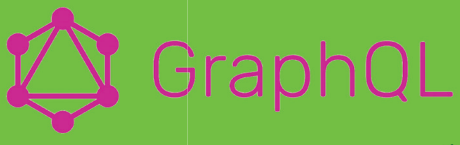
JAVA MAGAZINE

.nl.jug

01 > 2023

Onafhankelijk tijdschrift voor de Java-professional

JAVA IS FUTURE PROOF



› MIGREREN VAN JAVA NAAR KOTLIN
HOE & WAT

› SPRING 6 & SPRING BOOT 3
DE NIEUWE FEATURES

› JFALL
EEN TERUGBLIK

75% of our IT colleagues agree

ing.nl/careers/it

{ COLOFON } JAVA MAGAZINE 01-2023

Content Manager & Project Coördinatie:
Stijn van de Blankevoort

Eindredactie:
Eveline Kroese

Auteurs:
Ron Veen, David Vlijmincx, Soham Dasgupta, Ko Turk, Kitty van Leeuwen, Joris Kuipers, Jasper Sprengers, Bas Knopper

Redactiecommissie:
Stijn van de Blankevoort, Julien Lengrand-Lambert, Maja Reißner, Ted Vinke, Mark van der Walle, Ivo Woltring, Thomas Zeeman

Vormgeving:
Wonderworks, Haarlem

Uitgever:
Martin Smelt

Traffic & Media order:
Marco Verhoog

Drukkerij:
Senefelder Misset, Doetinchem

Advertenties:
Richelle Bussenius
E-mail: richelle.bussenius@nljug.org
Telefoon: 023 752 39 22
Fax: 023 535 96 27
Marc Post
E-mail: mpost@reshift.nl
Telefoon: 023 543 00 08

Abonnementenadministratie:
Tanja Ekel

Lidmaatschap van de NLJUG kost € 59,50 per jaar, waarbij Java Magazine gratis verschijnt. Naast het Java Magazine krijgt u gratis toegang tot de vele NLJUG workshops en het J-Fall congres. Het NLJUG is lid van het wereldwijde netwerk van JAVA user groups. Voor meer informatie of wilt u lid worden, **zie www.nljug.org**. Een nieuw lidmaatschap wordt gestart met de eerst mogelijke editie voor een bepaalde duur. Het lidmaatschap zal na de eerste (betalings)periode stilzwijgend worden omgezet naar lidmaatschap van onbepaalde duur, tenzij u uiterlijk één maand voor afloop van het initiële lidmaatschap schriftelijk (per brief of mail) opzegt. Na de omzetting voor onbepaalde duur kan op ieder moment schriftelijk worden opgezegd per wettelijk voorgeschreven termijn van 3 maanden. Een lidmaatschap is alleen mogelijk in Nederland en België. Uw opzegging ontvangen wij bij voorkeur telefonisch. U kunt de Klantenservice bereiken via: 023-5364401. Verder kunt u mailen naar **members@nljug.org** of schrijven naar NLJUG BV, Ledenadministratie, Nijverheidsweg 18, 2031 CP Haarlem. Verhuisberichten of bezorgklachten kunt u doorgeven via **members@nljug.org** (Klantenservice).

Wij nemen je gegevens, zoals naam, adres en telefoonnummer op in een gegevensbestand. De verwerking van uw gegevens voeren wij uit conform de bepalingen in de Algemene Verordening Gegevensbescherming. De gegevens worden gebruikt voor de uitvoering van afgesloten overeenkomsten, zoals de abonnementenadministratie en, indien je daar toestemming voor hebt gegeven, om je op de hoogte te houden van interessante informatie en/of aanbiedingen. Je kunt uw persoonsgegevens opvragen om inzicht te krijgen in welke gegevens wij van je hebben, deze te corrigeren of, na beëindiging van de abonnee-overeenkomst, te laten verwijderen. Stuur hiertoe een kaartje aan NLJUG BV, afd. klantenservice, Nijverheidsweg 18, 2031 CP Haarlem of een e-mail naar **members@nljug.org**.

VOORWOORD

Fris het nieuwe jaar in

We sloten vorig jaar af met een knaller: J-Fall 2022. Maar dit jaar wordt het nog een groter spektakel... Want dit jaar vieren we maar liefst de 20ste editie! Daarbij staan er nog heel veel andere leuke zaken in de stijgers dit jaar, daar krijg je via onze nieuwsbrieven binnenkort meer over te horen.

Maar niet alleen wij zijn bezig met innoveren. Ook het Java-landschap is constant in beweging. Zo duiken we in dit magazine in het nieuwe Project Loom, dat zorgt voor een lightweight concurrency construct. We kijken ook naar hoe development kan bijdragen aan een van de grootste wereldproblemen: sustainability. En niet te vergeten, Spring Boot 3 is uitgekomen. Joris Kuipers stelt je graag voor aan zijn nieuwe features.

Verder hebben we nog veel meer interessante artikelen waarin bijvoorbeeld het alom bekende GraphQL wordt vergeleken met het Netflix' DGS Framework. Ko Turk neemt je mee in een stappenplan voor het migreren van Java naar Kotlin. En we beseffen ons dat we stiekem allemaal wel data hoarders zijn. Dit was nog maar het puntje van de ijsberg, dus blader snel door!

Oh no, what is happening here? All of the sudden a switch to English? Yup! As you might've noticed, we are publishing more and more English written articles (don't worry, Dutch articles are also welcome 😊). So, do you have experience with a fun or interesting Java related tool and would you like to share your know-how? We want you! (Insert American Army meme). Don't hesitate and reach out to us via info@nljug.org. Even if it's your very first article. No worries, we would love to coach you!

Enjoy this first 2023 edition of Java Magazine!

V

Stijn van de Blankevoort
Content-/Communitymanager NLJUG
svdblankevoort@reshift.nl


We think there is a place for everyone within the bank. Including employees with autism.
Discover IT & how Rabobank creates an inclusive culture at **rabobank.jobs**

Rabobank

INHOUD

Spring GrapQL vs Netflix's DGS Framework

10 Spring Boot released 'Spring for GraphQL' in the beginning of 2022, but about two years ago Netflix open sourced their own GraphQL-library for Spring Boot (DGS framework), giving GraphQL in the microservice world an extra boost. Let's dive into these frameworks and put them side-by-side to understand the key advantages and disadvantages from an implementation perspective.

SustainabilitY

32 The past decade has been an amazing adventure for software engineering. Hardware became so abundantly available that it was hardly considered a constraint anymore. The cloud has especially helped breed the mantra "we can always throw more servers at it". Things are changing again though.

Karate-netty

40 Dit artikel is een beknopte kennismaking met de Karate Netty MockServer, een sub-project binnen het uitgebreide Karate-platform voor testautomatisering, dat ook tools bevat voor API-, GUI- en performancetesten (die hier niet aan de orde zullen komen).

SCHRIJVEN VOOR JAVA MAGAZINE?

Ben je een enthousiast lid van NLJUG en zou je graag willen bijdragen aan Java Magazine? Of ben je werkzaam in de IT en zou je vanuit je functie graag je kennis willen delen met de NLJUG-community? Dat kan! Neem contact op met de redactie, leg uit op welk gebied je expertise ligt en over welk onderwerp je graag zou willen schrijven. Direct artikelen inleveren mag ook.
Mail naar info@nljug.org en wij nemen zo spoedig mogelijk contact met je op.

VIRTUAL THREADS ARE LOOMING

There are a number of projects defined within Oracle that aim to modernize the Java language. A famous one from the past is Project Jigsaw, which splits the monolithic JVM into separate modules. Among the still active projects there is Project Panama, for foreign memory support and foreign function support, Project Valhalla, for bringing inline types (f.k.a. value types) to the JVM, and Project Loom which will bring a lightweight concurrency model to the JVM.

To understand why we need a different concurrency model in the JVM we should first examine how the current threading model works.

{ STANDARD THREADS (A.K.A. PLATFORM THREADS) }

Standard threads, also called platform threads, rely on an underlying operating system thread to perform their work. Creating and initializing an o/s thread is an expensive operation, both in time and memory usage. So you can only have a limited number of o/s threads, typically a few hundred. As creating threads is expensive, we started to cache them for re-use with the classes like the ThreadPoolExecutors. This helps to at least soften the cost of o/s thread creation.

The Java thread captures an o/s thread for the complete time of their running. So they are mapped 1:1 on an o/s thread. This means that if the code is blocking for some reason, like a database call or calling another web service, the o/s thread also blocks. This is of course an undesirable situation as they are such a scarce resource. It would be much more efficient if the underlying o/s thread could be freed and allowed to serve a different thread.

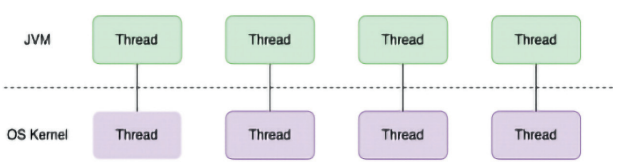


Image 1: see reference 1.

Ron Veen is a software developer and special agent at Team Rockstars IT. He has been working on the JVM for nearly two decades, focusing on Java and Kotlin. Ron is an international conference speaker and author.

David Vlijmincx is a software developer and special agent at Team Rockstars IT. He blogs and is an international speaker who likes to share knowledge with others.

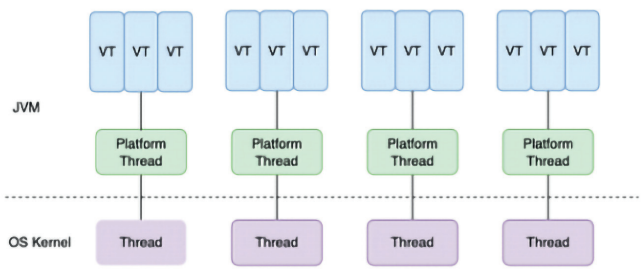


Image 2: see reference 3.

{ VIRTUAL THREADS (A.K.A. USER THREADS) }

Virtual threads are not mapped 1:1 on an o/s thread; instead, they are created and managed by the Java runtime. When they are created and become eligible for executing they will be connected to an o/s thread. Once their execution blocks, the JVM will suspend them, making the o/s thread available for another thread that is ready to execute. Once our blocked thread is ready to run again, it will be placed in a queue of threads waiting to continue their execution. When it is time to run again, it will be placed on an o/s

L1

```
public class VirtualThread {
   Runnable runnable = () -> {
       System.out.println("I am a virtual thread: " +      Thread.currentThread().isVirtual());
   };
   public static void main(String... args) {
       new VirtualThread().run();
   }
   private void run() {
       Thread.ofVirtual().start(runnable);
       Thread.ofPlatform().start(runnable);
   }
}
```

thread again. But it might be on a different o/s thread. So virtual threads have an n:m relationship with o/s threads.

JVM level thread switching is very lightweight which dramatically improves the scalability of virtual threads over standard threads. In fact, it is so efficient that it is an anti-pattern to cache virtual threads for re-use [1]. They very much follow the fire-and-forget concept. The Go language used the same concept with its Go routines.

{ HOW TO CREATE VIRTUAL THREADS? }

You might wonder how you can create a virtual thread. Turns out it is not very different from how you create a regular thread, see listing 1.

On line 14 (Image 3) we are creating a virtual thread. Whereas on line 15 we are creating a traditional thread. Once a thread is created it needs to be started by invoking `start()`.

If you rather use an executor then a new executor class was added to Java 19: ThreadPerTaskExecutor. It is an executor service that starts a new thread for each task and its size is unbounded. This is characteristically different from other executors that typically have some kind of upper bound (Listing 2).

{ HOW EFFICIENT ARE VIRTUAL THREADS? }

So how efficient are virtual threads? While the answer is 'very, very efficient', code speaks louder than words. I executed the little code snippet in listing 3 (next page) on my machine and came to some spectacular results. I urge you to do the same on your machine. I am sure you will be impressed.

The application creates and executes 10.000 threads. If you disable line 10 it will use only virtual threads, if you disable line 9 it will create platform threads.

When running with the virtual threads, they all complete within 1520 ms. Running the platform threads instead, the situation is quite different. On my MacBook Pro 2013 with 16GB memory, it simply runs out of memory. Reducing the number of threads to 2.500 makes it complete in around 2300 ms.

L2

```
private void run() {
   try(var executor = Executors.newVirtualThreadPerTaskExecutor()) {
       executor.submit(runnable);
   }
}
```

L3

```
public class ManyVirtualThreads {
   public static void main(String... args) {
       new ManyVirtualThreads().run();
       new ManyVirtualThreads().runClassic();
   }

   private void run()  {
       try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
           IntStream.range(0, 10_000).forEach( i -> {
               executor.submit(() -> {
                  Thread.sleep(Duration.ofSeconds(1));
               });
           });
       }
   }

   private void runClassic() {
       try (var executor = Executors.newThreadPerTaskExecutor(
               Executors.defaultThreadFactory())) {
           IntStream.range(0, 10_000).forEach( i -> {
               executor.submit(() -> {
                   Thread.sleep(Duration.ofSeconds(1));
               });
           });
       }
   }
}
```

The reason is clear, as soon as a virtual thread blocks it is suspended and a new virtual thread is allowed to run on the now available o/s thread. Thus blocking thread can be swapped out in rapid succession.

The platform threads however, keep blocking the o/s thread, thus reducing the number of concurrent tasks.

{ SILVER BULLET? }

So are virtual threads the silver bullet for every type of concurrency on the JVM? The answer is, 'not right now'. This is due to the fact that a high throughput can only be achieved by relying on tasks blocking at certain points. This means that virtual threads offer less of an advantage for CPU intensive tasks. This is simply down to the fact that these kinds of calculations block less frequency, thus not allowing new threads to join the party. But even then, in combination with the structured concurrency as described below, they offer many advantages over platform threads and the reactive programming model.

{ FINAL THOUGHTS }

Currently, there are two situations that can make a virtual thread 'pinned', meaning that it will block the underlying thread. The first one is when there is a native frame on the stack, so when you are making calls to native routines and those call back into Java.

The second situation is when using a synchronized block. A synchronized block, when blocked, will block the underlying thread. For that reason, it is advised that code that makes heavy use of synchronized blocks is rewritten to use a ReentrantLock. But when synchronized blocks are used scarcely no such rewrite is required.

Image 1.

Virtual threads are a preview feature in Java 19 and are there to gather from the community about its performance, potential bottlenecks and future improvements. Go and give them a try although you might be weary about running them in production.

Project Loom provides us with more proof that after more than 25 years, and against popular opinion, Java is still a language that keeps on evolving.

{ WHAT IS STRUCTURED CONCURRENCY? }

Java 19 introduced structured concurrency as an incubating module. As the name does suggest, the project brings structured concurrency capabilities to Java. The idea behind structured concurrency is to make the lifetime of threads behave the same as code blocks in structured programming. For example, when you call method A, which calls method B, you know that method call B is finished before you exit method A. Applying this idea to threads makes them easier to work with and reason about.

