Introducing Virtual Threads

José Paumard

PhD, Java Champion, JavaOne Rockstar

@JosePaumard | https://github.com/JosePaumard

Why virtual threads have been created

Module Agenda

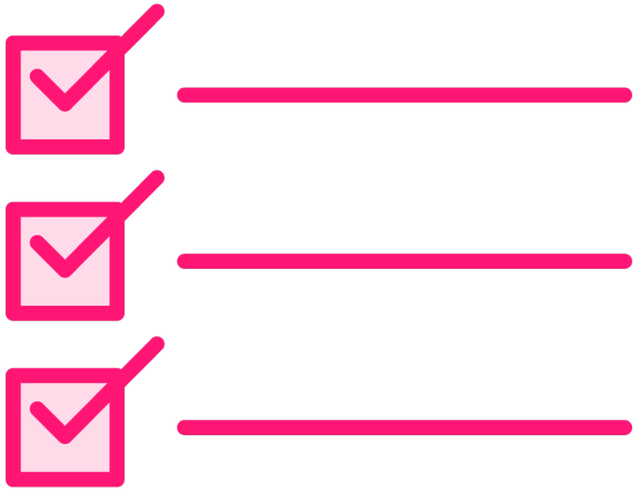
Why do you need virtual threads?

How reactive programming is working

What is the cost of the reactive approach?

Your first virtual threads in action

Handling Requests Concurrently



You want to execute several requests concurrently

To keep your CPU busy

One request per thread is not a solution

Because a thread is too expensive

```
HttpClient client = ...;
HttpRequest request = ...;

var response =
    client.send(request, ...);

String body = response.body();
```

- This is an in-memory processing, it does not block your thread
- This is a request on the network that blocks your thread
- This is again an in-memory processing

```java
Callable<String> task = () -> {
  HttpClient client = ...;
  HttpRequest request = ...;

  var response =
    client.send(request, ...);

  String body = response.body();
  return body;
};

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

String result = future.get();
```

- This is an in-memory processing, no code is executed when you create a Callable or a Runnable

- Submiting this taks to an executor triggers its execution in a thread from the executor. It blocks this thread
- Calling get() blocks your main thread

Giving several requests to one thread

Is about splitting a request

Into small operations

And wiring them together

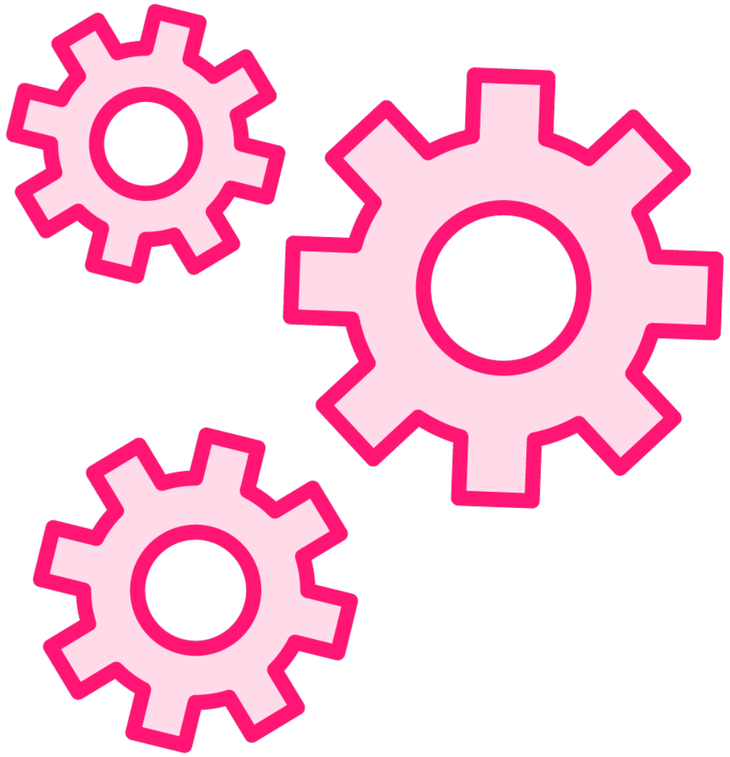
Using an ad-hoc framework

```
Supplier<Request> step1 =
  () -> {
    HttpRequest request = ...;
    return request;
  };

Function<Request, Response> step2 =
  request -> {
    var client =...;
    return client.send(request, ...);
  };

Function<Throwable, Response> step3 =
  throwable -> {...};

Function<Response, ...> step4 =
  response -> response.body();
```

- Create your request object.
- Send the request to the server.
- Handle the exceptions.
- Analyze your result.

```
CompletableFuture.supplyAsync(
  () -> {
    HttpRequest request = ...;
    return request;
  })

.thenApply(
  request -> {
    var client =...;
    return client.send(request, ...);
  })

.exceptionally(
  throwable -> {...})

.thenApply(
  response -> response.body())

.get();
```

- Creating a CompletableFuture pipeline can begin by passing a supplier that creates the request object.
- Then a second operation takes this object and uses it to send the request on the network .
- If an exception is thrown, you can use this method to handle it .
- If a response is available then this function can handle it .
- Calling get() is a blocking call, but there are other solutions to handle the result.



