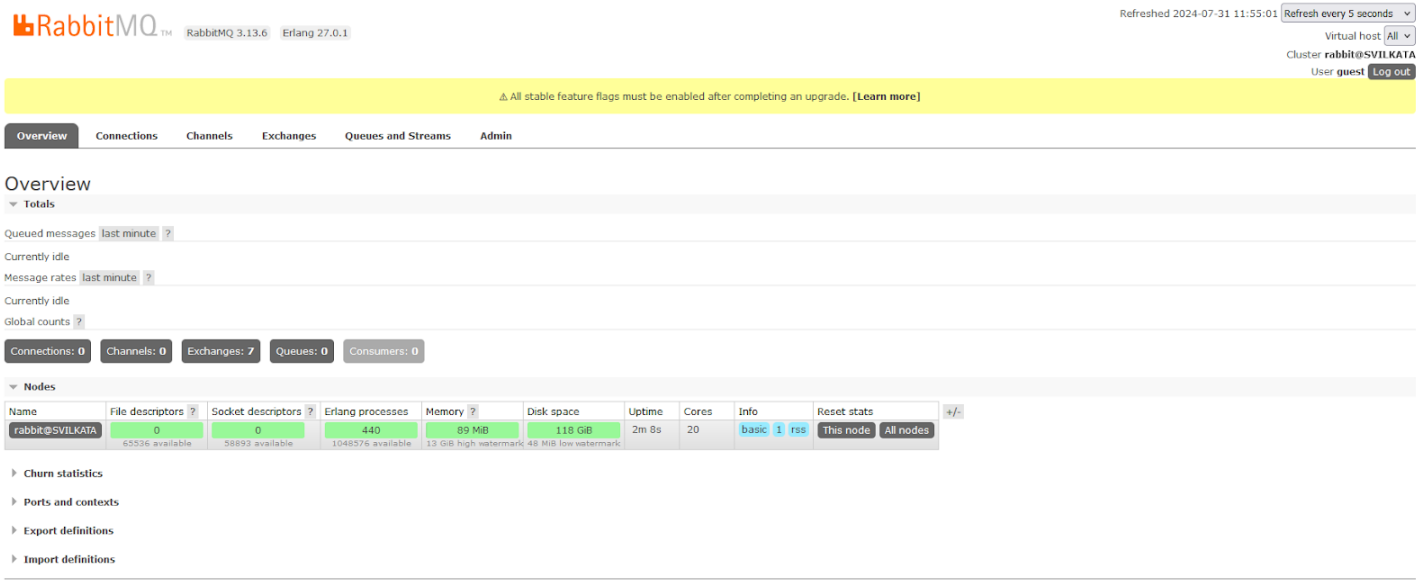
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>**rabbitmq-server.bat**

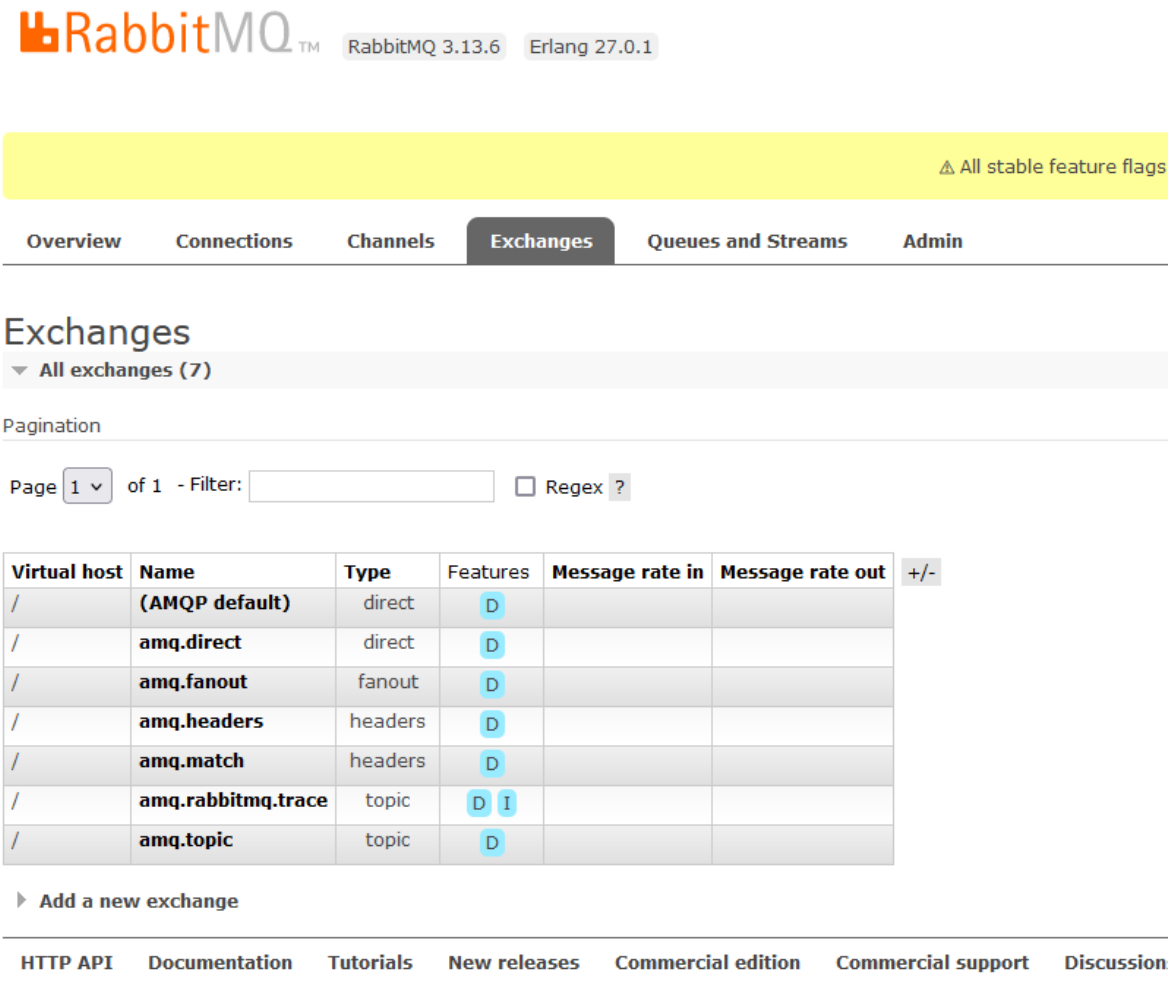
2024-07-31 11:52:47.571000+03:00 [warning] <0.134.0> Using RABBITMQ_ADVANCED_CONFIG_FILE: c:/Users/svilk/AppData/Roaming/RabbitMQ/advanced.config
2024-07-31 11:52:52.145000+03:00 [notice] <0.45.0> Application syslog exited with reason: stopped
2024-07-31 11:52:52.145000+03:00 [notice] <0.213.0> Logging: switching to configured handler(s); following messages may not be visible in this log output

```
  ##  ##      RabbitMQ 3.13.6
  ##  ##
  ##########  Copyright (c) 2007-2024 Broadcom Inc and/or its subsidiaries
  ######  ##
  ##########  Licensed under the MPL 2.0. Website: https://rabbitmq.com

  Erlang:      27.0.1 [jit]
  TLS Library: OpenSSL - OpenSSL 3.1.0 14 Mar 2023
  Release series support status: see https://www.rabbitmq.com/release-information

  Doc guides:  https://www.rabbitmq.com/docs
  Support:     https://www.rabbitmq.com/docs/contact
  Tutorials:   https://www.rabbitmq.com/tutorials
  Monitoring:  https://www.rabbitmq.com/docs/monitoring
  Upgrading:   https://www.rabbitmq.com/docs/upgrade

  Logs: <stdout>
        c:/Users/svilk/AppData/Roaming/RabbitMQ/log/rabbit@SVILKATA.log

  Config file(s): c:/Users/svilk/AppData/Roaming/RabbitMQ/advanced.config

  Starting broker... completed with 3 plugins.
```



username: guest
password: guest

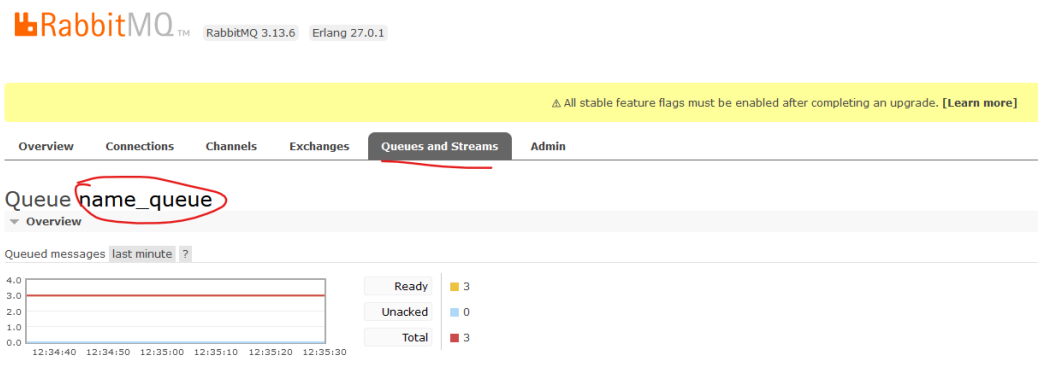


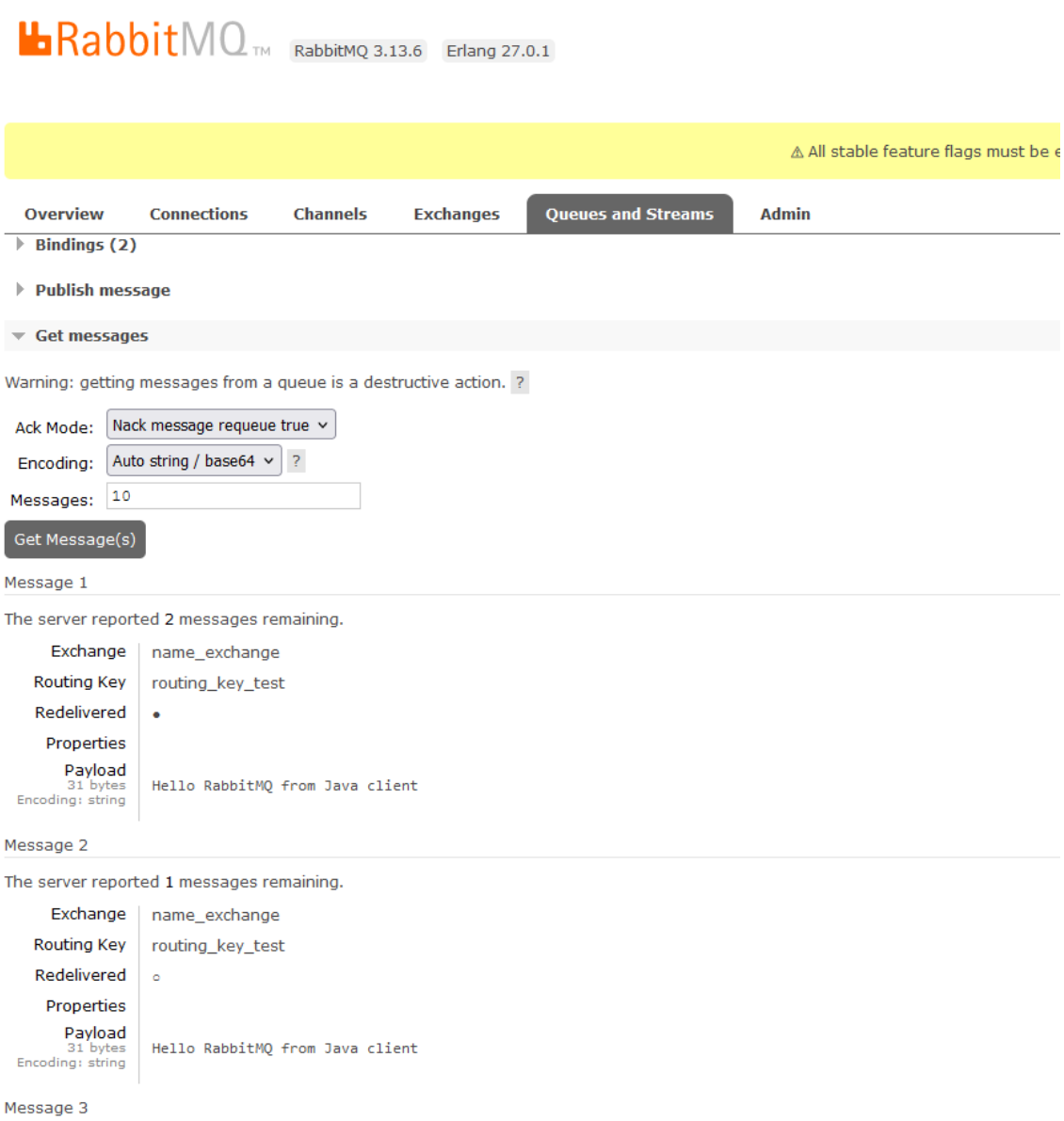


Using the Java Client

```
<dependency>
 <groupId>com.rabbitmq</groupId>
 <artifactId>amqp-client</artifactId>
 <version>5.20.0</version>
</dependency>
```

```java
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

public class Publisher {
   public static void main(String[] args) throws IOException, TimeoutException {
       Connection connection = null;
       Channel channel = null;

       try {
           ConnectionFactory connectionFactory = new ConnectionFactory();
           connectionFactory.setHost("localhost"); //by default on port 15672
           connection = connectionFactory.newConnection();
           channel = connection.createChannel();

           channel.exchangeDeclare("name_exchange", "direct"); //created only once on
the RabbitMQ server
           channel.queueDeclare("name_queue", false, false, false, null); //created
only once on the RabbitMQ server
           channel.queueBind("name_queue", "name_exchange", "routing_key_test");

           channel.basicPublish("name_exchange", "routing_key_test", null,
                   "Hello RabbitMQ from Java
client".getBytes(StandardCharsets.UTF_8));
       } finally {
           if (channel != null) {
               channel.close();
           }
           if (connection != null) {
               connection.close();
           }
       }
   }
}
```






```java
import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class Subscriber {
   public static void main(String[] args) throws IOException, TimeoutException,
InterruptedException {
       Connection connection = null;
       Channel channel = null;

       try {
           ConnectionFactory connectionFactory = new ConnectionFactory();
           connectionFactory.setHost("localhost"); //by default on port 15672
           connection = connectionFactory.newConnection();
           channel = connection.createChannel();
```

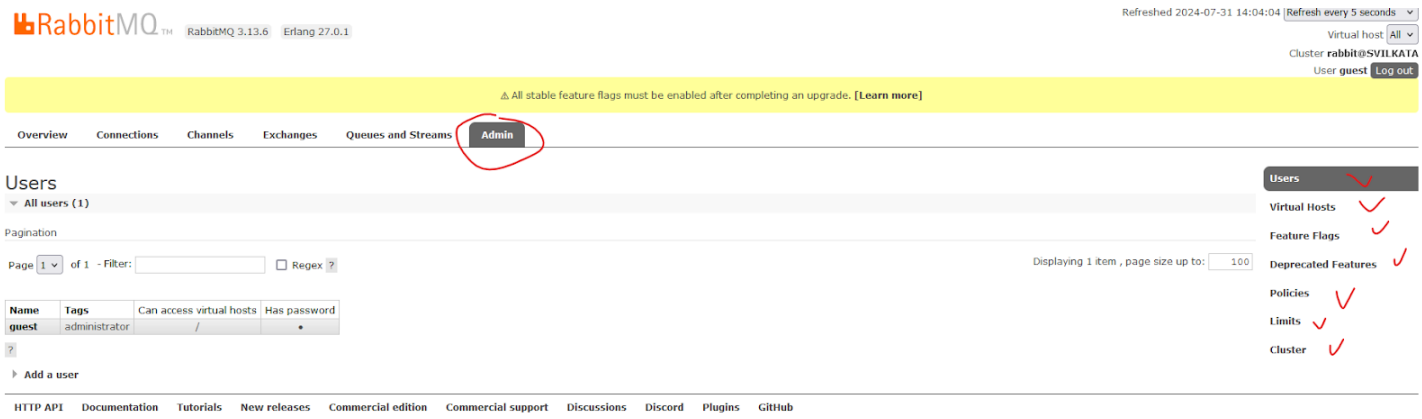
```
            channel.exchangeDeclare("name_exchange", "direct"); //created only once on
the RabbitMQ server
            channel.queueDeclare("name_queue", false, false, false, null); //created
only once on the RabbitMQ server
            channel.queueBind("name_queue", "name_exchange", "routing_key_test");

            while (true) {
                channel.basicConsume("name_queue", true, new DefaultConsumer(channel)
{
                    @Override
                    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                        // super.handleDelivery(consumerTag, envelope, properties,
body);  no-op no work to do
                        System.out.println(new String(body));
                    }
                });

                Thread.sleep(3000);
            }
        } finally {
            if (channel != null) {
                channel.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```

Administration

- Administration of the broker includes a number of activities such as:
  - Updating the broker
  - Backing up the broker database
  - installing/uninstalling and configuring plug-ins
  - Configuring the various components of the broker

- Apart from queues, exchanges and bindings we can also manage the following types of components:
  - vhosts (virtual hosts) - for logical separation of broker components
  - users
  - Parameters - defining upstream links to another brokers
  - Policies - for queue mirroring

- Administration of single instance or an entire cluster can be performed in several ways:
  - Using the management Web interface



- Using the management HTTP API - rest API
- Using the **rabbitmq-admin.py /** **rabbitmqadmin.py** script - written on Python
- Using the **rabbitmqctl** utility

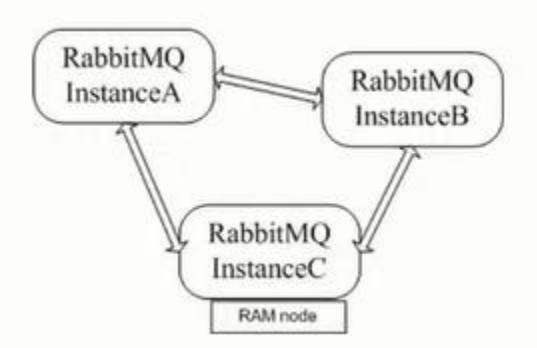
Scalability and High Availability in RabbitMQ

Basic default configuration

- RabbitMQ provides clustering support that allows new RabbitMQ nodes to be added on the fly
- Clustering by default does not guarantee that message loss may or may not occur - ако дадена инстанция примерно падне

- Nodes in a RabbitMQ cluster can be:
  - DISK - data is persisted in the node database
  - RAM - data is buffered only in-memory - когато не е критично да се запазват данните след рестарт например
- **Nodes share only broker metadata - messages are not replicated among nodes!! - съобщението не се репликира в останалите node-ве**

Example:
A и B са DISK nodes, а C е Ram node.



Instance A node DISK

```
set RABBITMQ_NODENAME=instanceA &
set RABBITMQ_NODE_PORT=5770 &
set RABBITMQ_SERVER_START_ARGS=
      -rabbitmq_management listener [{port,33333}] &
rabbitmq-server.bat -detached
```

Instance B node DISK

Пускаме инстанция B, спираме я, присъединяваме я след това

```
set RABBITMQ_NODENAME=instanceB &
set RABBITMQ_NODE_PORT=5771 &
rabbitmq-server.bat -detached
rabbitmqctl.bat -n instanceB stop_app
rabbitmqctl.bat -n instanceB join_cluster instanceA@MARTIN
rabbitmqctl.bat -n instanceB start_app
```

Instance C node RAM

Пускаме инстанция C, спираме я, присъединяваме я след това

```
set RABBITMQ_NODENAME=instanceC &
set RABBITMQ_NODE_PORT=5772 &
rabbitmq-server.bat -detached
rabbitmqctl.bat -n instanceC stop_app
rabbitmqctl.bat -n instanceC join_cluster -ram instanceA@MARTIN
rabbitmqctl.bat -n instanceC start_app
```

- If a node that hosts a queue buffers unprocessed messages goes down, then messages are lost
- Default clustering mechanism provides scalability in terms of queues rather than high availability

Mirrored queues

- **Mirrored queues** are an extension to the default clustering mechanism that can be used to establish **high availability** at the broker level
- Mirrored queues provide queue replication over different nodes that allows a message to survive node failure
- Queue mirroring is establishing by means of a mirroring policy that specifies:
  - Number of nodes to use for queue replication
  - Particular nodes designated by name for queue replication
  - All nodes for queue replication

- The node where the queue is created is the master node - all other nodes are slaves
- A new master node can be promoted in case the original one goes down
- A slave node is promoted to/as the new master in case it is fully synchronized with the old master

Example:

Let's define the test queue in the cluster and mirror it over all other nodes:

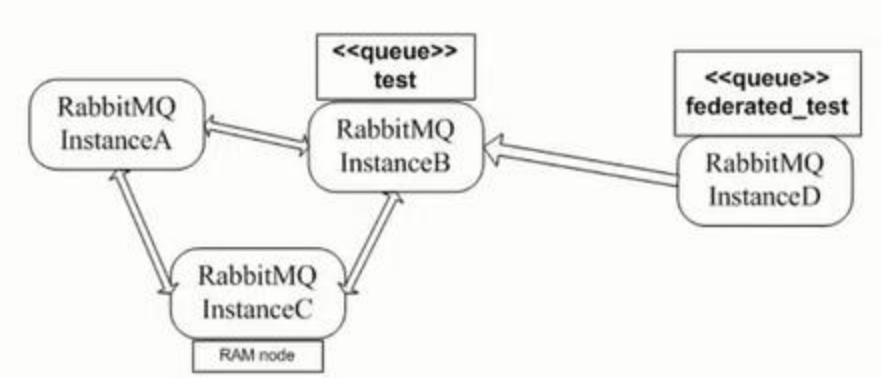
```
rabbitmqadmin.py -N instanceA declare queue name=test
durable=false
rabbitmqctl -n instanceA set_policy ha-all "test" "{""ha-
mode"":""all""}"
```

Federation and Shovel plugins

- The RabbitMQ clustering mechanism uses Erlang message passing along with a message cookie in order to establish communication between the nodes…….. which is **not reliable** over the Wide Area Networks!!
- In order to establish high availability among nodes in different geographic locations you can use the **federation, federation_management** and **shovel** plug-ins
- The shovel plug-in works at a lower level than the federation plug-in

Federation

- Ако искаме да репликираме опашката на отдалечена инстанция D:




```
set RABBITMQ_NODENAME=instanceD &
set RABBITMQ_NODE_PORT=6001 &
set RABBITMQ_SERVER_START_ARGS=
      -rabbitmq_management listener [{port,44444}] &
rabbitmq-server.bat -detached

rabbitmq-plugins -n instanceD enable rabbitmq_federation
rabbitmq-plugins -n instanceD enable
rabbitmq_federation_management
```

- Declare the **federated_test** queue

```
rabbitmqadmin.py -N instanceD -P 44444 declare queue
name=federated_test durable=false
```

Declare the upstream to the initial cluster and set a federation link to the **test** queue:

```
rabbitmqctl -n instanceD set_parameter federation-upstream
upstream
"{""uri"":""amqp://localhost:5770"",""expires"":3600000,
""queue"":""test""}"

rabbitmqctl -n instanceD set_policy federate-queue
--apply-to queues "federated_test"
"{""federation-upstream"":""upstream""}"
```

Shovel

The shovel plug-in provides two variants:

- **static** - all links between the source/destination nodes/clusters are defined statically in the RabbitMQ configuration file
- **dynamic** - all links between the source/destination nodes/clusters are defined dynamically via the RabbitMQ parameters

source \ destination	exchange	queue
exchange	federation dynamic shovel	dynamic shovel
queue	static shovel dynamic shovel	federation dynamic shovel

Integrations

Info

- RabbitMQ provides integrations with other protocols such as STOMP, MQTT and LDAP by means of RabbitMQ plug-ins
- Using the Java Client - already discussed above
- The Spring framework provides integration with AMQP protocol and RabbitMQ in particular
- The **Spring AMQP framework** provides:
  - **RabbitAdmin** class for automatically declaring queues, exchanges and bindings
  - **Listener container** for asynchronous processing of inbound messages
  - **RabbitTemplate** class for sending and receiving messages

- Utilities of the Spring AMQP framework can be used directly in Java or preconfigured in the Spring configuration
- The **Spring Integration framework** to **Spring Boot** provides adapters for the AMQP protocol

- Integration with Quarkus framework

Spring AMQP framework

```
<dependencies>
    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>1.4.5.RELEASE</version>
    </dependency>
</dependencies>
```

The **RabbitAdmin** class:

```
CachingConnectionFactory factory = new
       CachingConnectionFactory("localhost");
RabbitAdmin admin = new RabbitAdmin(factory);
Queue queue = new Queue("sample-queue");
admin.declareQueue(queue);
TopicExchange exchange = new TopicExchange("sample-topic-
       exchange");
admin.declareExchange(exchange);
admin.declareBinding(BindingBuilder.bind(queue).to(exchange)
       .with("sample-key"));
factory.destroy();
```

Listener container

```
CachingConnectionFactory factory =
       new CachingConnectionFactory(
"localhost");
SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer(
       factory);
Object listener = new Object() {
    public void handleMessage(String message) {
    System.out.println("Message received: " +
    message);
}};
MessageListenerAdapter adapter = new
       MessageListenerAdapter(listener);
container.setMessageListener(adapter);
container.setQueueNames("sample-queue");
container.start();
```

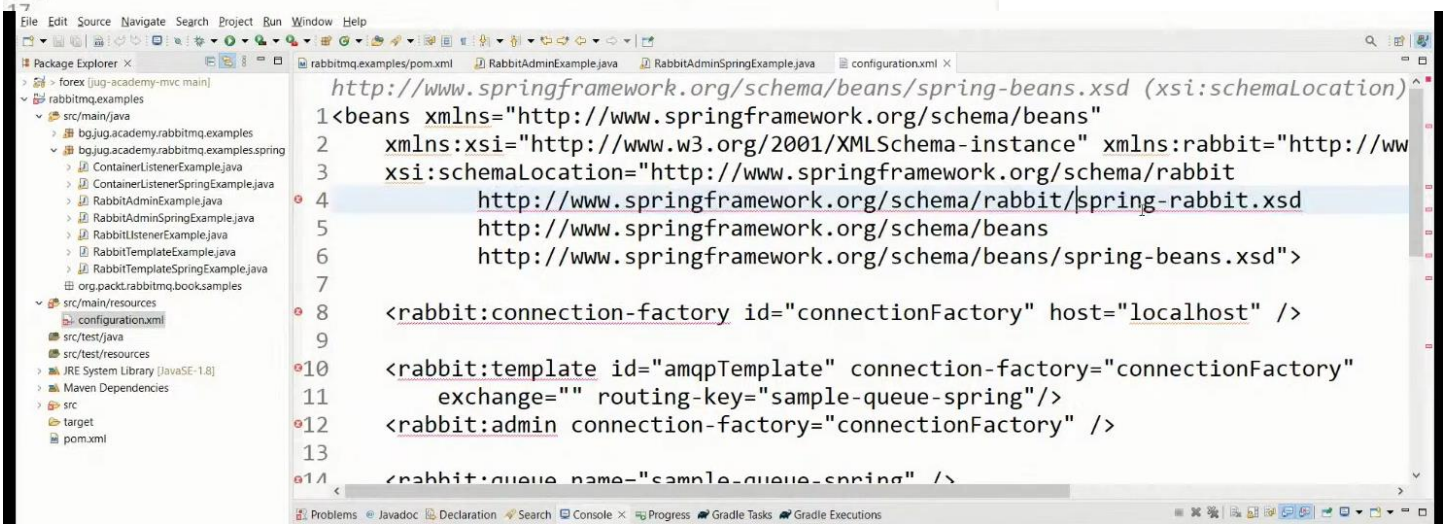
The **RabbitTemplate** class:

```
CachingConnectionFactory factory =
       new CachingConnectionFactory("localhost");
RabbitTemplate template = new
RabbitTemplate(factory);
template.convertAndSend("", "sample-queue",
        "sample-queue test message!");
```

- All of the above Spring AMQP framework examples can be configured using the Spring configuration - so that to be cleaner and to decouple RabbitMQ configuration from the business logic/Java code

For example within a **configuration.xml** file:

```
7 public class RabbitAdminSpringExample {
8
9     public static void main(String[] args) {
10
11         AbstractApplicationContext context = new ClassPathXmlApplicationContext(
12                 "configuration.xml");
13         RabbitAdmin admin = context.getBean(RabbitAdmin.class);
14     }
15
16 }
```



Spring Boot Starter AMQP - Spring Integration framework

In gradle

implementation ‘org.springframework.boot:**spring-boot-starter-amqp**’

Предоставя ни бийнове за **RabbitAdmin** class, **Listener container** and **RabbitTemplate** class.

Посредством дефиниране на бийнове - можем да си декларираме **exchange, queue** или **queueBinding**

```
@Component
public class EventingServiceImpl implements EventingService {

    private final RabbitTemplate template;

    public EventingServiceImpl(RabbitTemplate template) {
        this.template = template;
    }

    @Bean
    public Queue createExchangeQueue() {
        return new Queue("exchange_rate_queue");
    @Override
    public void publish(String message) {
        template.convertAndSend("exchange_rate_queue", message);
    }

}
```

Quarkus framework

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-smallrye-reactive-messaging-rabbitmq</artifactId>
</dependency>
```

In application.properties file

```
mp.messaging.outgoing.bi.use-ssl=true
mp.messaging.outgoing.bi.connector=smallrye-rabbitmq
mp.messaging.outgoing.bi.exchange.type=direct
mp.messaging.outgoing.bi.port=5672
```

```
import io.smallrye.reactive.messaging.rabbitmq.OutgoingRabbitMQMetadata;

@Inject
@Channel("asd")  //from Microprofile
Emitter<String> emitter;  //from Microprofile

public void emmitMessage(RabbitMqPayload payload) {
   String messagePayload = JsonUtils.toJsonString(List.of(payload));
   LOGGER.debugf("Emitting Bi message to RabbitMQ %s", messagePayload);
   OutgoingRabbitMQMetadata metadata = new OutgoingRabbitMQMetadata.Builder()
           .withRoutingKey(payload.routingKey())
           .build();

   Message<String> message = Message.of(messagePayload, Metadata.of(metadata));
   biEmitter.send(message); //from Microprofile
   LOGGER.infof("Bi message emitted to route %s", payload.routingKey());
}
```

Security

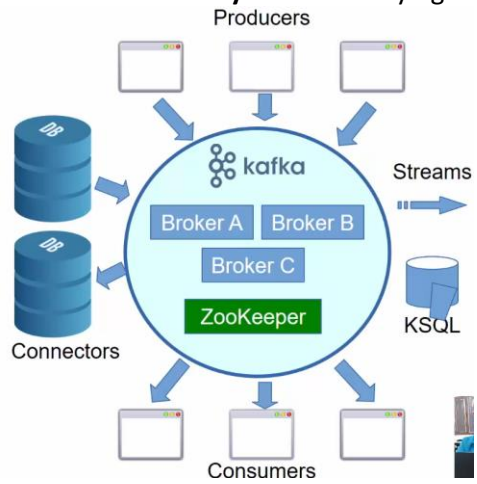
- RabbitMQ uses SASL Simple Authentication Security Layer for authentication (SASL PLAIN used by default)
- RabbitMQ uses access control lists (permissions) for authorization
- SSL/TLS support can be enabled for the AMQP communication channels
- SSL/TLS support can be enabled for node communication between nodes in a cluster
- SSL/TLS support can be enabled for the federation and shovel plug-ins

II. Apache Kafka

Book
Kafka – The Definitive Guide – Real-Time Data and Stream Processing at Scale

Main Concepts

- Kafka is run as a cluster on one or more servers (brokers) that can span multiple datacenters
- The Kafka **cluster** stores **streams of records** in one or more **brokers** in categories called **topics**
- Each record consists of a **key, value and timestamp**
- **Horizontally scalable** – trying to put more nodes

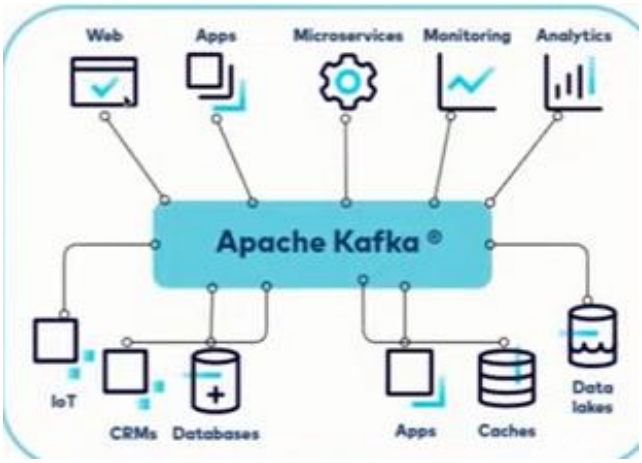


KSQL – стария подход без стриймове

- The **Producer API** – publish a stream of records to one or more Kafka topics
- The **Consumer API** – subscribe to one or more topics and process the stream of records produced
- The **Streams API** – a stream processor, consuming an input stream form one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams
- The **Connector API** – allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems – e.g. connector to a DB might capture every change in a table

Use cases

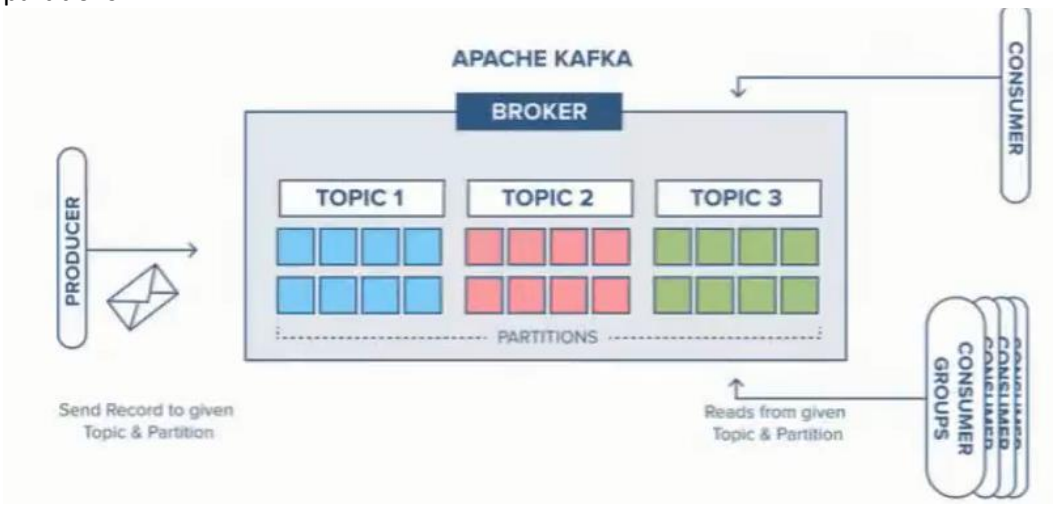
- For activity tracking – например LinkedIn да знае кой профил гледате/се гледа най-много
- Messaging
- Metrics and logging
- Commit log
- Stream processing – data pipeline – за един ден колко човека са кликнали на моята страница



Brokers and Clusters

A single Kafka server/instance is called a ***broker***. The broker receives messages from producers, assigns offsets to them, and writes the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands and millions of messages per second.