Image 1 visualizes the idea of structured concurrency. Here we start with the main thread that spawns a new thread called X. Now, for the main thread to finish, we first need to wait till thread X has finished running. Only then will the main thread be done.

To use structured concurrency inside your projects, you need to add this VM option `--add-modules jdk.incubator.concurrent`. This option will add the structured concurrency module to your project. With this module included, you can use the new StructuredTaskScope. Java 19 has two StructuredTaskScopes: ShutdownOnSuccess, and ShutdownOnFailure. These scopes are used to fork threads from and act as the shutdown policy.

In the following example (Listing 1), we use the ShutdownOnFailure shutdown policy. This StructuredTaskScope starts a bunch of threads and waits till all the threads are finished or they have been canceled because of an exception. The `scope.join()` call in the example is a blocking method that will make the parent thread wait till all the forked threads are done. When we continue after calling `join()`, we check if an exception has been thrown by any thread and propagate it to the caller.

Instead of `get()` to fetch the result from the future object instances, we use `resultNow()`. `Get()` is a blocking method; at this point in the code, we know the forked threads are finished.

L4

```
public String StructuredConcurrency() throws
ExecutionException, InterruptedException {

   try(var scope = new StructuredTaskScope.
ShutdownOnFailure()){
       Future<String> result1 = scope.
fork(RunningMultipleTasks::fetchResult);
       Future<String> result2 = scope.
fork(RunningMultipleTasks::fetchResult);

       scope.join();
       scope.throwIfFailed();

       return result1.resultNow() + result2.
resultNow();

   }
}
```

The other StructuredTaskScope ShutdownOnSuccess works almost the same as this one. The difference is that with ShutdownOnSuccess, we start a bunch of threads, wait until the first one is finished, and cancel the remaining ones.

Structured concurrency takes some time to get used to, but it is a powerful tool we can use to reason about threads. The crux of structured concurrency is not to have fire and forget concurrency. <

{ REFERENCES }

1 https://medium.com/javarevisited/how-to-use-java-19-virtual-threads-c16a32bad5f7
2 https://openjdk.org/jeps/436
3 https://medium.com/javarevisited/how-to-use-java-19-virtual-threads-c16a32bad5f7

A COMPARATIVE STUDY BETWEEN SPRING GRAPHQL AND NETFLIX'S DGS FRAMEWORK

Spring Boot released 'Spring for GraphQL' in the beginning of 2022, but about two years ago Netflix open sourced their own GraphQL-library for Spring Boot (DGS framework), giving GraphQL in the micro-service world an extra boost. Let's dive into these frameworks and put them side-by-side to understand the key advantages and disadvantages from an implementation perspective.

Soham Dasgupta is by role a Solution Architect at Capgemini working on cloud-native Java and OSS space.

{ WHAT IS GRAPHQL? }

GraphQL is a JSON based query language for APIs designed by Facebook to allow its mobile clients to define exactly what data should be sent back from an API and therefore avoid unnecessary roundtrips and data usage. Since it was open sourced in 2015, it was rapidly adopted, and many companies have already switched to the GraphQL way of building APIs.

{ BACKGROUND }

In this article I use a Spring Boot application which needs to fetch hierarchical data from a database (this can also be multiple different APIs in your scenario) and expose them as GraphQL APIs.

{ VERSION INFO }

In this article the following versions are used in all the examples:

- DGS framework: 5.4.1
- Spring for GraphQL: 1.0.2
- Maven plugin for code generation (graphqlcodegen-maven-plugin): 1.19

{ SCHEMA }

In GraphQL, the API definition comes in a schema file, which defines the queries and mutation one can execute, and it also contains the data model definition.

{ QUERYING DATA WITH GRAPHQL AKA DATAFETCHERS & DATALOADERS }

There are two ways we can retrieve data in GraphQL: DataFetchers when data comes from a single source and DataLoaders for referential data. For example, considering a data model Provider and Services, where a Service always has a Provider. If a user queries 'services' and also wants to retrieve the associated Provider, you would need DataFetcher to retrieve Services along with its attributes and a DataLoader to retrieve all the related Provider(s) at once. This is to prevent multiple calls for each related Provider and instead generating one call to retrieve all the related Provider(s) at once. This is called a N+1 problem in a hierarchical data model.

The code/class structure of Netflix's DGS framework is as shown in Image 2.

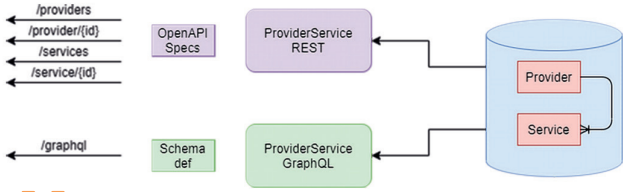


Image 1.

L1

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    services: [Service]
    serviceById(id: ID!): Service
    providers: [Provider]
    providerById(id: ID!): Provider
}

type Mutation {
    addService(name: String!, description: String,
providerId: ID!): Service
    addProvider(name: String!, description:
String): Provider
}

type Provider {
    id: ID!
    name: String
    description: String
    services: [Service]
}

type Service {
    id: ID!
    name: String
    description: String
    provider: Provider
}
```

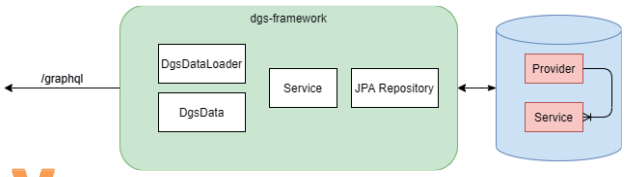


Image 2.

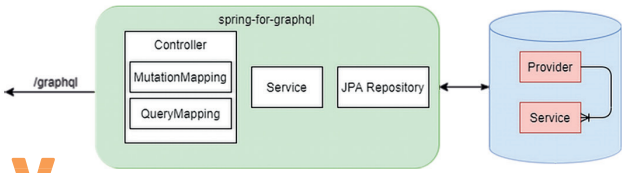


Image 3.

L2

```
@DgsDataLoader(name = “providers”)
public class ProviderDataLoader implements
BatchLoader<Integer, Provider> {

@Autowired
private ProviderService providerService;

@Override
public CompletionStage<List<Provider>>
load(List<Integer> keys) {
   return CompletableFuture.supplyAsync(() ->
providerService.findAllProvidersByIds(keys));
    }
}

@DgsComponent
public class ServiceDataFetcher {

@Autowired
private ServicesService service;

@DgsData(parentType = “Query”, field = “services”)
public List<Service> getAllServices() {
    return service.findAllServices();
}

@DgsData(parentType = “Service”, field = “provider”)
public CompletableFuture<Provider>
provider(DataFetchingEnvironment  dfe) {
    DataLoader<Integer, Provider> dataLoader = dfe.
getDataLoader(“providers”);
    Service service = dfe.getSource();
    return dataLoader.load(service.getProviderId());
}
}
```

Listing 2 shows the code example on how to implement DataFetcher and DataLoader in Netflix's DGS framework.

In Image 3 you'll find the code/class structure in Spring for GraphQL.

The implementation of DataFetcher and DataLoader looks like Listing 3 (next page).

NOTE: In both frameworks, one needs to register a DataLoader. In DGS you refer to the loader with a name and in Spring for GraphQL with the type of the object you are loading via the DataLoader.

{ TESTING }

Unit testing is always required to make sure that the code we write doesn't break in production or to prevent it breaking from future code updates and dependency updates.

L3

```
@Configuration
@RequiredArgsConstructor
public class DataLoaderRegistryConfig {

private final BatchLoaderRegistry registry;
private final ProviderService providerService;

@PostConstruct
public void registerDataLoader() {
   registry.forTypePair(Integer.class, Provider.
class)

.registerMappedBatchLoader((providerIds,
env) -> providerService.
findAllProvidersByIds(providerIds).
                map(v -> Map.entry(v.getId(), v))
                .collect(Collectors.toMap(Map.
Entry::getKey, Map.Entry::getValue)));
  }
}
}
…..
@Controller
@RequiredArgsConstructor
public class ServiceProviderController {

private final ServicesService servicesService;
private final ProviderService providerService;

@SchemaMapping(typeName = "Query", value =
"services")
public List<Service> findAllServices() {
   return servicesService.findAllServices();
}

@SchemaMapping(typeName = "Service")
public CompletableFuture<Provider>
provider(Service service, DataLoader<Integer,
Provider> loader) {
   return loader.load(service.getProviderId());
  }
}
```

L4

```
@SpringBootTest
public class ProviderDataFetcherTests {

@Autowired
private DgsQueryExecutor dgsQueryExecutor;


@Test
void testGetProviderById() {
    String providerName = dgsQueryExecutor.
executeAndExtractJsonPath(
            " { providerById(id: 2) { name id }}",
            "data.providerById.name");
    assertEquals("javaMag", providerName);
}
}
```

L5

```
@SpringBootTest
@AutoConfigureGraphQlTester
public class ServiceProviderControllerTest {

@Autowired
private GraphQlTester graphQlTester;

@Test
void getProviderById() {
    String query = "{ providerById(id:2) { name
}}";
    graphQlTester.document(query).execute()
            .path("data.providerById.name")
            .entity(String.class)
            .isEqualTo("javaMag");
}
}
```

The DGS framework provides `DgsQueryExecutor`, which can be injected into your test class to execute tests (Listing 4).

Spring for GraphQL provides `GraphQlTester`, which needs to be configured and can also be injected in order to run your unit tests. You can auto configure `GraphQlTester` using an annotation (Listing 5).

{ EXCEPTION HANDLING }

Handling exceptions during query execution and providing proper error messages to the client is absolutely essential when it comes to APIs. In GraphQL that's also not any different.

In the DGS framework, we need to define a Spring bean which implements `DataFetcherExceptionHandler`. Then, inside an overridden method you can put custom error information depending on which type of exception generated by the application. In the example in Listing 6 `NoDataFoundError` is the custom exception for which we are providing some extra information in the response.

L6

```
@Component
public class DataFetchingExceptionHandler implements DataFetcherExceptionHandler {

@Override
public  CompletableFuture <DataFetcherExceptionHandlerResult>
handleException(DataFetcherExceptionHandlerParameters params) {
    if(params.getException() instanceof NoDataFoundError noDataFoundError) {
        Map<String, Object> debugInfo = new HashMap<>();
        debugInfo.put("errorCode", noDataFoundError.getErrorCode());

        TypedGraphQLError graphqlError = TypedGraphQLError
                .newInternalErrorBuilder()
                .message(noDataFoundError.getMessage())
                .debugInfo(debugInfo)
                .path(params.getPath()).build();
        DataFetcherExceptionHandlerResult result = DataFetcherExceptionHandlerResult.newResult()
                .error(graphqlError).build();
        return CompletableFuture.completedFuture(result);
    } else {
        return DataFetcherExceptionHandler.super.handleException(params);
    }
}

}
```

In Spring for GraphQL, the implementation is also quite similar, but the bean is defined in spring configuration bean (Listing 7). Here also `NoDataFoundError` is the custom exception which the code is trying to handle.

The exception handling in Listing 7 will result in a response like Image 4, with a HTTP status code 200.

Image 4.

"THE KEY DIFFERENCE BETWEEN DATAFETCHER AND DATALOADER IS THE LATER YOU NEED FOR RELATIONAL DATA, FOR EXAMPLE RETRIEVING RELATED PROVIDER(S) FOR A SERVICE OBJECT."

L7

```
@Bean
public DataFetcherExceptionResolver
exceptionResolver() {
    return DataFetcherExceptionResolver.
forSingleError((ex, env) -> {
        if (ex instanceof NoDataFoundError ndf) {
            return GraphqlErrorBuilder.
newError(env).message(ndf.getMessage()).
errorType(ErrorType.NOT_FOUND).build();
        }
        else {
            return GraphqlErrorBuilder.
newError(env).message(ex.getMessage()).
errorType(ErrorType.INTERNAL_ERROR).build();
        }
    });
}
```

{ CODE GENERATION }

When it comes to writing code which consumes GraphQL APIs, it becomes quite handy if we can generate the response objects and API clients from schema definition.

In Spring for GraphQL there is no defined way to do this, but DGS on the other hand provides a plugin to generate clients and types. Officially this plugin is for Gradle, but there is a Maven community build variant of it as well. Listing 8 shows how you could do it.

Normally the query string for GraphQL requests varies depending on what a client wants to retrieve from the server. This forces the client to maintain and manipulate strings. With this plugin (Listing 9), the query string can be created in a type-safe manner.

The example in Listing 10 produces a query for all services with service names and descriptions, with associated provider descriptions.

{ CONCLUSION }

Both the frameworks are similar in many aspects, both are annotation driven and based on graphql-java library. In my opinion, where they stand apart is how DataLoader is implemented and the plugin to generate client and types in the DGS framework.

By design Spring for GraphQL uses reactive streams to execute DataLoader queries, which makes it useful when the underlying system(s) doesn't provide an endpoint to query all data at once.

This is different in the DGS framework, which expects the underlying system(s) to provide an endpoint to query all the data at once.

Additionally, in the DGS framework, creating clients and types are useful not only for consuming applications but also for writing automated test scripts, for example in Cucumber. This completely takes out string manipulation during query construction and also parsing responses from the server.

I hope this article gives you a gentle introduction to GraphQL in Spring Boot world and two frameworks which you can use to implement in your own project.

The complete code examples can be found via the QR codes. <

L8

```
GraphQLQueryRequest graphQLQueryRequest =
        new
GraphQLQueryRequest(ServicesGraphQLQuery.
newRequest().build(),
        new ServicesProjectionRoot().name().
description().provider().description());
```

L8

```
<plugin>
   <groupId>io.github.deweyjose</groupId>
   <artifactId>graphqlcodegen-maven-plugin</
artifactId>
   <executions>
      <execution>
         <goals>
            <goal>generate</goal>
         </goals>
      </execution>
   </executions>
   <configuration>
      <packageName>soham.spring.graphql.model</
packageName>
      <schemaPaths>
         <path>src/main/resources/schema</path>
      </schemaPaths>
      <generateBoxedTypes>true</
generateBoxedTypes>
      <generateClient>true</generateClient>
      <writeToFiles>true</writeToFiles>
   </configuration>
</plugin>
```

L10

