Move over JDBC: ABDA and R2DBC are coming!

I tend to use few words on slides, so a quick writeup of what I want to discuss seems to be the better way of getting this across.

The case for non blocking

First I'll get into the reason why I think non-blocking drivers are useful. I see both online and offline there is still quite a bit of confusion about what the real benefits are of a fully non blocking architecture.

I'll do that by looking at a bit of classic code, a code example of a classic servlet + JDBC query.

Pros:
- Very mature
- We've been writing code like this forever

Cons:
- The thread performing that request spends most of its time doing nothing
- Under stress (for example when the database performance is unstable) the number of threads will go up, which costs memory and CPU
- Small increases in latency between the application server and the database are very expensive, as that translate in more threads waiting -> more threads needed to do the same amount of work -> more cpu and memory is needed to do the same work.
- Fluctuations in memory usage will create more garbage collection pauses, which will create more memory usage fluctuations.
- In micro service architectures it is worse, as these 'waves of performance issues' propagate across services.

So we can strongly argue there are downsides to blocking code.

History of non-blocking stacks

Quickly go over existing non-blocking api's:
- HTTP Clients like Apache Async Http Client (2011)
- HTTP Server Servlet 3.1 (2013)
- Java NIO (2002)
- Many NoSQL databases

This is *not new*, but adoption has been slow, mostly because the API's have been hard to use.
- Example: 'Echo servlet' using Servlet 3.1 NIO is pages of really dense code, nearly impossible to understand
- Callback hell

Very hard to get right in threaded code paradigm.

Non blocking API’s

Relatively recent adoption of better api's made non blocking code more palatable, even though it still needs quite a shift in thinking.
Two ‘candidates'

- CompletableFutures
- Reactive Streams
- Java 9 Flow

(Will briefly explain all)
CompletableFutures: Embeds a result that might not be present yet. Can be chained using andThen().
Thinks in units of computation that need to be computed in a certain order. Included in modern Java runtimes.

Reactive Streams: Embed a stream that emits 0 or more results. Streams can be mapped, filtered and joined to other streams in many ways.
Implementations: RxJava2, Reactor, Akka Streams

Java 9 Flow: Official standard version of Reactive Streams, nearly identical to Reactive Streams, but still not as widely used as Reactive Streams.

All three of these api’s let us write maintainable non-blocking code.

Non blocking JDBC

First of all, I say JDBC to frame it as a SQL-database access api. It is not JDBC, JDBC will remain JDBC and fundamentally blocking.
We’re looking for non blocking *alternative* to JDBC

For a non-blocking version of JDBC, this split has continued, two different API's were published:
- ABDA (CompletableFuture based)
- R2DBC (Reactive Streams based)

Show example of ABDA.
- Show example
- Looks okay. Workable (I personally don’t really like the CompletableFuture)

Show example of R2DBC
- Show example
- Looks better I think. The ‘stream thinking’ seems better than completablefutures for data sources, but that might be personal preference.

Similar on the surface, both can work

Backpressure

Right now we didn't do much with the data, lets pretend we're running a servlet, and service a GET request that queries some data from a database, like our original blocking servlet.

First rewrite the ABDA one.
Callback writes to the outputstream.
Works, but it semi-blocking. We don't waste threads waiting on the database, but we very well might be wasting threads writing to the output, as the write operation might block.

We'll get back to that one, lets look at the R2DBC version:

Again similar, and here see the toIterable and as we know Iterable is a blocking api. Whenever you see a 'toBlocking' or 'toIterable' in reactive code that is a sign there is something wrong.

So lets convert this one (the R2DBC) to be fully non-blocking
(Using Servlet 3.1 non blocking API)

Pretty sweet:
We have now a fully non blocking query, it has some really nice properties
- Results are streamed through, even if the result set is huge it won't take much memory
- For the whole duration of the request, no threads block, so either when the database can't keep up, or when the client can't keep up.

Perhaps you already know, but let me explain what really happens here:

- When network is saturated servlet engine will indicate that the output is not ready, this signal is propagated upstream, all the way to the database driver, then though TCP (which always has backpressure) to the database, so in a nutshell: We only read data from the database when we can write it to the client.

We call this reactive back pressure. We have a stream of data that flows one way, and a stream of 'signals' that indicate when we can write.

Remember though, this is only important when we are talking about big data sets. If we are querying a single row none of this makes any sense, in that case it is completely ok to read it from the database and keep it in memory until the client is ready to read it.

If we get back to the ADBA version, and try to do the same, we run into a problem:
- While this reactive backpressure is built-in for R2DBC, it is not for ADBA, and if we think about it, it is basically impossible for the CompletableFuture model to do that.

(Explain in more detail)

So for ADBA, either we need to accept that this model won't work for ADBA, as CompletableFutures are going to complete, and they won't listen to anyone whether they need to slow down.

Fortunately it does not end there, the ADBA developers also needed to make a decision, between:
- Just not supporting reactive backpressure, and fully embrace the CompletableFuture model.
- Move away from CompletableFutures completely (and switch to Java 9 Flow)
- Create a hybrid between CompletableFutures and Flow

ADBA Hybrid
- Basically CompetableFuture based, uses Java 9 Flow where it makes sense.
- Show example

Works, but I think the API gets pretty messy.

I wonder if a ‘clean decision’ between Flow or CompletableFutures might have been better.

Conclusion

Non blocking code is definitely useful:
- Performance is better (same work with less threads)
- Deals better with performance issues in other parts of the system

“Nearly non blocking” and “completely non blocking” are quite different, most of the advantages of non blocking code we only get when we can get rid of all blocking code.
- ABDA and R2DBC are quite useful here, as SQL access is one of the last places where there are no standards for non blocking api’s.

Between ADBA and R2DBC
- Both very much in development
- R2DBC seems more pragmatic and clearer in what it wants to be
- R2DBC seems to move faster
- ADBA is a ‘stronger’ standard as it is an official OpenJDK spec.
- ADBA hybrid API seems confusing (to me)

Both claim to be aimed at library builders and meant to be used by application builders.
… but still. API’s should be understandable.

Epilogue

I was mostly surprised how few people involve themselves into something as important as this.

- Join mailing lists
- Try the samples of R2DBC and ADBA
- Try the code of my examples