Creating Performant Asynchronous Pipelines with CompletionStage

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

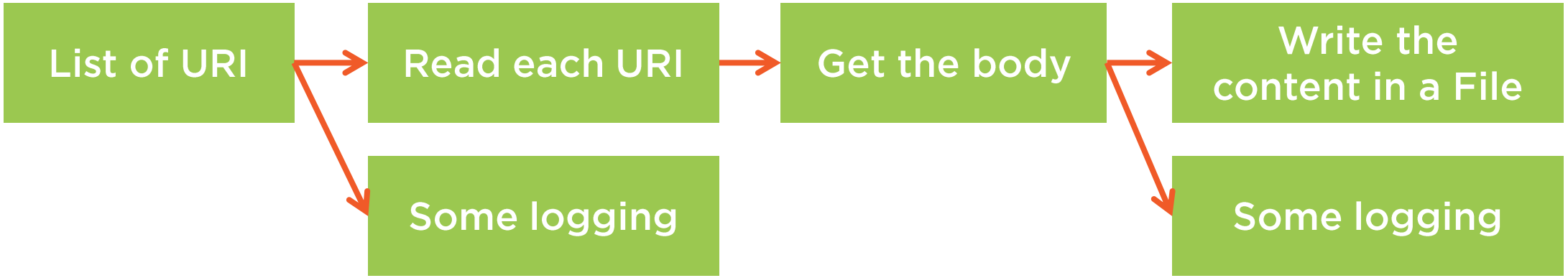
@JosePaumard https://github.com/JosePaumard

Agenda

- Let us talk about performance!
- Hints on how to write performant pipelines
- Understanding the multi-threaded nature of the CompletableFuture API
- How to control data transfer across threads

The HttpClient Example

An HTTP Request



This example uses a Java 10 incubator module

So to run it, you need this version

```
HttpClient client = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_1_1)
                .build();

HttpRequest request = HttpRequest.newBuilder()
                .GET()
                .uri(URI.create("https://somesite.com"))
                .build();

HttpResponse response = client.send(request,
                 HttpResponse.BodyHandler.asString());
```

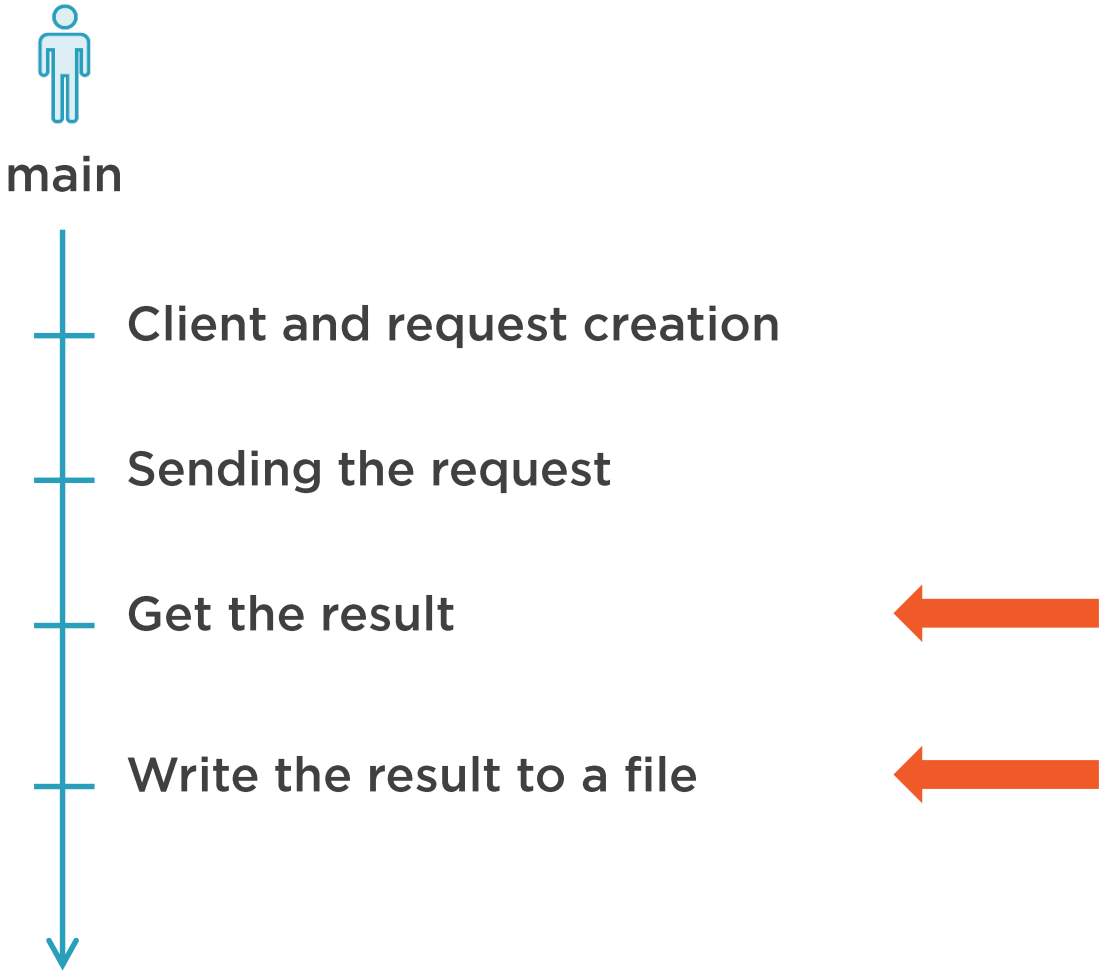
Creating an HTTP Client is really easy, and works in HTTP 2