```
query {
  services {
    name
    description
    provider {
      description
    }
  }
}
```

V *Spring for GraphQL.*

V *DGS Framework.*


Vandaag schrijf ik code voor de NS-app.
Morgen teken ik voor lead-developer.
Dennis, Developer
Lees meer op werkenbijns.nl

VERHUISBEDRIJF JETBRAINS HELPT MET HET MIGREREN NAAR KOTLIN

Ga eens terug naar het moment dat je ging verhuizen. Voor velen geen leuk moment, maar het hoeft niet altijd zo te gaan! Met een goed plan en de juiste instelling is het een fluitje van een cent. En raad eens, er zijn ook mensen die je willen helpen bij de verhuizing. Zo werkt het min of meer ook bij het migreren van Java naar Kotlin. Vraag je grote vriend IntelliJ IDEA om wat hulp. In dit artikel gaan we dieper in op hoe je kan migreren naar Kotlin.

Ko Turk (@KoTurk77) is een developer bij Blue4IT en heeft een opdracht bij de Rabobank. Hij spreekt graag over techniek op conferenties, jugs en local meet-ups.

{ MAAR EERST... }

Waarom zou je willen migreren? Er zijn talloze redenen te bedenken, maar de belangrijkste is de begrijpbaarheid van de code. Zoals professor en conferentiespreker Venkat Subramanian al jaren predikt, 'keep it simple'. Maak je code duidelijker door functioneel te programmeren. Kotlin heeft hiervoor wat handigheidjes. Duidelijkheid zit 'm ook in het feit dat er minder boilerplate-code nodig is.

Ook word je binnen Kotlin geforceerd om `null` af te vangen. Dus geen `NullPointerException` meer in je productiecode 😂!

Kotlin maakt developers productiever! Niet voor niets gebruikt Google het in hun codebase. Van de (Kotlin)-developers zegt 70% dat ze productiever zijn en 87% dat ze erg blij zijn met het gebruik van de taal. Ook crasht het minder snel en is er 0% degradatie in het gebruik van Java en Kotlin door elkaar. Daarnaast zijn er meer dan vijf miljoen gebruikers. Dus, interessante feiten die zijn verteld op Devoxx België 2022 (kijk vooral eens naar Afbeelding 1).

{ MAAK EEN (VERHUIS) PLAN }

Het is belangrijk om te bedenken wat je wil migreren, dus maak een plan. Wil je alles in één keer migreren, of per deel? Het belangrijkste is om er structuur in aan te brengen. Zo kun je ervoor kiezen om te migreren per class, package, feature, story of applicatie.

Afbeelding 1: redenen voor Google om voor Kotlin te kiezen.

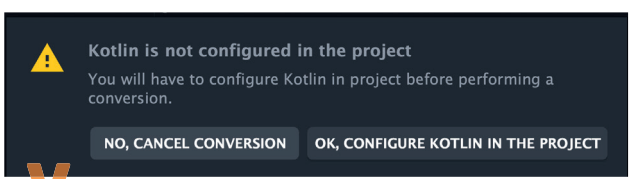
Zoals je bij een verhuizing ook doet: stop alles in verhuisdozen en zet een label op de doos. 'Keuken!' Zo weet je wat erin zit en het belangrijkste, waar het naartoe moet.

Dit gaan we ook doen bij onze migratie (van een Spring Boot-applicatie). In totaal maken we vijf dozen, die we gaan uitpakken in ons nieuwe huis:

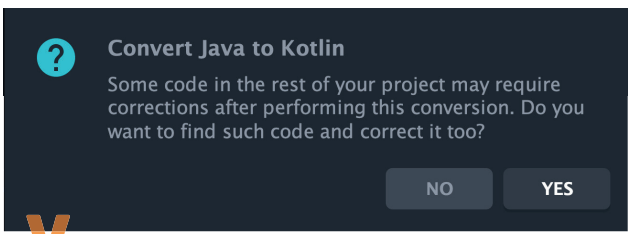
- Doos 1: Model.
- Doos 2: Applicatiefile.
- Doos 3: Controller.
- Doos 4: Service.
- Doos 5: Database.

{ DE VERHUIZING }

En dan is het zover! We gaan verhuizen en onze grote vriend IntelliJ IDEA komt ons helpen. Wist je bijvoorbeeld dat er een functie is om Java om te zetten in Kotlin? In de 'Project Explorer' (klik met de linker muisknop op het (Java)-bestand en scroll dan naar beneden) vind je een optie 'Convert Java File to Kotlin File'. Klik hier op en er verschijnt een nieuw venster (Afbeelding 2).
Ondanks dat Maven niet meer als default wordt weergegeven in de Spring Initializer, gebruiken we toch de meest gebruikte buildtool



Afbeelding 2: Kotlin configureren met de hulp van IntelliJ IDEA.



Afbeelding 3: de migratie van Java naar Kotlin.

[1] (van 2021). Voordat we verder gaan, moeten we Kotlin eerst configureren in onze pom.xml. Klik op 'OK' en onze IDE voegt de kotlin-maven-compiler toe aan de build (compile execution). Ook de betreffende dependency's om Kotlin te kunnen gebruiken in je code worden toegevoegd:

```
<dependency>
   <groupId>org.jetbrains.kotlin</groupId>
   <artifactId>kotlin-stdlib-jdk8</artifactId>
   <version>${kotlin.version}</version>
</dependency>
<dependency>
   <groupId>org.jetbrains.kotlin</groupId>
   <artifactId>kotlin-test</artifactId>
   <version>${kotlin.version}</version>
   <scope>test</scope>
</dependency>
```

Omdat we voor het eerst Kotlin gaan gebruiken, moeten we eerst de pom.xml herladen en de poging nog eens wagen, hierna volgt Afbeelding 3.

Uiteindelijk krijg je een gegenereerde Kotlin-file, zeg maar onze 'ingepakte doos'. Nu is het tijd om deze te gaan uitpakken en de spulletjes op de juiste plek te zetten. De code kan namelijk nog veel optimaler! We beginnen hierbij bij de eerste doos, het model.

{ DOOS ÉÉN: HET MODEL }

We beginnen met het simpelste, namelijk het migreren van ons model naar Kotlin. Dat kan er als volgt uitzien in Java (JDK8):

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Example {

    @NotNull
    private String variable;

    private String variableNotNull;
}
```

Maar wist je dat je helemaal geen Lombok meer nodig hebt (zie class annotaties)? Ook de overbodige `@NotNull`-annotaties zijn helemaal niet nodig. In Kotlin zijn alle variabelen bij default `not null`, we moeten deze afvangen. Op die manier verwijderen we al vier overbodige regels. Het wordt al wat minder complex! Laten we kijken hoe dit eruit ziet in Kotlin:

```
data class Example(val variable: String,
                   val variable2: String?)
```

Zoals je ziet, wordt het nog wat makkelijker gemaakt met de data class modifier. In JDK14 kan je ook Records gebruiken, hierbij is het verschil miniem. In Records is alles immutable, in de data class kan je nog wel met een `val` of `var` aangeven of iets final is of niet. In Kotlin heb je ook nog wat handigheidjes, zoals de `copy`-functie (als je niet alle waarden wil wijzigen):

```
var sample = Sample("Key-1", "Value-1")
val sample2 = sample.copy(key = "Key-2")
```

Pratende over boilerplate, wist je bijvoorbeeld dat als je een data class hebt in je applicatie, er ook destructing declarations gebruikt kunnen worden? Dus je hoeft niet al je variabelen apart te declareren en kan dit in een keer doen (met dank aan de persoon in het voorbeeld):

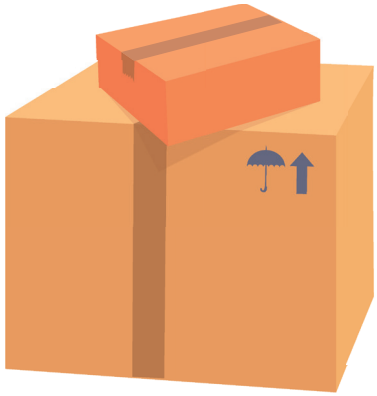
```
val person = Person(1, "Ali", 34)
val(id, name, age) = person

println(id)     // 1
println(name)   // Ali
println(age)    // 34
```

{ DOOS TWEE: HET MIGREREN VAN JE APPLICATIEFILE }

Als je werkt met Java en Spring Boot, dan ziet de volgende klasse (application.java) er bekend uit:

```
SpringApplication.run(PaymentEngineApplication.
class, args);
```

Nou is dit een simpel voorbeeld, maar totaal niet functioneel. Je leest niet duidelijk wat het doet. Voor de 'senior' onder ons niet lastig, maar voor menig beginnende developer wel. In Kotlin is het daarentegen wat makkelijker te volgen, het beschrijft wat het doet, namelijk:



```
runApplication<PaymentEngineApplication>(*args)
```

Jawel, het starten van de applicatie 😉. Wellicht een open deur, maar dit zijn van die kleine dingen om mee te beginnen. En dan zie je ook de kracht van Kotlin! Hou het simpel. Overigens, de methode `runApplication` is een 'inline function', wat betekent dat je de type niet hoeft mee te geven, maar gebruik maakt van een type parameter `<PaymentEngineApplication>`. Stuk duidelijker toch?

{ DOOS DRIE: DE CONTROLLER }

Nu wordt het wat leuker, want nu gaan we een hoop boilerplate-code weghalen. Zo heb je bijvoorbeeld een aantal annotaties boven je klasse wederom niet nodig. Gooi Lombok in de prullenbak (als je dat nog niet gedaan had) en ook de annotatie om je model te valideren (met spring-context) op `null` is niet nodig (of handig).

```
@RestController
@Slf4j
@RequiredArgsConstructor
@Validated
```

Laatstgenoemde kan natuurlijk wel handig zijn om andere validaties te doen (zoals size, `notEmpty`) maar als je alleen op `null` controleert, hoef je deze annotaties en dependency's niet te gebruiken. Mocht je wel willen vasthouden aan de Spring-validaties? Vergeet dan niet je annotatie te veranderen naar bijvoorbeeld `@field:Email` als het om de e-mailvalidatie gaat.

Maar er bestaat ook nog iets anders, namelijk Konform of Valiktor, waar validaties (idomatic) in je dataclass staan en waar je geen rare annotaties (fratsen) nodig hebt. Je hebt daarnaast beter zicht op wat er gecheckt wordt, want het verdwijnt niet zomaar op de achtergrond, zoals bij Spring. Ook moeilijke (cross-field) validaties zijn makkelijker te maken, want er zijn geen implementatie-classes nodig.

Een voorbeeld van wat je in je dataclass kan zetten met Valiktor:

```
init {
validate(Example::text).hasSize(min = 10, max =
10000)
}
```

En met Konform:

```
init {
Validation<Example>{
Example::text {
minLength(10)
maxLength(10000)
}
}
}
```

Wat je verder vaak ziet in Java zijn de 'try', 'catch' en 'finally'-blokken, niet heel erg functioneel. Dit kan mooier! Met de volgende code, is het in Kotlin een stuk functioneler:

```
kotlin.runCatching {
  // je logica hier
}.also(
  // soort van Java's finally
).fold(
    {  ResponseEntity.ok("Created payment") },
  // OK flow
    {  status(HttpStatus.INTERNAL_SERVER_ERROR).
build() }   // NOK flow
)
```

{ DOOS VIER: DE SERVICE }

Instance variabelen (vaak geannoteerd) plaatsen we nu in de constructor:

```
class Service(private val client: Client) {
```

In de service-laag zie je vaak alle logica verschijnen. Vaak maken we hierbij gebruik van de Streams API. Super handig! Maar in Kotlin hoef je na een list niet meer `.stream()` aan te roepen. We kunnen nu direct op de lijst een (`transform`-) functie aanroepen. Dit kan bijvoorbeeld een map zijn, maar ook andere Stream-achtige functies, zoals `find()`.

'Scoped functions', zoals `let` en `also` kunnen het verschil maken in je code! Met `let` kan je itereren over je variabele, ook al heeft die maar één waarde. Samen met de Elvis operator, het vraagteken (gevolgd door een dubbele punt), kunnen we ook controleren of het object `null` is. Een voorbeeld:

```
variabele.let {} ?: log.info("No variable set")
```

Dat is toch leesbaarder dan een (extra) `null`-check in Java, zoals op volgende regels?

```
if (variabele != null) {}
else  {}
```

Mocht je de `if`-check vergeten in Java, dan heb je (meestal) een groot probleem. Je herkent vast wel de situatie dat je geen `null` verwacht, maar die er uiteindelijk wel is. En zie daar: een NullPointerException in productie. Oei, pijnlijk. Hoe ga ik dit uitleggen? In Kotlin word je geacht om `null` af te vangen, dus deze situatie (en discussie achteraf) vermijd je.

'Extension functions': een krachtige feature in Kotlin, waarmee je methoden (of functions in Java) kan toevoegen aan klassen (ongeacht of je de owner bent). Dit kan super handig zijn als je een klasse gebruikt die niet alles doet wat je wil. Een simpel voorbeeld:

```
fun String.removeFirstWord(): String =  this.
substring(0, this.lastIndexOf(" "))

fun main(args: Array<String>) {
   val myString = "Hello Thomas"
   val result = myString.removeFirstWord()
   println("$result thanks for reviewing")
}
```

In dit geval voeg je een extra functie toe aan de String-klasse en kan je jouw functie aanroepen vanuit de variabele zelf. In Java moet je eerst de klasse extenden en krijg je meer boilerplate. Overigens het resultaat `$result` plaats je gewoon in je `println` om je variabele te gebruiken.

{ DOOS VIJF: DE DATABASE }

Wanneer je asynchroon je code wil aanroepen, heeft Kotlin (al jaren) iets leuks: coroutines! Een manier (tussenlaag) om te zorgen dat je niet uit je threads loopt. Handig als er veel verkeer is op je endpoints of deze deze wel eens blijven hangen (en daarna weer doorgaan) door iets magisch in je code 😇. Dan heb je op een gegeven moment geen threads meer over met als gevolg: een StackOverflowExeption. In Java kunnen we dit vergelijken met Project Loom (Virtual Threads), maar die zit nu nog als preview in Java 19.

Momenteel maken we in Java vaak gebruik van bijvoorbeeld CompletableFutures. In Kotlin kan dit ook, maar geven we aan of iets blocking is of niet. Het volgende voorbeeld laat zien hoe dit werkt:

```
fun main() = runBlocking {
   launch { joke() }
   println("0 is false and 1 is true, right?")
}

suspend fun joke() {
   delay(1000L)
   println("1")
}
```

Als eerste geven we een scope mee, in dit geval `runBlocking`. Dan roepen we `joke` aan binnen `launch` (om de coroutine te starten). Door de methode `joke` 'suspended' te maken, geven we aan dat deze functie opgepakt moet worden door een coroutine. Dan kan de JVM lekker aan de slag met andere zaken (mocht `joke` blocking zijn, zoals in dit geval omdat er een delay volgt). Het voorbeeld geeft zoals je verwacht `0 is false and 1 is true, right?` terug gevolgd door een `1`.