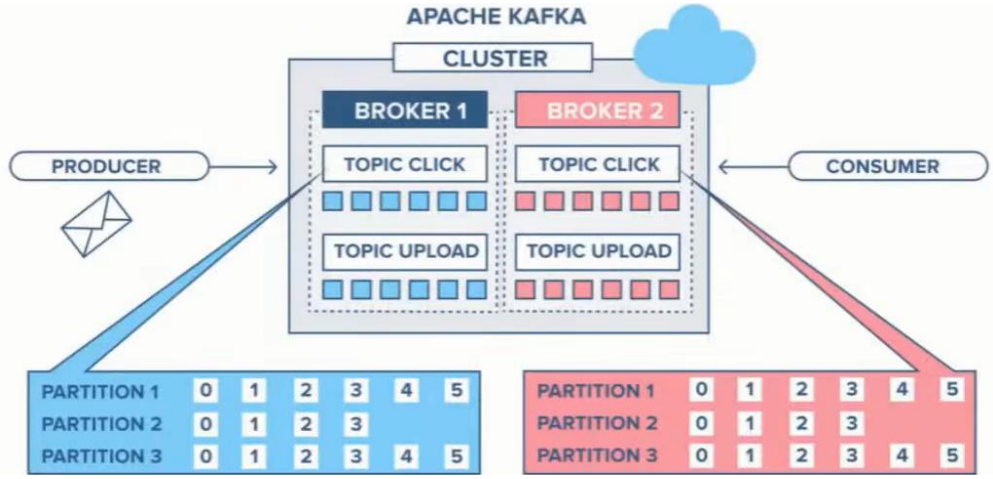
В един Kafka инстанция/брокер може да има 1 или повече топици, а във всеки топик има определен брой partitions.



Cluster – комбинация от няколко брокера (поне 3), т.е. няколко брокера които работят заедно и се координират заедно.

С други думи да има Fault tolerance, и когато един broker instance падне/или целият cluster падне, то да има опция да се репликират данните.

Като си правим cluster, то самите broker instances да бъдат географски на различни места – че ако падне нета на единия instance, да не паднат всичките брокер инстанции и дефакто целия клъстер.



- Kafka is maintained as clusters where each **node** within a cluster **is called a Broker**. Multiple brokers allow us to evenly distribute data across multiple servers and partitions. This load should be monitored continuously and brokers and topics should be reassigned when necessary.
- Each Kafka cluster will designate on of the brokers as the **Controller** which is responsible for managing and maintaining the overall **health of a cluster**, in addition to the basic broker responsibilities. Controllers are responsible for:
    - creating/deleting topics and partitions
    - taking action to rebalance partitions
    - assign partition leaders
    - handle situations when nodes fail or get added
- Controllers subscribe to receive notifications from **ZooKeeper** which tracks the state of all nodes, partitions and replicas

Topics and Partitions

- **Topic** = stream of records. A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one or many **consumers** that subscribe to the data.
- For each topic, the Kafka cluster maintains a **partition log** – each **partition** is an ordered, immutable sequence of records that is continually appended to – a structured **commit log.**
- The records in each partition are each assigned a sequential id number called the **offset** that uniquely identifies each record within the partition.
- Each partition is append-only. Records in the partition can be deleted automatically (whole segments are deleted) based on initial configuration about capacity.
- Ако in-memory буфера се напълни, то чак тогава данните се записват в база данни – подлежи на настройка

- With key – Messages are appended in the same partition if they have the same key
- Without key – messages are appended to the next partition in a round-robin fashion
- The order is guaranteed only in a partition and only if we have/use **key**

Kafka topic – основната единица

Колкото partitions има в даден топик, то толкова паралелни consumers може да имаме.

Ако имаме повече от 1 partition в даден топик, то нямаме гаранция за подредба. Освен разбира се ако не използваме **key!**

Partition представлява множество съобщения/елементи едно след друго, с определен индекс, който наричаме **offset!**

Producers

Producer commit

Producers publish data to the topics of their choice. The producer is responsible for **choosing which record to assign to which partition** within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (e.g. based on **key's hash value**).

From the producer side, "commit" refers to the acknowledgement received from the Kafka brokers that a message has been successfully written to the topic partitions.

Key points about **producer commits**:

It provides durability guarantees based on the acks configuration.
With acks=0, no acknowledgement is expected from the brokers.
With acks=all, the producer waits for all in-sync replicas to confirm receipt before considering the message committed.

Batching

spring.kafka.producer.batch-size=100
Sets the maximum number of records to be sent om one request. Sends the batch as soon as it is filled.

spring.kafka.producer.properties[linger.ms]
Gives the batch that much ms to be filled and sent automatically (see above). After the duration, if the batch is still not filled, sends it as it is.

spring.kafka.producer.buffer-memory
If a batch cannot be sent due to broker being down or whatever other recoverable issue, fills new batches in the buffer. After the buffer is filled, the **send()** method blocks for **spring.kafka.producer.properties[max.block.ms]**. This buffer **must be large enough** to contain at least one full batch or such batches will be lost.

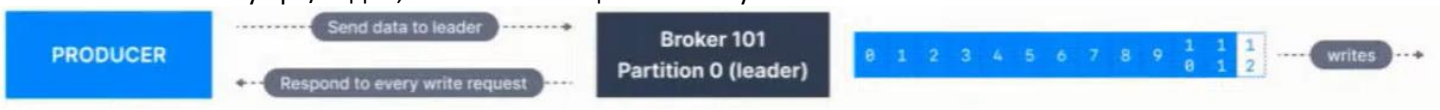
spring.kafka.producer.compression-type=lz4 avro gzip snappy
The batch will be compressed before sending. This directly increases throughput and latency (not much latency)
Следните неща ни интересуват като изпращаме съобщения:

- **acks=0**

Does not wait for a response to consider the sent successful

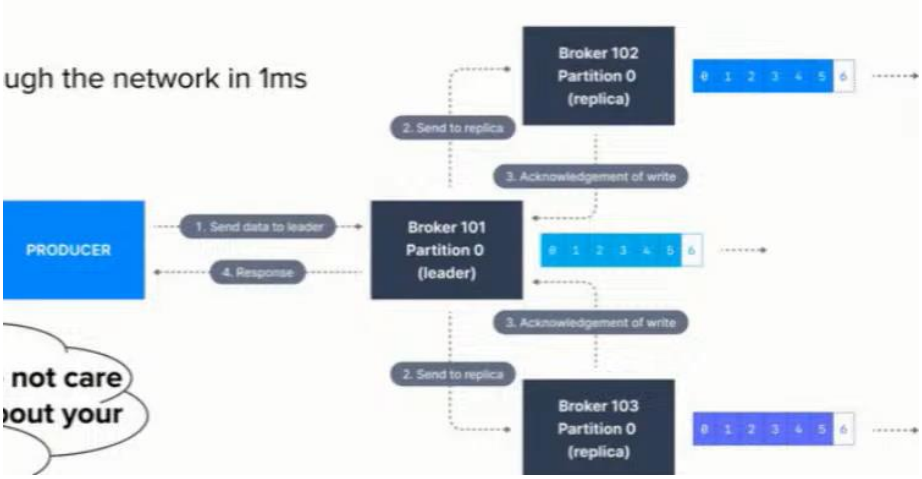
- **acks=1**

Wait for the leader to commit the message/s but do not wait for the replicas to commit!!
Изчакай поне Leader да ти върне, че е записано съобщението, но не чакаме да се случи репликацията. Проблем би бил ако leader-а умре/падне, и тогава съобщението се губи



- **acks=all**

Wait for the leader and all replicas(или еди колко си на брой реплики) to commit, so that to consider the message/s sent.
Note: all == -1



x2 slower – докато се получат повтържденията за репликите, и се забавя
Assuming that a message/s is sent through the network in 1ms. This leads to 2.5ms latency.
1000 / 2ms = 2micro s per message

Kafka Producer send() Method

• Asynchronously sends a record to a topic
new ProducerRecord(TOPIC , reading.getId (), reading)

• Allows sending many records without blocking for a broker response

• send() method returns a Future

• Two forms:
  - send() method without a callback
  - send() method with a callback - the callback gets invoked when the broker has acknowledged the send

[ACK = -1 (all ISR), 0 or 1 (leader)]

• Callbacks for records sent to same partition are executed in the sent order

• Callback receives RecordMetadata which contains a record's partition, offset, and timestamp, and possible Exception if there was an error sending.

Kafka send() Method Exceptions

•InterruptException - If thread is interrupted while blocking

•SerializationException - If key or value can not be serialized using configured serializers
•TimeoutException – when fetching metadata or allocating memory exceeds max.block.ms, or getting **acks** from Broker exceed timeout.ms, etc.
•KafkaException - when Kafka error occurs, but not in public API
•AuthenticationException - if authentication fails
•AuthorizationException - the producer is not allowed to write
•IllegalStateException -if a transactional.id has been configured and no transaction has been started, or when send() invoked on closed producer

Kafka Producer partitionsFor() Method

•partitionsFor(topic) - returns meta-data for partitions:
public List<PartitionInfo> partitionsFor(String topic)

•Used by producers that implement their own partitioning – for custom partitioning

•PartitionInfo consists of topic, partition, leader node (Node), replica nodes (Node[]) and inSyncReplicanodes.

•Node consists of id, host, port, and rack

Kafka Producer Interceptors

•Activate interceptors by adding them to interceptor.classes property of the producer

```
props.put(
ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
CountingProducerInterceptor.class.getName());
```

•Producer interceptor methods:

```
public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)
public void onAcknowledgement(RecordMetadatametadata, Exception exception)
```

Kafka Producer flush() and close() Methods

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        executor.shutdown();
        try {
            executor.awaitTermination(200, TimeUnit.MILLISECONDS);
            log.info("Flushing and closing producer");
            producer.flush();
            producer.close(10_000, TimeUnit.MILLISECONDS);
        } catch (InterruptedExceptione) {
            log.warn("shutting down", e);
        }}
);
```

Kafka Producer metrics() Method

•metrics() - used to get a map of metrics:
public Map<MetricName,? extends Metric> metrics()

•Returns a full set of producer metrics.

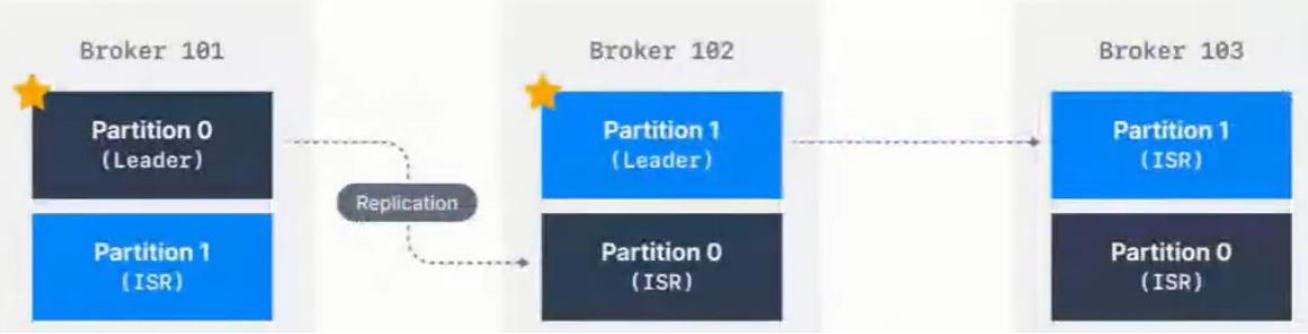
•MetricName consists of name, group, description, and tags(Map).

•Metric consists of a MetricName and a Measurablevalue(double) or Object (gauge).

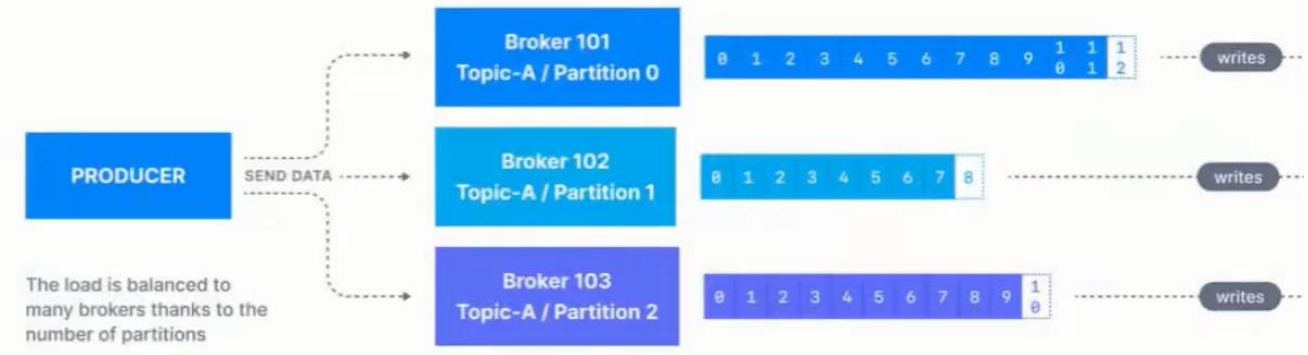
Topic replication and read balancing

- **min.insync.replicas** – how many brokers (including the leader) should have the data before it is considered stored and ready to serve. Recommended is **replication.factor -1** (можем да настроим да я има тази информация на само 1 брокер, на още 2 брокера, и т.н. – зависи от use case-а и от наличните брокери)

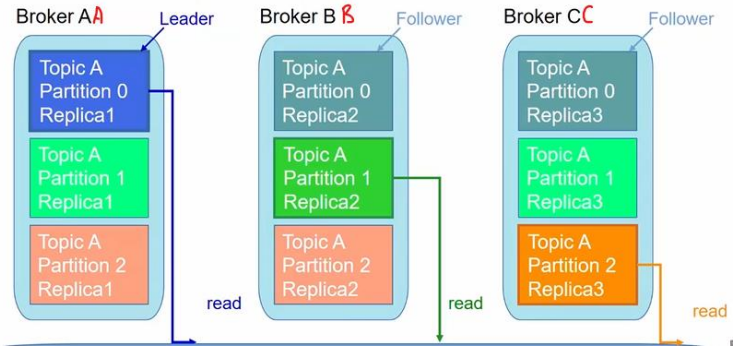
Each partition is replicated separately, has its own **broker leader** and ISR.
Обикновено единия брокер се води Leader на partition-а.



Когато изпращаме данни, ние изпращаме данни към съответния Leader на съответния partition.

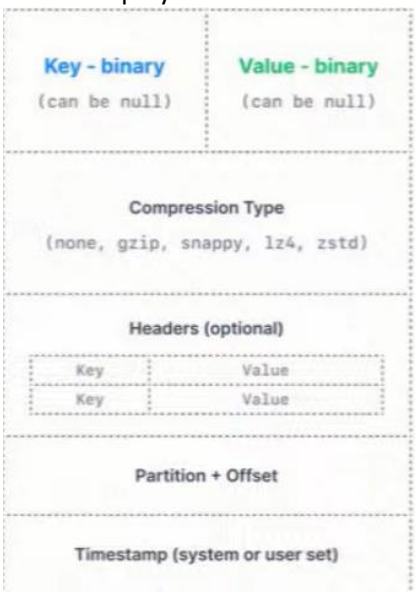


За **topic A в нашето Кафка приложение** примерно имаме 3 брокера/Кафка инстанции - имаме Leader и Followers – т.е. за topic A имаме **Leader брокер AA**, който преразпределя и казва дали ако информацията е **reading**, то да не я прави той, а да предостави възможност broker BB или broker CC да извърши тази операция – съответно през **replica2** на partition1 на брокер BB; или през **replica3** на partition2 на брокер CC.



What is a Kafka message

И key и value са **binary**! За Кафка всичко е byte-ове.
Възможност за компресия. Запазват се sequential – на следващия сегмент на диска.
Headers – например информация през кои сървиси е минало
Информация на кой partition е, и на кой офсет индекс е даденото съобщение.
Timestamp by default се слага от producer-а, а не вътрешно от Кафка брокера!



- Messages are stored inside topics within a log structured format, where the data gets written sequentially.
- A message can have a maximum size of 1MB by default, and while this is configurable, Kafka was not designed to process large size records. It is recommended to split large payloads into smaller messages, using identical key values so they all get saved in the same partition as well as assigning part numbers to each split message in order to reconstruct it on the consumer.
- Messages (aka Records) are always written in record batches. Record batches and records have their own headers. The detailed format of each is described in: http://kafka.apache.org/documentation/#recordbatch

Consumers

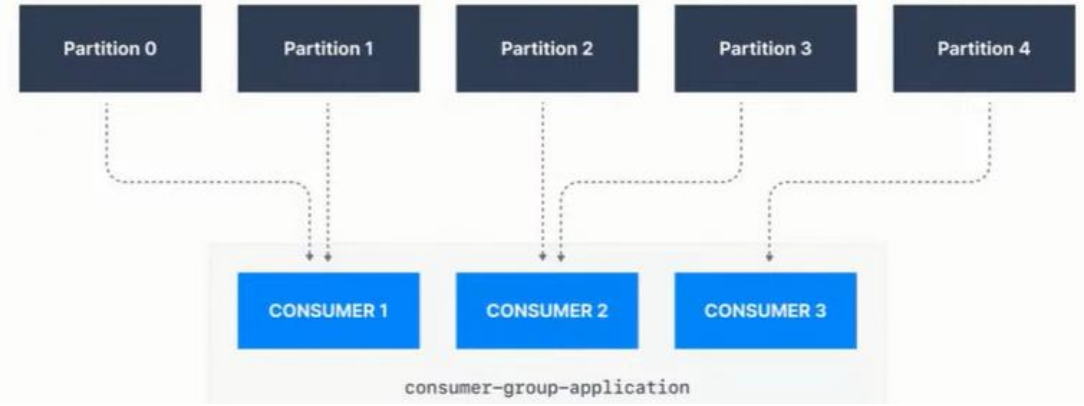
Kafka скалира не на базата на топици, а на базата на partitions!!!

- Consumers label themselves with a **consumer group**, and each record published to a topic **is delivered to one consumer instance within a consumer group**. Can be separate processes/machines
- Same consumer group – records will effectively be load balanced
- Different consumer groups – **broadcast** to all the consumers

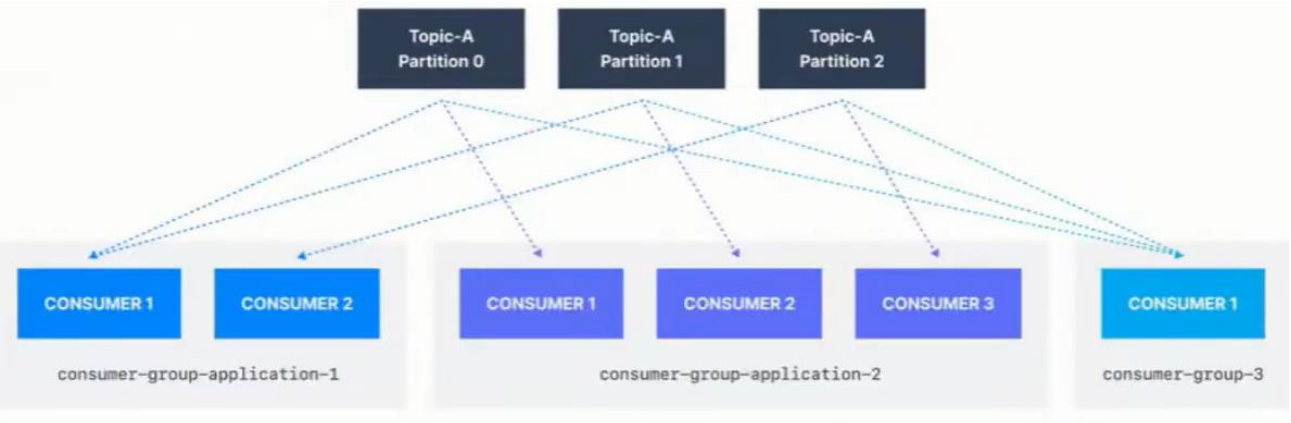
Consumer group

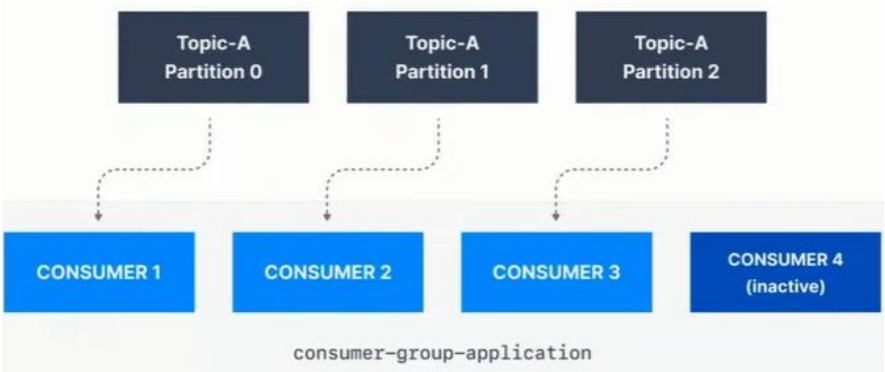
- **Consumer group = "logical subscriber"** -> many consumer instances for scalability and fault tolerance = publish-subscribe semantics where the **subscriber is a cluster of consumers** instead of a single process.
- The way consumption is implemented in Kafka is by **dividing up the partitions in the log over the consumer instances** so that each instance is the exclusive consumer of a 'share' of partitions at any point of time
- Group membership is handled by the Kafka protocol **dynamically**. If new instances join the group, they will take some partitions from other group members; if an instance dies, its partitions will be distributed.
- Kafka only provides a **total order over records within a partition**, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data **by key** is sufficient for most applications.
- **If you require a total order over records** ----> use a **topic that has only one partition** = only one consumer process per consumer

Пример: Имаме 5 партишъни. И разпределяме партишъните на различни consumers.
В текущият пример можем да сложим общо макс 5 consumers!!! Или с други думи правилото е че можем да скалираме consumers до броя на partitions!



Many consumers groups

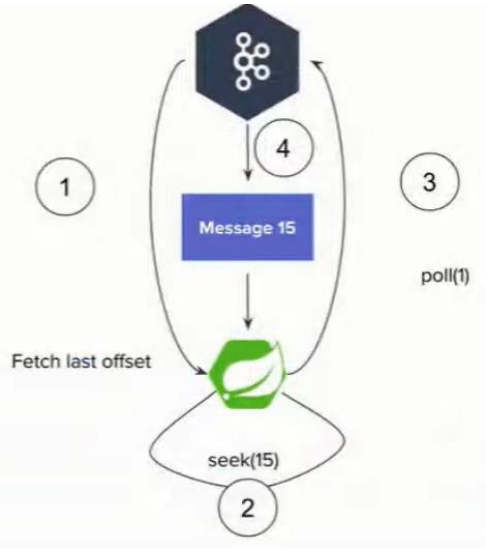




Consumer commit

- Consumer Commit is the process of telling Kafka to remember that you have processed **(consumed)** the messages of a partition to a specific offset
- When first started, the consumer looks up where it left off via the kafka commit facility
- A call to **seek()** is made to initialize an internal counter which determines the messages that will be poll()-ed for the lifetime of the **consumer**

Poll означава вземи съобщението от брокера, и го процесни при теб (при Spring framework-а например)



In Kafka consumers**, "commit" refers to the process of marking a message or offset as having been processed successfully**. This is typically done by the consumer application itself, not by the Kafka brokers 2.

Key points about committing in Kafka consumers:
- It allows the consumer to keep track of which messages it has already processed.
- It enables resuming from the last committed offset if the consumer crashes or restarts.
- It helps prevent duplicate processing of messages.

Consumer

Batching again

spring.kafka.consumer.max-poll-records=500
Sets the maximum number of records that will be returned by a poll()

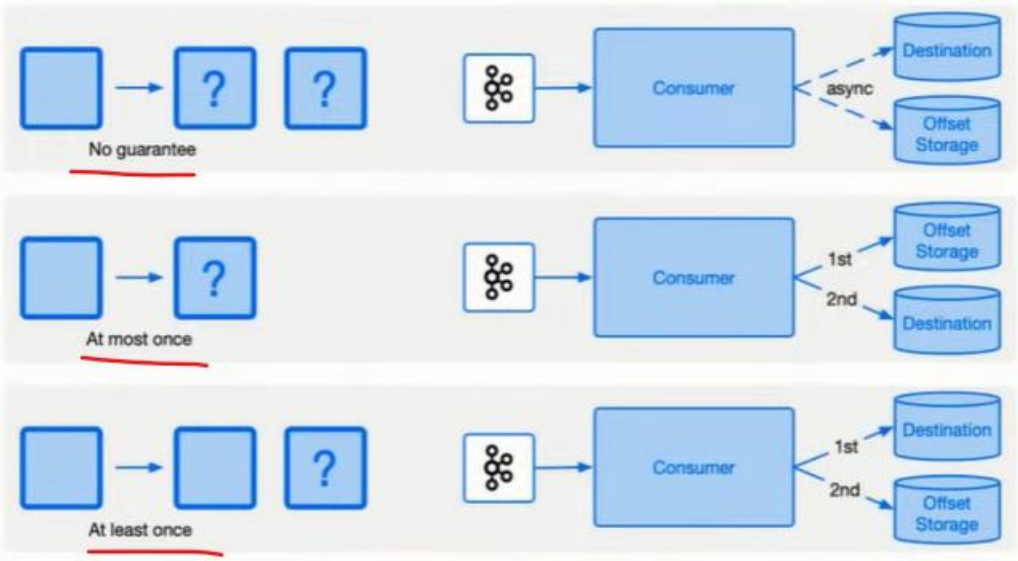
spring.kafka.listener.ack-mode=batch
Auto commit is mostly evil. Don't use it by default unless actually verifiably ok for your data
Като процеснеш batch-а, то commit-ни ръчно тогава offset-а. Ако се закача наново, то да не го процесна същото съобщение.
spring.kafka.consumer.enable-auto-commit=false

max.poll.interval.ms=5min
You must be able to process a full batch in that time or a rebalance will be triggered. На колко време да вземаме съобщенията обратно.

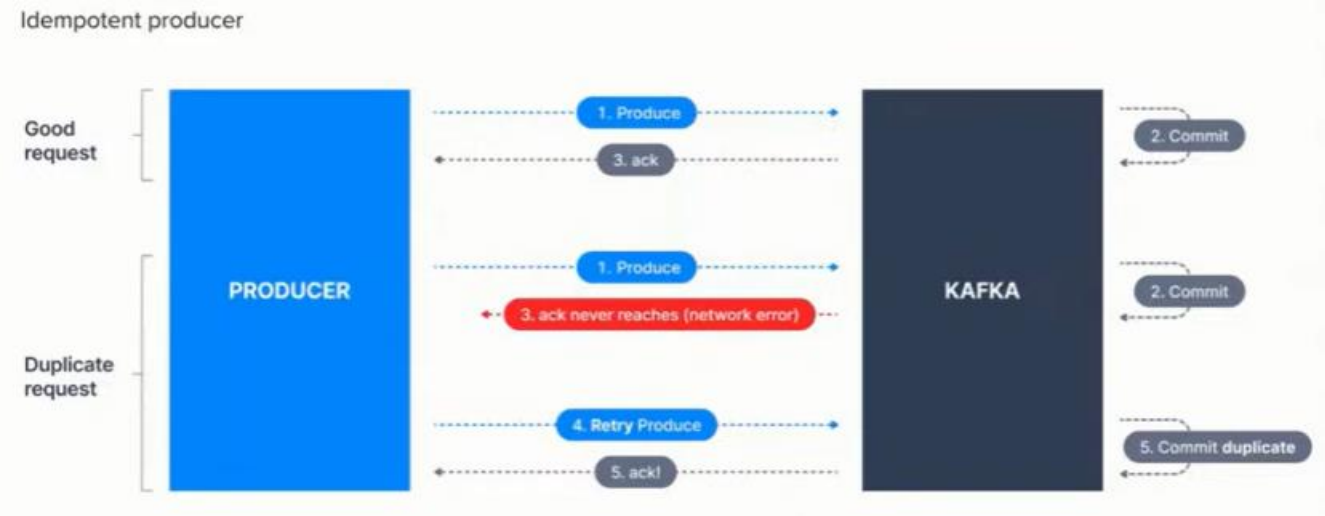
Delivery guarantees

Това съобщение защо сме го процеснали 2 пъти.
Или това съобщение защо изобщо не сме го процеснали.

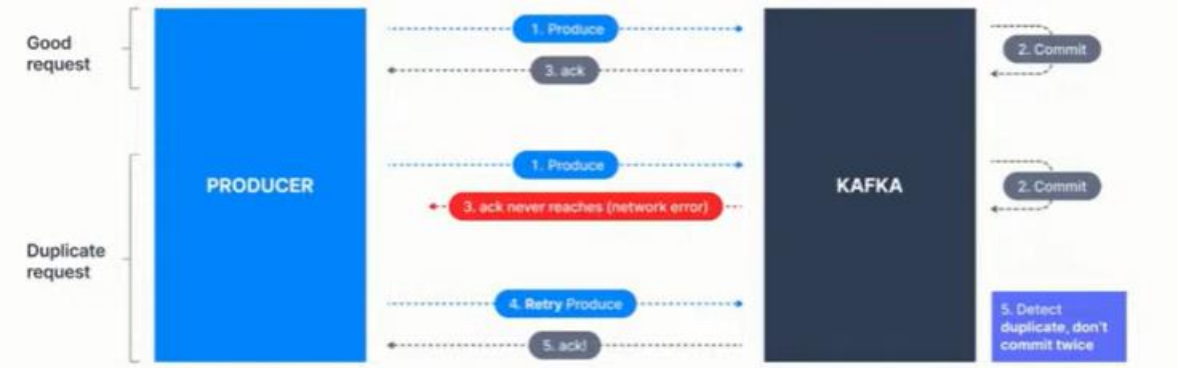
At **least once** means not that a single message might be duplicated but a whole batch it is scary with big batches 😊



Idempotent producer – получаване на дуплицирани съобщения
Поради network проблем може Кафка брокера да е приел съобщението и да не отговори, и ние да изпратим съобщението наново.



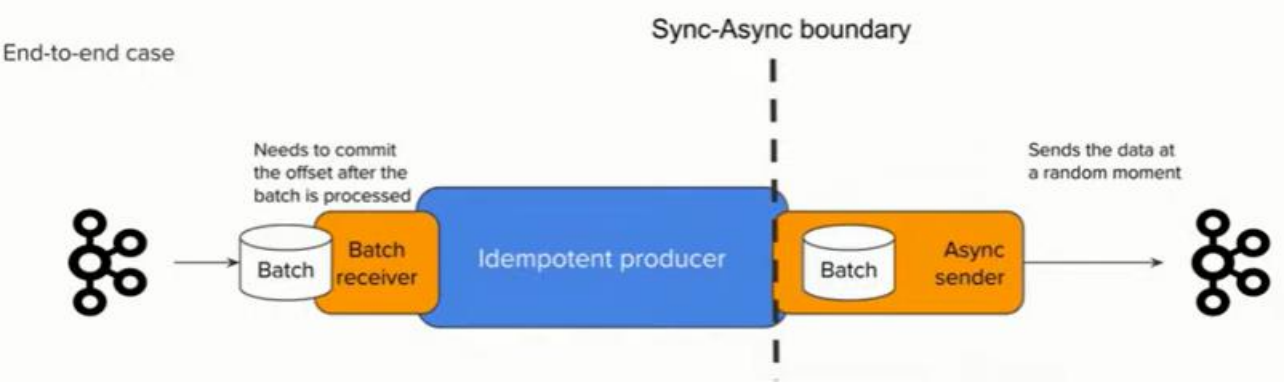
Idempotent producer
spring.kafka.producer.properties[max.in.flight.requests.per.connection]=5
spring.kafka.producer.properties[enable.idempotence]=true
It works for the lifespan of the producer. Required but not enough for proper idempotency across restarts.



Ако е Network проблем, това решава дуплицирането.

Consumer – producer problem

Ако чета в Кафка, и пиша в Кафка – имам проблем. Защото трябва да имаме/използваме нещо, което се казва трансакции (трансакции в Кафка).



kafkaTemplate.send(msg)

This is fully async. You don't know when the data is actually going to be sent. The method returns a future that can be waited but to do it for each message breaks the batching and is extremely slow.

Kafka Transactions

Ако ползваме база данни да записваме Kafka съобщения, то няма как едновременно да използваме Kafka transactions и db transactions! Няма да е атомарна цялата операция с 2 вида транзакции!

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());
producer.initTransactions();
try {
        producer.beginTransaction();
        for (inti= 0; i< 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i),Integer.toString(i)));
        producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException| AuthorizationException e) {
// We can't recover from these exceptions, so our only option is to close the producer and exit.
        producer.close();
} catch (KafkaExceptione) {
// For all other exceptions, just abort the transaction and try again.
        producer.abortTransaction();
}

producer.close();
```