Then we need a request

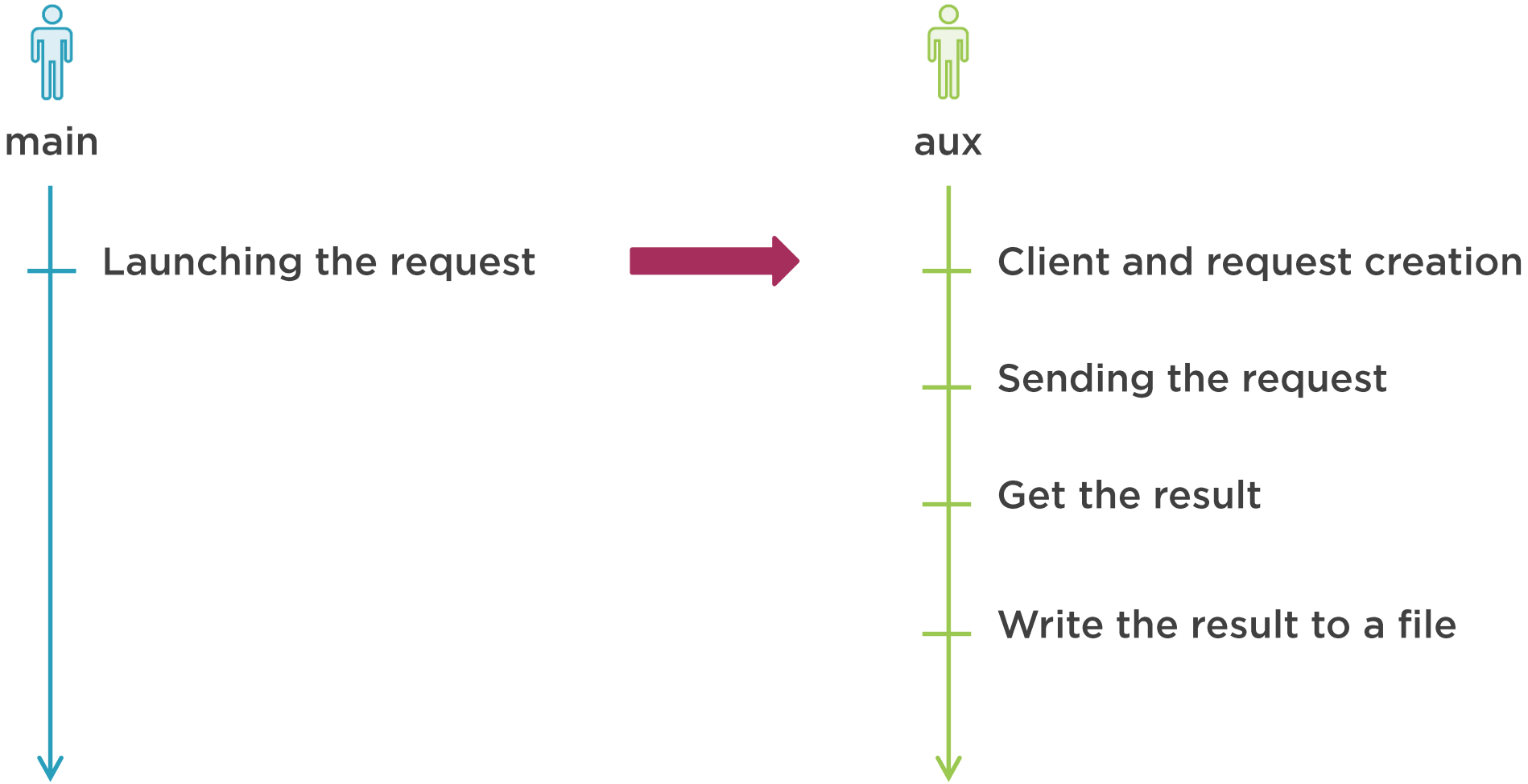
And from that request, get a response

There are several ways of storing the response, in memory or not

Analyzing the HTTP Request