Overigens als je non-blocking wil werken, werk dan met frameworks, zoals r2dbc en maak je methoden suspended:

```
@Query("SELECT * FROM samples")
suspend fun getSample(): List<Sample>
```

Om het helemaal af te maken, kun je gebruik maken van een non-blocking applicatieserver, zoals KTOR [2]. Overigens gaan we niet dieper in op deze coroutines en KTOR, omdat hier hele studies over zijn (en dit artikel te kort is om het allemaal uit te leggen). Kijk vooral eens op mijn GitHub-pagina.

{ VERHUISD! NIEUW HUIS, NIEUWE BUURT, NIEUWE MOGELIJKHEDEN }

Gefeliciteerd, je bent gemigreerd! Nu denk je wellicht, "veel kan ook in Java 17". Maar de leesbaarheid (meer functioneel, extension methods), minder boilerplate (minder code, geen ;), kwaliteitsverbeteringen (scoped functions) en coroutines maken een overstap zeker de moeite waard. Ook met Jetbrains aan je zijde en een bruisende community ziet de toekomst er rooskleurig uit voor Kotlin! <

Kijk voor alle werkende code op:
https://github.com/KoTurk/Kotlin/tree/main/NLJUG

{ REFERENTIES }

1 https://snyk.io/jvm-ecosystem-report-2021/
2 https://ktor.io/

SYLLOGOMANIE

Ik heb een telefoon die volstaat met foto's waar ik al jaren niet meer naar heb gekeken. Tot laatst, toen ik er uit verveling eens doorheen scrolde. Het was een trage donderdagavond en de streamingdiensten hadden niks te bieden, zoals mijn kast uit kan puilen van kleding en ik toch niets vind om te dragen. Ik bladerde door al mijn foto's, helemaal terug tot 2012. Ik herleefde stapavonden van toen ik begin twintig was en ogenschijnlijk groen fluwelen jurkjes met lange mouwen kon dragen in een club zonder me kapot te zweten. Ik reisde met retrospectieve vliegschaamte af naar Rome, Lissabon, Hong Kong, de Amerikaanse Westkust en Parijs. Ik moest hardop lachen om een verdwaasde John Travolta bij de fietstoren op Amsterdam C.S. en een paar keer glimlachen om de tientallen andere memes en gifjes in mijn filmrol (het was een fase).

Kitty van Leeuwen is Solution Lead Quality Improvement Services bij Ordina. Ze is geïnteresseerd in ethische vraagstukken rondom digitale technologie.

Ik heb drie laptops die vol staan met niet langer relevante apps, programma's die ik ooit eens downloadde, een keer gebruikte en daarna nooit meer naar omkeek. Mijn desktoppen bestaan uit een ongeorganiseerde, kleurige wirwar van screenshots. Ik heb een notitieapp met daarin meer dan 1000 losse notities; vage ideeën voor verhalen die ik zou willen schrijven, film- en boektitels die me zijn aangeraden en die ik op dat moment ook echt had willen zien of lezen. Uiteindelijk raakten ze zoek tussen alle andere ingevingen die eens heel belangrijk leken, maar later toch hun glans verloren.

Ik heb vier mailaccounts met in totaal meer dan drieduizend ongelezen mailtjes en tienduizenden gelezen berichtjes die ik nooit meer hoef terug te zien. Ik heb social media-accounts met foto's, berichten, 'vrienden', volgers. Connecties die ik niet eens allemaal meer ken of eigenlijk nooit heb gekend.

Ik ben Kitty en ik ben een datahoarder. Het duurde lang voordat ik het kon zeggen. Niet omdat ik het niet durf toe te geven, nu ik het weet praat ik er graag over. Het lucht op, misschien ontmoet ik wel andere mensen die net als ik zijn. Nee, schaamte was niet hetgeen dat me tegenhield om te verkondigen dat ik een datahoarder ben. De ware reden dat mijn extreme dataverzamelwoede zo lang onder de radar is gebleven, komt omdat het er geleidelijk is ingeslopen, ik had het zelf niet eens door.

Toen ik 14 was, kreeg ik mijn allereerste telefoon, een Nokia 3310. Er pasten maar 20 berichtjes op en één spelletje, Snake. Als ik een sms'je te veel kreeg, gaf mijn telefoon dat aan. Dan moest ik er een paar verwijderen, zodat ik nieuwe kon ontvangen en zo ging dat een tijdje door. Terwijl ik volwassen werd, evolueerde de technologie die ik gebruikte met me mee. Ik was 18 toen Steve Jobs een revolutionair product aankondigde dat "de wereld zou veranderen". Het duurde nog even voor ik daadwerkelijk mijn eerste iPhone had, maar de parallel is op zijn minst noemenswaardig.

In Nederland wordt in 9 op de 10 huishoudens de smartphone dagelijks gebruikt [1]. Ik schrik van die cijfers en vraag me af hoe huishoudens zonder zo'n ding hun leven leiden. Gebruiken ze uitvouwbare kaarten om de weg te vinden? Vertrouwen ze op het weerbericht van Gerrit Hiemstra? Maken ze foto's van het hartje in hun havercappuccino to-go met een analoge camera? Mijn punt moge duidelijk zijn, de revolutie die Jobs aankondigde is gekomen. Smartphones bezitten grotendeels ons leven. En zoals dat ging met mijn Nokia toen ik 14 was, maak ik het tegenwoordig nooit meer mee dat ik zaken moet verwijderen om ruimte vrij te maken. Berichtendiensten bieden mij namelijk onbegrensde cloudopslag. Ergens op een willekeurige plek op deze planeet, in een doosachtig gebouw, draait er een server met daarop al mijn correspondentie van het afgelopen decennium. Daardoor blijft er voor mij een mogelijkheid bestaan om comateuze WhatsApp groepen uit 2013 – waarin ik niet als enige ben overgebleven – te reanimeren.

NIET ALLE HOARDERS ZIJN TROTS OP HUN GEDRAG. RECENTELIJK KOMT ER STEEDS MEER AANDACHT VOOR HET OPRUIMEN VAN DATA. OP TIKTOK SPREKEN INFLUENCERS ME TOE DAT IK MIJN MAIL MOET VERWIJDEREN VOOR HET MILIEU.

Het klinkt misschien onschuldig, toch blijft het daar niet bij. Berichtendiensten zijn namelijk niet de enige die oneindige opslag aanbieden. Van mijn complete telefoon worden automatisch back-ups gemaakt. Daardoor hoef ik geen enkele foto te verwijderen, geen enkele notitie te verliezen, geen enkele instelling ooit kwijt te raken.

Een soortgelijk systeem hanteer ik voor mijn laptops. Bestanden kan ik opslaan op een willekeurig cloudplatform, het maximale geheugen zal nooit bereikt worden. Het hele model van cloudserviceaanbieders is ontworpen om mij nooit op de 'delete'-knop te laten drukken. Daar zit geen ingewikkeld kwaadaardig complot achter, niks kwaadaardiger dan kapitalisme tenminste. Toonaangevende technologiebedrijven, zoals Apple, Dropbox, Google en Microsoft hebben er belang bij om gebruikers hun gegevens naar cloudplatforms te laten verplaatsen. Ze hebben er belang bij om grote hoeveelheden data te verzamelen: hoe meer gegevens, hoe meer ruimte gebruikers nodig hebben. Hoe meer data, hoe meer mogelijkheden om als platform te gedijen. Onbeperkte opslag voor foto's lijkt misschien ontzettend genereus, maar vrijgevigheid is niet per se de belangrijkste drijfveer als we een groter bedrijfsmodel overwegen waarin data een essentiële hulpbron is voor Machine Learning en AI-Training.

Naar datahoarding werd tot voor kort niet veel onderzoek gedaan. Anders dan bij fysieke dwangverzameling heeft het opslaan van data geen direct effect op onze leefomgeving en hygiëne en dat maakt mensen die last hebben van excessieve dataverzameldrang nogal lastig te spotten. De term 'datahoarden' wordt als volgt gedefinieerd: '..het verzamelen van digitale bestanden tot het punt van het verlies van perspectief, wat uiteindelijk leidt tot stress en chaos' [2]. Voor datahoarders, maar ook voor mensen die zich niet zo identificeren, kunnen digitale objecten eenzelfde emotionele lading krijgen als fysieke objecten dat hebben [3]. Net als bij fysieke 'ontspulling' kan het zomaar gebeuren dat we iets weggooien dat later ontzettend belangrijk bleek. Maar waar we de kans op het per ongeluk weggooien van een stoffelijk ding verlagen, omdat we deze eerst door onze handen laten gaan, zijn digitale zaken vluchtiger van aard.

Jarenlang heb ik weinig tot niets verwijderd, waardoor ik onbewust een extensief digitaal archief van mijn leven heb aangelegd. In dat archief zitten best wat waardevolle momenten, maar er zit ook een hele hoop zooi tussen. Twintig versies van dezelfde dronken selfie heb ik niet per se nodig om een leuke avond te

herinneren. Net zomin als ik de 'Kruidvataanbieding van week 13, 2011' uit mijn inbox nog van nut kan maken meer dan elf jaar later. Ik weet dat ik moet opruimen, alleen de kennis ontbreekt mij om effectief te kunnen beoordelen wat ik wil bewaren en wat niet. Ik mis het overzicht, ik heb werkelijk geen idee waar ik moet beginnen. Tegelijkertijd kan ik niet inschatten hoeveel tijd het me gaat kosten om mijn digitale leven op te ruimen, de anticipatie alleen al verlamt me.

Alhoewel ik sinds kort iets minder per ongeluk aan het verzamelen ben, volg ik nog altijd schaapachtig wat de knappe koppen van Silicon Valley hebben uitgedacht dat hen het meeste geld oplevert. En waar het voor hun zakken heel lucratief is dat ik en met mij nog vele anderen nooit de 'delete'-knop indrukken, is het voor onze planeet een stuk minder gunstig dat wij te lui zijn om de bezem door ons digitale decor te halen. Op dit moment zijn datacenters verantwoordelijk voor 2% van wereldwijde CO2-uitstoot, dat is ongeveer evenveel als de gehele luchtvaartindustrie. Verwacht wordt dat dit zal stijgen naar 3.2% in 2025 en in het ergste doemscenario in 2040 de 14% aantikt [4]. Veel data op het internet is niks meer dan afval. En er wordt onnodig veel computerkracht besteed aan dat afval. De selfies, de junkmails, de memes, de gifjes, de vage Facebookvrienden. Heb ik ze echt ooit nog nodig?

Afscheid nemen van mijn digitale archief valt me zwaarder dan ik zou willen toegeven. Hoe sentimenteel ook, het bevat schimmen van mijn verleden die ik nog niet kan loslaten. De verspreide stukjes data houden mijn identiteit vast. Het zijn de sporen die ik heb achtergelaten, de artefacten die ik heb verzameld, ze zijn uniek voor mij. De gefragmenteerde onderdelen samen maken mijn digitale zelf. Ik heb een angst dat wanneer ik daar te rigoureus een bezem doorhaal, ik een type zelfmoord pleeg.

Dat we ons werkelijk verbonden kunnen voelen met digitale spullen, blijkt uit een recent onderzoek waar mensen met een Pinterest-account aan deelnamen. De participanten werden voor de gek gehouden door de onderzoekers, die zeiden dat een van de door hen zo zorgvuldig gecureerde borden was verwijderd. Uit een vragenlijst die de deelnemers daarna moesten invullen, kwam een positieve correlatie naar voren tussen het aantal digitale pins en het niveau van hechting aan die plaatjes. Te meten aan de stress die ze ervaarden bij het idee dat hun bord niet langer bestond [5].

De verontrusting van de deelnemers aan dat onderzoek kan ik me goed voorstellen. Gegevens online kunnen absoluut uit de openbare toegang verdwijnen zonder een spoor achter te laten. Soms doordat je zelf een knullige fout maakt, vaker omdat anderen besluiten gegevens weg te nemen. Websites kunnen offline gaan, muziek op Spotify of een video op YouTube kan door de eigenaar worden verwijderd. In de r/DataHoarder subreddit geven trotse datahoarders elkaar tips voor precies dat laatste probleem. Ze discussiëren welke harddrive je het beste kunt aanschaffen of welke cloudserviceaanbieder het goedkoopst is. Onder het motto "What do you mean DELETE?!" praten ze hun hoardinggedrag goed, omdat zij de nobele taak op zich hebben genomen het internet te archiveren. In de toekomst zal de mensheid hen dankbaar zijn. Ze bieden tegen elkaar op over hoeveel data ze door de jaren hebben vergaard (de winnaar 2.6 petabytes (PB) lokale opslag en 12 PB cloudopslag, of in andere niet te bevatten eenheden; 14,600 terabytes). Wanneer een forumlid zich afvraagt waarom ze eigenlijk zoveel data opslaan, geeft een ander het verlossende antwoord "Het vervult onze verzamelbehoeften vanuit ons jagers en verzamelaars instinct, maar omdat we leven in een tijdperk van digitale communicatie, verzamelen we data!" [6].

Niet alle hoarders zijn trots op hun gedrag. Recentelijk komt er steeds meer aandacht voor het opruimen van data. Op TikTok spreken influencers me toe dat ik mijn mail moet verwijderen voor het milieu. We kennen van oudsher de slogan 'een beter milieu begint bij jezelf' en vanuit die gedachte is het natuurlijk ontzettend mooi dat ik mijn steentje kan bijdragen door simpelweg mijn mailbox op orde te brengen. Maar zet dat echt zoden aan de dijk? Duurzamere oplossingen liggen denk ik bij het reguleren van de aanbieders en grootgebruikers van dataopslag. Wat als software ons vertelde welke data er irrelevant zijn, bijvoorbeeld omdat we er de afgelopen 10 jaar niet naar hebben gekeken? Of wat als het afnemen van extra opslagruimte minder goedkoop was dan het nu is, waardoor je eerder geneigd bent om je zooi op te ruimen in plaats van extra storage bij te kopen voor een euro. Bedrijven kunnen actief beleid voeren om data die lange tijd onaangeroerd zijn na een bepaalde periode naar 'cold storage' te verschuiven. Slechts een paar ideeën die kunnen helpen bij het verminderen van CO2-uitstoot en waarvan de implementatiemogelijkheden minstens onderzocht moeten worden. Tot die tijd, zal ik vaker de 'delete'-knop indrukken, beloofd. <

{ REFERENTIES }

1 https://longreads.cbs.nl/ict-kennis-en-economie-2020/ict-gebruik-van-huishoudens-en-personen/
2 https://www.researchgate.net/publication/282732234_A_case_of_digital_hoarding
3 https://www.researchgate.net/publication/343282119_Digital_Collection_What_makes_it_different_from_Digital_Hoarding
4 https://www.climatechangenews.com/2017/12/11/tsunami-data-consume-one-fifth-global-electricity-2025/
5 https://www.researchgate.net/publication/343282119_Digital_Collection_What_makes_it_different_from_Digital_Hoarding
6 https://www.reddit.com/r/DataHoarder/comments/v1y7t6/mental_health_issues_around_datahoarding_serious/

WHY YOU SHOULD MOVE FROM JAVA TO KOTLIN

Java is stable and has a rich ecosystem, so why move to Kotlin, right? I'm here to tell you why we at Rabobank think you should definitely consider using Kotlin, and what our challenges are around introducing it to our engineers.