Demo with kafka-clients

```
Docker-compose file for Zookeeper, Kafka and RedPanda
and also logback.xml configuration. – see BGJUG public repo.
```

build.gradle – Groovy style

```
dependencies {
    implementation group: 'org.apache.kafka', name: 'kafka-clients', version: '3.6.1'
    //SLF4J Api
    implementation 'org.slf4j:slf4j-api:1.7.32' //Use the latest version available

    //Logback (SLF$J implementation)
    implementation 'ch.qos.logback:logback-classic:1.2.6' //Use the latest version available
```

```
}
```

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class ProducerExample {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", "localhost:29092");
        properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        properties.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(properties);

        producer.send(new ProducerRecord<>("bgjug","KAFKA","Welcome to RabbitMQ"), (metadata,
exception) -> {
            if (exception == null) {
                System.out.println("Message sent successfully - Topic: " +
                        metadata.topic() + ", Partition: " + metadata.partition() +
                        ", Offset: " + metadata.offset());
            } else {
                System.err.println("Error sending message: " + exception.getMessage());
            }
        });

        producer.close();
    }
}
```

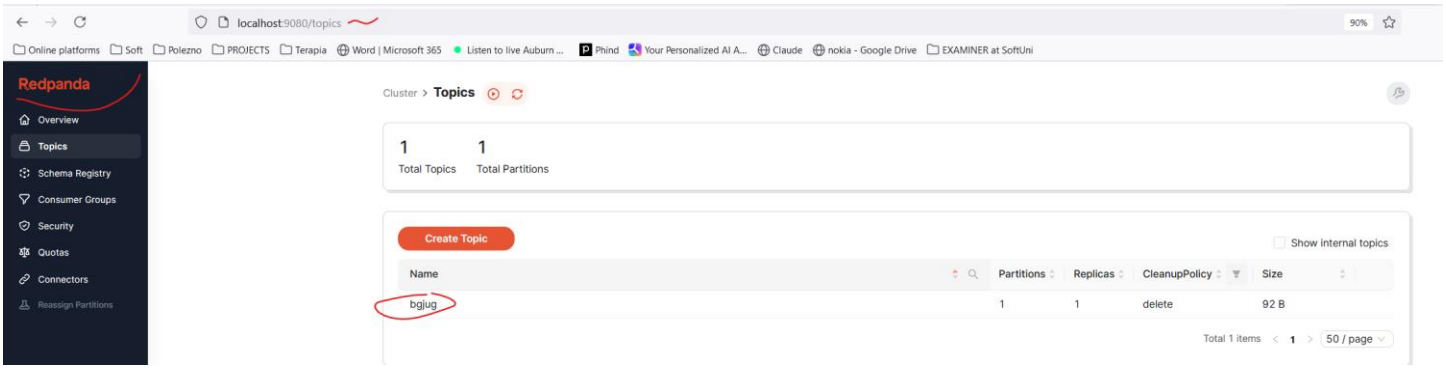
След като пуснем Producer-а, получаваме следния лог:

```
13:19:15.064 [main] INFO  o.a.k.c.producer.ProducerConfig - ProducerConfig values:
        acks = -1
        auto.include.jmx.reporter = true
        batch.size = 16384
        bootstrap.servers = [localhost:29092]
        buffer.memory = 33554432
        client.dns.lookup = use_all_dns_ips
        client.id = producer-1
        compression.type = none
        connections.max.idle.ms = 540000
        delivery.timeout.ms = 120000
        enable.idempotence = true
        interceptor.classes = []
        key.serializer = class org.apache.kafka.common.serialization.StringSerializer
        linger.ms = 0
        max.block.ms = 60000
        max.in.flight.requests.per.connection = 5
        max.request.size = 1048576
        metadata.max.age.ms = 300000
        metadata.max.idle.ms = 300000
        metric.reporters = []
        metrics.num.samples = 2
        metrics.recording.level = INFO
        metrics.sample.window.ms = 30000
        partitioner.adaptive.partitioning.enable = true
        partitioner.availability.timeout.ms = 0
        partitioner.class = null
        partitioner.ignore.keys = false
        receive.buffer.bytes = 32768
        reconnect.backoff.max.ms = 1000
        reconnect.backoff.ms = 50
        request.timeout.ms = 30000
```

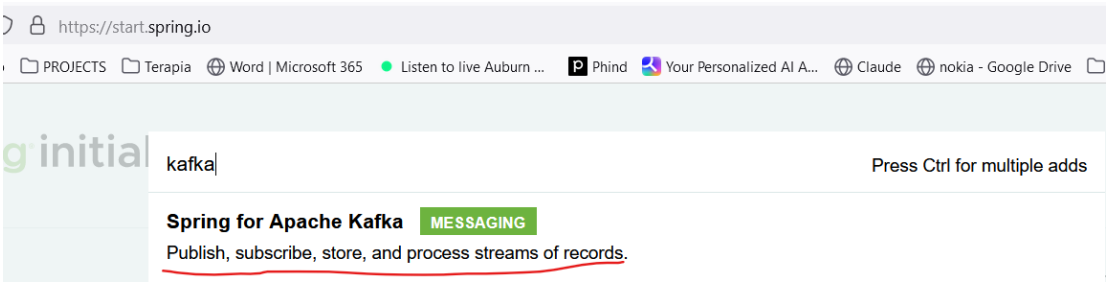
```
        retries = 2147483647
        retry.backoff.ms = 100
        sasl.client.callback.handler.class = null
        sasl.jaas.config = null
        sasl.kerberos.kinit.cmd = /usr/bin/kinit
        sasl.kerberos.min.time.before.relogin = 60000
        sasl.kerberos.service.name = null
        sasl.kerberos.ticket.renew.jitter = 0.05
        sasl.kerberos.ticket.renew.window.factor = 0.8
        sasl.login.callback.handler.class = null
        sasl.login.class = null
        sasl.login.connect.timeout.ms = null
        sasl.login.read.timeout.ms = null
        sasl.login.refresh.buffer.seconds = 300
        sasl.login.refresh.min.period.seconds = 60
        sasl.login.refresh.window.factor = 0.8
        sasl.login.refresh.window.jitter = 0.05
        sasl.login.retry.backoff.max.ms = 10000
        sasl.login.retry.backoff.ms = 100
        sasl.mechanism = GSSAPI
        sasl.oauthbearer.clock.skew.seconds = 30
        sasl.oauthbearer.expected.audience = null
        sasl.oauthbearer.expected.issuer = null
        sasl.oauthbearer.jwks.endpoint.refresh.ms = 3600000
        sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms = 10000
        sasl.oauthbearer.jwks.endpoint.retry.backoff.ms = 100
        sasl.oauthbearer.jwks.endpoint.url = null
        sasl.oauthbearer.scope.claim.name = scope
        sasl.oauthbearer.sub.claim.name = sub
        sasl.oauthbearer.token.endpoint.url = null
        security.protocol = PLAINTEXT
        security.providers = null
        send.buffer.bytes = 131072
        socket.connection.setup.timeout.max.ms = 30000
        socket.connection.setup.timeout.ms = 10000
        ssl.cipher.suites = null
        ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
        ssl.endpoint.identification.algorithm = https
        ssl.engine.factory.class = null
        ssl.key.password = null
        ssl.keymanager.algorithm = SunX509
        ssl.keystore.certificate.chain = null
        ssl.keystore.key = null
        ssl.keystore.location = null
        ssl.keystore.password = null
        ssl.keystore.type = JKS
        ssl.protocol = TLSv1.3
        ssl.provider = null
        ssl.secure.random.implementation = null
        ssl.trustmanager.algorithm = PKIX
        ssl.truststore.certificates = null
        ssl.truststore.location = null
        ssl.truststore.password = null
        ssl.truststore.type = JKS
        transaction.timeout.ms = 60000
        transactional.id = null
        value.serializer = class org.apache.kafka.common.serialization.StringSerializer
```

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.Collections;
```

```java
import java.util.Properties;

public class ConsumerExample {

    private static final Logger logger = LoggerFactory.getLogger(ConsumerExample.class);

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");
//        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "your.group.id");
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "svilen");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        Consumer<String, String> consumer = new KafkaConsumer<>(properties);
        consumer.subscribe(Collections.singletonList("bgjug"));

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

            records.forEach( record -> {
                logger.info("Consumed message - Topic: {}, Partition: {}, Offset: {}, Key: {},
Value: {}",
                        record.topic(), record.partition(), record.offset(), record.key(),
record.value());
            });
        }

    }
}
```



Demo with Spring




```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.kafka:spring-kafka'
```

```
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.kafka:spring-kafka-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

https://spring.io/projects/spring-kafka
https://spring.io/projects/spring-kafka#learn
https://docs.spring.io/spring-kafka/reference/index.html

https://docs.spring.io/spring-kafka/reference/kafka/receiving-messages/listener-annotation.html
https://docs.spring.io/spring-kafka/reference/kafka/sending-messages.html

Не е добре на production Кафка сама да си прави топиците – тази опция трябва да бъде настроена подходящо!!!

2024-08-12T13:36:36.054+03:00 INFO 9752 --- [live-demo] [main] o.a.k.clients.consumer.ConsumerConfig :
ConsumerConfig values:
allow.auto.create.topics = true
auto.commit.interval.ms = 5000
auto.include.jmx.reporter = true
auto.offset.reset = earliest
bootstrap.servers = [localhost:29092]
check.crcs = true
client.dns.lookup = use_all_dns_ips
client.id = consumer-application-3
client.rack =
connections.max.idle.ms = 540000
default.api.timeout.ms = 60000
enable.auto.commit = false
exclude.internal.topics = true
fetch.max.bytes = 52428800
fetch.max.wait.ms = 500
fetch.min.bytes = 1
group.id = application
group.instance.id = null
heartbeat.interval.ms = 3000
interceptor.classes = []
internal.leave.group.on.close = true
internal.throw.on.fetch.stable.offset.unsupported = false
isolation.level = read_uncommitted
key.deserializer = class org.apache.kafka.common.serialization.StringDeserializer
max.partition.fetch.bytes = 1048576
max.poll.interval.ms = 300000
max.poll.records = 500
metadata.max.age.ms = 300000
metric.reporters = []
metrics.num.samples = 2
metrics.recording.level = INFO
metrics.sample.window.ms = 30000
partition.assignment.strategy = [class org.apache.kafka.clients.consumer.RangeAssignor, class org.apache.kafka.clients.consumer.CooperativeStickyAssignor]
receive.buffer.bytes = 65536
reconnect.backoff.max.ms = 1000

reconnect.backoff.ms = 50
request.timeout.ms = 30000
retry.backoff.ms = 100
sasl.client.callback.handler.class = null
sasl.jaas.config = null
sasl.kerberos.kinit.cmd = /usr/bin/kinit
sasl.kerberos.min.time.before.relogin = 60000
sasl.kerberos.service.name = null
sasl.kerberos.ticket.renew.jitter = 0.05
sasl.kerberos.ticket.renew.window.factor = 0.8
sasl.login.callback.handler.class = null
sasl.login.class = null
sasl.login.connect.timeout.ms = null
sasl.login.read.timeout.ms = null
sasl.login.refresh.buffer.seconds = 300
sasl.login.refresh.min.period.seconds = 60
sasl.login.refresh.window.factor = 0.8
sasl.login.refresh.window.jitter = 0.05
sasl.login.retry.backoff.max.ms = 10000
sasl.login.retry.backoff.ms = 100
sasl.mechanism = GSSAPI
sasl.oauthbearer.clock.skew.seconds = 30
sasl.oauthbearer.expected.audience = null
sasl.oauthbearer.expected.issuer = null
sasl.oauthbearer.jwks.endpoint.refresh.ms = 3600000
sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms = 10000
sasl.oauthbearer.jwks.endpoint.retry.backoff.ms = 100
sasl.oauthbearer.jwks.endpoint.url = null
sasl.oauthbearer.scope.claim.name = scope
sasl.oauthbearer.sub.claim.name = sub
sasl.oauthbearer.token.endpoint.url = null
security.protocol = PLAINTEXT
security.providers = null
send.buffer.bytes = 131072
session.timeout.ms = 45000
socket.connection.setup.timeout.max.ms = 30000
socket.connection.setup.timeout.ms = 10000
ssl.cipher.suites = null
ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
ssl.endpoint.identification.algorithm = https
ssl.engine.factory.class = null
ssl.key.password = null
ssl.keymanager.algorithm = SunX509
ssl.keystore.certificate.chain = null
ssl.keystore.key = null
ssl.keystore.location = null
ssl.keystore.password = null
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX
ssl.truststore.certificates = null

ssl.truststore.location = null
ssl.truststore.password = null
ssl.truststore.type = JKS
value.deserializer = class org.apache.kafka.common.serialization.StringDeserializer

```
import lombok.RequiredArgsConstructor;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.kafka.config.KafkaListenerEndpointRegistry;
import org.springframework.kafka.core.KafkaTemplate;

@SpringBootApplication
@RequiredArgsConstructor
public class LiveDemoApplication implements CommandLineRunner {

    private final KafkaTemplate<String, String> kafkaTemplate; //injecting it

    private final KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry; //injecting it

    public static void main(String[] args) {
        SpringApplication.run(LiveDemoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
//      for (int i = 0; i < 25_000_000; i++) {
//          kafkaTemplate.send("welcometokafka", "When is next Java beer in Plovdiv?");
//      }
        kafkaListenerEndpointRegistry.getListenerContainer("bgjug").start(); //How to dynamically
start/stop Kafka Listener
    }
}

--------------------

import lombok.extern.slf4j.Slf4j;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Slf4j
@Component
public class KafkaConsumer {

//    @KafkaListener(topics = "welcometokafka", groupId = "application", concurrency = "3")
//    @KafkaListener(topics = "welcometokafka", groupId = "application")
    @KafkaListener(id = "bgjug", topics = "welcometokafka", groupId = "application", autoStartup =
"false")
    public void listen(String data) {
        log.info(data);
    }
}

------------------------

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

import java.util.HashMap;
import java.util.Map;
```

```
@Configuration
public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest"); //when app goes down and
then restart

        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 1); //
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100_000); //

        return props;
    }

    @Bean
    ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory(ConsumerFactory<String, String> consumerFactory) {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);

        return factory;
    }
}
```

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaProducerConfig {
    @Bean
    public KafkaTemplate<String, String> kafkaTemplate(ProducerFactory<String, String>
producerFactory) { //Injecting the producer factory
        return new KafkaTemplate<String, String>(producerFactory);
    }

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    @Bean
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip");
        // See https://kafka.apache.org/documentation/#producerconfigs for more properties
```

```
        return props;
    }
}
```

Demos from Trayan Iliev
https://github.com/iproduct/kafka-streams-javaland/
https://softuni.bg/trainings/resources/video/90522/video-2-28-10-2023-trayan-iliev-softuni-java-land-vol-2/4401

Consumer with kafka-clients

```
dependencies {
    implementation 'org.apache.kafka:kafka_2.12:3.6.0'
    implementation 'org.apache.kafka:kafka-clients:3.6.0'
}

import lombok.extern.slf4j.Slf4j;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;

import java.time.Duration;
import java.util.Collections;
import java.util.List;
import java.util.Properties;

@Slf4j
public class DemoConsumer {
    private Properties props = new Properties();
    KafkaConsumer<String, String> consumer;

    public DemoConsumer() {
        props.setProperty("bootstrap.servers", "localhost:9093");
        props.setProperty("group.id", "dmlConsumer");
        props.setProperty("enable.auto.commit", "true");
        props.setProperty("auto.commit.interval.ms", "1000");
        props.setProperty("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        consumer = new KafkaConsumer<>(props);

    }

    public void run() {
        // kafka автоматично разпределя
        consumer.subscribe(Collections.singletonList("events"));

        // указваме топика и задаваме само partition 0 започвайки от офсет 0.
        TopicPartition partition = new TopicPartition("events", partition 0);
        consumer.assign(List.of(partition));
        consumer.seek(partition, offset 0);

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
                if (records.count() > 0) {
                    for (ConsumerRecord<String, String> record : records) {
System.out.printf("offset = %d, key = %s, value = %s, headers = %s, topic = %s, partition = %d%n",
record.offset(), record.key(), record.value(), record.headers(), record.topic(),record.partition());
                    }
                }
            }
        } finally {
            consumer.close();
        }
```

```
    }

    public static void main(String[] args) {
        DemoConsumer consumer = new DemoConsumer();
        consumer.run();
    }
}
```

Producer with kafka-clients

```
import lombok.extern.slf4j.Slf4j;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

@Slf4j
public class DemoProducer {
    private Properties props = new Properties();
    private Producer<String, String> producer;

    public DemoProducer() {
       props.setProperty("bootstrap.servers", "localhost:9093");
       props.setProperty("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
       props.setProperty("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        producer = new KafkaProducer<>(props);
    }

    public void run() {
        for(int i = 0; i < 10; i++){
            ProducerRecord<String, String> record =
                    new ProducerRecord<>("events", "" + i, "IoT EVENT " + i);
//            producer.send(record);  //synchronous

            //asynchronous with lambda
            producer.send(record, (recordMetadata, exception) -> {
                System.out.println(">>>" +
                        String.format("Topic %s, Partition: %d Offset: %d, Timestamp: %d\n",
                                recordMetadata.topic(),
                                recordMetadata.partition(),
                                recordMetadata.offset(),
                                recordMetadata.timestamp()
                        )
                );
                if (exception != null) {
                    log.error("Consumer eror: ", exception);
                }
            });
        }
    }

    public static void main(String[] args) throws InterruptedException {
        DemoProducer producer = new DemoProducer();
        producer.run();
        Thread.sleep(5000);
    }
}
```

Some Kafka console commands

In docs/kafka-commands.txt:

kafka-topics.bat --list --bootstrap-server localhost:9093
kafka-topics.bat --create --bootstrap-server localhost:9093 --replication-factor 1 --partitions 1 --topic my-new-topic
kafka-topics.bat --create --bootstrap-server localhost:9093 --replication-factor 1 --partitions 1 --topic events

kafka-topics.bat --list --bootstrap-server localhost:9093
kafka-topics.bat --describe --bootstrap-server localhost:9093 --topic sweepDistances
kafka-console-producer.bat --broker-list localhost:9093 --topic sweepDistances
kafka-console-producer.bat --broker-list localhost:9093 --topic sweepDistances –sync
kafka-console-consumer.bat --bootstrap-server localhost:9093 --topic sweepDistances --from-beginning
kafka-topics.bat --describe --bootstrap-server localhost:9093 --topic temperature

kafka-consumer-groups --bootstrap-server localhost:9092 -list
kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group event-consumer

Replicated Topic Using Kafka

copy config\server.properties config\server-1.properties

copy config\server.properties config\server-2.properties

config/server-1.properties:
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=D:\\CourseKafka\\kafka_2.12-2.2.1\\kafka-logs-1

config/server-2.properties:
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=D:\\CourseKafka\\kafka_2.12-2.2.1\\kafka-logs-2

bin\windows\kafka-server-start config\server-1.properties

bin\windows\kafka-server-start config\server-2.properties

kafka-topics --describe --bootstrap-server localhost:9092 --topic my-replicated-topic

wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid

taskkill /F /PID pid_number

Some Spring application.properties

```
spring.kafka.bootstrap-servers=localhost:9093
spring.kafka.producer.batch-size=1
spring.kafka.producer.properties.linger.ms=0
spring.kafka.streams.properties.commit.interval.ms=0
spring.kafka.streams.application-id=kafka-streams-robot-demo
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

Quarkus integration

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-reactive-messaging-kafka</artifactId>
</dependency>
```

---

application.properties

```
########################
## Logging configuration
########################
quarkus.log.category."io.zerodt.service.KafkaEventLogProducer".level=INFO
%dev.quarkus.log.category."io.zerodt.service.KafkaEventLogProducer".level=DEBUG
%dev.quarkus.log.category."org.apache.kafka.clients".level=ERROR



########################
## Kafka configuration
########################
%dev.kafka.bootstrap.servers=localhost:9092
%prod.kafka.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required username="${KAFKA_USER}" password="${KAFKA_PASSWORD}";

mp.messaging.incoming.analytics-in.connector=smallrye-kafka
mp.messaging.incoming.analytics-in.topic=analytics
mp.messaging.incoming.analytics-
in.key.serializer=org.apache.kafka.common.serialization.StringSerializer
mp.messaging.incoming.analytics-
in.value.serializer=org.apache.kafka.common.serialization.StringSerializer
mp.messaging.incoming.analytics-in.group.id=wallet-service

mp.messaging.outgoing.analytics.connector=smallrye-kafka
mp.messaging.outgoing.analytics.topic=analytics
mp.messaging.outgoing.analytics.key.serializer=org.apache.kafka.common.serialization.
StringSerializer
mp.messaging.outgoing.analytics.value.serializer=org.apache.kafka.common.serializatio
n.StringSerializer
```

```
import io.smallrye.reactive.messaging.annotations.Blocking;
import io.smallrye.reactive.messaging.kafka.KafkaRecord;
import io.zerodt.bonus.dto.CancelBonusDTO;
import io.zerodt.dto.AccountStatusChangedDTO;
import io.zerodt.dto.UserProfileDTO;
import io.zerodt.exception.BadDataException;
import io.zerodt.service.WalletService;
import io.zerodt.util.ExcludeCodeCoverageGenerated;
import io.zerodt.util.JsonUtils;
import org.apache.kafka.common.header.Header;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.jboss.logging.Logger;
import org.jboss.logging.MDC;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.CompletionStage;

@ApplicationScoped
public class KafkaEventListener {

    private static final Logger LOGGER = Logger.getLogger(KafkaEventListener.class);

    @Inject
    WalletService walletService;


    @Incoming("analytics-in")
    @Blocking
    @Transactional
    public CompletionStage<Void> updateProfileStatus(KafkaRecord<String, String> message) {
```

```
        final String eventType = getHeaderAsString(message, "eventType");
        if (!"ACCOUNT_STATUS_CHANGED".equals(eventType)) {
            LOGGER.debugf("Received different type of event - %s. Skipping", eventType);
            return message.ack();
        }

        LOGGER.infof("Received new user status changed event. Updating wallet status");
        final AccountStatusChangedDTO statusChange =
JsonUtils.readJsonFromString(message.getPayload(), AccountStatusChangedDTO.class);
        if (statusChange == null) {
            LOGGER.warnf("Received invalid profile JSON: %s", message.getPayload());
            return message.ack();
        }

        MDC.put("profileId", statusChange.playerId());
        try {
            walletService.updateProfileWalletsStatus(statusChange.playerId(),
statusChange.newStatus(), statusChange.previousStatus());
        } finally {
            MDC.clear();
        }
        return message.ack();
    }
}
```

```
import io.smallrye.reactive.messaging.kafka.OutgoingKafkaRecord;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.eclipse.microprofile.reactive.messaging.Emitter;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

@ApplicationScoped
public class KafkaEventLogEmitter {

    @Inject
    @Channel("analytics")
    Emitter<String> analyticsEmitter;

    public void emmitAnalyticsEvent(OutgoingKafkaRecord<String, String> message) {
        analyticsEmitter.send(message);
    }
}
```

```
public class OutgoingKafkaRecord<K, T> implements KafkaRecord<K, T>


public interface KafkaRecord<K, T> extends Message<T>, ContextAwareMessage<T> {

    static <K, T> OutgoingKafkaRecord<K, T> from(Message<T> message) {
        return OutgoingKafkaRecord.from(message);
    }

    /**
     * Creates a new outgoing Kafka record.
     *
     * @param key the key, can be {@code null}
     * @param value the value / payload, must not be {@code null}
     * @param <K> the type of the key
     * @param <T> the type of the value
     * @return the new outgoing Kafka record
     */
    static <K, T> OutgoingKafkaRecord<K, T> of(K key, T value)
```

```
import io.smallrye.reactive.messaging.kafka.KafkaRecord;
import io.smallrye.reactive.messaging.kafka.OutgoingKafkaRecord;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;

import jakarta.inject.Inject;
import static jakarta.enterprise.event.TransactionPhase.AFTER_COMPLETION;


@ApplicationScoped
@ExcludeCodeCoverageGenerated
public class KafkaEventLogProducer {

    private static final Logger LOGGER = Logger.getLogger(KafkaEventLogProducer.class);

    @Inject
    KafkaEventLogEmitter emitter;

    public void publishLoyaltyLevelChangeEvent(@Observes(during = AFTER_COMPLETION)
LoyaltyStatusChangeDTO loyaltyStatusChange) {
        Long profileId = loyaltyStatusChange.profileId();

        final String loyaltyStatusChangeMessage = JsonUtils.toJsonString(loyaltyStatusChange);
        final var loyaltyStatusChangeRecord = KafkaRecord.of(profileId.toString(),
loyaltyStatusChangeMessage)
                .withHeader("id", UUID.randomUUID().toString())
                .withHeader("eventType", "LOYALTY_STATUS_CHANGE");

        LOGGER.debugf("Publishing the following message to analytics topic: %s",
loyaltyStatusChangeRecord);
        emitter.emmitAnalyticsEvent(loyaltyStatusChangeRecord);
    }
```

Kafka on Windows 10

Kafka is a message broker exchanging messages in two or more external microservices

https://kafka.apache.org/quickstart
https://www.youtube.com/watch?v=aKDWWICgfA0

STEP 1: GET KAFKA

server.properties файла log.dirs=/tmp/kafka-logs
запиши в основната директория на kafka, в случая отбелязано с /tmp
C:/SYSTEM/kafka_2.13-3.2.1/kafka-logs

zookeeper.properties dataDir= /tmp/zookeeper-data запиши в основната директория на kafka, в случая отбелязано с /tmp
C:/SYSTEM/kafka_2.13-3.2.1/zookeeper-data

STEP 2: START THE KAFKA ENVIRONMENT

Отваряме cmd в директория kafka_2.13-3.2.1
.\ е текущата директория
C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-server-start.bat .\config\server.properties

STEP 3: CREATE A TOPIC TO STORE YOUR EVENTS

C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-topics.bat --create --topic mynewprofessiontopic --bootstrap-server localhost:9092

Created topic mynewprofessiontopic.

C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-topics.bat --describe --topic mynewprofessiontopic --bootstrap-server localhost:9092

Topic: mynewprofessiontopic TopicId: qwAyRYs3Tj2gBW3BKlNJSA PartitionCount: 1 ReplicationFactor: 1 Configs: segment.bytes=1073741824
Topic: mynewprofessiontopic Partition: 0 Leader: 0 Replicas: 0 Isr: 0

STEP 4: WRITE SOME EVENTS INTO THE TOPIC

C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-console-producer.bat --topic mynewprofessiontopic --bootstrap-server localhost:9092

>Hello world
>My first topic event
>demo1
>demo2
>
>Terminate batch job (Y/N)? Y

STEP 5: READ THE EVENTS

C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-console-consumer.bat --topic mynewprofessiontopic --from-beginning --bootstrap-server localhost:9092

Hello world
My first topic event
demo1
demo2

Processed a total of 2 messages

C:\SYSTEM\kafka_2.13-3.2.1>.\bin\windows\kafka-console-consumer.bat --topic mynewprofessiontopic --from-beginning --bootstrap-server localhost:9092
Hello world
My first topic event
demo1
demo2

STEP 6: TRACE WHEN A NEW TOPIC/MESSAGE APPEAR in the consumer console

How to see all current topics
.\bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --describe
.\bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --list

How to see/read the events of each topic (i.e. the messages of each topics)
.\bin\windows\kafka-console-consumer.bat --topic po-write --from-beginning --bootstrap-server localhost:9092

Значи създаваме си топик, в него се записват event-и/ message-и - чрез producer. А когато искаме да ги прочетем/достъпим тези message-и, то използваме consumer.

STEP 7: - Run Kafka Commands inside the Docker container

Starting the KAFKA from Docker-compose.yaml file
От текущата директория

```
docker-compose up -d
```

docker-compose.yaml

```
version: '2'

services:

 zookeeper:
   image: quay.io/strimzi/kafka:0.29.0-kafka-3.1.1
   command: [
     "sh", "-c",
     "bin/zookeeper-server-start.sh config/zookeeper.properties"
   ]
   ports:
     - "2181:2181"
   environment:
     LOG_DIR: /tmp/logs

 kafka:
   image: quay.io/strimzi/kafka:0.29.0-kafka-3.1.1
   command: [
     "sh", "-c",
     "bin/kafka-server-start.sh config/server.properties --override
listeners=$${KAFKA_LISTENERS} --override
advertised.listeners=$${KAFKA_ADVERTISED_LISTENERS} --override
zookeeper.connect=$${KAFKA_ZOOKEEPER_CONNECT}"
   ]
   depends_on:
     - zookeeper
   ports:
     - "9092:9092"
   environment:
     LOG_DIR: "/tmp/logs"
     KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
     KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
     KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

docker ps
да видим контейнерите, които вървят в момента

```
PS C:\SVILEN\QUARKUS\QuarkusForSpringDevelopers\project> docker ps
CONTAINER ID   IMAGE                                  COMMAND                  CREATED             STATUS             PORTS                    NAMES
fc33cf454631   quay.io/strimzi/kafka:0.29.0-kafka-3.1.1   "sh -c 'bin/kafka-se…"   About a minute ago   Up About a minute   0.0.0.0:9092->9092/tcp   chapter-5-quarkus-kafka-streams_kafka_1
fb32aba53bf0   quay.io/strimzi/kafka:0.29.0-kafka-3.1.1   "sh -c 'bin/zookeepe…"   About a minute ago   Up About a minute   0.0.0.0:2181->2181/tcp   chapter-5-quarkus-kafka-streams_zookeeper_1
```

docker exec -it <kafka_conatiner_id> sh
docker exec -it fc33cf454631 sh
Now we are inside our Docker container.

cd /opt/kafka_<version>
cd /opt/kafka/bin

We are moving to the folder where all the executives are located.

Дебъгване на kafka
./bin/kafka-topics.sh --create --topic quickstart --bootstrap-server localhost:9092

./bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
./kafka-topics.sh --bootstrap-server localhost:9092 --list

./bin/windows/kafka-topics.bat --bootstrap-server localhost:9092 --list това не работи съвсем

Kafka streams

For many more info – see the *KafkaStreams_JavaLand.pdf* file.

```
dependencies {
implementation 'org.apache.kafka:kafka-clients:3.6.0'
implementation 'org.apache.kafka:kafka-streams:3.6.0‘
…
}
```

```
public static void main(String[] args) {
// Use the builders to define the actual processing topology, e.g. to specify from which input topics to read, // which stream operations (filter, map, etc.) should be called, and so on.

StreamsBuilderbuilder = ...;  // when using the DSL

Topology topology= builder.build();
//
// OR
//

Topology topology= ...; // when using the Processor API

// Use the configuration to tell your application where the Kafka cluster is,// which Serializers/Deserializers to use by default, to specify security settings, and so on.
Properties props = ...;KafkaStreamsstreams = new KafkaStreams(topology, props);


// Add shutdown hook to stop the Kafka Streams threads. You can optionally provide a timeout to`close`.
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

Conclusion

Всеки път/за всяка задача трябват да настройваме ръчно Kafka producer и kafka consumer настройките! Никога да не разчитаме на дефаултните конфигурации!