Analyzing the HTTP Request



From a technical point of view:

Can be done with the Executor pattern

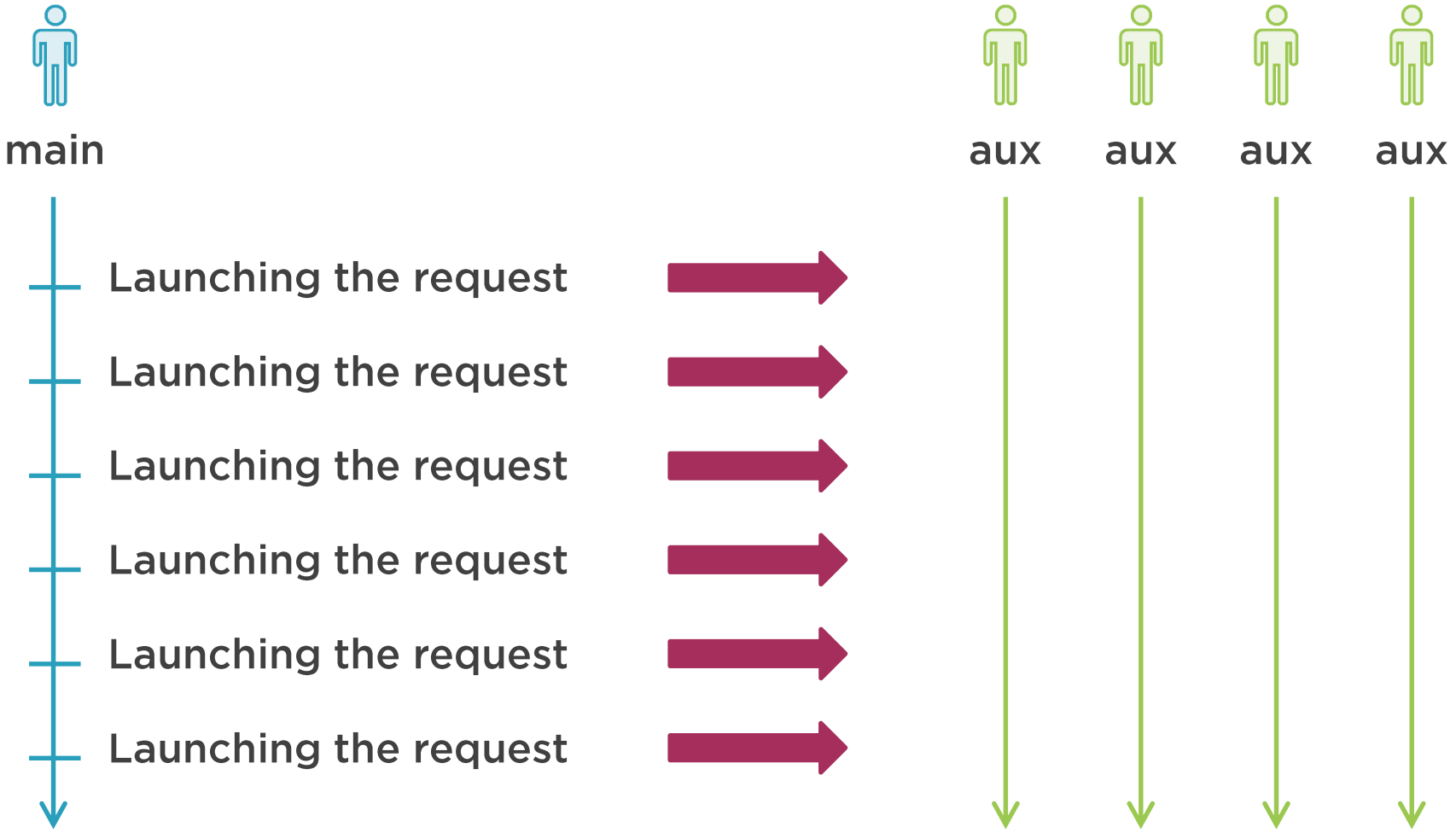
Should be done with the CompletableFuture pattern