{ KOTLIN: A PRAGMATIC & MODERN LANGUAGE }

Like Java, Kotlin is a JVM language. What makes Kotlin different is that, at heart, it is a pragmatic language. That means it was created with real-life use cases in mind, such as being interoperable with existing Java code and infrastructure. And the language will evolve to stay relevant and modern, by not only adding new features but also retiring ones that are outdated.

When designing Kotlin, the creators did not try to reinvent the wheel. Instead, they had a careful look at other languages such as Java and Scala and improved upon the features they wanted to use. Kotlin was created to be a language that combines Object-Oriented and Functional programming, focused on interoperability, safety, clarity, and tooling support.

{ MOVING TO KOTLIN IS EASY }

Although Rabobank seems quite orientated towards Java, moving to Kotlin is relatively easy. The bank has an advanced infrastructure and tooling support for Java (JVM) applications. All these tools and libraries are also used for Kotlin.

{ TAKING THE KOTLIN COMMUNITY TO THE NEXT LEVEL }

Last year, a Kotlin community was set up to unite Kotlin engineers scattered all over the bank. This year, the community is steering towards increasing the adoption of Kotlin. To achieve that, we need a more mature and diverse community.

Because most of our Java engineers are new to Kotlin, the pitfall is that they don't know all the best ways to use it. It's important to learn how to write idiomatic Kotlin from the start from a more experienced Kotlin engineer. This can help prevent the increase in technical debt and the spreading of bad practices throughout the team.

To grow the community, we need more experts that can offer support to less-experienced engineers. We have designed a new training program around the idea of training engineers who can train other engineers. It's called the "Kotlin Mastery Program", and will allow engineers from different entry levels to become excellent in Kotlin and even become Kotlin Ambassadors and Champions. Over 100 engineers have already followed the basic training, and about 30 engineers have followed the intermediate and advanced training.

Ali Meshkat – Software Engineer at Rabobank

{ IN SHORT }

Kotlin comes with extensive utilities and key features that allow you to create safe, expressive, and concise code. Because most engineers within Rabobank are just starting out using Kotlin, it's imperative to have more experts around to guide them. The Kotlin Mastery Program is there to train such experts by offering a tailor-made program based on the needs of Rabobank engineers. Only once the Kotlin Community has enough champions and ambassadors can we sit back and relax and let the community take over.

DO YOU WANT TO KNOW MORE ABOUT THE MOVEMENT FROM JAVA TO KOTLIN? SCAN THE QR CODE AND READ MORE AT RABO TECHBLOG.

Rabobank

J·FALL 2022

de Frietkraam was sponsored by

InTraffic

ABN AMRO KIJKT TERUG OP J-FALL

J-Fall, het was weer als vanouds gezellig en inspirerend! 1700 gepassioneerde mensen die volop de sessies en de stands bezochten. En bij ons op de ABN AMRO-stand was het dan ook gezellig druk.

De ochtend begon goed met leuke early-bird sessies. Onze Java-Chapter-Lead Andre Groeneveld beet de spits af in één van deze sessies met zijn topic 'Automate your home by building your own Zigbee bridge'. De zaal was zo vol dat er zelfs mensen op de trap gingen zitten om naar zijn verhaal te luisteren. Zijn score door de bezoekers was dan ook zeer hoog, waar wij zeer blij mee zijn.
Na de opening hebben wij als hoofdsponsor de keynote verzorgd. Sabyasachi (Saby) Sengupta nam het publiek mee hoe verandering en innovatie begint bij mensen. Hoe voorzie je vertragingen en wegversperringen en wat zou jij met jouw team hier aan kunnen doen? Met veel humor en interactie wist Saby het publiek te inspireren en is er na afloop nog veel nagepraat over zijn sessie.

Ondanks dat het buiten fris was, liep de temperatuur binnen best op. Gelukkig kon iedereen bij ons een heerlijk ABN AMRO of Java-ijsje halen die verzorgd werd door Bernardo's uit Ede. En als je dan toch op onze stand was, kon je gelijk deelnemen aan een Java-quiz. Veel bezoekers hebben dit gedaan en er waren leuke reacties. Aan het eind van de dag ging de top 10 dan ook naar huis met leuke Lego-prijzen.
De sfeer was de hele dag leuk en wij hebben veel mensen gesproken over wat wij aan Java-development doen binnen ABN AMRO. 2022 is een mooi jaar geweest en 2023 brengt ons, naast de nieuwe Java 21 LTE, vast veel meer inspirerende dingen.
In 2023 zal ABN AMRO ook veel MeetUps organiseren. Mocht je interesse hebben dan nodigen wij jou graag uit om onze MeetUp-pagina in de gaten te houden op meetup.com/abn-amro-developer.

Zien wij jou in 2023 op J-Spring, J-Fall of één van onze MeetUps?

ABN·AMRO


TIKKIE WON'T BE INNOVATIVE FOREVER
#STARTWITHUS
After successfully launching apps like Tikkie and Grip, it would be easy to say "Okay, we made it". But that's not how innovation works. Innovation is about constantly challenging yourself. Daring to acknowledge that something can always be better. Must be better. Because it's this mindset that makes the difference. And to make a difference we need you. Employees that will continue to motivate others, and us.
www.workingatabnamro.com/J-Fall
ABN·AMRO

DUO ZET IN OP VOLLEDIG CONTAINERIZED APPLICATIELANDSCHAP

Iedereen kent de Dienst Uitvoering Onderwijs (DUO) natuurlijk van studiefinanciering, maar de laatste jaren heeft deze organisatie zich ontpopt tot een ICT-first organisatie die kwaliteit en flexibiliteit richting de eindgebruiker hoog in het vaandel heeft staan. Samen met Overheidsdatacenter-Noord (ODC-Noord) zetten ze in op een volledig containerized applicatielandschap, dat draait op OpenShift. Daarmee zijn ze een voorloper binnen de Rijksoverheid.

Bas van Driel is senior developer bij de Dienst Uitvoering Onderwijs (DUO).

Martijn van Tiel is software architect bij de Dienst Uitvoering Onderwijs (DUO).

Senior developer Bas van Driel legt de keuze voor containerplatform uit. ‘We willen snel schakelen en bijblijven met techniek en security. Dit platform geeft ons die flexibiliteit. Daarnaast is ieder DevOps-team in control over hun eigen applicatielandschap. Ze kunnen er zelf voor kiezen om te werken met de nieuwste technieken. Bijvoorbeeld voor een upgrade naar Java 11 hoeven ze niet te wachten tot de hele organisatie over is.’

{ INTERESSANTE TECHNIEK }

Dat maakt het werk leuker, vertelt software architect Martijn van Tiel. ‘Voor de meeste ontwikkelaars is de toolset waarmee we werken interessant. Je wordt niet meer beperkt door een platform, wat je in een keurslijf drukt. We werken in OpenShift en cloud first. Dat geeft veel vrijheid voor maatwerk, al gaan we bijvoorbeeld de pipeline wel standaardiseren. De shift naar een meer cloud native opzet betekent ook dat we ons meer richten op microservice architectuur.’

{ VERNIEUWEN BETEKENT OOK VERANDEREN}

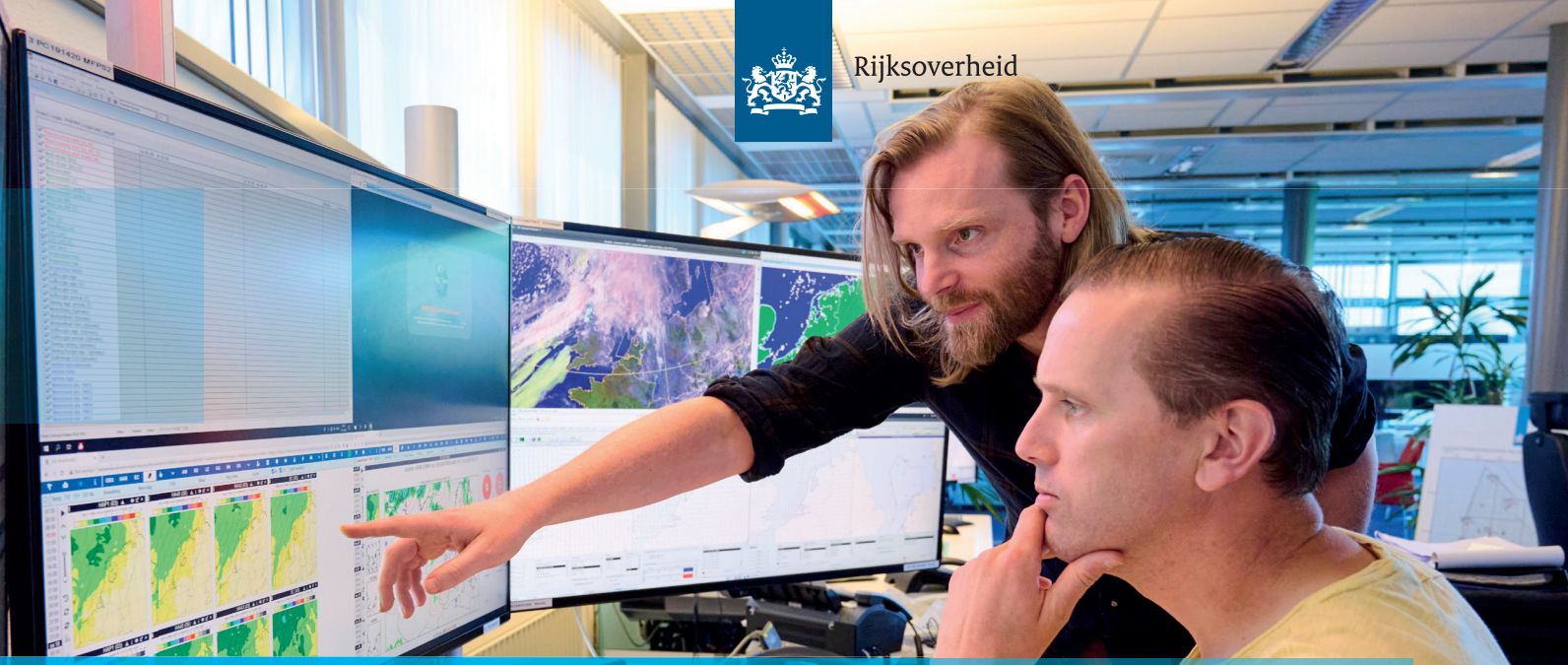
De overgang naar een volledig containerized applicatielandschap gaat niet zonder slag of stoot. In de vorige stack waren veel zaken centraal ingericht, zoals databases, queue managers en (virtual) servers. Voor een cloud first strategie met GitOps-gedachte komen deze centrale diensten en werkwijzen in het gedrang. Het worden obstakels. Het doel van de nieuwe architectuur is dat zowel infrastructure-as-code als configuration-as-code de gebruikelijke werkwijze moet worden voor de DevOps-teams.

{ DEDICATED TEAM }

Essentieel voor het succes: een multidisciplinair team heeft dedicated aan deze migratie gewerkt. Dit cloud migratie support-team geeft workshops, verspreidt kennis en maakt voorbeeldapplicaties. Bas: ‘Het klinkt als een open deur maar voor zo’n grote migratie heb je ook business- en productowners nodig. Een duidelijke richting vanuit de directie, een centrale regie-aanpak (maar decentrale uitvoering) en vooral het enthousiasme van de DevOps-teams hebben geleid tot adoptie bij de business.’

{ SPITS AFGEBETEN }

Last but not least: een paar enthousiaste teams hebben zich opgeworpen als early adopters. Deze teams hebben het spits afgebeten en voor bepaalde onderwerpen uitgezocht wat wel werkt en wat niet. Martijn: ‘Op basis van hun ervaringen maken we werkinstructies en bouwblokken die de andere teams kunnen gebruiken om ermee aan de slag te gaan. We hebben dit project op een agile manier aangepakt. Klein beginnen, en stap voor stap. Nu merken we dat steeds meer teams aanhaken, wat natuurlijk een mooie ontwikkeling is.’



Rijksoverheid

Werk jij mee aan IT voor 17 miljoen Nederlanders?

Als IT-professional bij de Rijksoverheid werk je mee aan een zo goed mogelijk functionerende digitale samenleving voor iedereen in Nederland.
Je houdt je als developer bezig met de ontwikkeling van applicaties om het weer te voorspellen. Innoveert als data-expert met big data, slimme algoritmes en toolsvoor verkeersinformatie.
Of draagt als securityspecialist bij aan de beschermingvan systemen die Nederland draaiende houden.

Wat wij jou bieden? Volop mogelijkheden om je breed te ontwikkelen in een werkomgeving waar je dagelijks werkt aan complexe IT-vraagstukken.
Denk aan RPA, front-end en back-end technologieën, programmeren met bestaande of zelfontwikkelde tools in Java en het automatiseren van testanalyses, uitvoeren van risicoanalyses of inzetten van data-tools.
Benieuwd wat jij bij de Rijksoverheid kunt doen?

Werken bij de Rijksoverheid is werken voor Nederland.

Scan de QR-code en check onze vacatures.

www.werkenvoornederland.nl

NLJUG INNOVATION AWARD 2022

THE WINNER IS OMODA!

WINNER NLJUG INNOVATION AWARD 2022!

During the 19th edition of the J-Fall Conference, which took place on November 3rd in Pathé Ede, the winner of the NLJUG Innovation Award was chosen live on stage by 1.500 Java developers.

The three nominees were announced before J-Fall took place: Adyen, Maqqie en Omoda. They presented their innovative project to the audience via a video pitch. After which the audience could vote live on their personal favorite.

Omoda successfully convinced the audience with their innovative "PhotoChopper: automating photo adjustments for an online clothing shop"-project. In this project Omoda automated and simplified a labor intensive photo editing & processing process using AI and interesting technology stacks.

MO
DA
OMODA

The NLJUG congratulates Martijn van der Maas & the developer team of Omoda with this magnificent victory!

MASTERS OF JAVA 2022

Woensdag 2 november - Masters of Java 2022: De jaarlijkse code challenge voor Java Developers van elk niveau.

Het was een gedenkwaardige editie. Een andere locatie, een record aantal deelnemers van maar liefst 34 actieve teams. Juist door het grote aantal teams had de game-server problemen, dit bleek tijdens de 1e opdracht. We konden overschakelen naar Plan B waardoor de deelnemers allemaal 3 opdrachten konden maken. Zo werd het alsnog EN wederom een spannende wedstrijd. Extra spanningselement was dat de tussenstanden nu niet getoond konden worden, de teams waren namelijk onderverdeeld in 2 shifts.