8. UBUNTU as WSL

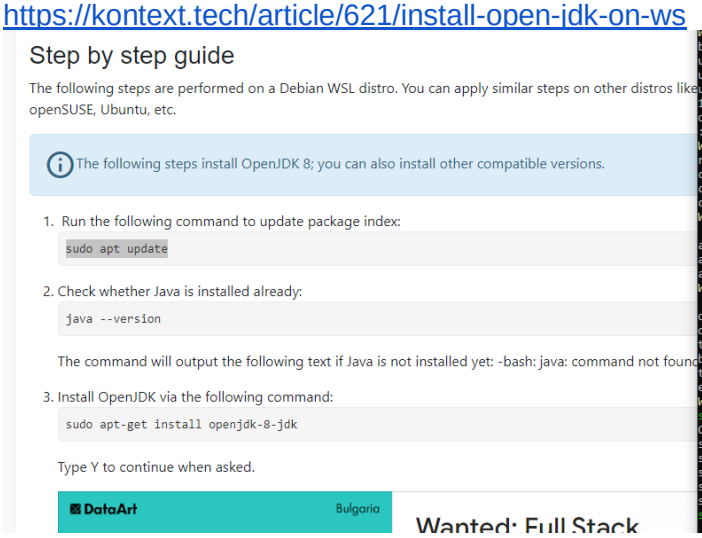
Installing Ubuntu from Microsoft store
https://apps.microsoft.com/store/detail/ubuntu-22041-lts/9PN20MSR04DW?hl=en-us&gl=us

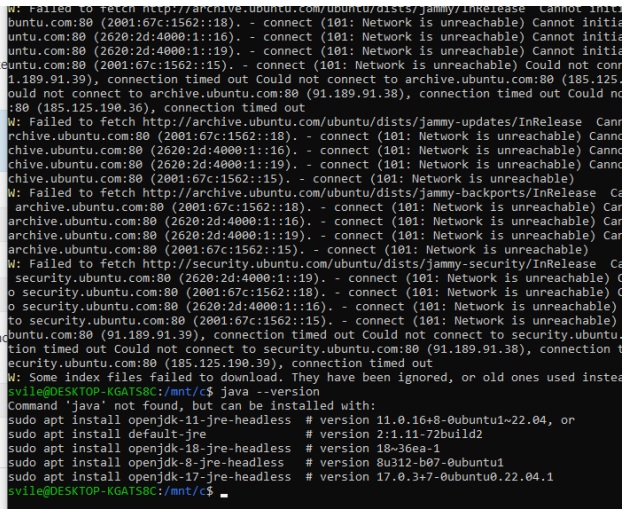
Отиди на C:\ директорията

cd /mnt/c

Installing Java on Ubuntu/WSL

https://kontext.tech/article/621/install-open-jdk-on-ws





sudo apt install openjdk-11-jre-headless # version 11.0.16+8-0ubuntu1~22.04, or
sudo apt install default-jre # version 2:1.11-72build2
sudo apt install openjdk-18-jre-headless # version 18~36ea-1
sudo apt install openjdk-8-jre-headless # version 8u312-b07-0ubuntu1
sudo apt install openjdk-17-jre-headless # version 17.0.3+7-0ubuntu0.22.04.1

/usr/lib/jvm/java-11-openjdk-amd64/bin/

Configuring JAVA_HOME

Configure JAVA_HOME

Configure **JAVA_HOME** environment variable is optional but I highly recommend it especially if you want to install Hadoop, Spark, Hive, HBase or any other Java based frameworks.

To do this, we need to find out where JDK is installed:

```
$ readlink -f $(which java)
/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
```

As printed out, JDK is installed in folder **/usr/lib/jvm/java-8-openjdk-amd64**.
Run the following command to edit .bashrc file:

```
vi ~/.bashrc
```

Add the following line to the content:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Save the file and then run the following command:

```
source ~/.bashrc
```

Run the following command to make sure it is effective:

```
~$ echo $JAVA_HOME
/usr/lib/jvm/java-8-openjdk-amd64
```

Insert

Cancel

Basic Linux commands
https://gitlab.com/vv-b-s/operating-systems/-/blob/master/docs/7-LinuxIntroductionBaseCommands-en.adoc

https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners

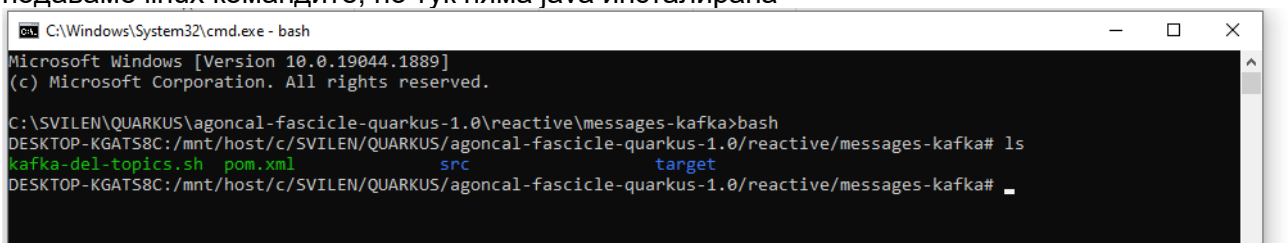
Type :wq to exit
: denotes that you are going to run a command
w means you want to write the file
q means you want to quit the editor

Другият вариант за изход е Shift + ZZ

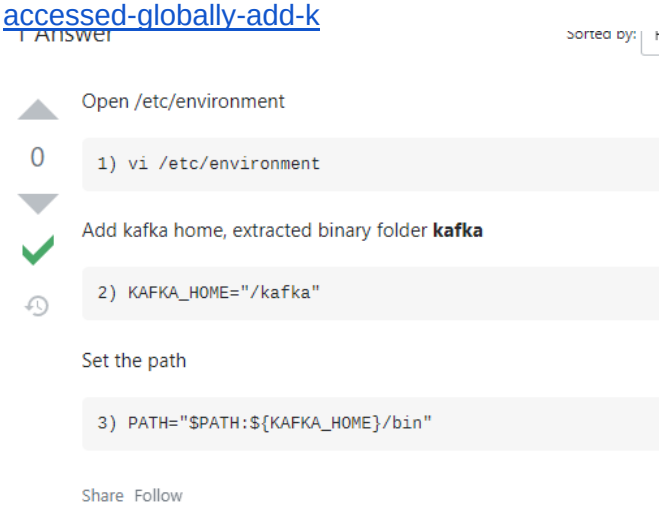
Installing Kafka on Ubuntu/WSL

Следвам стъпките от Apache Kafka, като трябва да ползвам и **sudo** ("SuperUser Do") в началото на командата

От която и да е директория, през cmd-то като достъпим, и след като напишем bash и то можем да подаваме linux командите, но тук няма java инсталирана



https://stackoverflow.com/questions/63006982/add-kafka-path-to-etc-environment-file-for-kafka-to-be-accessed-globally-add-k



Executing a shell script in Linux
https://www.cyberciti.biz/faq/how-to-execute-a-shell-script-in-linux/

1. Create a new file called demo.sh using a text editor such as nano or vi in Linux:

```
nano demo.sh
```

2. Add the following code:

```
#!/bin/bash
echo "Hello World"
```

3. Set the script executable permission by running chmod command in Linux:

```
chmod +x demo.sh
```

4. Execute a shell script in Linux:

```
./demo.sh
```

Let us see all steps in details.

kafka-del-topics.sh

```
#!/bin/bash
echo "Hello $USER"
echo "Today is $(date)"
echo "Bye for now"

TOPICS=$(kafka-topics.sh --bootstrap-server localhost:2181 --list )

for T in $TOPICS
do
 if [ "$T" != "__consumer_offsets" ]; then
   kafka-topics.sh --bootstrap-server localhost:2181 --delete --topic $T
 fi
Done
```

kafka-del-topics.sh

```
#!/bin/bash
echo "Hello $USER"
echo "Today is $(date)"
echo "Bye for now"

TOPICS=$(kafka-topics --bootstrap-server localhost:2181 --list )

for T in $TOPICS
do
 if [ "$T" != "__consumer_offsets" ]; then
   kafka-topics --zookeeper-server localhost:2181 --delete --topic $T
 fi
done
```

9. Docker and containerization

I. Containers, Docker, Images

Containerization and images

OS-level virtualization refers to an operating system paradigm in which the kernel allows the existence of **multiple isolated** user **space instances** known as **containers, zones, jails, …**

Работят в изолация /Standalone/.
Не може единия апп да източи паметта на другия апп. Ако единия апп му свърши паметта, то другия апп продължава да работи.
Цял сървър е доста скъпо също.
Затова използваме контейнеризация.

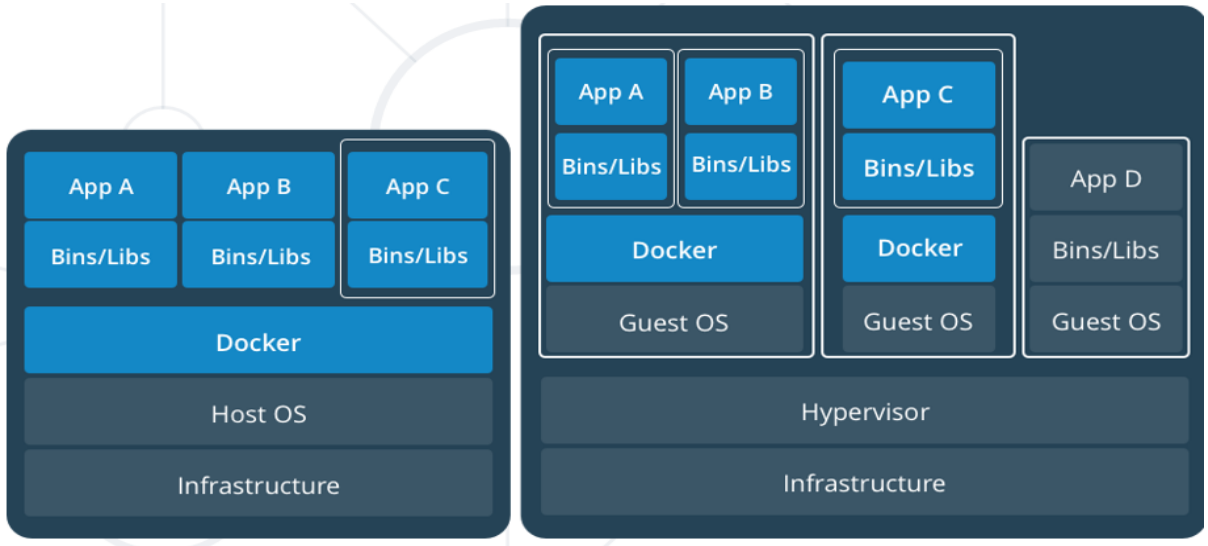
Container:

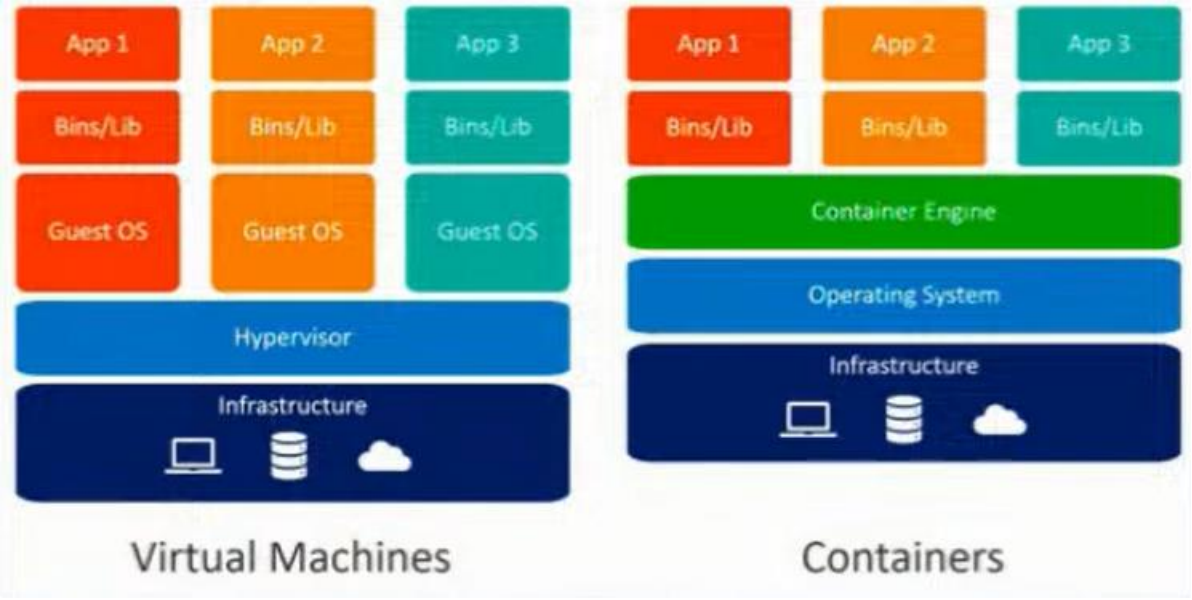
- A sandboxed process that is isolated from all other processes on a host machine
- Should contain only a single application and its dependencies
- **Portable and lightweight**

Image:

- Standalone executable package
- The specification of a container

VMs (виртуални машини) vs Containers





Първоначално е имало само виртуални машини.
Виртуална машина означава на даден хост машина/сървър да има изолирана операционна система, която можем да я ползваме без достъп до някакви други неща.

Идеята, ако има тази хост машина досттъчно ресурси, то да пуснем две незасивими изолирани виртуално операционни системи. Идеята е на всяка виртуална машина да се пуска един application.

Hypervisor – да дистрибутира и менъджира всички виртуални операционни системи и техните ресурси.

Един Docker container е процес, а не цяла виртуална машина! Една хост машина, една операционна система, различни процеси като един от тях е Docker cointainer!

В самият Docker container се казва също че има инсталирана малка операционна система – това всъщност не е съвсем вярно! Има файлова операционна система.

Docker Daemon(Docker engine) менъджира отделните контейнери.

- VMs virtualize the hardware
- Complete isolation
- Complete OS installation. Requires more resources
- Runs almost any OS

- Containers virtualize the OS
- Lightweight isolation
- Shared kernel among all the containers. Requires fewer resources
- Runs on the same OS == Linux

Solutions

- **rkt** by CoreOS
  - Application container engine
  - https://coreos.com/rkt
- **Docker** by Docker Inc
  - Application container engine

- https://www.docker.com/

Containers Concepts (Docker View)

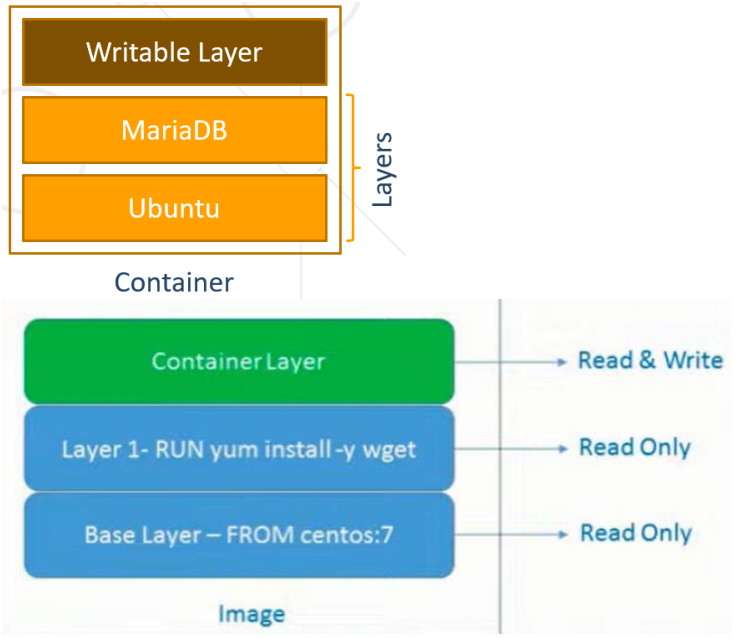
- **Container image** shows the state of a container, including registry or file system changes = все едно **container image** е нещо клас, а самия **контейнер** е нещо като клас. Самият container image е работещ container.
- **Container OS image** is the first layer of potentially many image layers that make up a container = основния image, върху който се гради контейнера
- **Container repository** stores container images and their dependencies = мястото, където се пазят тези container images

Definitions

- Container
  - Containers are processes with much more isolation

- Image
  - Images provide a way for simpler software distribution

Container layers



docker pull nginx

```
❯ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
25d3892798f8: Pull complete
42de7275c085: Pull complete
c459a9332e03: Pull complete
48882f13d668: Pull complete
49180167b771: Pull complete
da4abc2b066c: Pull complete
20dc44ab57ab: Pull complete
Digest: sha256:84c52dfd55c467e12ef85cad6a252c0990564f03c4850799bf41dd738738691f
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Като дръпнем един докер image, той сваля няколко слоя/layers с няколко id-та, като всеки layer е една от следните команди: FROM, MAINTAINER, RUN, COPY, ADD, EXPOSE, ENTRYPOINT, CMD.

Ако имаме едно и също начало на два докер файла за изпичане на image то при pull-ване на 2 различни Image-а, то те ще се преизползват – няма да ги дърпа два пъти! Т.е. всеки един layer се кешира поотделно и представлява определена част от тази lightweight файлова система. Това също означава, че image-те се оптимизират. Всички layer-и се кешират, с изключение на последния Read & Write layer.

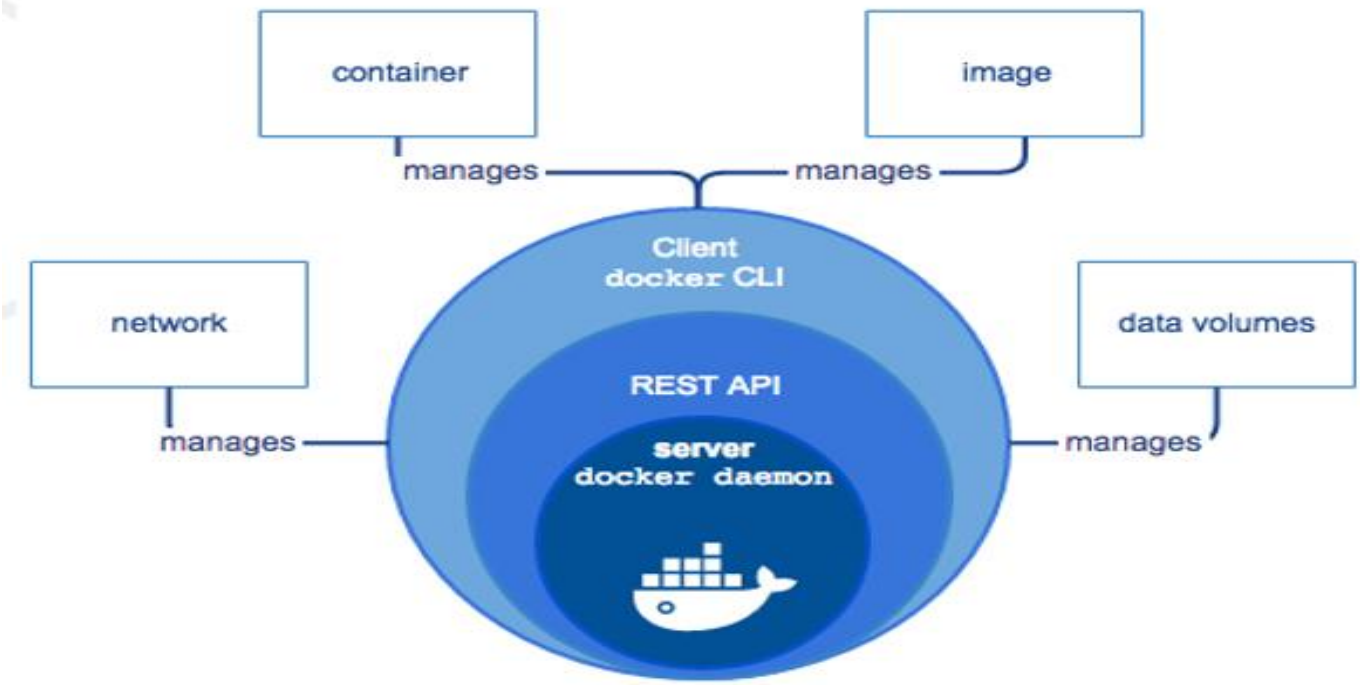
Когато създадем нов контейнер от image, то само тогава се създава и то **наново Read & Write layer-а**.

```
> docker pull httpd
Using default tag: latest
latest: Pulling from library/httpd
25d3892798f8: Already exists
7a5857226826: Pull complete
4f4fb700ef54: Pull complete
ec9858119e90: Pull complete
54e79f99f33b: Pull complete
e6520ad62ebb: Pull complete
Digest: sha256:5ee9ec089bab71ffcb85734e2f7018171bcb2d6707f402779d3f5b28190bb1af
Status: Downloaded newer image for httpd:latest
docker.io/library/httpd:latest
```

II. Docker

Docker Engine

- Docker Mission – **Build**, **Ship**, **Deploy**
- Client-server application
- Components
    - dockerd daemon
    - REST API
    - docker CLI



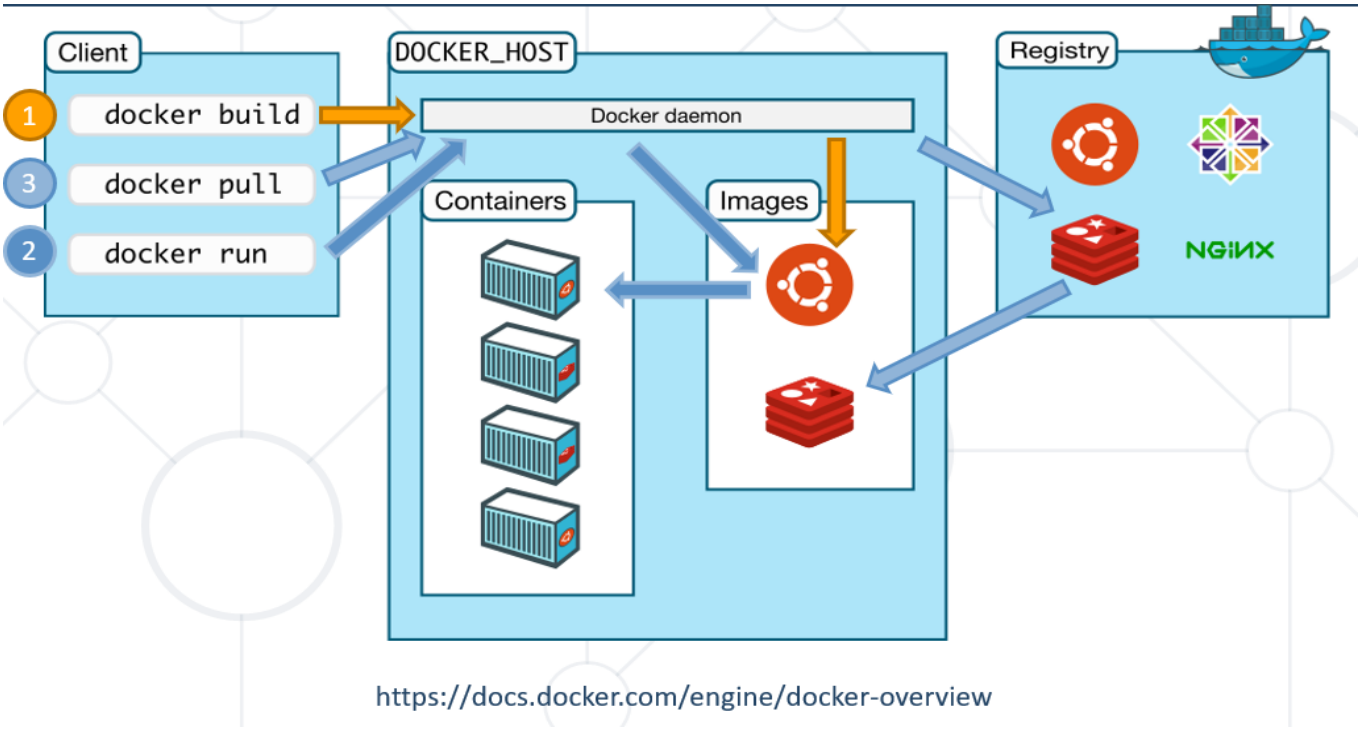
Registries – мястото, където се пазят тези images

- Provided by Docker
    - Cloud
        - Docker Hub (https://hub.docker.com/explore/)
        - Docker Store (https://store.docker.com/)

    - On-premise
- Provided by 3rd parties
    - Quay.io, Artifactory, Google Container Registry

Workflow

docker build – създаване на docker image
docker pull – дърпане на image от регистри
docker run – стартирай ми контейнер
docker push – ако локално ние направим update и искаме нашия image да го дистрибутираме към регистрито.



https://docs.docker.com/engine/docker-overview

Client – Docker CLI (Docker Command Line Interface)
Docker host – това реално е Docker engine-а == Docker daemon
Не е задължително CLI и Docker host да са на една и съща машина. Но в прости дема – това е една и съща машина.

Docker платформата върви само на Линукс. С други думи на Windows и на MacOS пуска виртуална машина на Линукс, за да може да запали Докера.

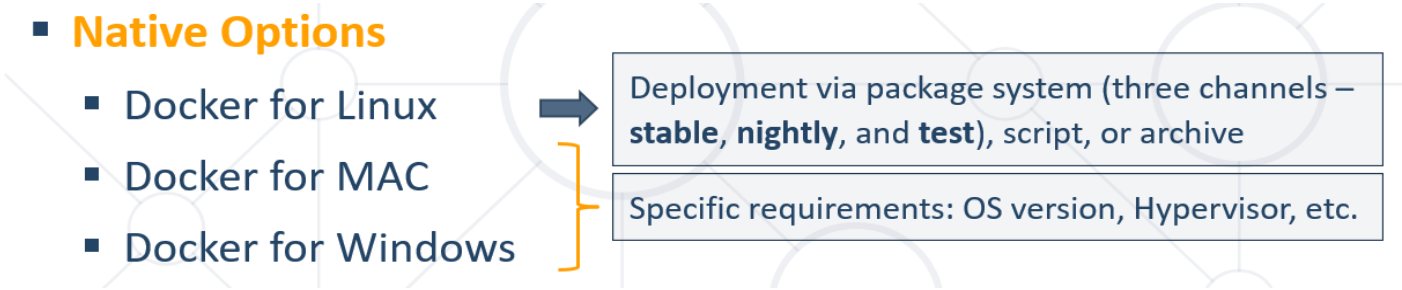
Registry – място където се споделят images – нещо като MVN repository
https://hub.docker.com/

III. Docker Installation

What We Need to Know?

- **Two Editions** (Community and Enterprise)
- **Native Options**
    - Docker for Linux – без Desktop 😊
    - Docker Desktop for MAC

- Docker Desktop for Windows
- **Docker Toolbox** (deprecated) - All-in-one solution
  - For Mac and Windows



IV. Working with Docker

Docker е оптимизиран откъм бързина, но не и откъм дисково пространство

Pull / Image Pull

- Purpose
  - Pull an image or a repository from a registry
- Old syntax

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

- New syntax

docker image pull [OPTIONS] NAME[:TAG|@DIGEST]

- Example

docker image pull ubuntu:latest

Run / Container Run

- Purpose
  - Run a command in a new container
- Old syntax

docker run [OPTIONS] IMAGE [COMMAND] [ARG]

docker run -it ubuntu :latest //-it идва от interactive TTY – да имаме достъп отвън до контейнера отвътре
docker run -it ubuntu :latest /bin/bash стартирай ми контейнера с имидж Ubuntu, и стартирай отвътре bash-а
docker run -it nginx самият nginx може да пуска web приложение
docker run -p 8080:80 nginx **-p** опция за forwarding of traffic (port forwarding)

Лявата страна на порта “8080:8080” е на нашата линукс сървърна машина, а дясната страна на порта е каквото е вътре в пуснатия един или няколко контейнера от image nginx.

docker run -d -p 8080:80 nginx **-d** опция за пускане на процеса в background/long run – като се изпълни процеса, то процеса остава и не се самоизчиства!
docker run -d -p 8081:80 nginx

```
❯ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS                    NAMES
2b171cfb6d1d   nginx     "/docker-entrypoint.…"   17 seconds ago       Up 16 seconds       0.0.0.0:8081->80/tcp     heuristic_wozniak
44f014f68fe6   nginx     "/docker-entrypoint.…"   About a minute ago   Up About a minute   80/tcp                   heuristic_tharp
5bd2b56352b7   nginx     "/docker-entrypoint.…"   3 minutes ago        Up 3 minutes        0.0.0.0:8080->80/tcp     hardcore_torvalds
```

- New syntax

docker container run [OPTIONS] IMAGE [COMMAND] [ARG]

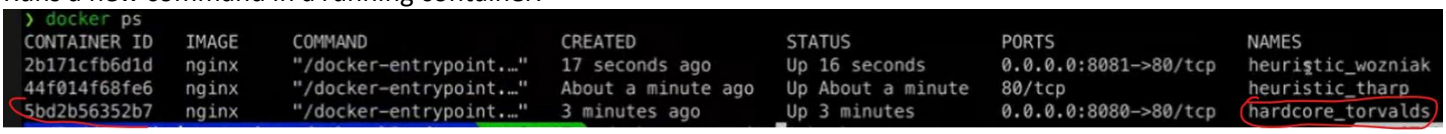
- Example

docker container run -it ubuntu

Exec

Ако имаме няколко контейнера активни, то в нов терминал да си достъпим отвътре някой действащ контейнер примерно.

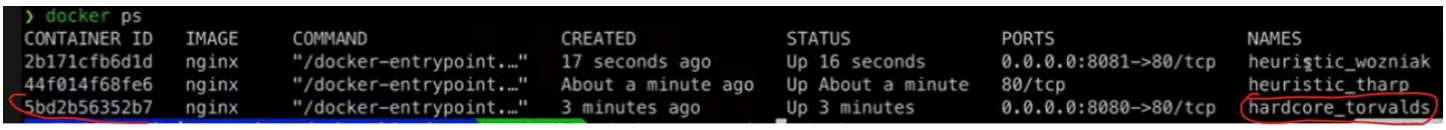
Runs a new command in a running container.



docker exec -it hardcore_torvalds bash

docker exec -it 5bd bash само с част от container id-то стават доста от докер командите

Kill

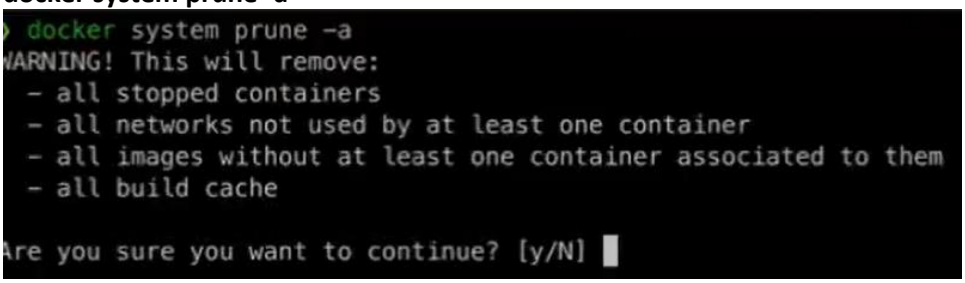


docker kill 5bd само с част от container id-то стават доста от докер командите

Prune

Deletes both containers and images

docker system prune -a



Run container with a specific volume / Bind Mount

docker run -d -p 8080:80 **-v** /tmp/nginx:/user/share/nginx/html nginx

Рънни контейнер от nginx имидж, като копирай всичко от папка /tmp/nginx вътре в контейнера на път /user/share/nginx/html – като **-V** указва, че един вид това е volume. В случая този ред означава **Bind Mount** – директна връзка между външния свят и контейнра вътре, и е по-различно от volume. Volume заема произволно място на хост машината, докато в нашия случай ние сме си направили папка отвън, която да се копира вътре в контейнера.

Images / Image Ls

- Purpose
  - List locally available images

- Old syntax

docker images [OPTIONS] [REPOSITORY[:TAG]]

- New syntax

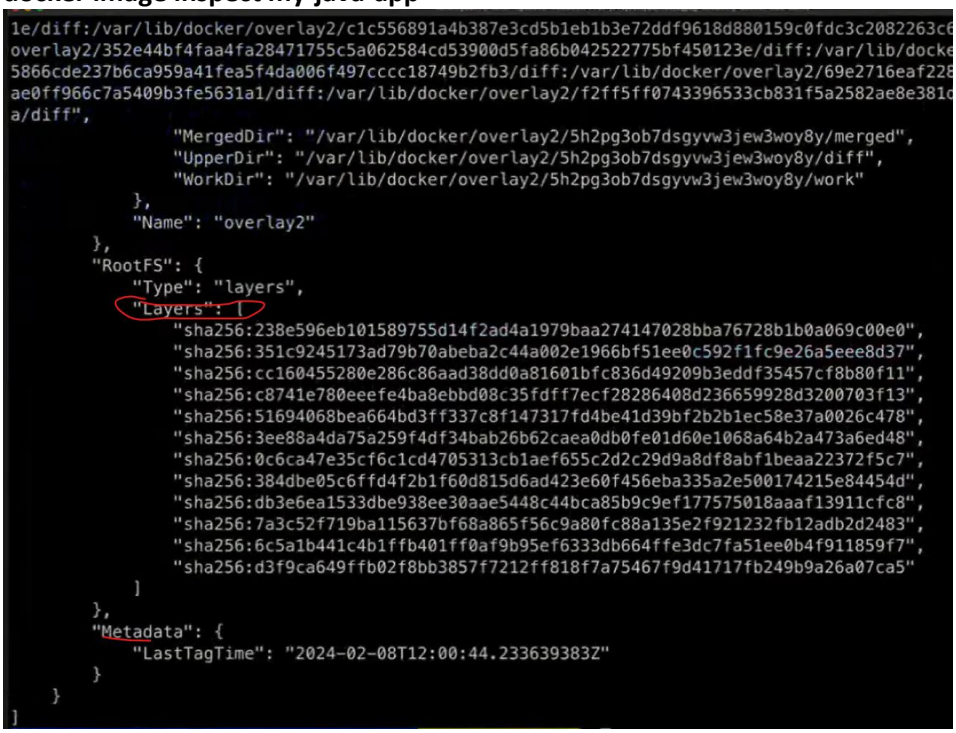
docker image ls [OPTIONS] [REPOSITORY[:TAG]]

- Example

docker image ls fedora

Image inspect

docker image inspect my-java-app



Ps / Container Ls

- Purpose
  - List containers
- Old syntax

docker ps [OPTIONS]

- New syntax

docker container ls [OPTIONS]

- Example

docker container ls –a -q

Rm / Container Rm

- Purpose
  - Remove one or more containers
- Old syntax

docker rm [OPTIONS] CONTAINER [CONTAINER]

- New syntax

docker container rm [OPTIONS] CONTAINER [CONTAINER]

- Example

docker container rm weezy_snake

Rmi / Image Rm

- Purpose
  - Remove one or more images
- Old syntax

docker rmi [OPTIONS] IMAGE [IMAGE]

- New syntax

docker image rm [OPTIONS] IMAGE [IMAGE]

- Example

docker image rm ubuntu fedora

Start / Container Start

- Purpose
  - Start one or more stopped containers
- Old syntax

docker start [OPTIONS] CONTAINER [CONTAINER]

- New syntax

docker container start [OPTIONS] CONTAINER [CONTAINER]

- Example

docker container start -a -i 0cbf27183

Restart / Container Restart

- Purpose
  - Restart a one or more containers
- Old syntax

docker restart [OPTIONS] CONTAINER [CONTAINER]

- New syntax

docker container restart [OPTIONS] CONTAINER [CONTAINER]

- Example

docker container restart 0cbf27183

Stop / Container Stop

- Purpose
  - Stop one or more running containers
- Old syntax

docker stop [OPTIONS] CONTAINER [CONTAINER]

- New syntax

docker container stop [OPTIONS] CONTAINER [CONTAINER]

- Example

docker container stop 0cbf27183

Unpause / Container Unpause

- Purpose
  - Unpause all processes within one or more containers
- Old syntax

docker unpause CONTAINER [CONTAINER]

- New syntax

docker container unpause CONTAINER [CONTAINER]

- Example

docker container unpause 0cbf27183

Attach / Container Attach

- Purpose
  - Attach to a running container
- Old syntax

docker attach [OPTIONS] CONTAINER

- New syntax

docker container attach [OPTIONS] CONTAINER

- Example

docker container attach 0cbf27183

docker -exec ……

Push / Image Push

- Purpose
  - Push an image or repository to a registry
- Old syntax

docker push [OPTIONS] NAME[:TAG]

- New syntax

docker image push [OPTIONS] NAME[:TAG]

- Example

docker image push repo-name/test:latest

Login to Docker registry

- Purpose
  - Log into a Docker registry
- Old syntax

docker login [OPTIONS] [SERVER]

- New syntax

docker login [OPTIONS] [SERVER] като стария синтаксис

- Example

docker login

Logout from Docker registry

- Purpose
  - Log out from a Docker registry
- Old syntax

docker logout [SERVER]

- New syntax

docker logout [SERVER] като стария синтаксис

- Example

docker logout

V. Image from DockerFile – изпичане на docker файл

General Structure (Dockerfile)

- Script, composed of commands and arguments
- Always begins with FROM instruction

 # Set the base image
FROM nginx Comment

 # Set the maintainer
MAINTAINER John Smith Command (Instruction)

 # Copy files
COPY index.html /usr/share/nginx/html/

FROM

- Purpose
  - Defines the base image to use to start the build process

- Syntax