The question is: what is the cost?

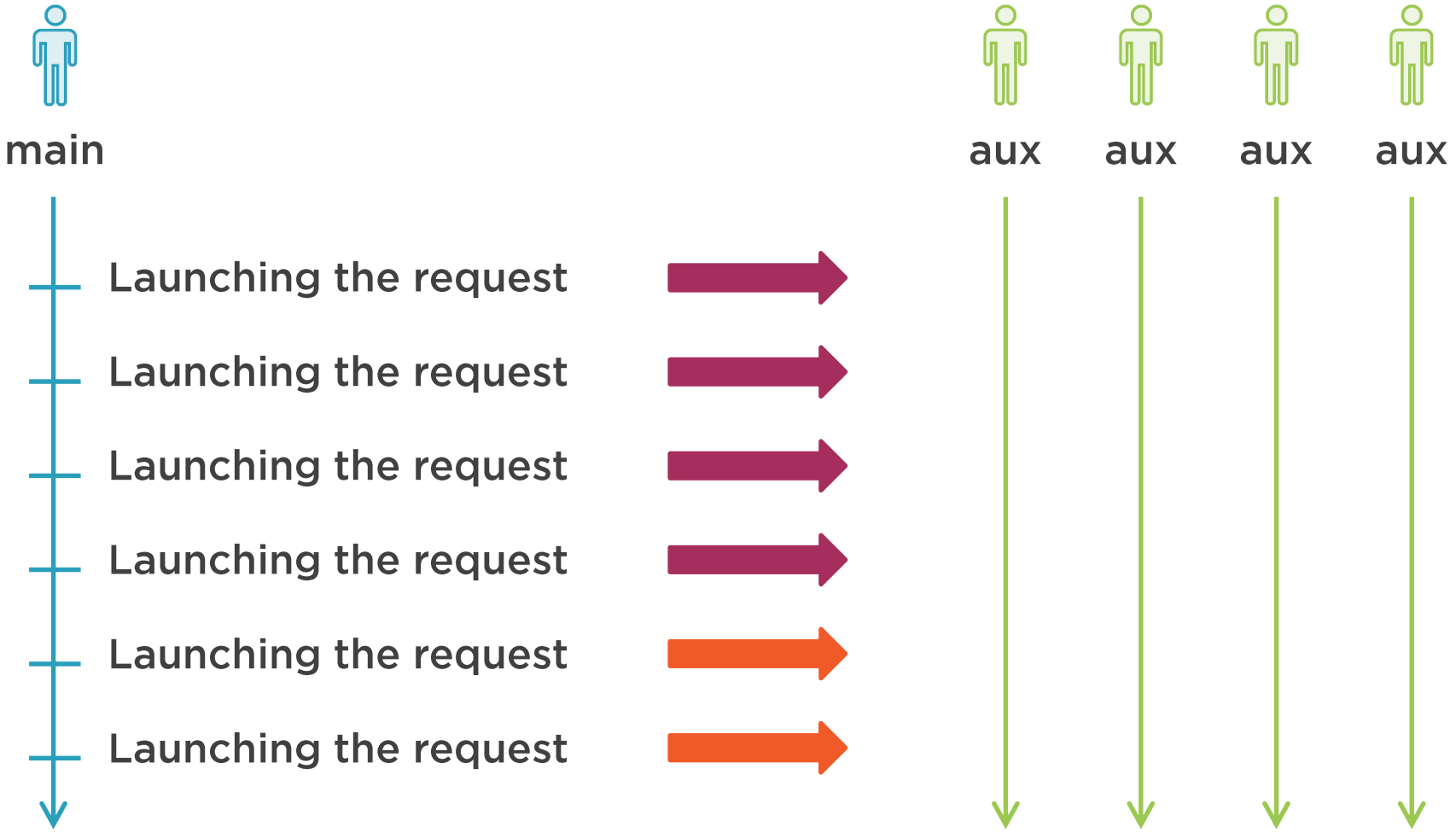
The obvious part: data has to be moved from one thread to the other

Is there another part?