{ DE OPDRACHTEN }

Per opdracht kregen de deelnemers een beknopt Java project waarvan je in regel slechts één bestand kan aanpassen. Ze konden alleen gebruikmaken van de standaard Java SDK, dus er was geen framework kennis nodig. Door gebruik te maken van een paar meegeleverde tests konden de teams hun oplossing controleren en daar soms ook extra punten mee verdienen. Dachten ze de juiste oplossing te hebben, dan konden ze die indienen. Was de opdracht goed, dan kregen ze de resterende tijd aan punten plus een bonus.

De 3 opdrachten moesten worden gemaakt binnen de limiet van 30 minuten. Je had echt je hardcore java skills nodig en de opdrachten varieerden van het uitwerken van algoritmen tot het gebruik van bekende en minder bekende Java API's.

{ DE UITSLAG }

1 Jasper van Merle | team camel_case | 5.112 punten
2 Johan de Jong | team EatSleepCodeRepeat | 3.944 punten
3 Thomas Withaar en Maarten van der Zwaart | team OVSoftware | 3.886 punten

Een grote dank aan iedereen voor de flexibiliteit, inzet en uiteraard fun tijdens deze dag. We kijken al uit naar Masters of Java 2023!

FIRST 8 | CONCLUSION

Benieuwd naar de opdrachten? Check:

ACHIEVING SUSTAINABILITY IN SOFTWARE ENGINEERING

The past decade has been an amazing adventure for software engineering. Hardware became so abundantly available that it was hardly considered a constraint anymore. The cloud has especially helped breed the mantra "we can always throw more servers at it". Things are changing again though.

Jan-Hendrik is a passionate developer who is not afraid to break something open in order to find out how it works. He is also passionate about eliminating tedious tasks from his routines.

{ THE COMING STORM }

Since the Paris agreement where countries pledged to curb Green House Gas (GHG) emissions, most production industries have seen increased scrutiny and regulation on their environmental impact. The digital industries have so far been largely overlooked, since their GHG emissions are indirect and only measurable as a by-product of their energy consumption.

Thanks to an increasing sense of urgency, recent news reports on the energy consumption of data centers and cryptocurrencies as well as the current energy crisis, awareness of the impact of computing is now growing. Some of us have argued for a while that it is inevitable for this wave of change to reach our industry and people are now seeing the massive amounts of energy needed by some technologies. For example, cryptocurrencies have been in the news a couple of times for consuming more energy than some countries do. Bitcoin alone generates roughly 65 megatons of C02 emissions yearly, which is comparable to the emissions of the entirety of Greece and almost half of the 163 megatons the Netherlands emit. It stands to reason that all these factors weigh into the increased scrutiny that is applied to new data center plans. They no longer seem to be automatically welcomed with open arms.

Is the dream of infinite computing power lost to us? How is this change going to impact us?

{ OPPORTUNITIES AWAIT US }

Change may feel scary when it is heading for you, but change is usually also a source of opportunity. If you worry that these changes will make IT a dirty industry or that jobs may be lost, fear not. Society has become digital at a staggering pace and for years now, most companies and governments have become completely reliant on their IT systems. This means that it will be nearly impossible to simply turn things off and remain operational. Instead, we will have to take a close look at the software that is currently in production and find ways to optimize and minimize. Which means more work for us!

{ CHANGES IN OPTIMIZATIONS }

In 1955 a CPU hour would cost around $600, while a programmer was paid $2 per hour. The scale of the IT industry has brought the cost of computing down, while the need for programmers has brought wages up significantly. AWS hourly EC2 rates currently average at $0,10 while programmer wages can be anywhere between $60-$120 per hour. This change caused the focus of optimization to shift. Initially, we were focussed on minimizing processing time at the cost of human working hours. Currently, our industry is highly optimized for developer efficiency at the cost of the increased use of computing resources. The costs associated with GHG emissions and environmental certification will undoubtedly shift this focus back to increasing the need for more efficient code.

One area where we may have to rethink the direction of optimizations is the availability of our services. It seems every modern project feels the need to be online 24/7 no matter what. Making an architectural decision to have an availability of 99.999% actually sets a product up as a massive resource sink. These uptime guarantees require at least one level of redundancy, immediately doubling or tripling the overall impact of the service. Instead, we should investigate how we can gracefully handle outages and hours of unavailability. A venture into more asynchronous processing models may alleviate this problem, while still maintaining a consistent and reliable service to our customers. An interesting 'outside the box'-thinking example is how some Christian webshops and news outlets close their sites on Sunday. Even though

it may not be inspired by climate awareness, it does show that opening hours may not always have to be round-the-clock.

{ CURRENT EFFORTS: GLOBAL AND LOCAL }

A lot of initiatives are already underway to work on better practices, sustainable alternatives and efficient products. There are datacenter providers recycling the heat of their servers, like LeafCloud and BlockHeating, both Dutch companies. The Green Software Foundation (GSF) is working on defining what makes software green, or how we can be aware of the cost of our computing in real time. The Sustainable Digital Infrastructure Alliance (SDIA) is hosting industry working groups trying to create concrete plans and steps for companies to reach sustainability goals.

On a scene closer to home to us developers, an IT company in Utrecht organized a meetup last November that hosted Bernard van Gastel, an assistant professor at Radboud University leading the new Sustainable Digital Transformation research department. Personally, after finding each other on Twitter, the authors of this piece recently hosted two sessions at Devoxx and J-Fall where we had lively discussions with other developers about ways to increase the sustainability of our work. With these sessions and the accompanying blog post, we hope to inspire you to look for ways to contribute.

{ HOW CAN I HELP? }

This massive transformation coming our way is not something that can be taken care of by a few people. It will need all of us as an industry to stand up and take responsibility for our contributions to create a better world. As massive as this task may feel, change is in the smallest beginnings. We thank you for reading this article and ask you to look through the references. The next step is to become part of the change. This is as simple as recommending this article to a peer, or joining a newsletter from the GSF or SDIA. Taking a look at the offerings of a greener cloud provider, or visiting any meetup on the subject.

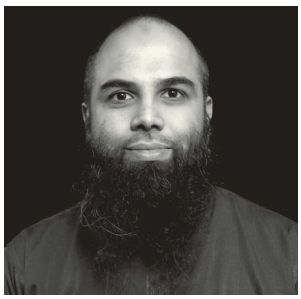
Closer to home, play around with JoularJX which is a new Java-agent for measuring power consumption at the source code level. For more ways to be inspired, we invite you to read the blog post summarizing the many great ideas collected at our conference sessions. Whatever you decide to do, talk about it, share it and in the process we will all be shaping this new direction of our IT industry for a greener tomorrow! <

{ REFERENCES }

- Green Software Foundation (https://greensoftware.foundation/)
- Sustainable Digital Infrastructure Alliance (https://sdialliance.org/)
- Recap Devoxx and JFall sessions https://blog.yoink.nl/posts/2022/11/14/exploring-sustainability-in-tech.html
- BitCoin's emissions (https://www.bbc.com/news/technology-60521975)

SERVERLESS JAVA APPLICATIONS ON AWS

Many teams are building modern Java applications by implementing modular architectural patterns, serverless computing, and agile development processes. When building serverless web applications, it is common to use AWS Lambda functions as the compute layer for the business logic. In this article, you will understand the basics behind how Lambda execution environments operate and the different ways to improve the startup-time and performance of Java applications on Lambda.



Mohammed Fazalullah Qudrath is a Senior Developer Advocate at AWS for the MENA region.

{ WHY OPTIMIZE JAVA APPLICATIONS ON AWS LAMBDA? }

AWS Lambda's pricing is designed to charge you based on the execution duration, rounded to the nearest millisecond. The cost of executing a Lambda function will be proportional to the amount of memory allocated to the function. Therefore, the performance optimization may also result in long-term cost optimizations.

For Java managed runtimes, a new JVM is started and the Java application code is loaded. This results in additional overhead compared to interpreted languages and contributes to initial start-up performance. To understand this better, let's see how AWS Lambda execution environments work.

{ A PEEK UNDER THE HOOD OF AWS LAMBDA }

AWS Lambda execution environments are the infrastructure upon which your code is executed. When creating a Lambda function, you can provide a runtime (e.g. Java 11). The runtime includes all dependencies necessary to execute your code (for example, the JVM). A function's execution environment is isolated from other functions and the underlying infrastructure by using the Firecracker microVM technology [1]. This ensures the security and integrity of the system.

When your function is invoked for the first time, a new execution environment is created. It will download your code, launch the JVM, and then initializes and executes your application. This is called a cold-start. The initialized execution environment can then process a single request at a time and remains warm for subsequent requests for a given time. If a warm execution environment is not available for a subsequent request (e.g. when receiving concurrent invocations), a new execution environment has to be created which results in another cold-start.

Therefore, to optimize Java applications on Lambda one has to look at the execution environment, the time it takes to load dependencies and how fast can it be spun up to handle requests.

{ IMPROVEMENTS WITHOUT CHANGING CODE: CHOOSING THE RIGHT MEMORY SETTINGS WITH POWER TUNING }

Choosing the memory allocated to Lambda functions is a balancing act between speed (duration) and cost. To automate the process of finding the right configuration, use the AWS Lambda Power Tuning application [2].

You can graph the results to visualize the performance and cost trade-off. In the example in Image 1, you can see that a function has the lowest cost at 2048 MB memory, but the fastest execution at 3072 MB.

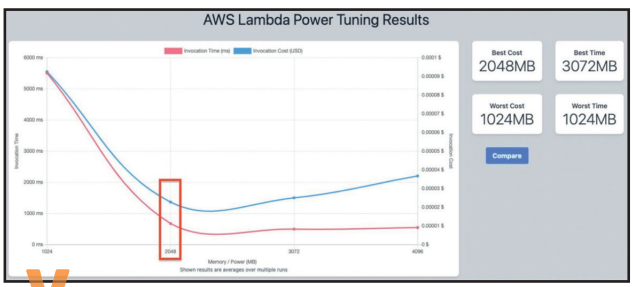


Image 1.

{ ACTIVATING SNAPSTART ON A LAMBDA FUNCTION }

SnapStart is a new feature announced for AWS Lambda functions running on Java Coretto 11. When you publish a function version with SnapStart enabled, Lambda initializes your function and takes a snapshot of the memory and disk state of the initialized execution environment. It encrypts the snapshot, and caches it for low-latency access. When the function is invoked for the first time, and as the invocations scale up, Lambda resumes new execution environments from the cached snapshot instead of initializing them from scratch, improving startup latency. This feature can improve startup performance of latency-sensitive applications

L1

```
public class App implements
RequestHandler<Object, Object> {
    private final S3Client s3Client;
    public App() {
        s3Client = DependencyFactory.s3Client();
    }
    @Override
    public Object handleRequest(final Object
input, final Context context) {
        ListBucketResponse response = s3Client.
listBuckets();
        // Process the response.
    }
}
```

by up to 10x at no extra cost, typically with no changes to your function code. Follow the steps to activate SnapStart on a Lambda function [3].

Other options to consider are enabling tiered compilation on your Lambda function to improve the startup time and performance, and configuring provisioned concurrency to spin up the specified number of execution environments to respond to invocations.

{ CODE REFACTORING: OPTIMIZING FOR THE AWS SDK }

Ensure that any costly actions, such as making an S3 client connection or a database connection, are performed outside the handler code. The same instance of your function can reuse these resources for future invocations. This saves cost by reducing the function duration.

In the example in Listing 1, the S3Client instance is initialized in the constructor using a static factory method. If the container that is managed by the Lambda environment is reused, the initialized S3Client instance is reused.

When utilizing the AWS SDK to connect to other AWS services you can accelerate the loading of libraries during initialization by making dummy API calls outside the function handler. These dummy calls will initialize any lazy-loaded parts of the library and additional processes such as TLS handshakes [4].

{ LEVERAGING CLOUD NATIVE FRAMEWORKS WITH GRAALVM }

GraalVM is a universal virtual machine that enables Ahead-of-Time (AoT) compilation of Java programs into a self-contained native executable, called a native image. The executable is optimized and contains everything needed to run the application, and it has faster startup time and smaller heap memory footprint compared to a JVM.

Modern cloud frameworks such as Micronaut and Quarkus (and the recent Spring Boot 3 release) support building GraalVM native images as part of the build process.

In conclusion, you have seen the various approaches to optimizing Java applications on AWS Lambda. Further references go in-depth with the above suggestions, along with a workshop that you can follow along to do the above and more optimizations on a sample Spring Boot application. <

L2

```
public class App implements RequestHandler <
Object, Object > {
    public App() {
        ddb = getDynamoDBClient();
        DescribeTableRequest request =
DescribeTableRequest.builder()
            .tableName(System.getenv("DYNAMODB_
TABLE_NAME"))
            .build();
        try {
            TableDescription tableInfo = ddb.
describeTable(request).table();
                System.out.println("Table
found:" + tableInfo.tableArn());
        } catch (DynamoDbException e) {
            System.out.println(e.getMessage());
        }
    }
    @Override
    public Object handleRequest(final Object
input, final Context context) {
        //Handler code
    }
}
```

{ REFERENCES }

1 https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html#lambda-microvms-and-workers
2 https://github.com/alexcasalboni/aws-lambda-power-tuning
3 https://docs.aws.amazon.com/lambda/latest/dg/snapstart-activate.html
4 https://aws.amazon.com/blogs/developer/tuning-the-aws-java-sdk-2-x-to-reduce-startup-time/
5 https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/
6 https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-2/
7 https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html
8 https://aws.amazon.com/blogs/compute/reducing-java-cold-starts-on-aws-lambda-functions-with-snapstart/
9 https://catalog.workshops.aws/java-on-aws-lambda/en-US

SPRING BOOT 3

Al het goede komt in drieën

November 2022 was een maand vol met releases vanuit de diverse Spring-teams. De belangrijkste waren de nieuwe generaties van het core Spring Framework en van Spring Boot, met versies 6 respectievelijk 3. Dit artikel geeft een overzicht van de belangrijkste features en wat er bij komt kijken om bestaande applicaties naar deze nieuwe versies te migreren.

Joris Kuipers is bij Trifork werkzaam als CTO en hands-on architect in diverse domeinen, zoals educatie, healthcare, nieuwsmedia, overheid en loterijen.

{ NIEUWE BASELINES: JAVA 17 EN JAKARTA EE }

Om te beginnen is er met het uitbrengen van twee major releases van de gelegenheid gebruik gemaakt om een aantal minimumvereisten op te schroeven. Zo is Java 17 nu de minimaal ondersteunde Java-versie en is de switch naar de Jakarta EE namespaces gemaakt.