All the tasks you create are non-blocking

Instead of blocking, they should return

When they are done, they free the thread that is running them

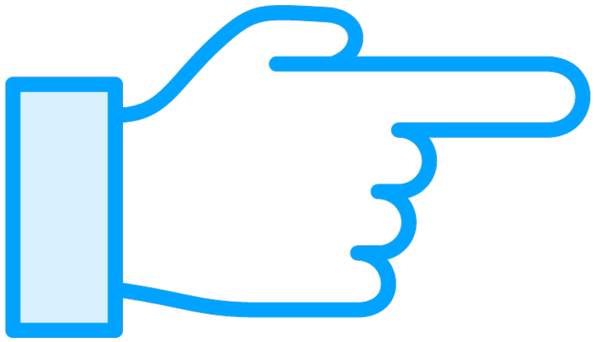
How does CompletableFuture know that the data is available?

It registers a handler on the O/S signal

When the signal is triggered, it calls your lambda with the data

So no thread is ever blocked

As long as your lambdas are non-blocking



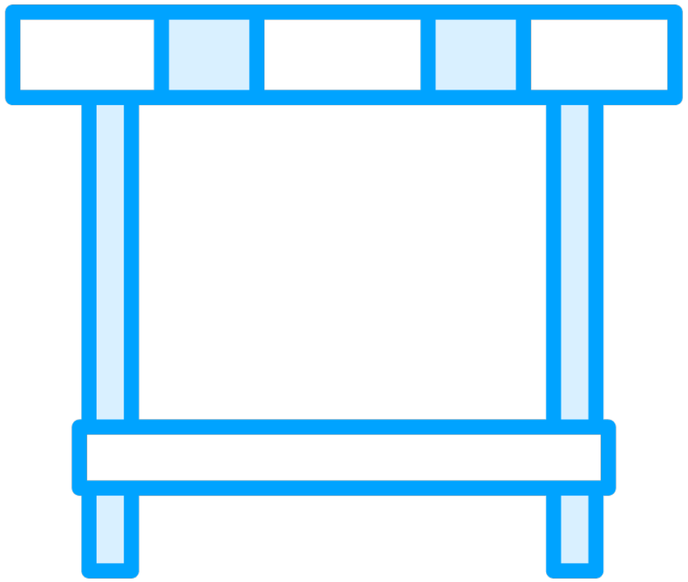
How many platform threads do you need for this system to work?

As few as possible

Some systems are working with a single platform thread

One thread per core works well

Reactive Programming Drawbacks



Reading and writing this code is hard

Writing unit tests is hard

Debugging is hard

Error handling is hard

Profiling is impossible

The maintenance cost is very high!

Reading and writing
this code is hard

Reactive programming gives
good performances

But with drawbacks

Maintenance cost is high

The goal of virtual threads is to have
the best of both worlds:

Performance and a simple programming model

Building a New Model of Thread

Reactive programming works with platform threads

With a lighter model of thread

You could avoid reactive programming

And keep your code simple

The Goal of Virtual Threads

The goal of Virtual Threads is to build a model of thread that is lighter than the platform threads

So that you can run your blocking tasks using them, without having to use reactive programming

How Much Lighter?

The goal is to be able to run 1 million tasks concurrently to keep your CPU busy

Cost of Platform Thread

Memory:	1 MB
Start-up time:	1 ms
Context switching:	0.1 ms

With 10 cores and 10 platform threads, each thread needs to handle 100k tasks

With 1 task per virtual thread, you need 1 million virtual threads to avoid having to use reactive programming

How Much Lighter?

The goal is to launch 1 million virtual threads on a regular machine

Cost of Platform Thread

Memory:	1 MB
Start-up time:	1 ms
Context switching:	0.1 ms

You can launch in the order of several thousand of them

To be able to launch 1 million virtual threads, where you can have 1000 platform threads, 1 virtual thread need to be 1000 times lighter than a platform thread

How Much Lighter?

The target performance of a virtual thread is the following

Cost of Platform Thread

Memory:	1 MB
Start-up time:	1 ms
Context switching:	0.1 ms

Target cost of Virtual Thread

Memory:	1 kB
Start-up time:	1 micro sec

Virtual Threads in Action

Let us create a first virtual thread

And see how it works

Module Wrap Up

The two strategies to handle your network requests, while keeping your CPU busy:

Reactive programming

Virtual threads

How to launch a virtual threads

And what's in it for you in the API

Up Next:

Using Virtual Threads to Increase Your Throuhput