Analyzing the HTTP Request



Analyzing the HTTP Request



You can use the cached thread pool

That creates threads on demand

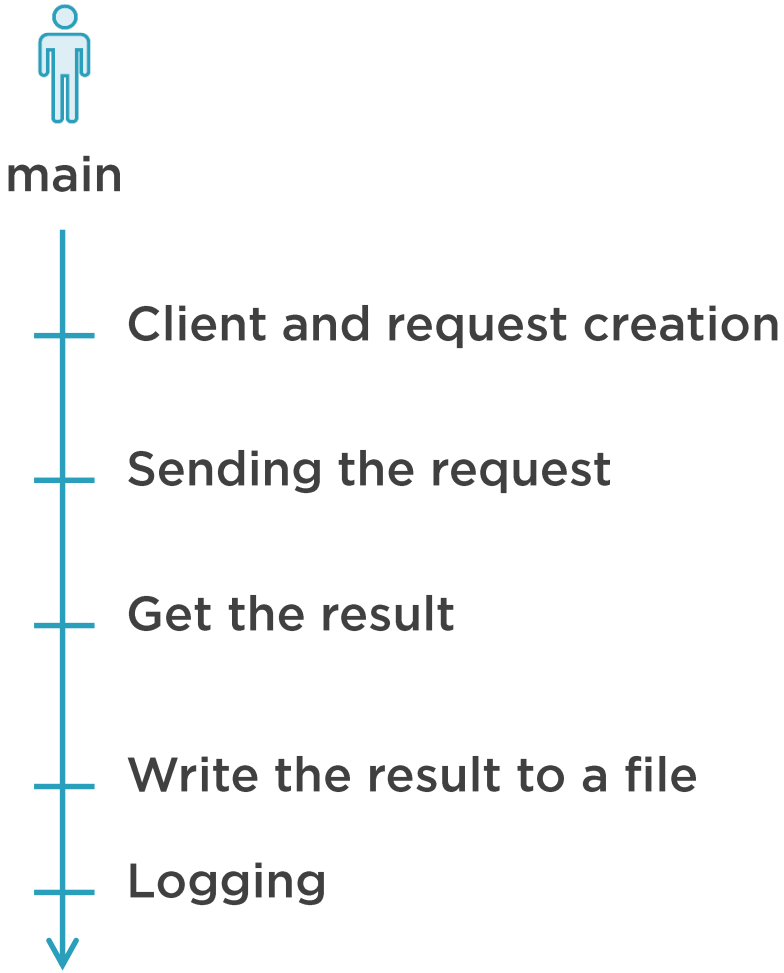
And free the unused ones

But it has a cost!