```
FROM <image>[:<tag>] [AS <name>]
```

- Example

```
# it is a good practice to state a version (tag)
FROM ubuntu:18.04

# for the latest version the tag could be skipped
FROM ubuntu
```

MAINTAINER

- Purpose
  - Sets the author field of the image. It is deprecated
- Syntax

```
MAINTAINER <name>
```

- Example

```
# deprecated
MAINTAINER John Smith
# newer variant is this:
LABEL maintainer="John Smith"
```

RUN

- Purpose
  - Used during build process to add software (forms another layer)
- Syntax

```
RUN <command>
```

- Example

```
# single command
RUN apt-get -y update
# more than one command
RUN apt-get -y update && apt-get -y upgrade
```

COPY

- Purpose
  - Copy files between the host and the container
- Syntax

```
COPY [--chown=<user>:<group>] <src>... <dest>
```

- Example

```
# Copy single file
COPY readme.txt /root

# Copy multiple files
COPY *.html /var/www/html/my-web-app
```

ADD

- Purpose
  - Copy files to the image
- Syntax

```
ADD [--chown=<user>:<group>] <src>... <dest>
```

- Example

```
# Add single file from URL
ADD https://softuni.bg/favicon.ico  /www/favicon.ico

# Add tar file content
ADD web-app.tar /var/www/html/my-web-app
```

EXPOSE

- Purpose
  - Informs Docker that the container listens on the specified ports
- Syntax

```
EXPOSE <port> [<port>/<protocol>...]
```

- Example

```
# single port
EXPOSE 80

# multiple ports
EXPOSE 80 8080
```

ENTRYPOINT

- Purpose
  - Allows configuration of container that will run as an executable
  - **Задължителната команда, която ще се изпълни в началото**
- Syntax

```
# exec form, this is the preferred form
ENTRYPOINT ["executable", "param1", "param2"]

# shell form
ENTRYPOINT command param1 param2
```

CMD

- Purpose
  - Main purpose is to provide defaults for an executing container
  - **Дефаултната команда, която ще се изпълни в началото**
- Syntax

```
# exec form, this is the preferred form
CMD ["executable","param1","param2"]

# as default parameters to ENTRYPOINT
CMD ["param1","param2"]

# shell form
```

CMD command param1 param2

CMD vs ENTRYPOINT

- Both **define** what **command** gets **executed** when **running** a container
- **Dockerfile** should specify at **least one** of them
- **ENTRYPOINT** should be defined when using the **container** as an **executable**
- **CMD** should be used as a way of defining **default** **arguments** for an **ENTRYPOINT** command or for **executing** an **ad-hoc** command in a container
- **CMD** will be overridden when **running** the container with **alternative** arguments

- Both have **exec** and shell **form**
- When used **together always** use their **exec** form

Когато се използват заедно или поотделно:
CMD командата може да заема няколко аргумента

		ENTRYPOINT		
		N/A	**exec_entry p1_entry**	**["exec_entry", "p1_entry"]**
CMD	**N/A**	Error	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
	["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	**exec_entry p1_entry exec_cmd p1_cmd**
	["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	**exec_entry p1_entry p1_cmd p2_cmd**
	exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Build / Image Build

- Purpose
  - Build an image from a Dockerfile
- Old syntax

docker build [OPTIONS] PATH | URL | -

- New syntax

docker image build [OPTIONS] PATH | URL | -

- Example - **-t** означава сложи му таг, а точката е текущата директория

docker image build -t new-image .

Ако създаваме локално docker image на M1, M2, M3 MacOS компютър (ARM или ARM64 процесорна архитектура), то би имало проблем този имидж като го качим в регистрито и други потребители си го изтеглят и пуснат контейнер на обикновен Линукс. Затова ако сме на MacOS пускаме следната команда и тогава е доста голям шанса да работи:
docker build buildx –platform linux/amd64 my-java-app .

Доста често в практиката тези docker images се билдват от Jenkins/GitHub actions агенти, и тъй като се билдват (чрез buildAaA.yml) на Линукс сървър, тогава го нямаме този проблем с ARM64! Т.е. преместваме създаването/изпичането на докер image в CI/CD pipeline-а.

Dockerfile example workflow

Example 1:

Dockerfile-build

```
# We start from openjdk17 base image to have JDK installed
FROM eclipse-temurin:17-jdk
WORKDIR /app

COPY build/libs/petclinic-0.0.1-SNAPSHOT.jar app.jar   //копирай jar файла като app.jar вътре в контейнера

CMD [”java”, ”-jar”, ”app.jar”]     //Read & Write layer
```

In the terminal:

docker build -t my-java-app . **-t** означава сложи му таг, а точката е текущата директория

docker run -p -d 8080:8080 **my-java-app**

Example 2:

Dockerfile-build

```
# We start from openjdk17 base image to have JDK installed
FROM eclipse-temurin:17-jdk
WORKDIR /app      //same as RUN mkdir app && cd app

//организираме layers по честота на промяна – т.е. src ще се променя често, тогава измести копиранията за gradle по-горе – това ще ги сложи в кеша за следващ image build ще се изпълнява по-бързо
COPY gradlew .          //копирай gradlew файла вътре в контейнера
COPY gradle gradle/      //копирай съдържанието на gradle папката вътре в контейнера в директория gradle/
COPY build.gradle.kts .
COPY settings.gradle.kts .

COPY src src/            //копирай съдържанието на src папката вътре в контейнера в директория src/
RUN ./gradlew bootJar

EXPOSE 8080

CMD sh -c 'java -jar build/libs/*.jar'     //Read & Write layer
```

In the terminal:

docker build -t my-java-app . **-t** означава сложи му таг, а точката е текущата директория

След като image-а е изпечен/създаден, то може да пуснем container от него:

docker run -it my-java-app /bin/bash

Example 3 – with maven offline:

Dockerfile-build

```
# We start from openjdk17 base image to have JDK installed
FROM eclipse-temurin:17-jdk
WORKDIR /app      //same as RUN mkdir app && cd app
```

```
COPY maven???*** .

# 3 layers runs
RUN apt update
RUN apt install java
RUN apt install python

# preferable to be only 1 layer instead of 3 layers!!!
RUN apt update && apt install java && apt install python

RUN mvn dependencies:go-offline

COPY src src/

RUN ./mvnw package

EXPOSE 8080

CMD sh -c 'java -jar build/libs/*.jar'
```

Example 4 – multistage

Заема по-малко място самият готов изпечен image

Dockerfile-multistage

```
# phase 1
FROM eclipse-temurin:17-jdk AS build
WORKDIR /app

COPY gradlew .
COPY gradle gradle/
COPY build.gradle.kts .
COPY settings.gradle.kts .

COPY src src/
RUN ./gradlew bootJar

# phase 2 - run the jar file
FROM eclipse-temurin:17-jre
WORKDIR /app

COPY --from=build /app/build/libs/*.jar app.jar

EXPOSE 8080
CMD sh -c 'java -jar app.jar'
```

docker build -t "petclinic" -f Dockerfile-multistage .

Recommendations

- Don’t create large images
- Don’t use only the “latest” tag
- Don’t run more than one process in a single container
- Don’t rely on IP addresses
- Put information about the author

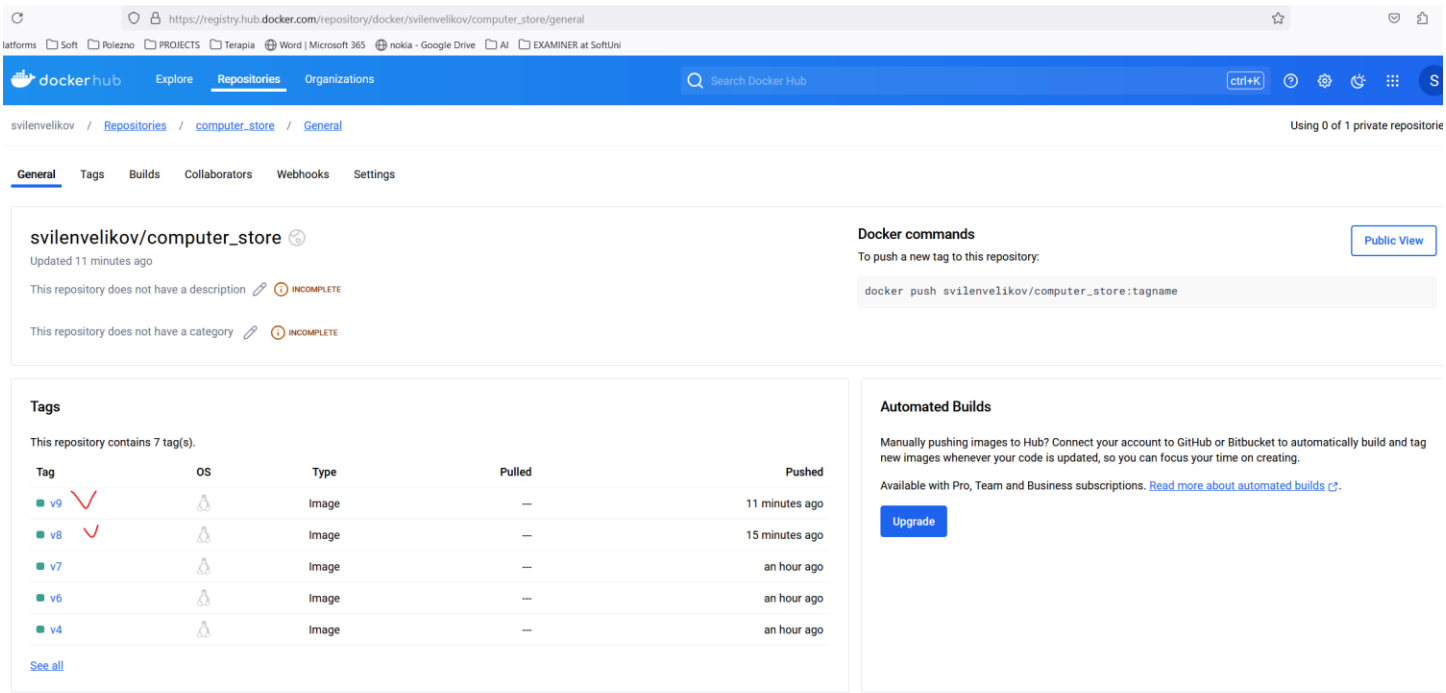
Local docker image to docker hub

How to push a local docker image to docker hub
https://docs.docker.com/get-started/workshop/04_sharing_app/

mvn clean package
docker build -t computer_store:v3 .

D:_GitHub locale\PROJECTS\computer_store> **docker login -u svilenvelikov**
Password:
Login Succeeded

docker tag computer_store:v3 svilenvelikov/computer_store:v3
docker push svilenvelikov/computer_store:v3



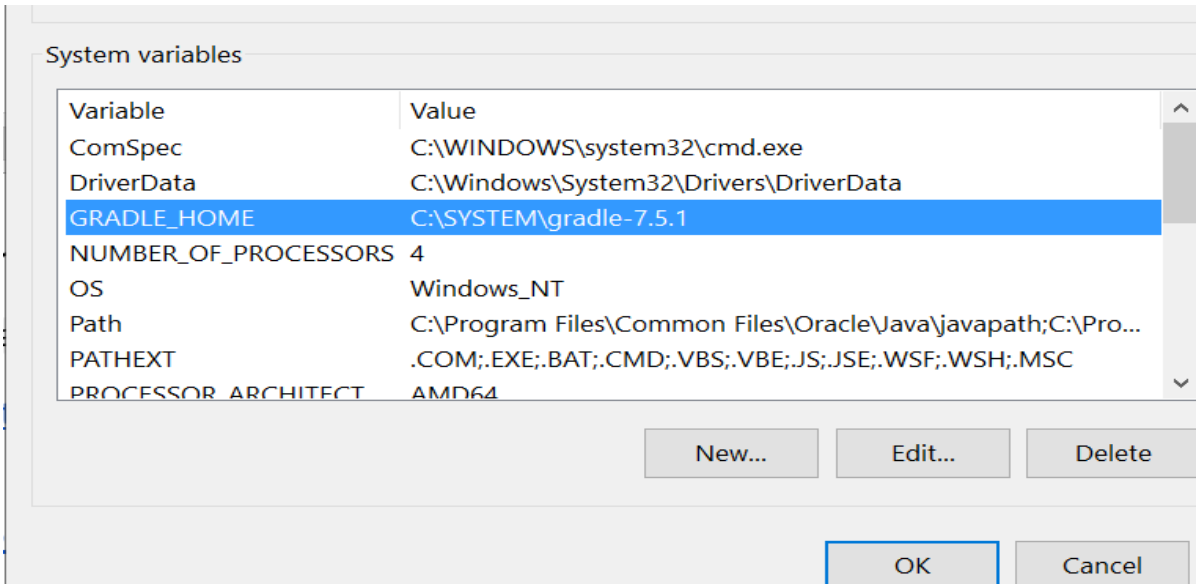
VI. Demo summary + Gradle Wrapper and Gradle for Windows

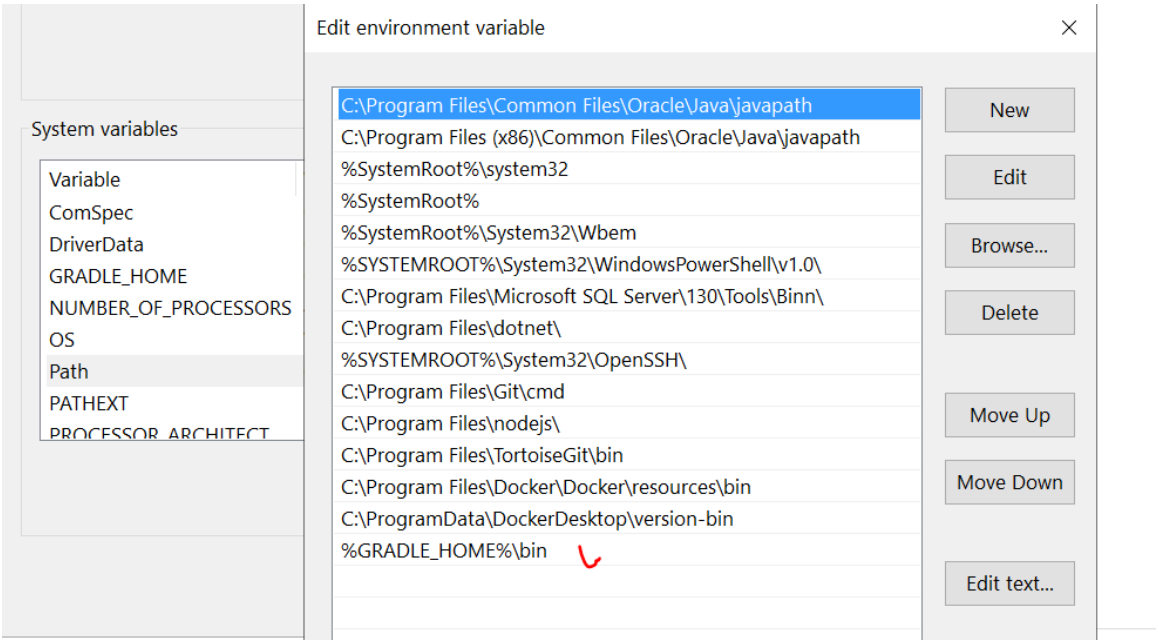
Gradle

https://tomgregory.com/what-is-the-gradle-wrapper-and-why-should-you-use-it/

The Gradle wrapper is a script you add to your Gradle project and use to execute your build. The advantages are:

you don’t need to have Gradle installed on your machine to build the project
the wrapper guarantees you’ll be using the version of Gradle required by the project
you can easily update the project to a newer version of Gradle, and push those changes to version control so other team members use the newer version





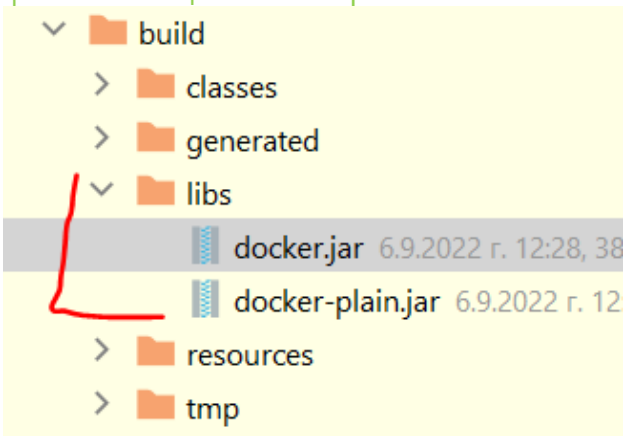

```
gradle init
```

Running a build

```
gradle build
gradle clean build
```

When you run the **build** command you'll see output like this.
This means that your application was assembled and any tests passed successfully. Gradle will have created a build directory if it didn't already exist, containing some useful outputs you should be aware of.

1. **the *classes* directory** contains compiled .class files, as a result of running the Java compiler against your application Java source files

2. **the *libs* directory** contains a generated jar file, an archive with all your compiled classes inside **ready to be executed or published** – ако махнем версията от build.gradle, то .jar файловете ще са с по-кратки имена.
3. 

4. **the *reports* directory** contains an HTML report summarising your test results. If your build fails, then consult this report to see what went wrong.

You now know that running the Gradle build command is equivalent to running the build task, which you do in Windows with `gradlew build` and on Linux/Mac with `./gradlew build`. The *build* task depends on other tasks, and in some situations it may make sense to run a more specific task.

След което можем да го пуснем/run-нем тези jar
java -jar ./build/libs/docker.jar

За да създадем .jar файл с Gradle, то тестовете трябва да минават

Maven
За да създадем .jar файл с Maven, то тестовете трябва да минават

Файлът /deployment/Dockerfile:


```
FROM openjdk:11-jre

VOLUME /tmp
COPY build/libs/docker.jar app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom -Xms256m -Xmx512m", "-jar","/app.jar"]
```

Ако използваме Maven, тези думички били различни
build/libs

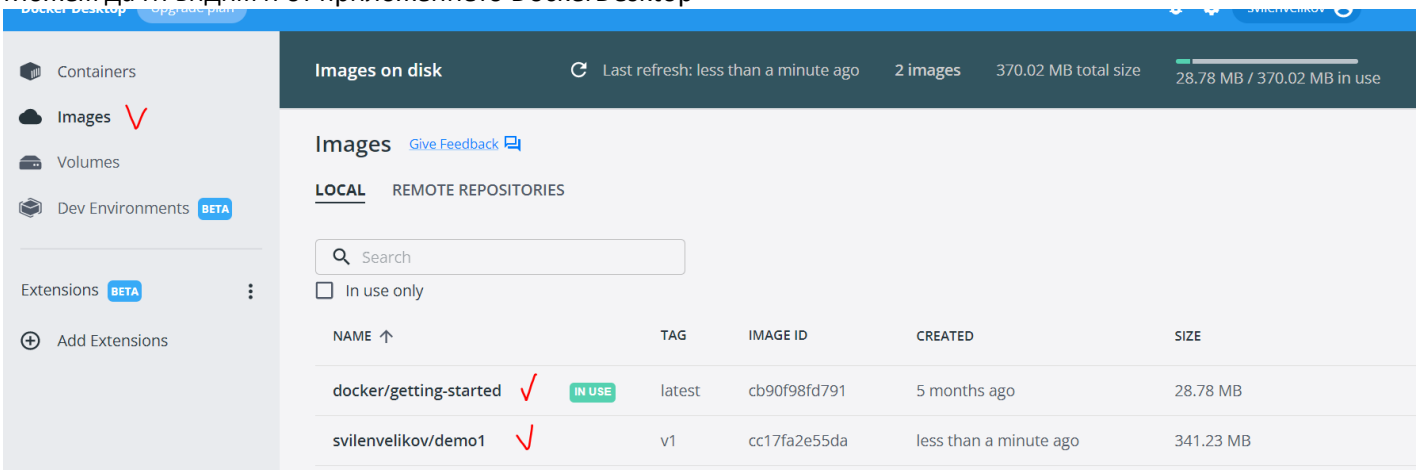
От основната директория на проекта ни, под CMD на Windows пускаме следната команда
C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example>
docker build -t svilenvelikov/demo1:v1 -f deployment/Dockerfile .
създай от .jar файла **image** с име demo1 версия 1 на user svilenvelikov и от текущата директория която е точка, изпълни Dockerfile.

docker images

в терминала показва всички имиджи качени на докер към момента

```
C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example>docker images
REPOSITORY                TAG       IMAGE ID       CREATED         SIZE
svilenvelikov/demo1       v1        cc17fa2e55da   5 minutes ago   341MB
docker/getting-started    latest    cb90f98fd791   4 months ago    28.8MB
```

Можем да ги видим и от приложението DockerDesktop



I) Пускане на docker image-а

C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example>
docker run -p 8080:8080 svilenvelikov/demo1:v1

Когато image-а се run-не, то той става контейнер.

docker ps

покажи текущо активни контейнери

Качи image-а на облака docker repositories

docker push svilenvelikov/demo2:v2

останалите images са част от jdk и няма нужда да се пушват

```
PS C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example> docker push svilenvelikov/demo2:v2
The push refers to repository [docker.io/svilenvelikov/demo2]
4ced9a8f798d: Pushed
5a7e7a880634: Mounted from library/openjdk
3dccaa93bb0e: Mounted from library/openjdk
5c384ea5f752: Mounted from library/openjdk
293d5db30c9f: Mounted from library/openjdk
03127cdb479b: Mounted from library/openjdk
9c742cd6c7a5: Mounted from library/openjdk
v2: digest: sha256:3232e698c894f814aea10bb0e5ecbd61af3562e9b8bdea73dab5cb4d49acc547 size: 1794
```

Remove-ване на спрян контейнер

При опит да изтрием docker image:
Error response from daemon: conflict: unable to delete **5d4cfe7396db** (must be forced) - image is being used by stopped container **d61b2cca0024**
PS C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example> **docker rm d61b2cca0024**

Remove-ване на самия docker image

PS C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example> **docker rmi 5d4cfe7396db**

Untagged: svilenvelikov/demo2:v2
Untagged: svilenvelikov/demo2@sha256:3232e698c894f814aea10bb0e5ecbd61af3562e9b8bdea73dab5cb4d49acc547
Deleted: sha256:5d4cfe7396db1e26c28c46c664427ad0041b8afa23c23aeb5c8f4b28091e9ac6

Търси първо локално, след това търси от облака

Ако пуснем команда за рънване на изтрития локално image, то първо търси локално, след това търси на облака docker repositories
docker run -p 8080:8080 svilenvelikov/demo2:v2

PS C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example> docker run -p 8080:8080 svilenvelikov/demo2:v2
Unable to find image 'svilenvelikov/demo2:v2' locally
v2: Pulling from svilenvelikov/demo2
001c52e26ad5: Already exists
d9d4b9b6e964: Already exists
2068746827ec: Already exists
8510da692cda: Already exists
b6d84395b34d: Already exists
bf03fea6c3ad: Already exists
7ac2c538825b: Already exists
Digest: sha256:3232e698c894f814aea10bb0e5ecbd61af3562e9b8bdea73dab5cb4d49acc547
Status: Downloaded newer image for svilenvelikov/demo2:v2

II) Пускане на повече docker images - как 2 image-а си говорят когато са качени на Docker

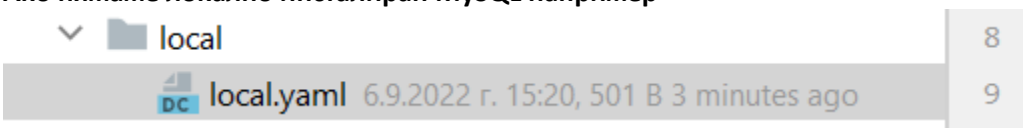
PS C:\Temp projects\Spring Advanced\12 Containerization & Documentation\docker_example>

Docker Compose is **a tool that was developed to help define and share multi-container applications**. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

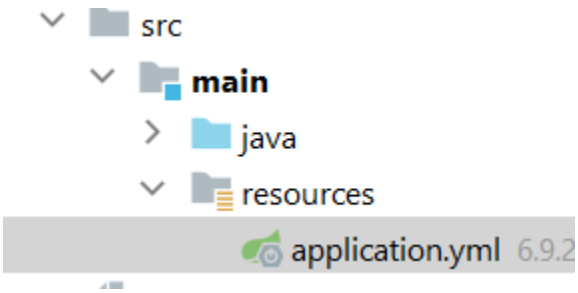
docker-compose -f ./local/local.yaml up

docker-compose -f ./local/local.yaml **down**

Ако нямаме локално инсталиран MySQL например




```
version: '3.3'
services:
  db:
    image: arm64v8/mysql:oracle
    ports:
      - "3306:3306"
    command: ['--character-set-server=utf8mb4', '--collation-server=utf8mb4_bin', '--default-
authentication-plugin=mysql_native_password']
    environment:
      - MYSQL_ALLOW_EMPTY_PASSWORD="yes"
      - MYSQL_DATABASE=intro
  docker-demo:
    image: svilenvelikov/docker-demo:v3
    ports:
      - "8080:8080" # web ui
    environment:
      - MYSQL_HOST=db
      - MYSQL_USER=root
      - MYSQL_PASSWORD=
```



application.yml

```
spring:
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    url:
"jdbc:mysql://${MYSQL_HOST:localhost}:3306/demodocker?useSSL=false&createDatabaseIfNotExist=true&s
erverTimezone=UTC"
    username: "${MYSQL_USER:root}"
    password: "${MYSQL_PASSWORD:}"
```

III) Пускане на docker image с CMD опция

При стартиране на docker image-а да прави ping

При първата команда ping-ва от локалния сървър localhost/от локалния компютър.

При втората команда овъррайдваме CMD-то да бъде google.com и сега ping-ва от google.com

ENTRYPOINT не може да се override-не

```
FROM debian:wheezy
ENTRYPOINT ["/bin/ping"]
CMD ["localhost"]

