Cost of Layered ArchitectureLayered architecture is probably one of the most popular choices in the Object Oriented Systems, and for a good reason.

When done right it provides great separation of concerns making each layer replaceable without a need to transform everything around it.

But layers require a lot of additional models, in this post, we take a look at how additional models and constant mappings impact performance of our app.

Glance at the Benchmark app

Most common use cases of layer architecture define 4 layers: UI, Application, Domain and Persistance. To keep better separation, each layer

is supposed to have its own data model, that will be used only inside this layer. This means that we are creating a lot of versions of very similar classes

just to satisfy the needs of the layer.

This post is all about benchmarking an example JVM app. We take a look at speed and memory usage with and without using layers.

You can find the source code here, feel free to run it and see what is outcome on your machine.

Our benchmark tool it's a Kotlin app with just a single dependency on com.neovisionaries:nv-i18n (We are going to use it for the

CountryCode). The premiss is pretty simple. We will be creating User with some basic properties. User will be stored inside an in-memory repository.

After that, we will try to retrieve the data and return it to the caller.

The same flow will be executed in two different ways:

with using all the layers: UI, Application, Domain and Persistance - handleThroughoutAllLayers(), with skipping most of the layers - handleSkippingLayers()

While using all layers we need to map our model 6 times: UserRequest -> †'UserDto ->†User ->†' UserDao -> User -> UserDto ->†' UserResponse

Skipping layers simplifies flow a lot: UserRequest -> UserDao ->†' UserResponse. We are mapping objects only 2 times, that 4 fewer mappings to different models.

With that out of the way, we can take a look at the actual benchmark numbers.

All the tests were made on MacBook Pro with Intel I7-9750H and 16GB of RAM. Benchmark app was running on OpenJDK 18.0.2. Numbers are an average from 10 consecutive runs.

Case studies

Let's take a look at the numbers!

No. of elements - means number of objects that will be dragged on through the whole flow. In the web app, this is comparable to the number of requests.

From the graph, we see that time is growing with the number of requests in a linear way. On average execution with all layers is 40-50% slower.

Looking at memory usage the differences are even clearer. While using layers app consumes about 80-90% more memory.

To no one surprise, additional layers provide overhead for the app and consume more memory. But this is not the end of the story.

We are missing one important factor, JIT! JVM can improve our performance in the runtime.

We just need to give it a little bit of time to figure out possible optimisations. Let's see how the performance will change if we do give

JVM some time to warm-up.

For the warm-up tests, I will execute the same query N times before start measuring the execution time.

By increasing the number of warm-up runs we are decreasing the difference between both approaches. It means that JIT slowly finds a way to

reduce time spend on mapping objects and after ~20 runs both methods seem to even out to a similar outcome.

A quick look at memory usage. But looks like JIT does not help here, and the discrepancy between both methods stays on the same level.

Conclusion

Performance discrepancies are normalised and after a couple of runs are getting close to 0. The bigger memory footprint remains the main difference

factor. It's possible that tinkering with GC settings or using different GC could yield better results.

You may ask yourself how big of a problem it is? There are two downsides (performance-wise) to layered architecture: Slower "boot time"

Since JIT need a couple of runs to catch up the speed. This might be a problem when your app is on the cloud and you are constantly shuffling instances. Then you might not be getting the full benefit from JIT. But if that happens to you, there is probably a much bigger problem

with your app.

Higher memory footprint

This might be an issue if your RAM is limited. You are running on the server down in the basement. But again this is a sign of some other problems

with your infrastructure. RAM is pretty cheap nowadays and grounding our architecture decisions on the RAM cost on might not be the right trade to make.