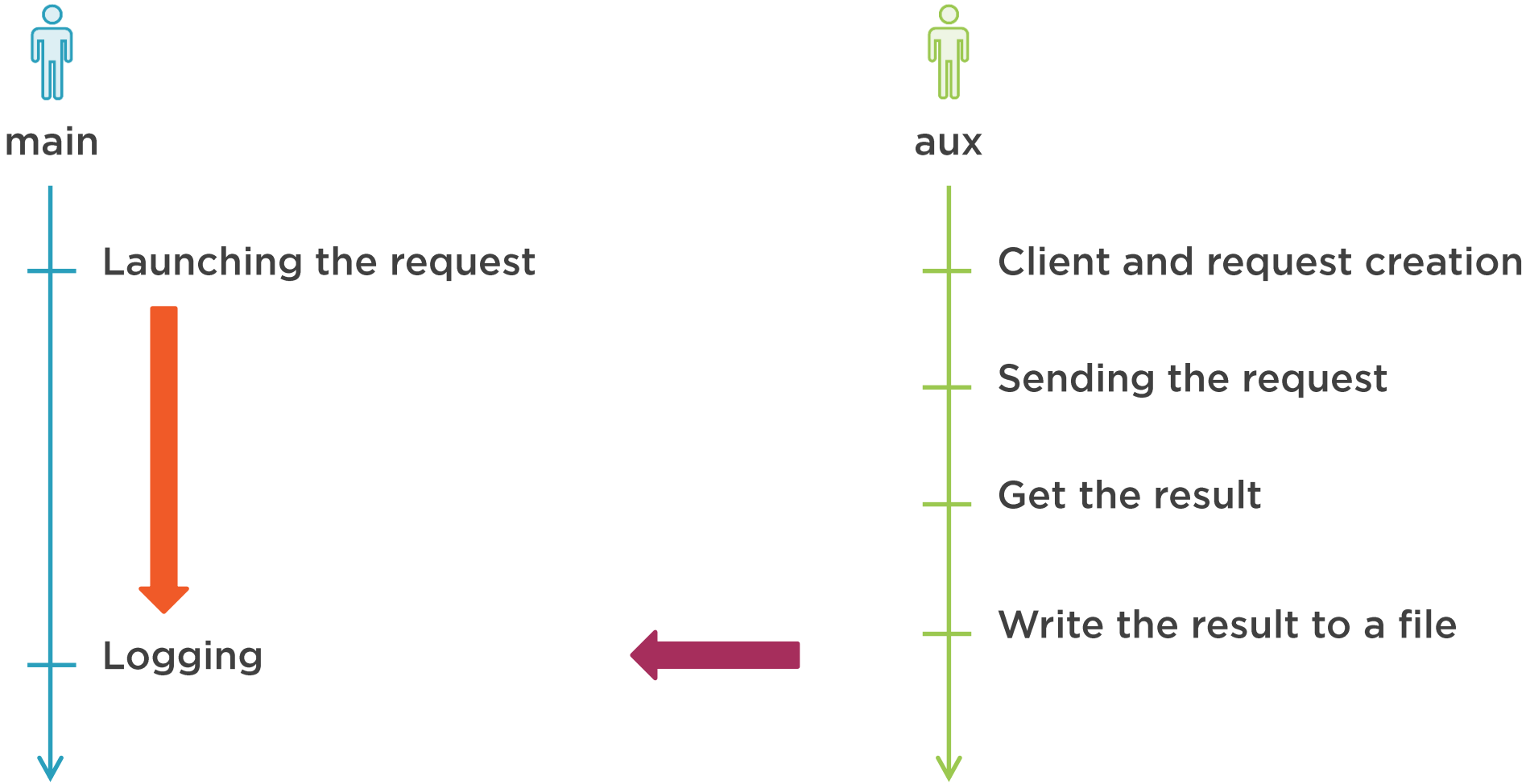
```
HttpResponse response = client.send(request,
                HttpResponse.BodyHandler.asString());

logger.info("Done!");
```

And what about the final logging?