$ docker run -it test
PING localhost (127.0.0.1): 48 data bytes

$ docker run -it test google.com
PING google.com (173.194.45.70): 48 data bytes
```

10. Kubernetes and orchestration

History Overview

- At the beginning software was mostly installed **on prem** (on physical hardware)
- Then the cloud paradigm started emerging and **hypervisors** like VirtualBox and HyperV became important players in that area
- However virtualization posed significant limitations due to the strict isolation of VMs in terms of resources (CPU and memory)

- The evolution of hardware and operating systems provided the better possibilities to run applications as separate isolated processes
- An example of such possibilities are **namespaces** and **cgroups** in the Linux Kernel – изолираме ресурсите като памет и процесор за всеки процес без да е нужно външно приложение като **hypervisor**
- This enabled the implementation of container solutions like Docker

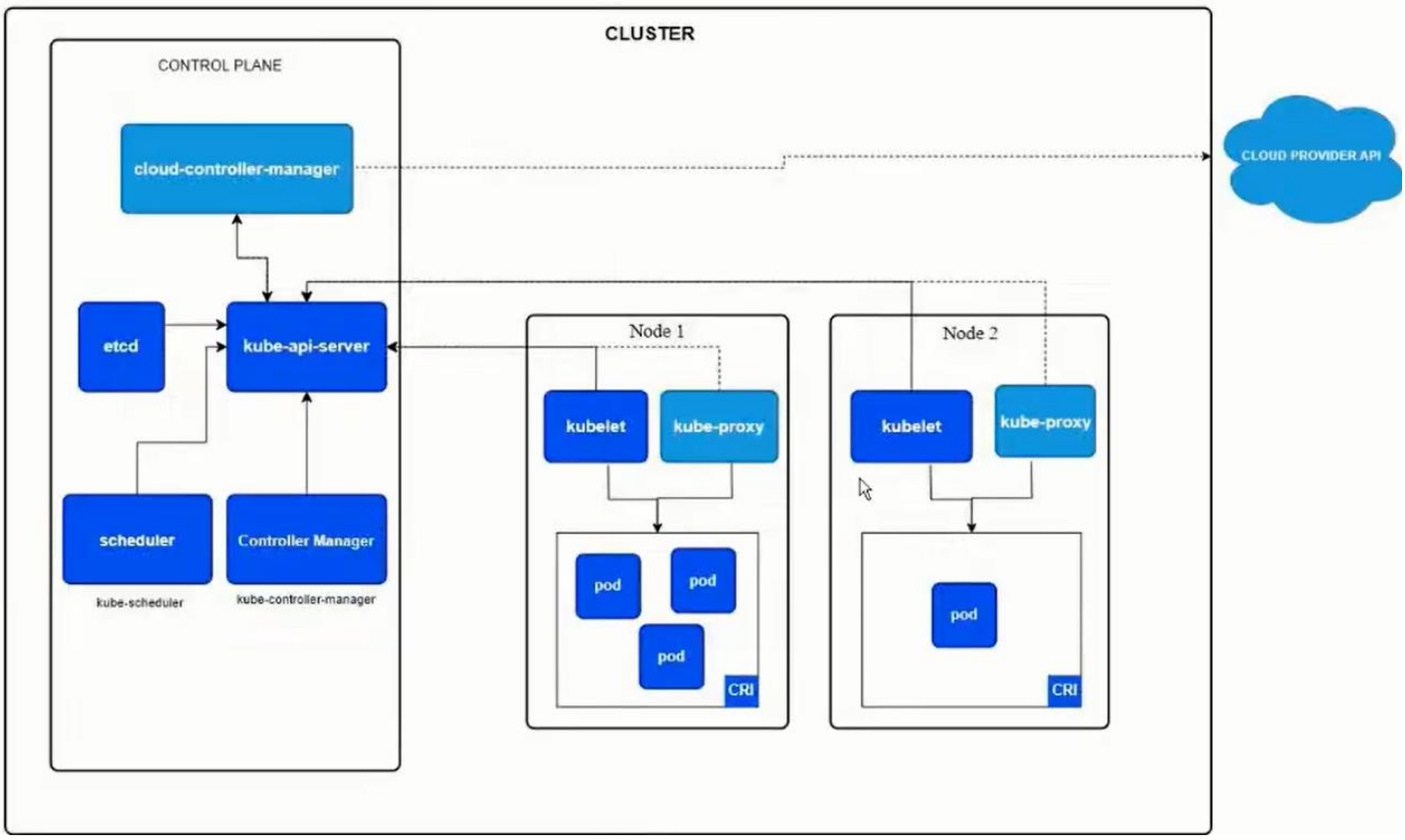
- However, the use of container technology like Docker still implied a lack of good tooling to automate the process of management and orchestration of the containers:
    - How do we manage configuration of containers?
    - How do we scale up/down using containers? (i.e. Docker Swarm)
    - How do we automate deployment of a group of containers? (i.e. Ansible scripts or Docker compose)
    - How do we manage the administrative aspects of a container (i.e. networking, traffic redirection, security, resource allocation, etc…)

- Due to the increasing needs to optimize resources utilization and management aspects of containers Google developed Borg as an internal large-scale cluster management system
- Borg allowed Google to run thousands of jobs from a number of applications
- Borg was later succeeded by Omega as a second generation cluster management system which emerged at about the same time as Docker

- Thus Kubernetes was born out of the idea to combine the possibilities of a container technology like Docker with a scalable cluster management system like Borg….

Architecture

https://kubernetes.io/docs/concepts/architecture/



Control Plane – на отделна физическа машина

etcd – разпределена файлова система, която се използва за да persist-ва цялата информация в Kubernetes cluster-а.

kube-api-server – rest interface, с който можем да менъджираме различните елементи в Kubernetes клъстера. На Java, C#, Python, и т.н. Този kube-api-server менъджира също node-вете.

node – виртуална машина или физическа машина или набор от няколко мощни сървъра - машина на която да се деплойват **pod**-вете

kubelet – комуникира двустранно – с kube-api-server и с pod-вете. Като менъджира pod-вете. С други думи kube-api-server казва на kubelet-а стартирай ми pod с тези и тези контейнери на този node в клъстера.

kube-proxy – се грижи да ротира трафика към pod-овете в рамките на node-а.

scheduler – се грижи за стартирването на pod-вете в node-вете – като например кой node има достатъчно ресурси в момента да стартира дадения pod.

Controller Manager – се грижи да изпълнява различни процеси/задачи – като си минава по пътя kube-api-server -> kubelet:

- като например деплоймент на приложение – еднократни job-ве от рода на стартирай нов node и му сложи първоначални pod-ве.
- Ако даден node не е reachable, то да го маркира да не се деплойват на него pod-ве в момента.
- Replicaset-controller – да се погрижи да има поне една инстанция например за nginx **pod**-а

Cloud-controller-manager – API, което може да се имплементира от различни Cloud providers (GoogleCloudProvider, AmazonWebServices). Дава възможност да се менъджират различни дейности в рамките на съответния cloud provider, последния който деплойва Kubernetes cluster.

Pod

- The basic unit of operation in Kubernetes is a **pod**
- A **pod** may contain one or more containers (Docker containers for example)
- **Pods** are logically grouped in Kubernetes **namespaces** – не е задължително да използваме namespaces, но когато организацията е голяма определни хора или тиймове да имат определени права свързани с този namespace
- **Namespaces** provide isolation of resources, users and permissions
- За всеки един pod се алокира отделен IP address

Workload

- Workloads are a logical grouping of pods that form an application
- Workloads in Kubernetes can be different types – in yaml syntaxis:
    - ReplicaSets – как да репликираме даден pod
    - Deployments – описание как можем да деплойваме pod
    - StatefulSets
    - DaemonSets

Services

Контейнерите на всеки pod могат да стартират на определени портове. Но за всеки отделен pod се алокира отделен IP address.

Всички pod-s от даден node са организирани в т.нар. клъстерна мрежа.
За да може един pod да достъпи сървиси от друг pod, то не е видимо как да стане от самите контейнери! За да може това нещо да се реализира, то имаме т.нар. сървиси/services.

- As the name implies: provides the possibility to define a ‘service’ on top of pods
- The service has an IP address, port and a DNS name
- The service provides the possibility to route traffic to pods either from inside the Kubernetes cluster or from outside
- A service is also described in yaml file

Ingress

Ако искаме да ротираме трафик между подовете на базата на конкретни правила.

- Ingress elements provide the possibility to route traffic to pods
- It can be configured on a rule-basis specifying how traffic coming to the cluster can be routed to pods
- Ingress routes traffic to services – тези правила са вече декларирани в **service** yaml файловете.

ConfigMaps

- ConfigMaps are the main mechanism used to supply configuration to pods
- That configuration can be passed in either of two ways from a ConfigMap to a pod:
    - As environment variables
    - Mounted on a pod container’s file system

Secrets

- Secrets like ConfigMaps can be used to store configuration data
- Secrets are intended to store **sensitive data (such as passwords and certificates)**
- Data stored as Kubernetes secrets is not encrypted-at-rest by default
- Secrets can also be derived from external access management applications like HashiCorp Vault or Keycloak, etc.

Имаме няколко различни механизма, с които можем да интегрираме Keycloak в Kubernetes:

- Ако сме на Spring/Quarkus проект – самият проект ни дава възможност да се интегрираме с Keycloak
- Kubernetes може да извлича данни от Keycloak и да ги записва като Kubernetes secrets

Volumes

- Containers provide by default temporary storage that is wiped out on container restart (in the case of Kubernetes – a pod restart)
- Kubernetes provides the possibility to create and manage persistent volumes used by the pods
- Volumes can be mounted to the file system on a pod container – например да пазим данни в база данни

Deployment and tools

One local instance of Kubernetes

- Deploying a Kubernetes cluster form scratch is not a trivial task…
- For that purpose for local development there are a number of options available like:
    - **minikube – една инстанция на една виртуална машина или на един докер контейнер**
    - Kind
    - K3s
    - MicroK8s
    - Kubeadm
    - DockerDesktop with installed also Kubernetes plugin run in a docker container

API clients

- **kubectl** – standard (and most used) CLI client for Kubernetes

kubectl create deployment nginx --image nginx //create deployment е по-скоро за локално тестване
kubectl apply -f deployment.yaml
kubectl get pods -n somenamespace
kubectl describe pod pod_id -n somenamespace какви контейнери и други неща в дадения под
kubectl logs pod_id -n somenamespace по подразбиране се листват логовете на първия контейнер от пода
kubectl logs pod_id -c containerid -n somenamespace

- language API clients (i.e. for Java)
- other tools – like **k9s**

Web UIs

Different third party web interfaces for managing Kubernetes:

- **Kubernetes Dashboard** (standard and most popular)
- Kubernetes Lens (IDE for Kubernetes)
- Octant
- others

Package management

- **Helm** is a widely used package manager for Kubernetes
- It is used to build distributable **charts** that are packages of Kubernetes resources
- Charts са група от Kubernetes ресурси – например deploy.yaml, който деплойва **pod** с nginx и Spring приложение с/на 3 инстанции, services, ConfigMaps, и т.н.
- Тези **charts** можем да ги създаваме, да ги качваме в централно Helm хранилище, и да бъдат преизползвани в различни Kubernetes клъстери.
- Charts can be stored in a central repository and used by applications

Инсталираме първо helm.
След това:
helm install petclinic k8s/charts

values.yaml

```yaml
namespace: petclinic
app:
  name: petclinic
  image: petclinic
  tag: latest
  pullPolicy: IfNotPresent
  port: 8080
  serviceType: LoadBalancer
  replicas: 4
db:
  name: postgres
  image: postgres
  tag: latest
  port: 5432
  serviceType: ClusterIP
  replicas: 1
  username: petclinic
  password: petclinic
  database: petclinic
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.app.name }}-deployment
  namespace: {{ .Values.namespace }}
spec:
  replicas: {{ .Values.app.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.app.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.app.name }}
    spec:
      containers:
        - image: "{{ .Values.app.image }}:{{ .Values.app.tag }}"
          imagePullPolicy: {{ .Values.app.pullPolicy }}
          name: {{ .Values.app.name }}
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: {{ .Values.db.name }}
            - name: SPRING_DATASOURCE_URL
              value: jdbc:postgresql://{{ .Values.db.name }}-service:{{ .Values.db.port }}/{{
.Values.db.database }}
          ports:
            - name: http
```

```
            containerPort: {{ .Values.app.port }}
            protocol: TCP
      restartPolicy: Always
```

Installing

Combination 1

1) DockerEngine Или DockerDesktop
2) kubectl

https://kubernetes.io/docs/tasks/tools/

`kubectl version –client`

3) minikube

https://minikube.sigs.k8s.io/docs/start/

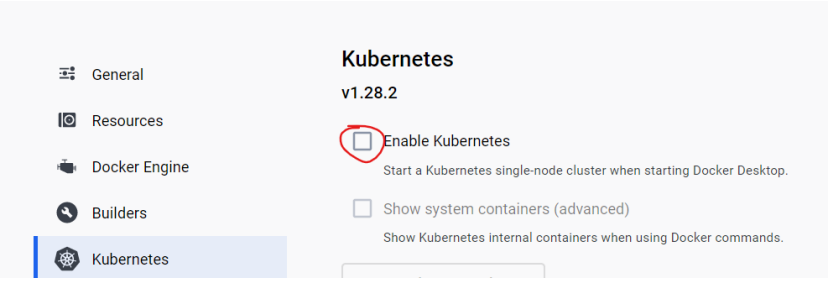
Add the minikube.exe binary to your PATH.
Make sure to run PowerShell as Administrator.

```
$oldPath = [Environment]::GetEnvironmentVariable('Path',
[EnvironmentVariableTarget]::Machine)
if ($oldPath.Split(';') -inotcontains 'C:\minikube'){
  [Environment]::SetEnvironmentVariable('Path', $('{0};C:\minikube' -f $oldPath),
[EnvironmentVariableTarget]::Machine)
}
```

K8S === Kubernetes //8 букви между K и S

Combination 2

Use DockerDesktop with Kubernetes plugin enabled



minikube not needed in this option

Some commands:

minikube start –driver=docker
docker ps
kubectl get pods
kubectl get pods -A all including system

kubectl create deployment nginx --image nginx

Основната разлика между Kubernetes и Docker(и Docker compose) е, че вече минаваме към декларативен начин на работа – чрез въпросните yaml файлове за Kubernetes:
C:\Windows\System32>**kubectl run nginx --image nginx** //създава си служебен yaml файл

pod/nginx created

C:\Windows\System32>**kubectl get pods**
NAME READY STATUS RESTARTS AGE
nginx 1/1 Running 0 84s

C:\Windows\System32>**kubectl delete pod nginx**
pod "nginx" deleted

C:\Windows\System32>**kubectl delete pod/nginx**
pod "nginx" deleted

C:\Windows\System32>**kubectl run nginx --image nginx -o yaml** //-o принтни служебно създадения yaml файл
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2024-08-15T11:38:32Z"
  labels:
    run: nginx
  name: nginx
  namespace: default
  resourceVersion: "14000"
  uid: 67bb5e2b-f0cd-482c-9663-196c6fef2a0e
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-knh6t
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300

```
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: kube-api-access-knh6t
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
      - configMap:
          items:
          - key: ca.crt
            path: ca.crt
          name: kube-root-ca.crt
      - downwardAPI:
          items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace
status:
  phase: Pending
  qosClass: BestEffort
```

Доста от служебно създадените неща за yaml файла са излишни.

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl apply -f k8s/app-service.yaml**

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl get all**

```
NAME        READY   STATUS    RESTARTS   AGE
pod/nginx   1/1     Running   0          11m
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3h3m
```

За следене на промени в терминала да се зареждат автоматично – този watch е допълнителен plug-in
watch -kubectl get all

app-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: petclinic-deployment
  namespace: petclinic
spec:
  replicas: 1    //една инстанция в случая - това нещо Docker и Docker Compose не могат да го направят!
  selector:
    matchLabels:
      app: petclinic
  template:
```

```
    metadata:
      labels:
        app: petclinic
    spec:
      containers:
        - image: petclinic:latest
          name: petclinic
          imagePullPolicy: IfNotPresent
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: postgres
            - name: SPRING_DATASOURCE_URL
              value: jdbc:postgresql://postgres-service:5432/petclinic
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
      restartPolicy: Always
```

app-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: petclinic-service
  namespace: petclinic
spec:
  ports:
    - name: http
      port: 8080
      targetPort: 8080
  selector:
    app: petclinic
  type: LoadBalancer  #ClusterIP   NodePort
```

C:\Windows\System32>**kubectl get pods**

NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 22m

Логовете

C:\Windows\System32>**kubectl logs nginx**

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/08/15 11:38:33 [notice] 1#1: using the "epoll" event method
2024/08/15 11:38:33 [notice] 1#1: nginx/1.27.0
2024/08/15 11:38:33 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/08/15 11:38:33 [notice] 1#1: OS: Linux 5.15.133.1-microsoft-standard-WSL2
2024/08/15 11:38:33 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024/08/15 11:38:33 [notice] 1#1: start worker processes
2024/08/15 11:38:33 [notice] 1#1: start worker process 29
2024/08/15 11:38:33 [notice] 1#1: start worker process 30
2024/08/15 11:38:33 [notice] 1#1: start worker process 31
2024/08/15 11:38:33 [notice] 1#1: start worker process 32
2024/08/15 11:38:33 [notice] 1#1: start worker process 33
2024/08/15 11:38:33 [notice] 1#1: start worker process 34
2024/08/15 11:38:33 [notice] 1#1: start worker process 35
```

2024/08/15 11:38:33 [notice] 1#1: start worker process 36
2024/08/15 11:38:33 [notice] 1#1: start worker process 37
2024/08/15 11:38:33 [notice] 1#1: start worker process 38
2024/08/15 11:38:33 [notice] 1#1: start worker process 39
2024/08/15 11:38:33 [notice] 1#1: start worker process 40
2024/08/15 11:38:33 [notice] 1#1: start worker process 41
2024/08/15 11:38:33 [notice] 1#1: start worker process 42
2024/08/15 11:38:33 [notice] 1#1: start worker process 43
2024/08/15 11:38:33 [notice] 1#1: start worker process 44
2024/08/15 11:38:33 [notice] 1#1: start worker process 45
2024/08/15 11:38:33 [notice] 1#1: start worker process 46
2024/08/15 11:38:33 [notice] 1#1: start worker process 47
2024/08/15 11:38:33 [notice] 1#1: start worker process 48

kubectl delete deploy nginx-deployment
kubectl delete deployment nginx-deployment

Идеята на контекста, е че можем да достъпваме няколкок Kubernetes клъстъра – единият клъстър може да го менъджира нашата фирма, у нас локално друг клъстър, и т.н.
C:\Windows\System32>**kubectl config get-contexts**
CURRENT NAME CLUSTER AUTHINFO NAMESPACE
* docker-desktop docker-desktop docker-desktop

C:\Windows\System32>**minikube start**
W0815 15:24:53.997133 22260 main.go:291] Unable to resolve the current Docker CLI context "default": context "default": context not found: open C:\Users\svilk\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33f0688f\meta.json: The system cannot find the path specified.
* minikube v1.33.1 on Microsoft Windows 11 Pro 10.0.22631.3880 Build 22631.3880
* Automatically selected the docker driver. Other choices: hyperv, ssh
* Using Docker Desktop driver with root privileges
* Starting "minikube" primary control-plane node in "minikube" cluster
* **Pulling base image v0.0.44 ...**
* Downloading Kubernetes v1.30.0 preload ...
> gcr.io/k8s-minikube/kicbase...: 481.58 MiB / 481.58 MiB 100.00% 3.43 Mi
> preloaded-images-k8s-v18-v1...: 342.90 MiB / 342.90 MiB 100.00% 2.44 Mi
* **Creating docker container** (CPUs=2, Memory=8000MB) ...
* Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
- Generating certificates and keys ...
- Booting up control plane ...
- Configuring RBAC rules ...
* Configuring bridge CNI (Container Networking Interface) ...
* Verifying Kubernetes components...
- Using image gcr.io/k8s-minikube/storage-provisioner:v5
* Enabled addons: storage-provisioner, default-storageclass

* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

Автоматично сменя контекста на kubectl да използва minikubea, не docker-desktop
C:\Windows\System32>**kubectl config get-contexts**
CURRENT NAME CLUSTER AUTHINFO NAMESPACE
docker-desktop docker-desktop docker-desktop
* minikube minikube minikube default

C:\Windows\System32>**kubectl config use-context docker-desktop**
Switched to context "docker-desktop".

C:\Windows\System32>**kubectl config use-context minikube**
Switched to context "minikube".

C:\Windows\System32>**kubectl create namespace petclinic**
namespace/petclinic created

C:\Windows\System32>**kubectl config use-context minikube --namespace=petclinic**
Switched to context "minikube".

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl apply -f k8s/app-deployment.yaml**
deployment.apps/petclinic-deployment created

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl run petclinic --image petclinic**
pod/petclinic created

C:\Windows\System32>**minikube stop**
W0815 16:15:12.563473 4696 main.go:291] Unable to resolve the current Docker CLI context "default": context "default": context not found: open C:\Users\svilk\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33f0688f\meta.json: The system cannot find the path specified.
* Stopping node "minikube" ...
* Powering off "minikube" via SSH ...
* 1 node stopped.

Minikube си има собствен контекст. Обаче за локални images, трябва да му се даде тази команда при MacOS (за Windows обаче не бачка). И след това наново билдваме image-а petclinic (D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **docker build -t "petclinic" -f Dockerfile-multistage .**) и текущия терминал го засича вече Image-а като локален такъв.

```
# Set docker env
eval $(minikube docker-env)              # Unix shells
minikube docker-env | Invoke-Expression # PowerShell
```

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl get all**
NAME READY STATUS RESTARTS AGE
pod/petcliniccontainer 0/1 ImagePullBackOff 0 22s

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl apply -f k8s цялата папка**
deployment.apps/petclinic-deployment created
service/petclinic-service created
deployment.apps/postgres-deployment created
service/postgres-service created

D:_GitHub locale\BGJUG-public\bgjug-academy-docker-k8s-demo-main> **kubectl get all**
NAME READY STATUS RESTARTS AGE
pod/petclinic-deployment-85b48d56b4-sldnb 1/1 Running 2 (33s ago) 42s
pod/petcliniccontainer 0/1 ImagePullBackOff 0 2m32s
pod/postgres-deployment-6cdb694745-bskt8 1/1 Running 0 42s

NAME TYPE **CLUSTER-IP** EXTERNAL-IP PORT(S) AGE

```
service/petclinic-service   LoadBalancer   10.108.107.116   localhost    8080:31560/TCP   42s
service/postgres-service    ClusterIP      10.102.100.199   <none>       5432/TCP         42s

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/petclinic-deployment  1/1     1            1           42s
deployment.apps/postgres-deployment   1/1     1            1           42s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/petclinic-deployment-85b48d56b4  1        1         1       42s
replicaset.apps/postgres-deployment-6cdb694745   1        1         1       42s
```

Kubernetes е предвиден да работи с много сървъри с много клъстери, най-често Cloud услуги. Като му зададем LoadBalancer, то трябва да има друг Load Balancer, към който Kubernetes да се кънектва!

Kubernetes изпълнява image-те по същият начин както Docker.

Kubernetes влиза в полза особено много когато проекта ни има микросървиси.

11. JVM innerworkings

I. Hotspot JVM

Virtual machines

A typical virtual machine for an interpreted language provides:

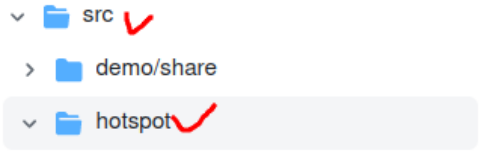
- Compilation of source language **into** VM specific **bytecode - компилира се до bytecode** (съвкупност от компилирани Java класове)
- Data structures to contains instructions and operations (the data the instructions process) - структури от данни за това какви са операциите и процесорните инструкции, които се поддържат от **bytecode-а** на JavaVirtualMachine (JVM)
- A **call stack** for function call operations

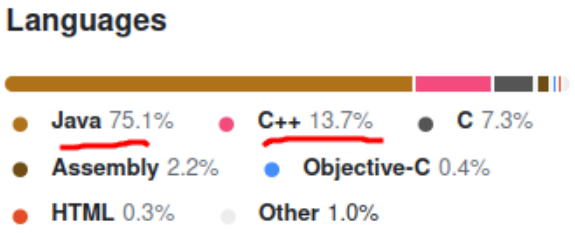
- An '**Instructor Pointer**' (IP) pointing to the next processor instruction to execute

- A **virtual 'CPU'** - the instruction dispatcher that:
  - *Fetches* the next instruction (addressed by the instruction pointer)
  - *Decodes* the operands
  - *Executes* the instruction

The HotSpot JVM

Кода на JVM го има в интернет:
https://github.com/openjdk/jdk/





HotSpot is the standard JVM distribution of Oracle Corp. that provides:

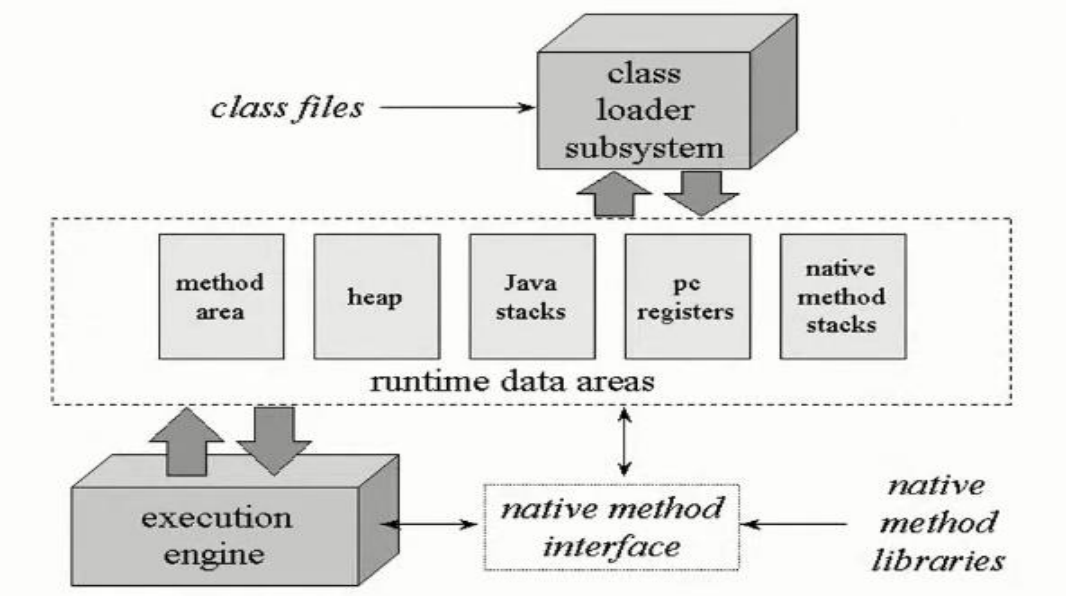
- **bytecode execution**
  - using an **interpreter - bytecode here**,
  - *На практика в някои случаи минаваме и през тази стъпка!!* - two **runtime compilers** (**JIT** (Just-In-Time) compilers) for optimization - например ако един метод се вика 50 000 пъти, то JVM преценява дали да го **компилира до машинен код**. Като компилирания машинен код е по-бърз от компилирания bytecode!! Client and server - C1 and C2 in JVM. Now they work together and not separately
    - **Client JIT** - важно приложението да стартира бързо, а цялото зареждане допълнение
    - **Server JIT** - за големи сървърни приложения. JVM по време на компилиране прави много оптимизации и стартира по-бавно. Така че когато е заредило вече приложението, то логиката в него пък да се изпълнява по-бързо
  - and **On-Stack Replacement** - JVM може да прецени да не изпълнява компилирания машинен код, и вместо това да изпълни компилирания bytecode. Т.нар. деоптимизация.

- **storage allocation** and **garbage collection**

Алокиране на памет за обектите, които създаваме в нашето приложение. Т.нар. HEAP памет се заделя при стартиране на нашето Java приложение.
Колко памет да използва, колко минимална и максимална памет да използва.
Бързодействието на самата JVM машина се определя от garbage collection-а - колко бързо можем да освободим памет без да нарушим изпълнението на нашата програма.

- **runtimes** - start up, shut down, class loading, threads, interaction with OS and others

Architecture of the HotSpot JVM



1.Class-loading

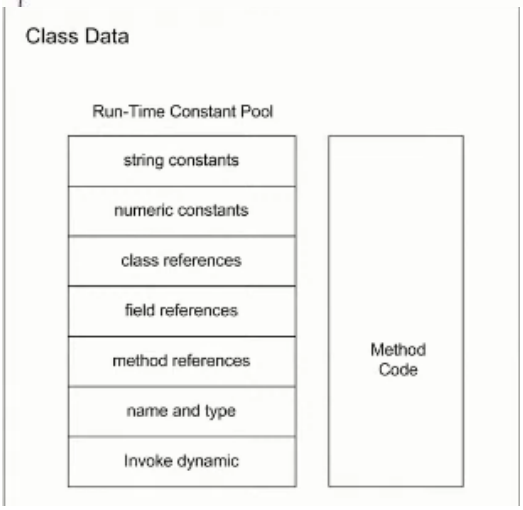
Зареждане до класфайловете от .java файлове

Three phases of class-loading:

- **Loading**

С коя версия на Java compiler е компилиран дадения .java файл. Това се записва в minor и major versions.
Constant_pool - Информация/масив за всички константи, които имаме в дадения клас
Access_flags - public, private, protected
Interfaces - масив с всички интерфейси на класа
Fields - масив с всички полетата на класа
Methods - масив с всички методи на класа
Attributes - масив с допълнителни атрибути, които може да сме дефинирали в класа

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```



- **Linking** - създават се връзки между съответните класфайлове в JVM
- **Initialization** - например инициализация на статични блокове, на константи

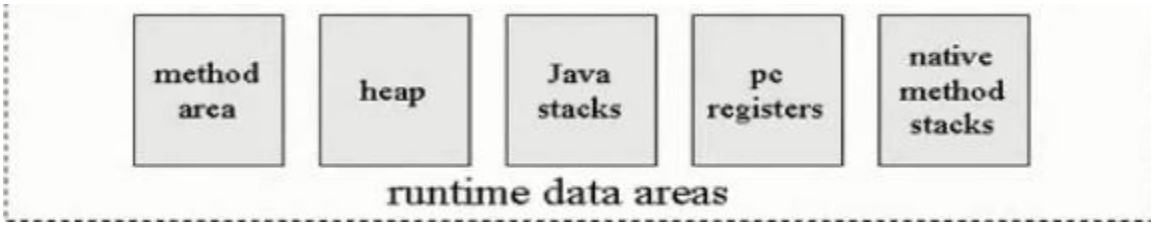
```
static int volume(int width,
        int depth,
        int height) {
    int area = width * depth;
    int volume = area * height;
    return volume;
}
```

⇒

```
0 iload_0
1 iload_1
2 imul
3 istore_3
4 iload_3
5 iload_2
6 imul
7 istore 4
9 iload 4
11ireturn
```

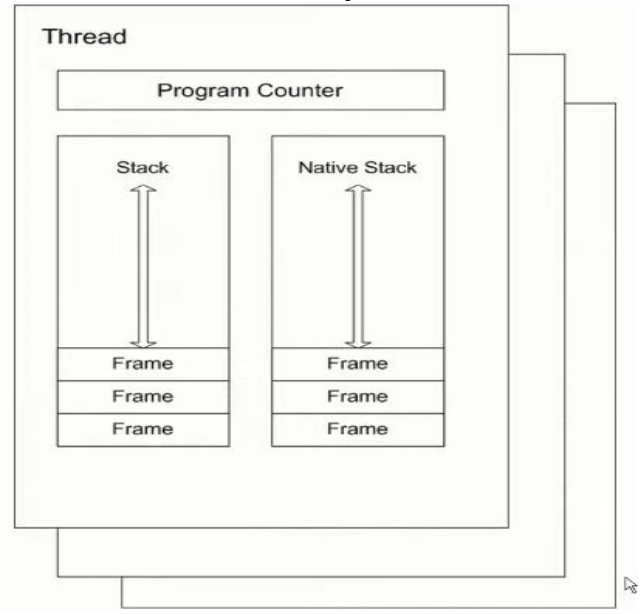
2.Runtime data areas

Main info



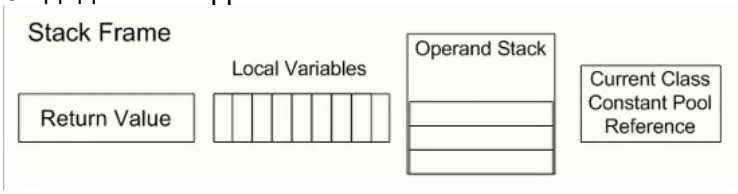
В JVM се зареждат различни области, които се използват от приложението ни. Като например:

- Област за HEAP паметта - за алокиране на обектите от приложението
- Method area - описват се методите, които се изпълняват
- Java stacks - създават се стекови от нишките, които се стартират в рамките на приложението
- **Program counter registers**
- Stacks за изпълнение на native методи - JVM отдолу в голяма степен е имплементирана на C++, така че в някои случаи може да се викат и методи на C++



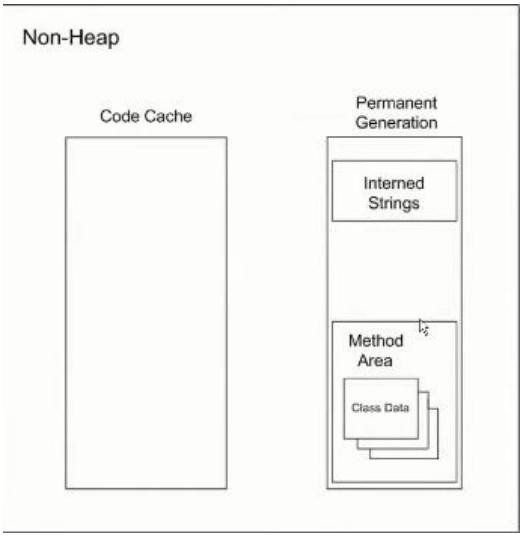
- Отделно JVM може да реши да използва и **native threads (на ниво ОС)** и за тях се създава **отделен Stack**!

За даден стек фрейм имаме:



Before JDK 8

Permanent generation - отделна област за съхраняване на информация за string-овите константи и за методите заредени от JVM



As of JDK 8

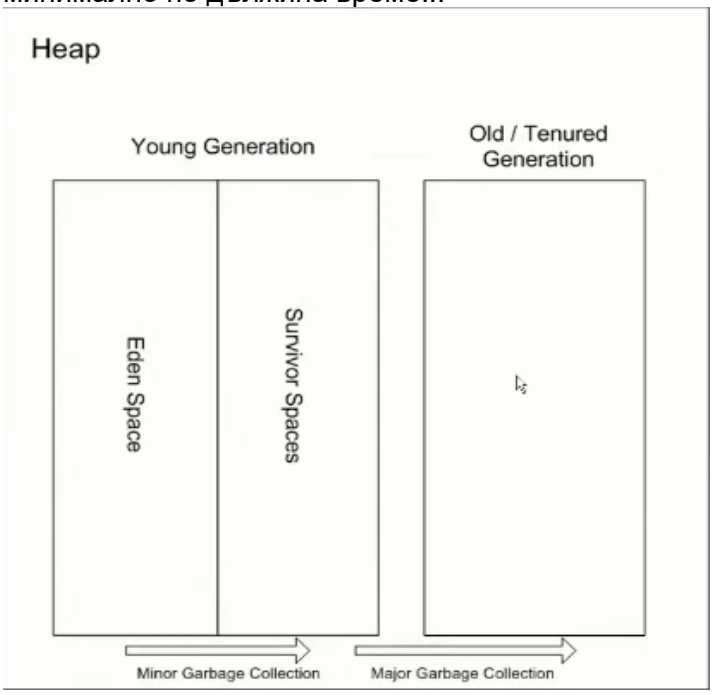
Permanent generation is part of the HEAP

HEAP

Young generation - пазят се тук обекти, които живеят кратко
Old / Tenured Generation - например за логер, който живее през цялото време.
Обекти, за които JVM прецени че живеят по-дълго от очакваното се местят от Young generation към Old / Tenured Generation!

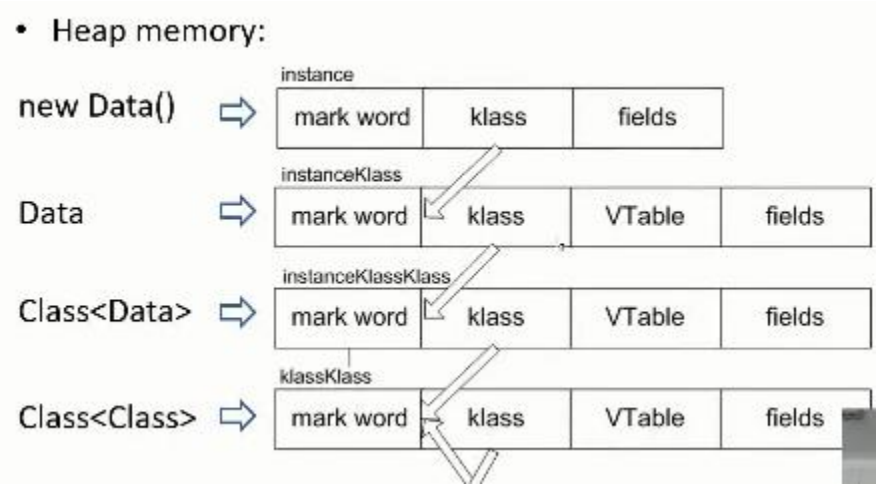
Обектите от Young generation се изчистват по-лесно и бързо от garbage collector-a, без да се прави задълбочен анализ от garbage collector-a.

Чистенето на обекти от Old / Tenured Generation става чрез паузиране, което за някои приложения е ключово. Затова трябва да се стремим чистенето на обекти от Old / Tenured Generation да бъде минимално по дължина време!!!