Java 17 werd al lange tijd ondersteund door Spring, dus als je huidige project nog een oudere versie gebruikt, is het updaten naar Java 17 een logische eerste stap. Spring 6 ondersteunt versies tot en met 19.

De Jakarta EE adoptie betekent dat Servlet-containers, JPA-providers, Bean Validation implementaties etc. allemaal gebruik moeten maken van de Jakarta-packages, daar waar voorheen Javax werd gebruikt. Oracle heeft de Java EE specificaties bij de Eclipse Foundation ondergebracht, maar daarbij het voorbehoud gemaakt dat de Javax namespace aangepast moest worden, omdat deze onder het Java-copyright van Oracle valt. Daarmee is deze qua packages incompatibel met de vorige versies van de bevatte standaarden, hoewel er verder in eerste instantie niets is veranderd.

Spring Boot 3 ondersteunt Jakarta EE 9 en 10. De eerste bevat eigenlijk alleen de package change, de tweede bevat daadwerkelijk nieuwe functionaliteit zoals onder andere geboden door Servlet 6.0 en JPA 3.1

De dependency's die Spring Boot standaard levert, zijn allemaal geüpgrade naar recente Jakarta EE-compatibel versies: denk bijvoorbeeld aan Tomcat en Hibernate ORM & Bean Validator. Voor Hibernate 6 zul je waarschijnlijk wel wat aanpassingen moeten maken in je applicaties: JBoss heeft een migration guide [1] gepubliceerd.

Voor je eigen code geldt ook dat deze de nieuwe namespaces moet gebruiken, dus je moet het een en ander migreren. Dit is grotendeels een simpele 'search & replace': je IDE of migratietools kunnen je hier ook bij helpen. Hou er rekening mee dat je ook je dependency's moet bijwerken, dus waar je eerder gebruik maakte van javax.servlet:javax.servlet-api wordt dat nu jakarta.servlet:jakarta.servlet-api, bijvoorbeeld.

Voor leveranciers van implementaties van Enterprise Java-standaarden is het onbegonnen werk om op langere termijn versies te ondersteunen voor zowel Java EE als Jakarta EE, zodat de switch naar de laatste voor een nieuwe generatie van de Spring-frameworks een logische stap is.

Dit betekent wel dat alles wat je gebruikt nu dus compatibel moet zijn met Jakarta EE; denk hierbij ook aan tooling als de OpenAPI Generator die code genereert die gebruik maakt van bean validation annotations! De 'spring'-generator van deze tool biedt inmiddels een `useSpringBoot3`-optie, maar wellicht ondersteunt niet al je tooling reeds Jakarta EE: check dit dus als je een upgrade overweegt.

{ AOT EN NATIVE COMPILATION }

De afgelopen paar jaar is er hard gewerkt aan de ondersteuning van native compilation met GraalVM, in een apart Spring Native-project. Het eindresultaat van dit werk is nu beschikbaar in

Spring 6 en Spring Boot 3. Hiermee is Spring Native als project komen te vervallen.

Hiermee kun je je applicaties omzetten naar machinecode: in plaats van bytecode die op een JVM runt, krijg je een native executable.

Het voordeel hiervan is dat deze veel sneller start dan een JVM-based applicatie en veel minder geheugen gebruikt. Het nadeel is dat de buildtime daarmee flink toeneemt, maar belangrijker is dat door het ontbreken van een JVM veel dynamische aspecten van Java (zoals runtime classloading en reflection) niet zomaar werken. Voor een framework als Spring levert dat flink wat uitdagingen.

Een belangrijk deel van de oplossing wordt geleverd door een nieuwe AOT-module. AOT staat voor 'ahead of time': er wordt tijdens het build-proces al veel werk verricht op basis van analyse van je applicatie, zodat er 'at runtime' niet meer geleund hoeft te worden op functionaliteit die niet beschikbaar is in een native image.

Spring Boot moet hier wat aannames voor maken over zaken. Bijvoorbeeld hoe je classpath er at runtime uit zal zien, zodat er geen classpath scanning meer plaats hoeft te vinden. En welke Spring-profiles er actief zullen zijn, zodat vaststaat welke beans er in je ApplicationContext gedefinieerd zullen zijn. Daarmee staat ook op voorhand vast welke auto-configuration er actief moet zijn. Anders zou je native image alle mogelijke Spring Boot @AutoConfiguration classes moeten bevatten.

Deze informatie wordt vervolgens gebruikt om code te genereren die de setup uitvoert die bij een JVM-based Spring-applicatie dynamisch bepaald en uitgevoerd zou worden tijdens het op-

starten. Door deze 'closed world assumptions' lever je een stukje dynamisch gedrag in om er een sneller startende en efficiëntere applicatie voor terug te krijgen.

Veel onderdelen van het Spring-ecosysteem werken al samen met native compilation, maar controleer vóórdat je hiermee gaat werken of dat voor al je dependency's geldt!

Het resultaat van de AOT-module is code die veel geschikter is om met GraalVM tot een native image gecompileerd te worden. Ook worden GraalVM-specifieke hints gegenereerd. De AOT-functionaliteit kan ook zonder GraalVM gebruikt worden om applicaties sneller op te laten starten en, door het verminderde gebruik van reflectie, in sommige situaties ook wat minder geheugen te laten gebruiken. Die voordelen zijn bij gebruik van een JVM echter een stuk minder dramatisch dan bij gebruik van een native image.

Dat gezegd hebbende vindt er momenteel veel onderzoek plaats naar hoe JVM-based applicaties ook sneller kunnen opstarten, waar het Spring-team actief aan meewerkt. Zo is het de verwachting dat latere 6.x versies van Spring ondersteuning zullen bieden voor Project CRaC [2].

Voer je een native image build uit op Windows of MacOS met gebruik van de GraalVM Native Build Tools? Dan komt daar een executable voor die specifieke platformen uit. Vaak zul je liever een docker image met een Linux-executable genereren: hiertoe biedt Spring Boot opties aan die gebruikmaken van Cloud Native Buildpacks. Beide werken zowel met Maven als Gradle. Lees voor meer informatie de native image documentatie [3].

{ WE ZULLEN ZIEN: MICROMETER-BASED OBSERVABILITY }

Spring Boot biedt in de 2-serie al langer ondersteuning aan voor het produceren van metrics via Micrometer. De Actuator-module instrumenteert diverse Spring-onderdelen, zoals de web frameworks, en via Micrometer worden deze metrics aan observability-tools zoals Prometheus of Datadog beschikbaar gemaakt. Voor distributed tracing wordt daarbij gebruik gemaakt van Spring Cloud Sleuth, een framework dat leunt op OpenZipkin's Brave.

Met distributed tracing zorg je dat alle services die betrokken zijn bij het afhandelen van een request een gedeeld trace-ID ter beschikking hebben voor gebruik in logging, die vervolgens gecorreleerd kan worden. Traces bestaan uit een of meerdere 'spans', die indien gewenst ook geëxporteerd kunnen worden naar tools als OpenZipkin. Daarmee wordt de flow binnen een servicesarchitectuur inzichtelijk en vooral waar in de keten bottlenecks in de verwerking zitten.

Sleuth komt te vervallen. In plaats daarvan bevat de laatste versie van Micrometer een nieuwe abstractie voor metrics én tracing. Deze nieuwe Micrometer-observation-module is in Spring 6 een directe dependency van Spring-onderdelen, zoals spring-web en spring-rabbit.

Daarmee is er één observability-abstractie gekomen en kan Spring-code die bovendien direct aanspreken vanuit de code, in plaats van dat Actuator- of Sleuth-code van buitenaf instrumentatie moet verzorgen. Dit maakt veel simpelere, betere en diepere integratie mogelijk dan voorheen het geval was.

De nieuwe Micrometer ondersteunt naast OpenZipkin ook OpenTelemetry [4] en maakt standaard gebruik van W3C context propagation [5] in plaats van OpenZipkins B3 headers [6].

Al met al is dit een hele grote stap in de support voor observability als first class citizen binnen het Spring-ecosysteem. Hopelijk worden andere onderdelen in het Spring-ecosysteem ook snel uitgebreid met deze nieuwe observability-ondersteuning.

{ DECLARATIEVE HTTP-CLIENTS }

Spring 6 biedt een nieuwe feature aan waarbij HTTP-clients gegenereerd kunnen worden op basis van interfaces en annotaties. Dit lijkt erg op wat OpenFeign ook mogelijk maakt, maar is nu dus direct ondersteund vanuit het framework.

Hierbij een voorbeeld uit de Spring-documentatie. Allereerst definieer je een interface met een of meerdere methods die je voorziet van een nieuwe exchange-annotatie:

```
interface RepositoryService {

   @GetExchange("/repos/{owner}/{repo}")
   Repository getRepository(@PathVariable String
owner, @PathVariable String repo);

   // more HTTP exchange methods...
}
```

Zoals je kunt zien worden er, naast de nieuwe `@GetExchange`, ook bestaande Spring Web-annotaties, zoals `@PathVariable` gebruikt.

Vervolgens vraag je aan Spring om hier een implementatie bij te maken:

```
WebClient client = WebClient.builder().
baseUrl("https://api.github.com/").build();
HttpServiceProxyFactory factory =
HttpServiceProxyFactory.builder(WebClientAdapter.
forClient(client)).build();
RepositoryService service = factory.
createClient(RepositoryService.class);
```

Merk op dat hier de reactive WebClient als onderliggende client wordt gebruikt. Dit verplicht je echter niet om in je interface ook reactive types als Mono en Flux te gebruiken.

Overigens werkt het Spring-team aan een Loom-compatible versie van de WebClient als alternatief voor de reactive implementatie. Daarmee zou eindelijk een volwaardige opvolger voor de RestTemplate beschikbaar komen voor applicaties die geen gebruik maken van Reactor en Webflux.

Raadpleeg voor meer informatie de documentatie [7].

{ HOUSTON, WE HAVE A PROBLEM }

Spring Boot heeft altijd ondersteuning aangeboden voor het genereren van error responses in webapplicaties voor uncaught exceptions. Het formaat van de errors is echter vrij beperkt en bovendien Boot-specifiek.

Met Spring 6 is er ondersteuning gekomen voor de 'Problem Details for HTTP APIs' specification, RFC 7807 [8]. Hiermee kun je fouten op een standaard manier beschrijven in een JSON-response met zowel voorgedefinieerde als custom velden. Het contenttype van zo'n response is `"application/problem+json"`.

Dit is een voorbeeld van zo'n error response:

```
{
 "type": "https://example.com/probs/out-of-
credit",
 "title": "You do not have enough credit.",
 "detail": "Your current balance is 30, but that
costs 50.",
 "instance": "/account/12345/msgs/abc",
 "balance": 30,
 "accounts": ["/account/12345", "/account/67890"]
}
```

Zoals je ziet, bevat het type-veld een URI. Hoewel niet verplicht, is het wel de verwachting dat dit een URL is die leidt naar een webpagina met foutdocumentatie. Balance en accounts zijn custom velden.

Persoonlijk ben ik nog geen API's tegengekomen die deze standaard gebruiken: wellicht brengt deze integratie daar verandering in.

De ondersteuning is er voor zowel Spring-MVC als Webflux. Zie voor meer informatie over hoe je deze feature aanzet de documentatie [9].

Een tip: ben je geïnteresseerd in meer opties voor web-error-handling in Spring Boot applicaties? Kijk dan eens naar de error-handling-spring-boot-starter library van Wim Deblauwe [10]. Met versie 4.0.0 biedt deze ook al ondersteuning voor Spring Boot 3!

{ WHAT ELSE? }

In dit artikel zijn de belangrijkste grote veranderingen beschreven, maar er zijn nog heel veel kleinere zaken aangepakt. In de release notes, blog post en andere bronnen kun je daar mee over vinden.

Voor meer informatie over het upgraden naar Spring Boot 3 is het goed om de migration guide [11] door te nemen. Zo zul je wellicht je eigen autoconfiguration moeten aanpassen als deze nog gebruik maakt van de spring.factories-file, of moeten updaten naar een nieuwe versie van de Apache HTTP-client.

In de nieuwe versies van frameworks als Spring Security zijn ook wel wat zaken aangepast; zo zijn de antMatchers en mvcMatchers methodes vervangen door een requestMatchers methode, iets waar je waarschijnlijk tegenaan gaat lopen als je Spring Security gebruikt.
Ook is er een dedicated tool beschikbaar, de Spring Boot Migrator (SBM) [12], gebaseerd op OpenRewrite waar je in de volgende editie van dit magazine een artikel over vindt.

{ REFERENTIES }

1 https://docs.jboss.org/hibernate/orm/6.0/migration-guide/migration-guide.html
2 https://github.com/CRaC/docs
3 https://docs.spring.io/spring-boot/docs/3.0.0/reference/html/native-image.html
4 https://opentelemetry.io/
5 https://www.w3.org/TR/trace-context/
6 https://github.com/openzipkin/b3-propagation
7 https://docs.spring.io/spring-framework/docs/current/reference/html/integration.html#rest-http-interface
8 https://www.rfc-editor.org/rfc/rfc7807.html
9 https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-rest-exceptions
10 https://github.com/wimdeblauwe/error-handling-spring-boot-starter
11 https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide
12 https://github.com/spring-projects-experimental/spring-boot-migrator

DRIE VLIEGEN IN ÉÉN TRAP MET KARATE

Dit artikel is een beknopte kennismaking met de Karate Netty MockServer, een sub-project binnen het uitgebreide Karate-platform voor testautomatisering, dat ook tools bevat voor API-, GUI- en performancetesten (die hier niet aan de orde zullen komen).

Jasper Sprengers is senior Java-engineer bij Team Rockstars IT en enthousiast blogger over onder andere testen en clean code.

Een mock HTTP-server stelt je in staat om client- en server-communicatie in detail te simuleren. Dat heeft drie voordelen, waar we in detail naar zullen kijken.

1. Frontenders kunnen zelfstandig experimenteren met edge-case scenario's en zo hun applicatie robuust maken.
2. De configuratiefiles die het gedrag van de server bepalen, vormen een belangrijk stuk documentatie als informeel contract tussen frontend en backend.
3. GUI-testen (met bijvoorbeeld Cypress, Selenium of Playwright) kun je ook draaien tegen een MockServer. Dat valideert dat de MockServer getrouw is aan productie. Bovendien kun je op die manier veel uitgebreidere scenario's testen, die ook nog sneller en stabieler draaien.

{ DE GROTE ONTKOPPELING }

Ruim twintig jaar geleden, in de begindagen van webapplicaties, was er nauwelijks scheiding tussen frontend- en backend-expertise zoals we die nu kennen. Als goede ontwikkelaar hield je model, view en controller hopelijk netjes gescheiden, maar je moest wel van alle markten thuis zijn. Iedereen was full stack. Door de tomeloze opmars van de browser als ontwikkelplatform werd frontend-ontwikkeling een volwaardige discipline en carrière. Ook bij de backend sloeg de specialisering toe, met als gevolg dat diepe kennis van beide werelden nu noodzakelijkerwijs over meerdere hoofden verdeeld is.