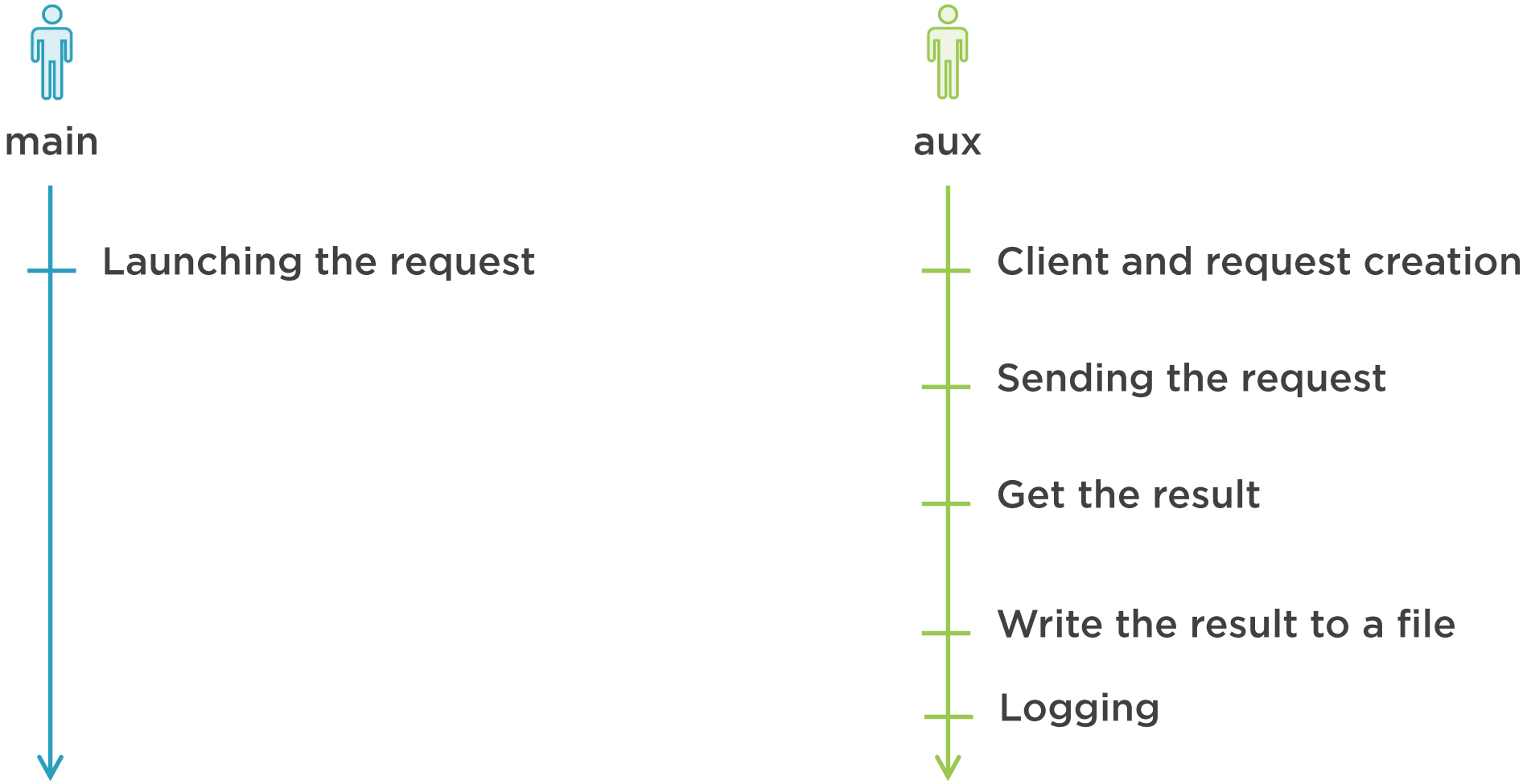
Logging a Message



Logging a Message with the Executor Pattern



Logging a Message with CompletableFuture



This kills the Executor pattern!

The CompletableFuture pattern is the right tool for this job

Examining the Request Itself

```
HttpResponse response =
    client.send(request,
                HttpResponse.BodyHandler.asFile(path));
```

Let us analyze this code

```
HttpResponse response =
    client.send(request,
                HttpResponse.BodyHandler.asFile(path));
```

Let us analyze this code

Sending the request and getting the response is costly

```
HttpResponse response =
    client.send(request,
                HttpResponse.BodyHandler.asFile(path));
```

Let us analyze this code

Sending the request and getting the response is costly

Writing to a file is also costly

The solution we need is to launch those two requests asynchronously

It can easily be done with CompletableFuture

```
CompletableFuture<HttpResponse> response =
    client.sendAsync(request,
                     HttpResponse.BodyHandler.asFile(path))
```

There is a sendAsync() method on HttpClient

That returns a response wrapped in a completable future

```
client.sendAsync(request,
                 HttpResponse.BodyHandler.asFile(path))
      .thenApply(response -> response.getBody())
      .thenApply(path -> path.toFile())
      .thenApply(file -> logger.info(file + " has been created");
```

There is a sendAsync() method on HttpClient

That returns a response wrapped in a completable future

So now we can chain the other tasks

What if you need to chain costly tasks?

You can run them asynchronously

You can compose them

Creating the Chain of Tasks

What if the creation of the chain is costly?

It should be created asynchronously!