При създаване на нов обект:
VTable = VirtualTable - при полиморфизъм влиза в употреба

Създават се няколко масива в хийпа:
- един за инстанциирания обект
- един за типа на обекта - от кой клас е - from reflection .getClass
- Class Генерик от тип нашия клас
- Class Генерик от тип Class



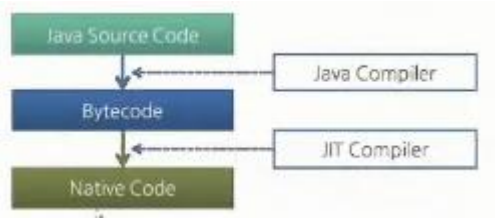
И така за всеки един обект!!!
`mark word` contains:
- identity hash code - от нашето Java приложение
- age
- lock record address - при синхронизация
- monitor address
- state (unlocked, light-weight locked, heavy-weight locked, marked for GC GarbageCollection)
- biased / biasable (includes other fields such as thread ID) - пристрастен - скъпо е една нишка да я изпълняваме на един процесор, после да я изпълняваме на друг процесор в друг етап от изпълнението ѝ. За тази цел JVM може да зададе дадена нишка да се изпълнява само на конкретно ядро от процесора! Т.е. не се случва т.нар. Context Switching! И това подобрява бързодействието!

3.Execution engine

```
while(true) {
        bytecode b = bytecodeStream[pc++];
        switch(b) {
                case iconst_1: push(1); break;
                case iload_0: push(local(0)); break;
                case iadd: push(pop() + pop()); break;
        }
}
```

Different execution techniques:

- Interpreting - **вземаме следващата bytecode процесорна инструкция и я изпълняваме**
- Just-in-time (JIT) compilation - **вземаме native code процесорна инструкция и я изпълняваме**
- Adaptive optimization (determines “hot spots” by monitoring execution) - преценява дали се компилира и изпълнява впоследствие на bytecode или на native code



JIT compilation

- Triggered asynchronously by counter overflow for a method/loop (interpreted counts method entries and loopback branches) - всеки път проверява колко пъти би се извикал дадения метод
- Produces generated **native code (машинен код)** and relocation info (transferred on next method entry)
- In case JIT-compiled code calls not-yet-JIT-compiled code, then control is transferred to the interpreter! - когато JIT-compiled code (**native машинен код**) метод извиква друг метод от нашето приложение, който е само интерпретиран, то JIT компилатора се грижи кой метод как да го изпълни и дали да не изпълни текушия метод на принципа на **On-Stack Replacement** (да го върне текущия метод да му се изпълни байткода вместо native машинния код)
- Compiled code may be forced back into interpreted bytecode (deoptimization)
- Is complemented by On-Stack Replacement - turn dynamically interpreted to JIT compiled code and vice-versa - dynamic optimization/deoptimization
- JIT compiler is more optimized for server VM - the so called server mode - hits big start-up time compared to client VM, but executes faster the logic afterwards.

JIT compilation flow

To native machine code during normal bytecode execution:

1. Bytecode is turned **into a graph**
2. The graph is turned **into a linear sequence** of operations that manipulate an infinite loop of virtual registers (each node places its result in a **virtual register**)
3. **Physical registers** are allocated for virtual registers (the program stack might be used in case virtual registers exceed physical registers)
4. **Native machine code** for each operation is generated using its allocated registers

Tiered Compilation in JVM

Client and server - C1 and C2 in JVM. Now they work together and not separately
https://www.baeldung.com/jvm-tiered-compilation

A JIT compiler **compiles bytecode to native code for frequently executed sections**.

The **client compiler**, also called **C1**, is a type of a **JIT compiler optimized for faster start-up time**. It tries to optimize and compile the code as soon as possible.

The **server compiler**, also called **C2**, is a type of a **JIT compiler optimized for better overall performance**. C2 observes and analyzes the code over a longer period of time compared to C1. This allows C2 to make better optimizations in the compiled code.

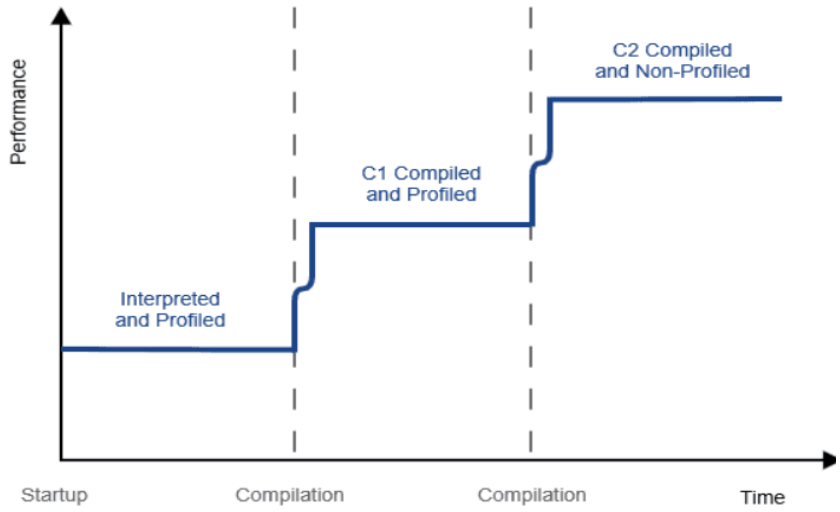
The C2 compiler often takes more time and consumes more memory to compile the same methods. However, it generates better-optimized native code than that produced by C1.

The tiered compilation concept was first introduced in Java 7. Its goal was to **use a mix of C1 and C2 compilers in order to achieve both fast startup and good long-term performance**.

Compilation Levels when tiered compilation enabled

Tiered compilation is **enabled by default since Java 8**. It's highly recommended to use it unless there's a strong reason to disable it.

Even though the JVM works with only **one interpreter** and **two JIT compilers**, there are **five possible levels of compilation**. The reason behind this is that the C1 compiler can operate on three different levels. The difference between those three levels is in the amount of profiling done.



Level 0 – Interpreted Code

Initially, JVM interprets all Java code. During this initial phase, the performance is usually not as good compared to compiled languages.

However, the JIT compiler kicks in after the warmup phase and compiles the hot code at runtime. The JIT compiler makes use of the profiling information collected on this level to perform optimizations.

Level 1 – Simple C1 Compiled Code

On this level, the JVM compiles the code using the C1 compiler, but without collecting any profiling information. The JVM uses level 1 for **methods that are considered trivial**.

Due to low method complexity, the C2 compilation wouldn't make it faster. Thus, the JVM concludes that there is no point in collecting profiling information for code that cannot be optimized further.

Level 2 – Limited C1 Compiled Code

On level 2, the JVM compiles the code using the C1 compiler with light profiling. The JVM uses this level **when the C2 queue is full**. The goal is to compile the code as soon as possible to improve performance.

Later, the JVM recompiles the code on level 3, using full profiling. Finally, once the C2 queue is less busy, the JVM recompiles it on level 4.

Level 3 – Full C1 Compiled Code

On level 3, the JVM compiles the code using the C1 compiler with full profiling. Level 3 is part of the default compilation path. Thus, the JVM uses it in **all cases except for trivial methods or when compiler queues are full**.

The most common scenario in JIT compilation is that the interpreted code jumps directly from level 0 to level 3.

Level 4 – C2 Compiled Code

On this level, the JVM compiles the code using the C2 compiler for maximum long-term performance. Level 4 is also a part of the default compilation path. The JVM uses this level to **compile all methods except trivial ones**.

Given that level 4 code is considered fully optimized, the JVM stops collecting profiling information. However, it may decide to deoptimize the code and send it back to level 0.

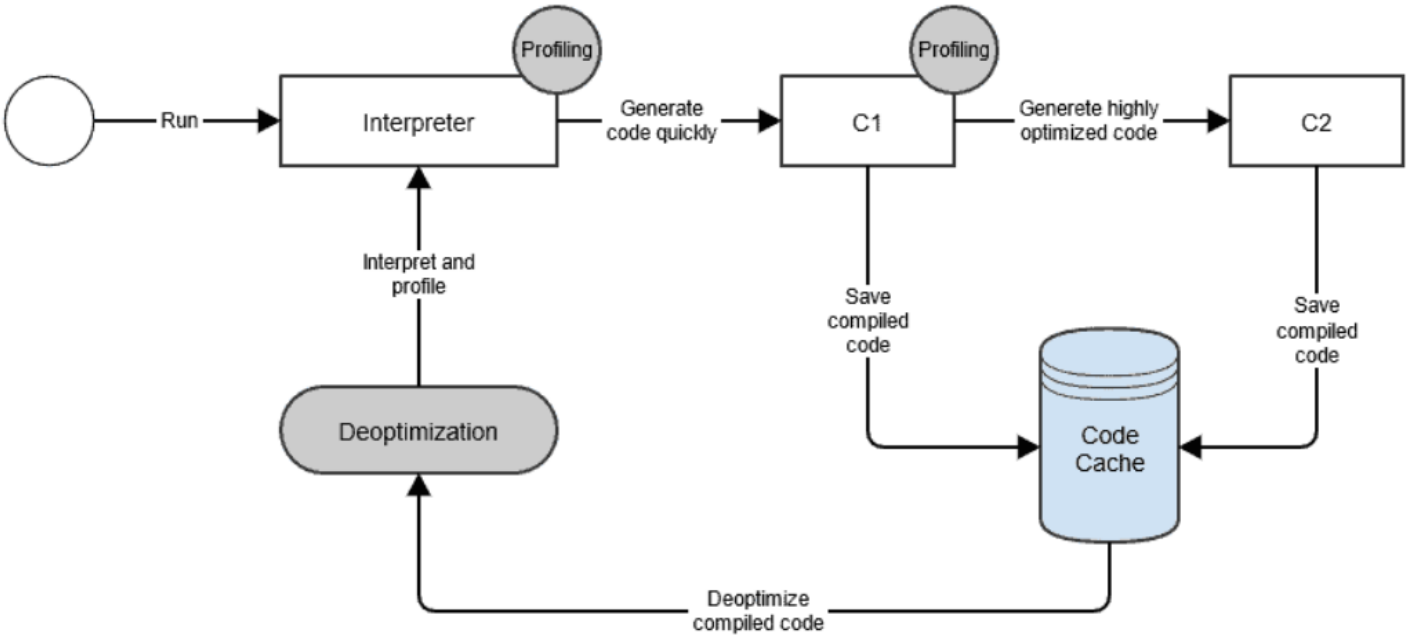
Thresholds for Levels and Method Compilation

In order to check the default thresholds used on a specific Java version, we can run Java using the *-XX:+PrintFlagsFinal* flag:

```
java -XX:+PrintFlagsFinal -version | grep CompileThreshold
intx CompileThreshold = 10000
intx Tier2CompileThreshold = 0
intx Tier3CompileThreshold = 2000
intx Tier4CompileThreshold = 15000
```

We should note that the **JVM doesn't use the generic *CompileThreshold* parameter when tiered compilation is enabled**.

In summary, the JVM initially interprets a method until its invocations reach the *Tier3CompileThreshold*. Then, it **compiles the method using the C1 compiler while profiling information continues to be collected**. Finally, the JVM compiles the method using the C2 compiler when its invocations reach the *Tier4CompileThreshold*. Eventually, the JVM may decide to deoptimize the C2 compiled code. That means that the complete process will repeat.



Typical execution flow

When using the **java/javaw** launcher:

1. Parse the command line options - като например -d
2. Establish the heap sizes and the compiler type (client or server)

3. Establish the environment variables such as **CLASSPATH** (компилираните до bytecode .java класове от нашето приложение – тоест .class видими за нас декодирани byte файлове)
4. If the java Main-Class is not specified on the command line, then fetch the Main-Class name from the JAR's manifest
5. Create the VM using **JNI_CreateJavaVM** (Java Native Interface, на C++) in a newly created thread (non primordial thread)
6. Once the VM is created and initialized, load the Main-Class
7. Invoke the **main** method in the VM using **CallStaticVoidMethod**
8. Once the **main** method completes check and clear any pending exceptions that may have occurred and also pass back the exit status

Detach the main thread using DetachCurrentThread, by doing so we decrement the thread count so the **DestroyJavaVM** can be called safely.

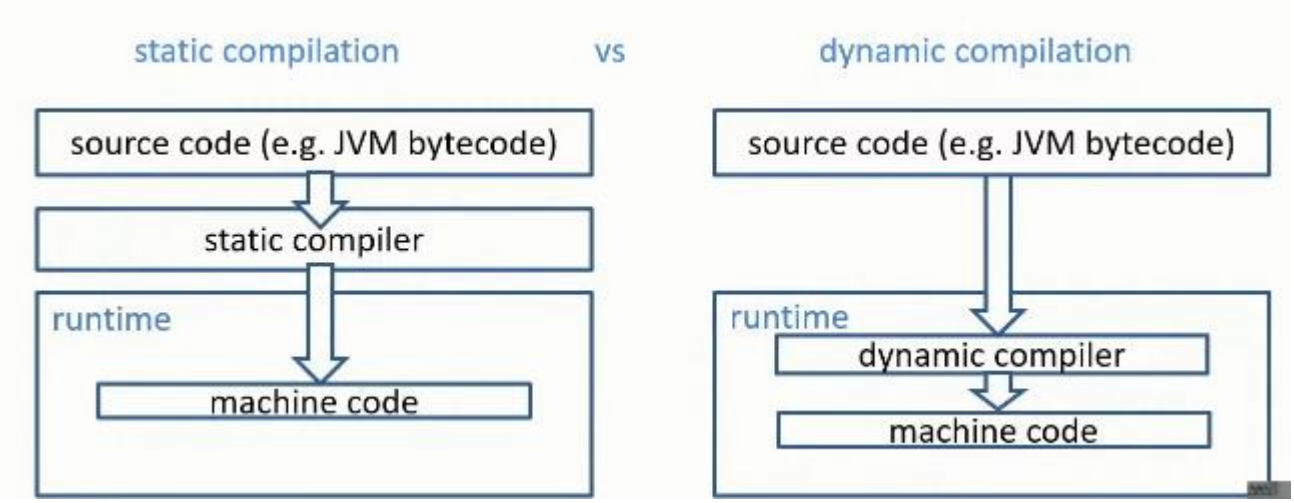
II. GraalVM

Some background

Статична компилация - когато нашия код се компилира до bytecode със **javac** компилатора/командата. И по време на изпълнение средата(JVM например) изпълнява bytecode-а **или** машинния native code.

Динамична компилация - например това го прави JIT компилатора - **изпълнява само машинен код**!

- The JVM performs **static compilation** of Java sources to **bytecode**
- The JVM may perform **dynamic compilation** of bytecode (**from bytecode) to machine code** for various optimizations using a JIT (Just-in-Time) compiler



Essentially, **static** linking involves compiling libraries into your app or program **as part of the build process**.
Dynamic linking lets the operating system hold off and load shared libraries into memory only when the app is launched.

Info

- GraalVM is a new JIT compiler for the JVM
- **Brings the performance of Java to scripting languages (via the Truffle API) - възможността да се използват предимствата на Java за по-бързо интерпретиране/изпълнение на компилирания код в сравнение ако просто даден браузър само интерпретира дадено JavaScript приложение/web страница**
- Written in Java
- GraalVM е нещо средно между клиентски и сървърен JIT компилатор (при стартиране прави повече оптимизации/опреации в сравнение с клиентския JIT, но все пак по-малко

оптимизации/операции в сравнение със сървърния JIT), но усъвършенстван да оптимизира още повече.
- Предназначен за скриптови езици като JS, Ruby
- Под капака GraalVM за някои оптимизации/операции си взаимодейства и с HotSport JVM

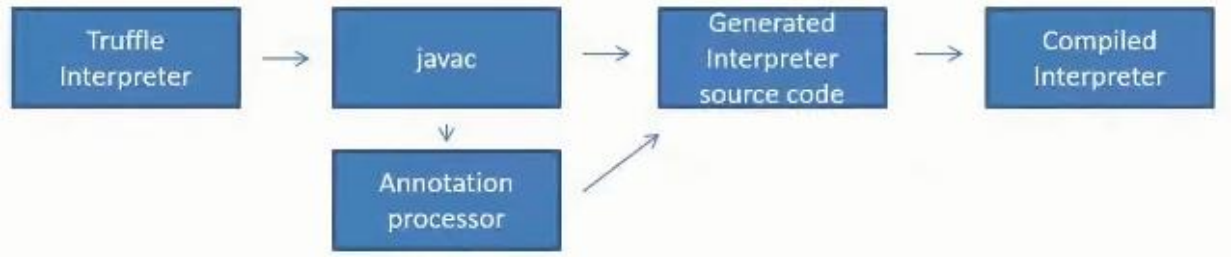
- In essence, the Graal JIT compiler generates machine code from an optimized **AST**(**abstract syntax tree**) rather than bytecode
- However, the Graal VM has both **AST**(**abstract syntax tree**) and bytecode interpreters

- The Graal VM supports the following types of compilers:

–vm server-nograal //server compiler
–vm server //server compiler using Graal
–vm graal //**Graal compiler** using Graal
–vm client-nograal //client compiler
–vm client //client compiler running Graal

The Truffle API

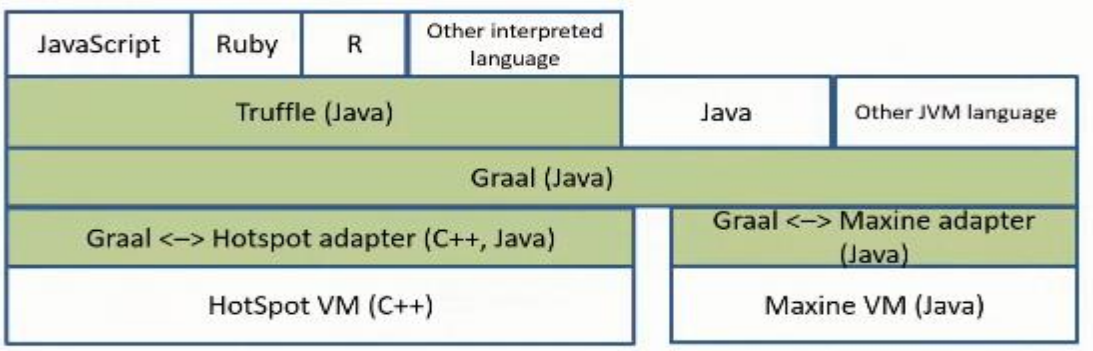
- is a Java API
- Provides AST (Abstract Syntax Tree) representation of source code
- Самото АПИ предоставя възможност за реализация на имплементация на даден програмен език, която имплементация да може да се изпълни от/на GraaVM. За целта различни елементи от нашия интерпретиран език се представят в рамките на една структура (предоставена от Thruffle API-то) и тази структура се използва от GraalVM за да се генерира т.нар. **AST (Abstract Syntax Tree)**
- Provides a mechanism to convert the generated AST into a **Graal IR (intermediate representation)**
- Възможно е и от bytecode-а да стигнем до **Graal IR (intermediate representation)**
- The Truffle API is declarative (uses Java annotations)
- The AST graph generated by Thruffle is a mixture of **control flow**(условни структури, for цикли, и т.н.) and **data flow graph** (местата през които да преминат данните в рамките на нашето приложение)
- Essential feature of the Truffle API is the ability to specify node specializations used in node rewriting
- The Truffle API is used in conjunction with custom annotation processor that generates code based on the Truffle annotations used in the interpreter classes.



Ruby & RubyJ пример - бекенд зарежда в пъти по-бързо през TruffleApi-то на RubyJ

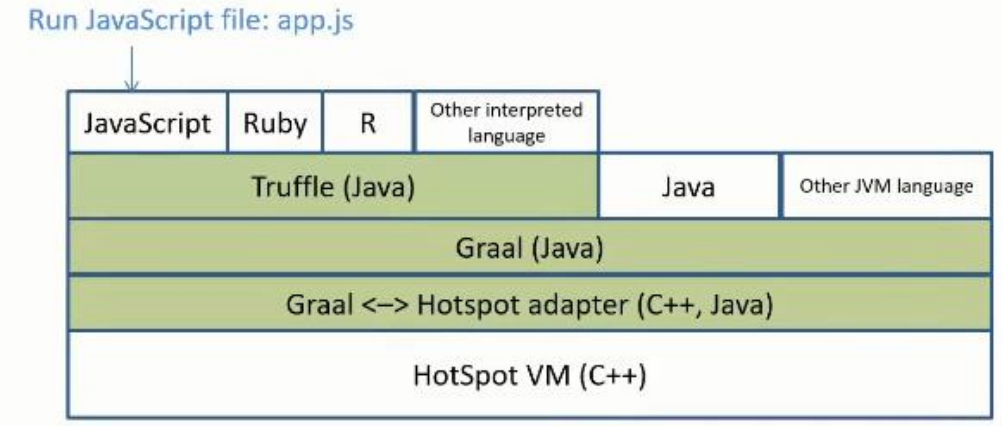
А какво прави Java компилацията до bytecode и евентуално до native machine code? - като види цикъл с 5 и го прави на 5 метода. И по-бързо се изпълнява. Та и затова като мине през TruffleApi и е в пъти по-бързо отколкото ако се интерпретира. **Но тук говорим за бекенда на дадения скриптов език**.

General structure of GraalVM



Let's see, for example, how a JavaScript interpreter works in Graal …

1. Run JavaScript file: app.js
2. Подаваме app.js за изпълнения от GraalVM - JavaScript Interpreter parses app.js and converts it to Truffle **AST (Abstract Syntax Tree).** Трябва да имаме интерпретатор (който сме написали), който да парсне този app.js файл до AST - графова структура
3. The Thruffle AST is converted into a new graph - т.нар. **Graal IR** AST **(intermediate representation)**
4. The Graal VM has bytecode and AST interpreter**s** along with a standard JIT compiler from HotSpot (optimizations and AST lowering is performed **to generate machine code** and perform partial evaluation)
5. When parts of app.js are compiled to machine code by the Graal compiler, then the compiled code is transferred to HotSpot for execution by means of HotSpot APIs and with the help of a **Graal-HotSpot adapter interface**
6. Compiled code can also be deoptimized and **control is transferred back to the interpreter** (from HotSpot VM back to Graal VM - e.g. when an exception occurs, an assumption fails or a guard is reached)



Optimizations done on the Graal IR (intermediate representation) include:

- Method inlining
- Partial escape analysis
- Inline caching
- Constant folding
- Arithmetic optimizations and others…

Реално такъв вид оптимизации могат да бъдат направени и от стандартното HotSpot JVM!

Currently supported languages from Graal VM include:

- JavaScript - (Graal.JS)
- Език R - за статистически изчисления се използва най-вече - (FastR)
- Ruby - (RubyTruffle)
- Experimental interpreters for C, Python, Smalltalk, LLVM IR and others

- Graal.JS - shows improvement of 1.5x in some cases with a peak of 2.6x compared to V8 (running Google Octane's benchmark) - https://github.com/oracle/graaljs

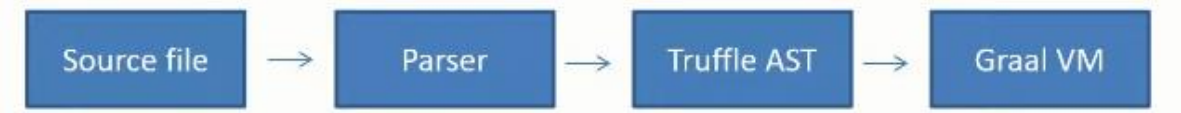
- RubyTruffle - shows improvements between 1.5x and 4.5x in some cases with a peak of 14x compared to JRuby

- FastR - shows improvements between 2x and 39x in some cases with a peak of 94x compared to GnuR

SimpleLanguage project

https://github.com/graalvm/simplelanguage

The SimpleLanguage project provides a showcase on how to use the Truffle APIs



III. Garbage collection

Info

- Garbage collection is the process of scanning heap memory for unused objects and cleaning them thus reclaiming memory
- Garbage collection in the JVM is implemented by means of special modules called garbage collectors
- Garbage collectors are optimized to work over different heap areas (**young and old generations**)

- Whenever the young generations (minor collections) fills up or the old generation (major collections) need to be cleaned **a "Stop The World" event** appears
- In a "Stop The World" event application threads are paused during garbage collection
- In that regard garbage collection might be disruptive for performance in some high-frequency applications, i.e. in fintech. **Затова за такива приложения даже се елиминира изцяло процеса по garbage collection-а!**

Обекти, за които JVM прецени че живеят по-дълго от очакваното се местят от Young generation към Old / Tenured Generation!

Обектите от Young generation се изчистват по-лесно и бързо от garbage collector-а, без да се прави задълбочен анализ от garbage collector-а.

Чистенето на обекти от Old / Tenured Generation става чрез паузиране, което за някои приложения е ключово. Затова трябва да се стремим чистенето на обекти от Old / Tenured Generation да бъде минимално по дължина време!!!

Types of garbage collectors

Different types of Java collectors exist at present and **it is an area of active research and development**:

-XX:-UserSerialGC - в допълнение към основната терминална команда когато стартираме приложението - можем да зададем какъв garbage collector да използва JVM.

- **Serial GC**: garbage collection is done **sequentially**

```
-XX:-UseSerialGC
```

- **Parallel GC**: uses **multiple threads** for garbage collection (default in JDK 7 and 8):

```
-XX:-UseParallelGC
```

```
-XX:-UseParallelOldGC
```

- **Concurrent Mark Sweep (CMS)**: works concurrently along the application threads trying to minimize pauses - самият garbage collector ползва няколко нишки И отделно от това тези нишки работят паралелно върху/по различните thread-ове/нишки от нашето приложение.

```
-XX:-UseConcMarkSweepGC
```

- **G1:** available as of JDK 7, parallel, concurrent and incrementally compacting low-pause collector (**default in JDK 9 and later**)

```
-XX:+UseG1GC
```

- **Epsilon GC:** this is a no-op/null garbage collector introduced in JDK 11 that allocates memory but does not do garbage collection - използва се например при тези fintech приложения където все пак трябва да се зададе като настройка тип колектор, и в случая колектора прави всичко останало, но без реалното събиране на боклука от паметта!!!!

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseEpsilonGC
```

- **ZGC:** also introduced in JDK 11, still experimental, tries to reduce pause times in large scale Java applications

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseZGC
```

- **Shenandoah GC:** available in JDK 13 and later (also in some earlier JDK versions), aims to reduce pause times by working concurrently with the application

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseShenandoahGC
```

Като цяло, всеки по-нов garbage collector цели да се справя по-добре в случаите когато има а “Stop The World” event.

12. Jenkins

- Jenkins is an open-source automation server that helps automate various tasks in software development and delivery processes
- It provides a robust and extensible platform for continuous integration (CI) and continuous delivery (CD)

- Решава същата задача като GitHub actions, CI & CD на GitLab, на GitBucket

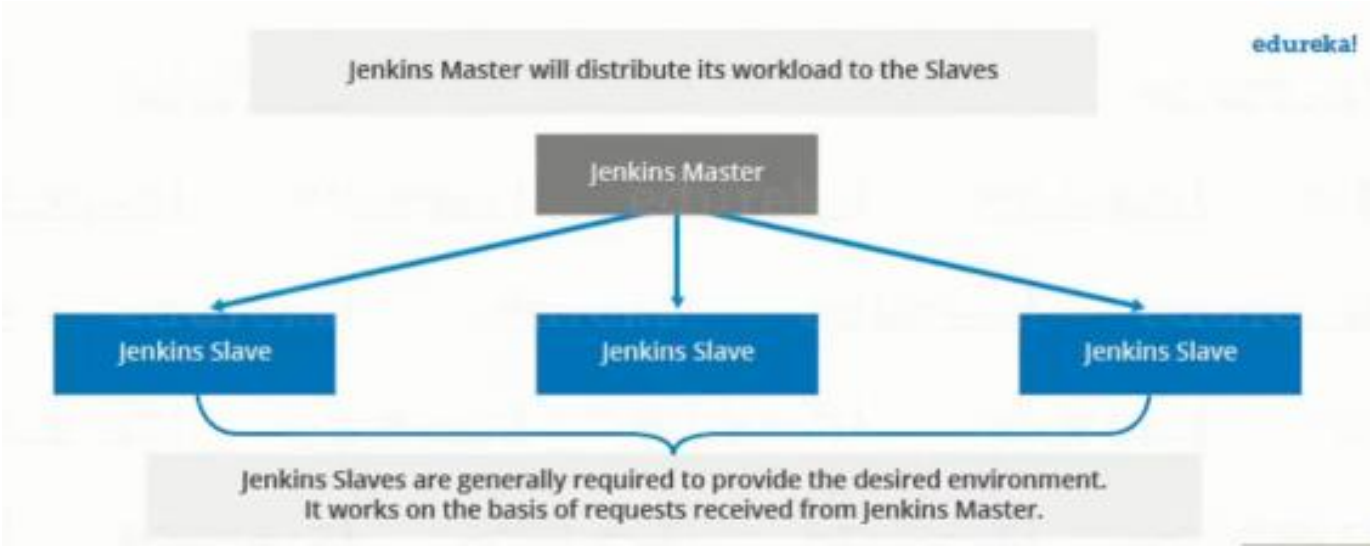
Key features

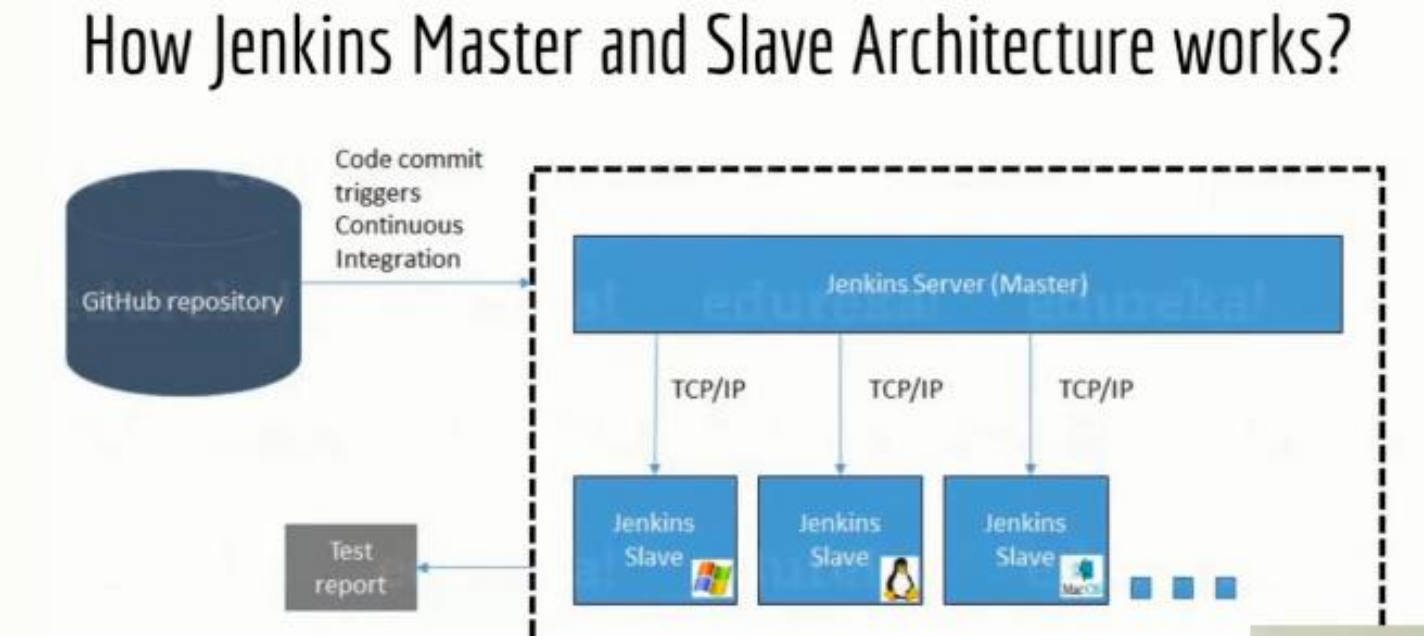
- **Continuous Integration (CI)**: Jenkins enables developers to integrate code changes regularly, ensuring early detection of issues and improving collaboration
- **Continuous Delivery (CD)**: Jenkins facilitates the automated deployment of applications to various environments, streamlining the release process
- **Extensibility**: Jenkins offers a vast ecosystem of plugins that extend its functionality, allowing integration with different tools and technologies.
- **Distributed Architecture:** Jenkins supports distributed builds, allowing for efficient resource utilization across multiple machines or agents - може на повече от една машина да се изпълняват тези pipelines (ако една компания има 500 човека, то не може един човек да чака 4 часа да му мине билда!)

Use cases

- Continuous Integration and Testing: Jenkins can build, test and validate code changes automatically, ensuring a stable and reliable software product
- Continuous Delivery and Deployment: Jenkins enables the automated deployment of applications to multiple environments such as development, staging and production
- Task Automation: Jenkins can automate various repetitive tasks such as generating reports, sending notifications, and scheduling jobs.

How Jenkins work (Master - Slave)





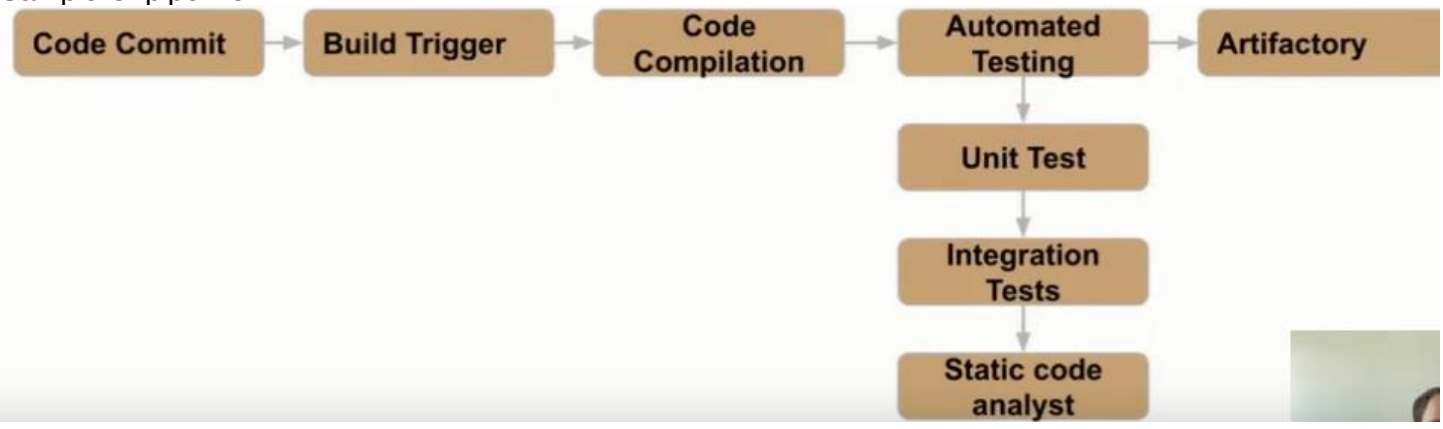
В случая когато ще използваме Java програмен език, то JVM машината веднъж инсталирана/конфигурирана тя работи на всяка операционна система.
Обаче ако проекта ни е с JS/ReactJS, то има в по-голяма степен значение slave-а ни на коя ОС е нагласен!

CI & CD

What is Continuous Integration (CI)?

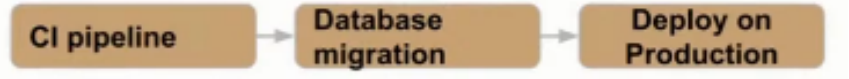
- Continuous Integration **pipeline** is a development practice where developers frequently integrate their code changes into a shared repository
- The integration process is automated and accompanied by various checks and validations
- Could be executed on 1 or more Jenkin slave nodes simultaniously

Sample CI pipeline:



What is Continuous Delivery (CD)?

- Continuous Delivery is a software development approach that focuses on automating and streamlining the software release and deployment processes
- It aims to deliver software changes more frequently, reliably, and with reduced manual effort
- Надгражда CI pipeline-а като добавя някои от следните неща: миграция на база данни, деплойване на съответния docker image на съответната среда

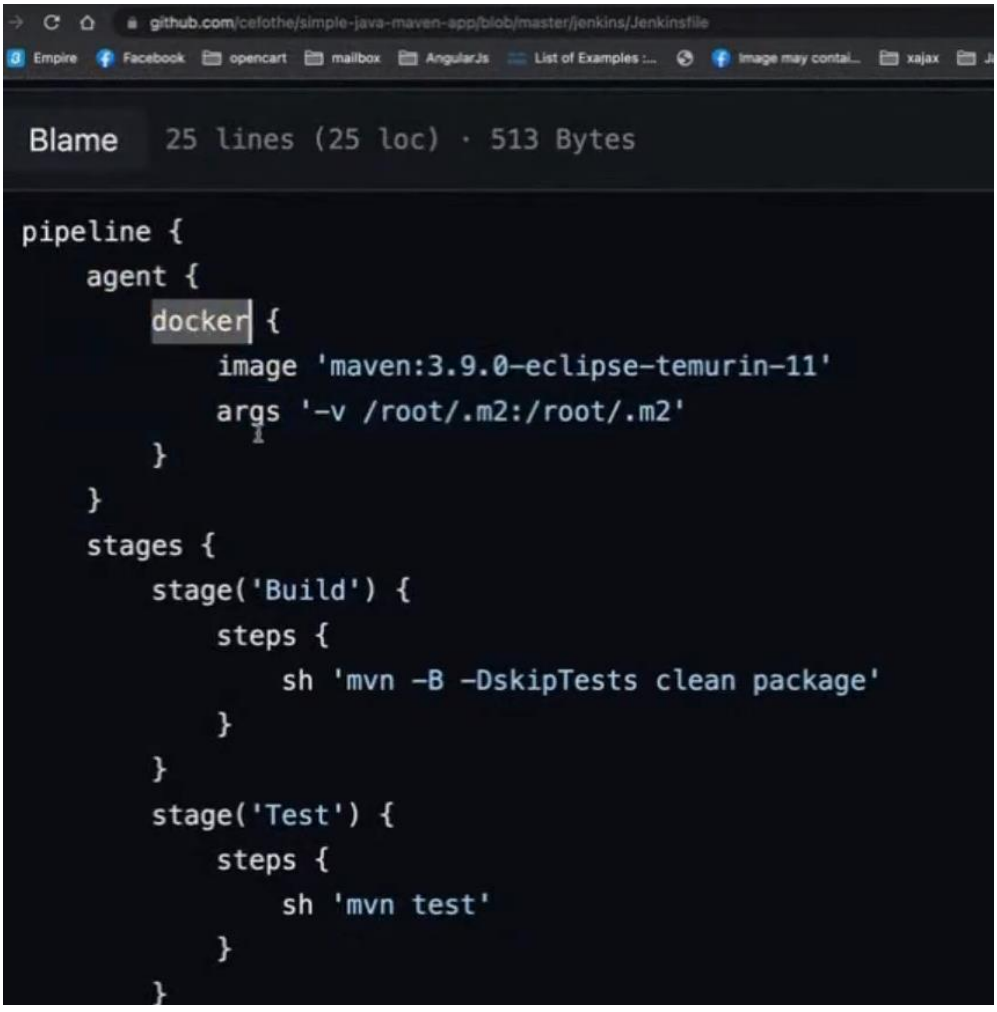


How Jenkins file will look like

Sample of a Jenkins file

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

agent Java, Maven, etc.



```
stage('Deliver') {
    steps {
        sh './jenkins/scripts/deliver.sh'
    }
}
```

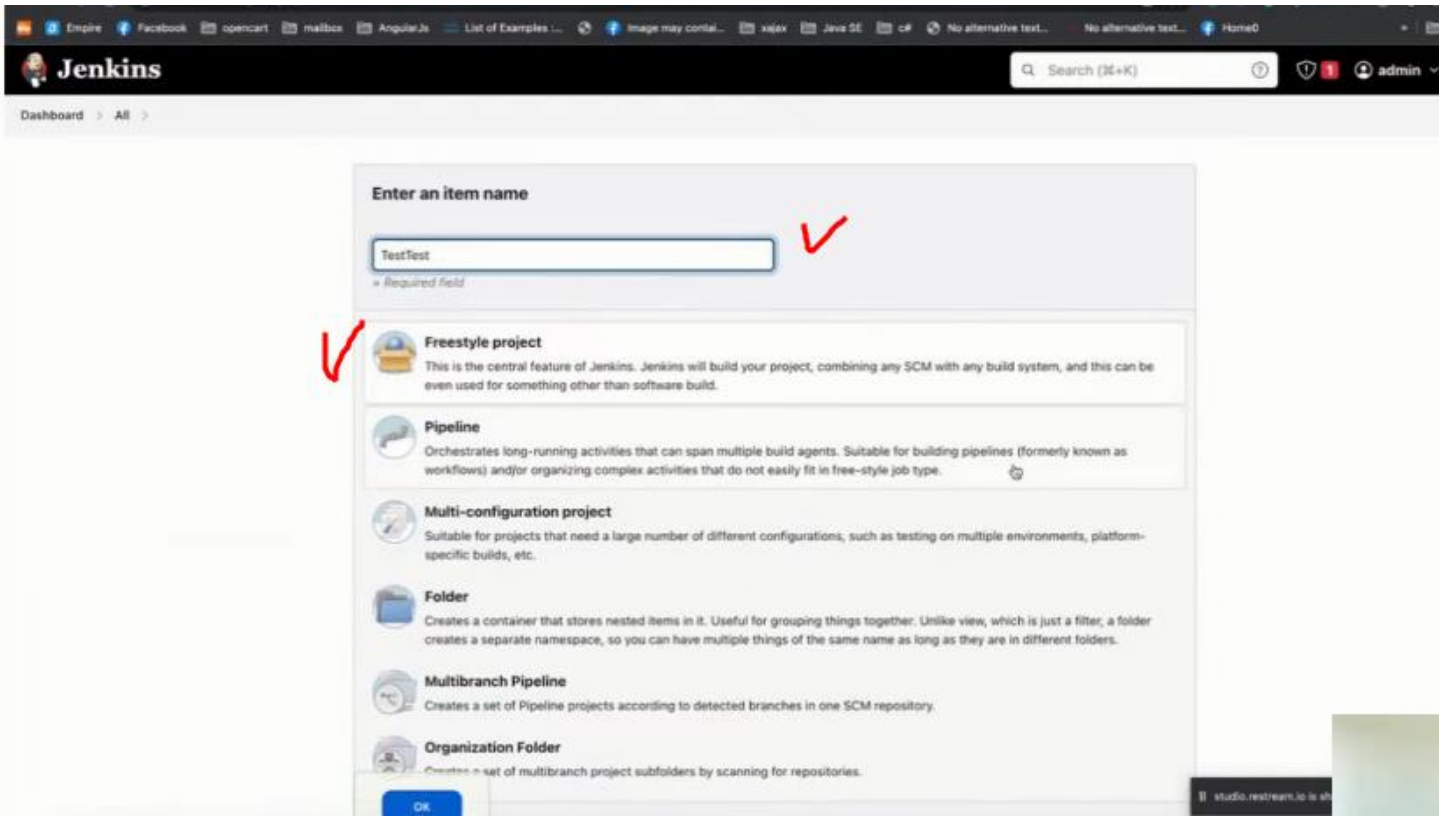
Installing Jenkins java app with maven

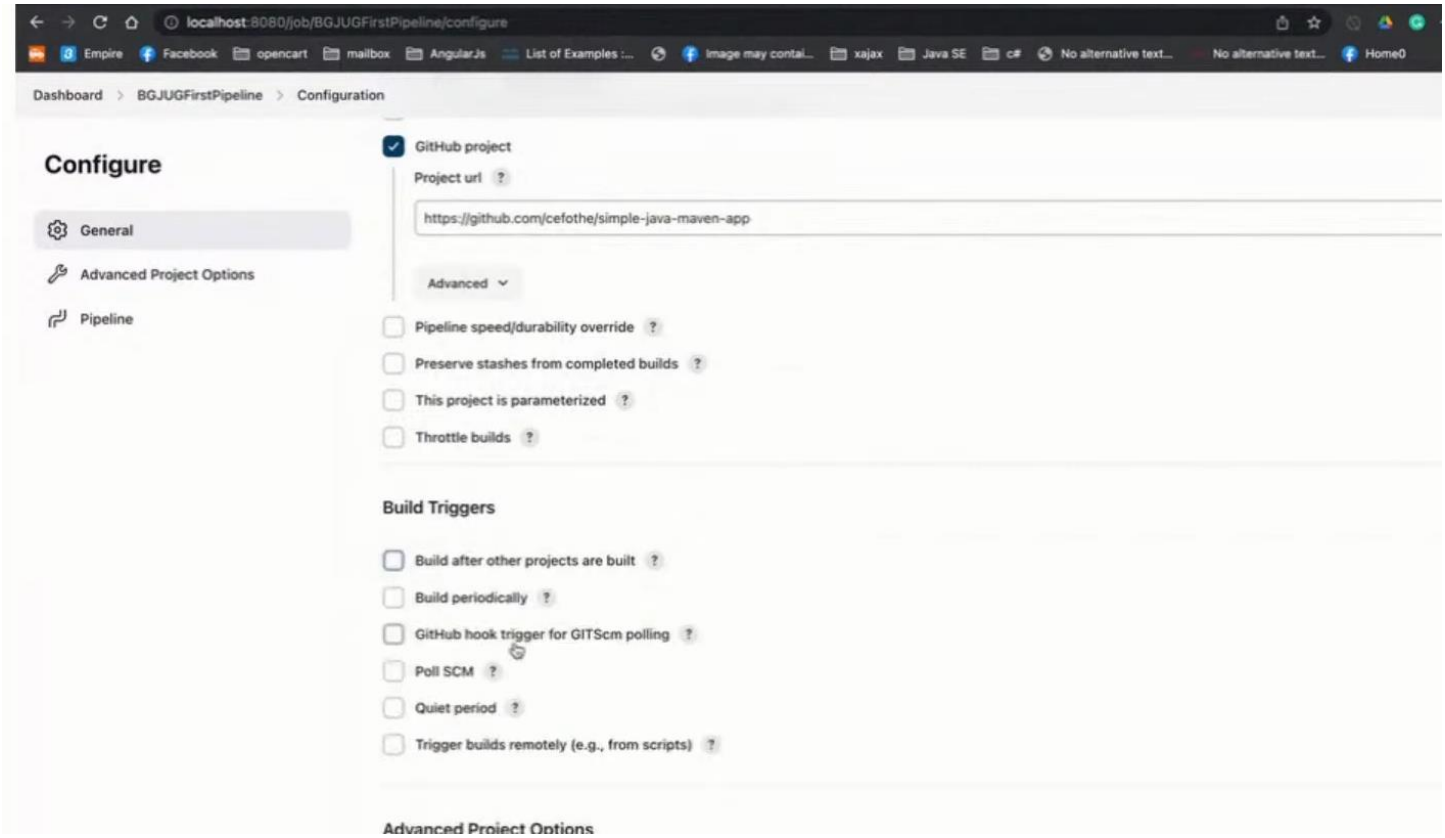
How to run Jenkins locally:
https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/

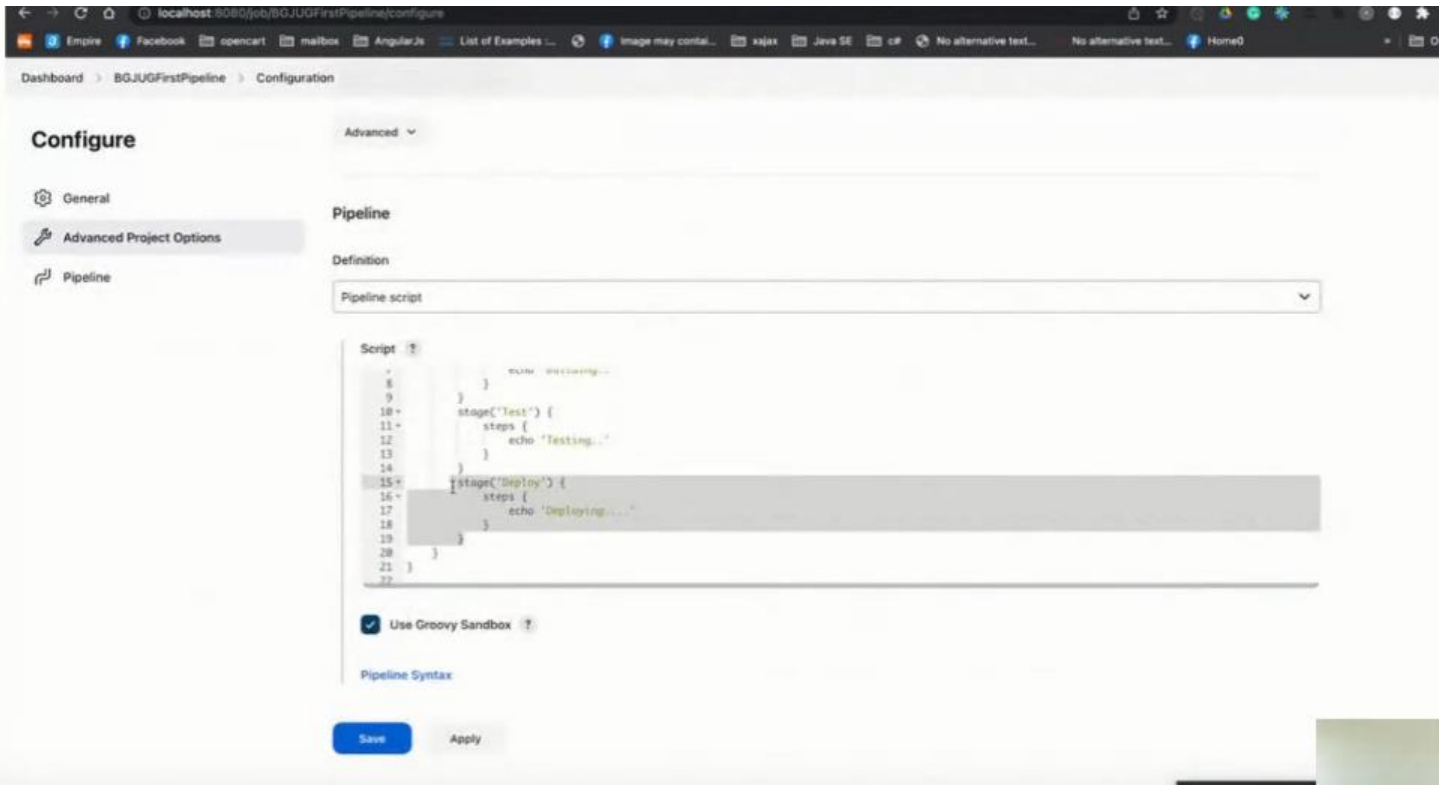
Тази инсталация посочена в линка работи в docker контейнер, като преди да започнем да работим с този контейнер, то имаме инсталиран в него JDK and Maven.

В други записки има примери как това става даже по-лесно ако работим с JS.

Creating Freestyle project

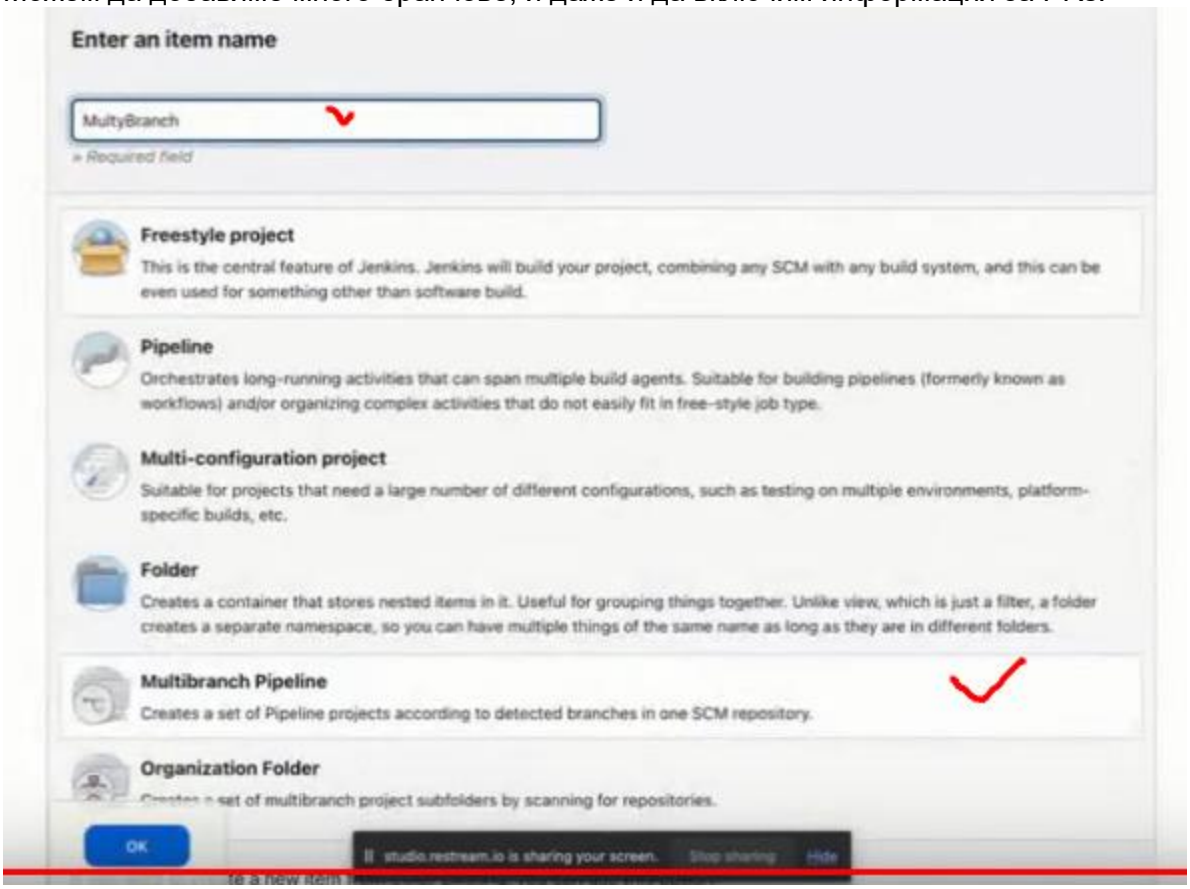


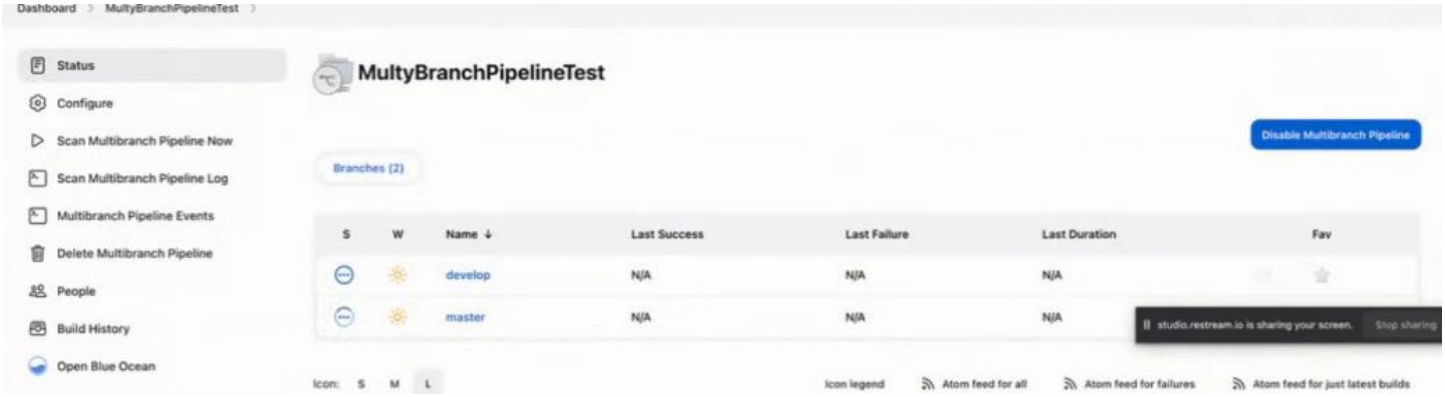

localhost:8080/job/BGJUGFirstPipeline/configure
Dashboard > BGJUGFirstPipeline > Configuration
Configure
General
Advanced Project Options
Pipeline
GitHub project
Project url
https://github.com/cefothe/simple-java-maven-app
Advanced
Pipeline speed/durability override
Preserve stashes from completed builds
This project is parameterized
Throttle builds
Build Triggers
Build after other projects are built
Build periodically
GitHub hook trigger for GITScm polling
Poll SCM
Quiet period
Trigger builds remotely (e.g., from scripts)
Advanced Project Options


Dashboard > BGJUGFirstPipeline > Configuration
Configure
General
Advanced Project Options
Pipeline
Advanced
Pipeline
Definition
Pipeline script
Script
stage('Test') {
steps {
echo 'Testing..'
}
}
stage('Deploy') {
steps {
echo 'Deploying....'
}
}
Use Groovy Sandbox
Pipeline Syntax
Save
Apply

Creating Multibranch pipeline

Можем да добавяме много бранчове, и даже и да включим информация за PRs.





13. Github actions

https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions
https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven
https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-gradle

What is Git Actions

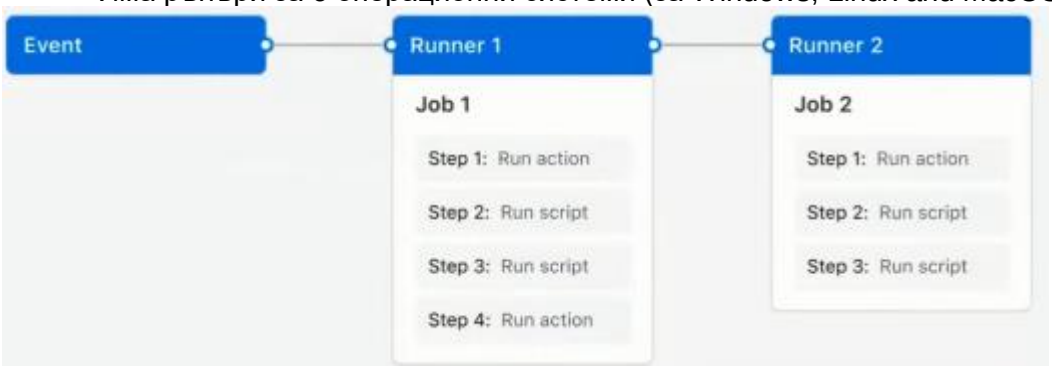
GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

Също така може да си добавим/плъгнем и допълнителни actions като например cron jobs.

The components of GitHub Actions

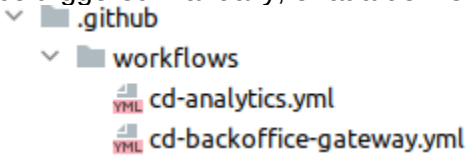
Тригърване на ивента ръчно - например при Jenkins ръчно при **build**-а. Но може да става и автоматично тригърването на ивента.

Има рънъри за 3 операционни системи (за Windows, Linux and MacOS) – като при Jenkins.



Workflows

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked into your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.



Events

An event is a specific activity in a repository that tirggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository.

Jobs

A job is a set of steps in a workflow that is executed on the same runner. Each step is either a shell script that will be executed, or an action that will be run. Steps are executed in order and are dependent on each other. Since each step is executed on the same runner, you can share data from one step to another. For example, you can have a step that builds your application followed by a step that tests the application that was built.

Actions

There are ready pre-built actions in GitHub. They are similar to the Jenkins plugins.

An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. An action can pull your git repository from GitHub, set up the correct toolchain for your build environment, or set up the authentication to your cloud provider.

Runners

A runner is a server that runs your workflows when they are triggered. Each runner can run a single job at a time. GitHub provides Ubuntu Linux, Microsoft Windows, and MacOS runners to run your workflows. Each workflow run executes in a fresh, newly-provisioned virtual machine.

Пример за ci или cd .yml файл:

build-java-with-maven.yml

```yaml
name: Build with Java and Maven
run-name: Java with maven
#on: [push]
on:
 push:
   branches:
     - 'develop'

 pull_request:
   branches:
     - 'master'

jobs:
 build-with-maven:
   runs-on: ubuntu-latest
   steps:
     - run: echo "Starting job"
     - name: Checkout repository
       uses: actions/checkout@v3

     - uses: actions/setup-java@v3
       with:
         java-version: '11'
         distribution: 'temurin'
         cache: maven

     - name: Build with maven
       run: mvn clean install
```

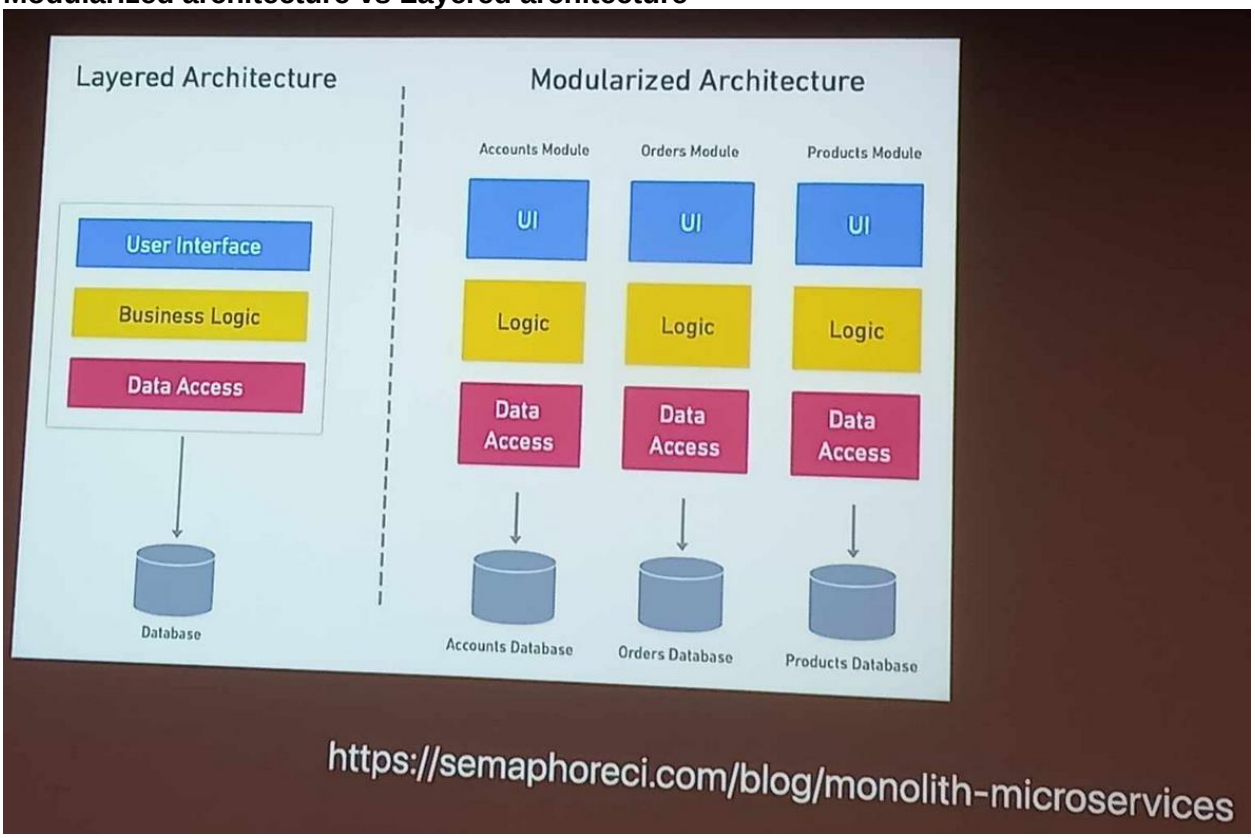
14. Architecture and microservices

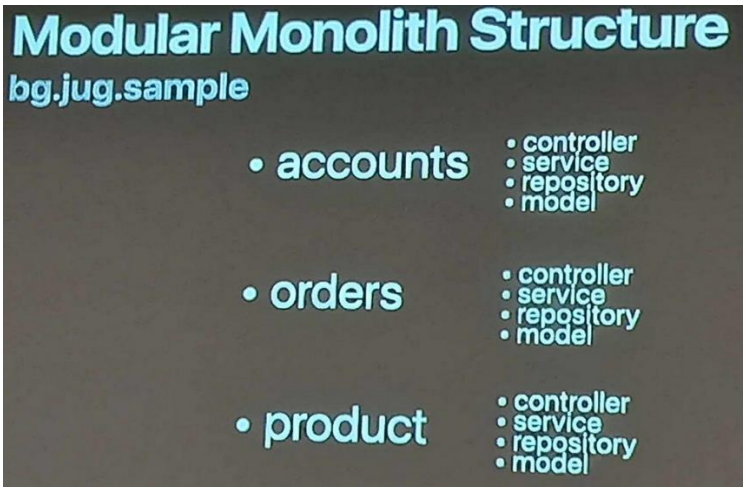
Cohesion - same matter at one place in the code
Coupling - interacting one another

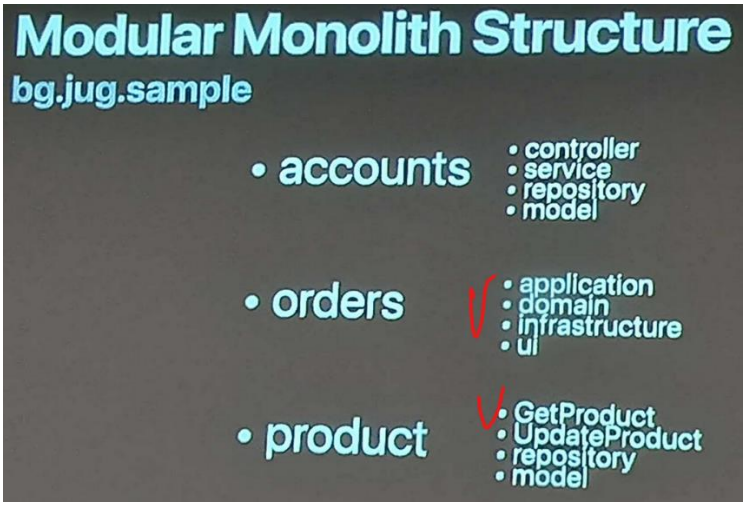
Loose coupling and high cohesion make it a microservice.
High coupling makes it a monolith.

Eventual consistency - I ask the bank do I have money, and the bank answers don't know, maybe yes, maybe u can buy that thing. Who knows?

Modularized architecture vs Layered architecture







Rober “Uncle Bob” Martin: **Your architecture should tell readers about the system, not about the frameworks you used in your system!**

Screaming architecture - to scream what it is