Using async will move your data from one thread to another

You can use the delayed start

```
CompletableFuture<Void> start = new CompletableFuture<>()
start.thenCompose(nil -> getUserIDs())
     .thenCompose(ids -> getUsersFromDB(ids))
     .thenCompose(users -> sendEmails(users))
```

All the chain is linked to the technical CompletableFuture: `start`

```
CompletableFuture<Void> start = new CompletableFuture<>()
CompletableFuture<List<Long>> ids =
          start.thenCompose(nil -> getUserIDs())

ids.thenRun(ids -> logger.info("Users read"));
ids.thenCompose(ids -> getUsersFromDB(ids))
   .thenCompose(users -> sendEmails(users))
```

You can even create several branches in your chain

The trick is: nothing happens until `start` completes!

```
CompletableFuture<Void> start = new CompletableFuture<>()
CompletableFuture<List<Long>> ids =
          start.thenCompose(nil -> getUserIDs())

ids.thenRun(ids -> logger.info("Users read"));
ids.thenCompose(ids -> getUsersFromDB(ids))
   .thenCompose(users -> sendEmails(users))


start.complete(null);
```

So to trigger the computation, you need to complete `start`

In this case, start completes in the main thread

```
CompletableFuture<Void> start = new CompletableFuture<>()
CompletableFuture<List<Long>> ids =
          start.thenCompose(nil -> getUserIDs())

ids.thenRun(ids -> logger.info("Users read"));
ids.thenCompose(ids -> getUsersFromDB(ids))
   .thenCompose(users -> sendEmails(users))


start.completeAsync(() -> null, executor);
```

So to trigger the computation, you need to complete `start`

In that case, `start` completes in a thread from executor

Demo

Time to see some code in action!

Let us take a closer look at which thread is executing what

Module Wrap Up

What did you learn?

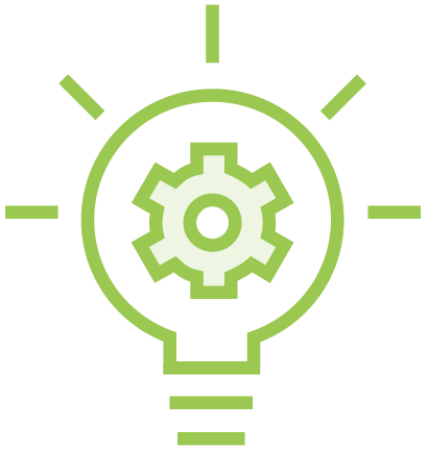
Performance!

Control your threads:

- async gives control and predictability

- non-async gives performance

And you can use delayed start

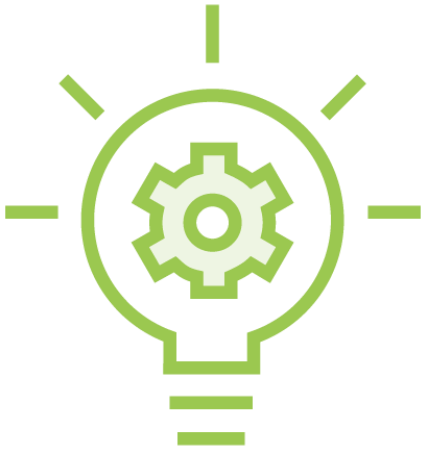


1) Do not block threads

Use async call for long running tasks

Use pools of threads of the right size

Identify what is executing each task

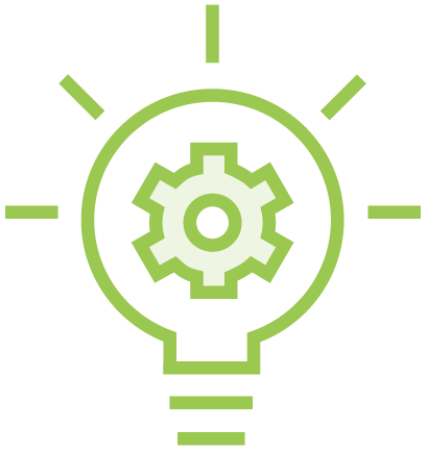


2) Avoid moving data across threads

Identify map / filter / reduce

Execute everything in the same thread

Identify what is executing each task



3) Use the “task triggering trick”

It helps for costly chains

It gives control on which thread is executing each task

Course Wrap Up

Completable Future API

What is a task?

Create pipelines of asynchronous tasks

Triggering tasks on the completion of other tasks

Dealing with exceptions

Controlling your threads

Understanding performance

Course Wrap Up

Thank you!

@JosePaumard

https://github.com/JosePaumard