Die 'loose coupling' tussen disciplines kan goed werken, zolang er maar overeenstemming is over de API: het cruciale contactpunt tussen frontend en backend. Zodra we het eens zijn over welke URL's welke berichtstructuren verwachten en terugsturen, is het fijn als de frontenders die al op een werkende omgeving kunnen aanspreken, nog voordat de features geïmplementeerd zijn op de ontwikkelstraat.

Een MockServer voorziet daarin. Deze draait op je eigen ontwikkelmachine, maar voldoet wel volledig aan het API-contract tussen frontend en backend, alleen dan met statische, gefingeerde data. Invoer en respons voor elke interactie kun je volledig naar je eigen hand zetten. Dit is inmiddels gevestigde technologie. Zo is binnen het Java-domein Wiremock populair om REST-verkeer tussen microservices te simuleren. Dit framework kun je programmatisch (in Java) configureren, of met JSON-files.

In Karate schrijf je daarentegen alles in het beter leesbare Gherkin-formaat. De MockServer gebruikt een syntax, waarbij de interacties tussen client en server zijn uitgedrukt als een scenario. Als je gewend bent met Gherkin te werken, moet je wel bedenken dat dit géén uitvoerbare BDD-scripts zijn. De HTTP-server ontvangt een client-request en zoekt in de beschikbare scenario's naar een match. Deze staan in feature files, zoals in dit voorbeeld, dat een `POST` en `GET` simuleert op de URL /api/v1/person.

```
File: api.feature
Scenario: pathMatches('/api/v1/person') &&
methodIs('post')
  * def response = {name: "John Doe"}
Scenario: pathMatches('/api/v1/person') &&
methodIs(get)
  * def response = [{name: "John Doe"}, {name:
"Jane Doe"}]
```

De standalone server is een jar-bestand dat je opstart met een verwijzing naar een of meerdere van deze feature files en een poortnummer.

```
java -jar karate.jar -m product.feature -m order.
feature -p 8090 -W
```

Met de `-w`-flag zijn wijzigingen meteen beschikbaar zonder herstart. Frontenders hebben enkel een recente JDK nodig, maar verdere Java-kennis is niet nodig. In bijbehorend GitLab-project vind je alle voorbeelden uit dit artikel, met extra uitleg, commentaar en links (zie Links).

Karate heeft een gebruiksvriendelijke, bondige syntax die toch enorm expressief is. Lange responsberichten kun je in een eigen bestand zetten, zodat je vaak in één scherm alles kunt uitdrukken wat de clientapplicatie van de backend nodig heeft. Naast een implementatie voor de succes-scenario's kun je dan per endpoint alternatieve scenario's configureren die getriggerd worden door meer specifieke input: een request header, query parameter, of een specifieke key ergens in de request payload. Zelfs reageren op de naam in een bestandsupload is mogelijk. Alles wat je terugstuurt is te manipuleren, zoals HTTP-responscode, headers en uiteraard de body. Of je zet de responstijd op drie seconden om te checken of de browser een juiste melding geeft. Een paar voorbeelden:

```
Scenario: pathMatches('/friends/1') &&
methodIs('get')
  * def response = {id: 1, "name": "John"}

Scenario: pathMatches('/friends/666') &&
methodIs('get')
  * def responseCode = 500

Scenario: pathMatches('/friends') &&
paramValue('age') > 30 && methodIs('get')
  * def response = [{id: 1, "name": "John"}]

Scenario: pathMatches('/friends') &&
methodIs('get')
  * def response = read('friends.json')

Scenario: pathMatches('/friends/search') &&
request.married == false > 20 && methodIs('post')
  * def response = [{id: 2, "name": "Holly"}]

Scenario: pathMatches('/v1/image/upload') &&
requestParts['photo'][0].name == 'valid.jpg'
  * def responseDelay = 1000
  * def response = "ACCEPTED"

Scenario: pathMatches('/v1/image/upload') &&
requestParts['photo'][0].name == 'invalid.jpg'
  * def responseDelay = 4000
  * def response = "REJECTED"
  * def responseCode = 400
```

Zo creëer je allerhande 'edge cases' en die zijn vanuit testoogpunt waardevol. Je merkt veel sneller of je frontend robuust genoeg is als je voor elke URL een alternatief scenario met http 500 of timeout toevoegt. Het is veel lastiger om dit soort gedrag op een echte backend te simuleren, als je tenminste je productiecode niet wil vervuilen met test-specifieke switches. Bovendien heb je dan andermans hulp en expertise nodig, plus een complete backend-instantie, die vaak te groot en complex is om volledig lokaal te draaien.

Dit soort edge case-configuraties kun je het beste in een apart bestand zetten dat je inlaadt vóór de scenario's die enkel matchen op URL en method. Zoals we al zagen, gebruikt Karate namelijk het eerste scenario dat een goede match is en zoekt daarna niet verder naar een meer specifieke match. Vergelijk het maar zoals je in Java een FileNotFoundException altijd vóór de IOException moet afvangen, omdat de laatste generieker is. Alleen is er bij Karate geen compiler die je dat verhindert.

Tot zover de syntax over de feature files. Er is nog veel meer mogelijk en het is allemaal goed gedocumenteerd. Zo ondersteunt Karate ook variabelen in een feature file, die je kunt manipuleren via `POST`'s en uitlezen met `GET`'s. Ziedaar: een heuse stateful server, die zelfs inline JavaScript ondersteunt. Dat klinkt aantrekkelijk voor sommigen, maar wees hier terughoudend in. Het reële gevaar is dat je stukken backend-logica gaat nabouwen en dat is ongewenst. De waarheid wat betreft de bedrijfsregels ligt bij de backend, niet bij de MockServer. Die helpt ons enkel om het contract te valideren, zoals dat is vastgelegd in de feature files. Implementatiedetails horen daar niet in thuis.

De overeenkomst die je in Karate-feature files beschrijft, is een consumer-driven-contract tussen frontend en backend. De implementatie van de backend kan uit meerdere (micro)services bestaan: dat maakt voor de frontend niet uit. Het contract legt enkel de interacties vast die door de frontend gebruikt worden. In het volgende voorbeeld kunnen bijvoorbeeld `hr` en `finance` verwijzen naar twee verschillende REST-services (de een in Java, de ander in Python en mogelijk extern), die middels een proxy via dezelfde host toegankelijk zijn gemaakt.

```
Scenario: pathMatches('/hr/api/v1/employee/12345')
  * def response = {id: 12345, "name": "John"}
Scenario: pathMatches('/finance/api/v3/payslips)
  * def response = []
```

Bij provider-driven-contracts daarentegen, zoals OpenAPI of het aloude WSDL, laat een service in detail weten wat hij in de aanbieding heeft. Een individuele webclient heeft van alle beschikbare endpoints misschien maar een klein deel nodig, maar doet zijn boodschappen dan weer bij zes verschillende services, via een proxy. Dankzij de bondige syntax van Karate kun je zo'n consumer-driven-contract beknopt en toch volledig formuleren.

{ EIGENAARSCHAP EN CONSISTENTIE }

Prima voor een hobby-project denk je misschien, maar zo'n set handmatig bijgehouden bestanden die de waarheid behoren te emuleren gaat onherroepelijk afwijken van de werkelijkheid, toch? Dat is de wet van de software-entropie. Iemand in de organisatie moeten we eigenaar maken van de MockServer, maar wie? Het frontend-team is weliswaar de belangrijkste gebruiker ervan, maar zij zijn van de backend afhankelijk, niet omgekeerd. Het gedrag van de MockServer moet exact overeenkomen met dat van de echte server(s). Dit moet je op elk moment kunnen valideren en liefst als onderdeel van de build.

Over dat automatisch valideren zo dadelijk meer. Ja, de backenders zijn eigenaar van de API, maar zij zijn wel dienstig aan de frontend (het woord 'server' verraadt het al). Een goede API ontwikkel je altijd in nauwe samenwerking met de gebruikers. Het is dus goed als de backend-ontwikkelaars die de API bouwen al meteen een 'reference implementation' (de basale happy-flow-scenario's) schrijven en beheren. Vraag ze vervolgens steeds om een review als je als frontender wijzigingen aanbrengt.

Dat handmatig onderhoud valt trouwens best mee als de API eenmaal stabiel is (ik spreek hier uit ervaring). Als je bovendien strikt vasthoudt aan versienummers voor je API hoef je die happy-flow-scenario's nog maar zelden te wijzigen. Je zult alleen nieuwe toevoegen. In de wat meer fluïde beginfase van een project moet je extra opletten dat je mock-configuraties niet uit de pas gaan lopen, maar dan is constante afstemming sowieso onontbeerlijk, met of zonder MockServer. Bovendien hoeven de backenders zich alleen te bekommeren om de happy-flows en zoals je in de voorbeelden kon zien, is dat echt niet veel werk. Alle alternatieve (faal)scenario's kunnen de testers en frontenders vervolgens afleiden van die happy-flows.

Kunnen we die feature files niet laten genereren vanuit een OpenAPI-bestand en zo inconsistenties voorkomen? Dat kan zeker, maar een consumer-driven-contract wordt per definitie niet door de backend gedicteerd. Die weet immers niet welke subset de frontend nodig heeft. Zelfs in Karate wordt zo'n configuratie dan topzwaar en onoverzichtelijk. Je kunt schrappen wat je niet nodig hebt, maar dat betekent een terugkerende handmatige stap.

{ COMPONENT-TESTS VAN DE FRONTEND }

Als je gebruik maakt van een GUI-testframework als Cypress, Selenium of Playwright, is het niet lastig om te valideren dat je MockServer zich hetzelfde gedraagt als de echte backend. Voer dezelfde testscripts uit op beide omgevingen en je hebt je antwoord. In een volwassen testopzet ben je toch al gewend om je tests tegen verschillende omgevingen in de OTAP-straat te draaien. In de praktijk betekent dat niet meer dan het configureren van een extra URL voor je lokale MockServer, bijvoorbeeld localhost:8090. De tests zouden onveranderd groen moeten blijven. Zie je namelijk discrepanties tussen de mock-omgeving en de test- of acceptatieomgeving? Dan is er mogelijk iets mis met je Karate configuratie.

Het testen van je clientapplicatie via de browser tegen een gemockte backend is sneller en stabieler dan tegen een echte backend. Let wel op het principiële verschil tussen deze werkwijze en de ouderwetse end-to-end testen, waar we GUI-frameworks meestal voor inzetten. GUI-tests met een MockServer zijn namelijk component-tests. Je focust je enkel op het gedrag van de applicatie in de browser. Of het berichtverkeer met de backend correct is, beschouwen we als gegeven, want die is gemockt. We zijn niet geïnteresseerd of de implementatie werkt, cru gezegd. Die focus is goed, want je kunt zodoende veel uitgebreidere validaties doen op de frontend, die tegen een echte omgeving vaak traag en instabiel blijken.

Door een subset van dezelfde scripts alsnog tegen de echte omgeving te draaien, krijg je de bevestiging dat de MockServer een getrouwe dubbelganger van productie is. Dat weet je natuurlijk pas zodra er ook een omgeving is om tegen te testen. Reden te meer om de backend regelmatig uit te rollen op een ontwikkelomgeving en het testen tegen de MockServer onderdeel te maken van de build pipeline.

{ CONCLUSIE }

Met Karate bouw je in no-time een zeer flexibele MockServer waarmee frontend-ontwikkelaars en testers allerhande (fout) scenario's in detail kunnen simuleren, zonder assistentie van backend-ontwikkelaars. Mits je goede afspraken maakt over onderhoud en eigenaarschap van de configuraties zijn deze een waardevolle bron van documentatie en bevorderen ze de samenwerking tussen teams met verschillende cycli van oplevering.

{ LINKS }

- https://gitlab.com/jasper-sprengers/karate-netty-demo
- https://karatelabs.github.io/karate/karate-netty/

BYTE SIZE

{ GIVE IT A CHANCE: PROPOSE AN ABSTRACT }

Our passion is first and foremost Java. But that's not where our interest ends. Just looking at what our editorial committee members use on a daily basis or what they use for hobby projects, shows subjects such as frontend development, AI & machine learning, testing and the list goes on and on... What we are trying to say is: if our interest already diversifies like this, we think our readers wouldn't mind some trips to these sort of subjects as well. So, do you want to write for Java Magazine, but are you on a personal or professional level actually also busy with other software than just Java? Why not just give it a chance and propose an abstract?

› **Java Editorial Committee**

{ WHAT'S IN A NAME }

Ontwikkelaars zijn over het algemeen slecht in het bedenken van namen. Hier goed over nadenken zal voor duidelijkere code zorgen. Zo zal een variabele genaamd 'customers' duidelijker communiceren wat dat betreft dan 'theList'. Dit principe geldt natuurlijk ook voor method en class-namen.

› **Mark Hendriks**

{ LESS COMPLEXITY AND CLEANER CODE }

Formly is a nice Angular library that enhances maintainability. A module based on Reactive Forms has been migrated to Formly. This reduced HTML from 887 to 29 lines of code. TypeScript code grew a bit from 3934 to 4256 lines (8%). After the migration, new stories turned out to take much less time to implement. Formly's features such as templateOptions, wrappers, expressionProperties, and hooks are a powerful tool that reduces complexity and supports clean code. Check out: formly.dev

› **Erik Kerkhoven**

{ TRIP DOWN MEMORY LANE }

Kotlin just turned 12! Turns out, the initial commit of the Kotlin language was created on the 8th November, 2010! Have a look down for a trip down memory lane 😊. It's a nice way to see the very first implementation of the data structures we still use today (with a .jetl extension back then!)

https://github.com/JetBrains/kotlin/commit/369b1974782b821e44b7aa6cd68e2e41eb2ba036

› **Julien Lengrand-Lambert**

HOW TO WRITE A BYTE SIZE' BYTE SIZE

Do you have a tip to share, an opinion, or simply want to share something short? That's what the byte size format is for! Byte sizes are short bits of information, in a Twitter format: you have roughly 280 characters, and space enough for a link, or a picture! Send us your byte sizes by @JavaMagazineNL on Twitter or by mailing **info@nljug.org**!

› **Java Magazine Editorial Committee**

CHILIT

VOOR DEVELOPERS MET PIT!

Ben jij een developer die méér brengt? Méér waard is? Méér pit heeft? Ontdek dan wat CHILIT voor jou kan betekenen.

Zo geniet jij straks van;

- ☑ een vast salaris,
- ☑ schaalbare bonussen,
- ☑ veel vrijheid,
- ☑ volop zekerheid!

WIL JIJ JE SPRING BOOT APPLICATIE IN MINDER DAN EEN SECONDE LATEN STARTEN? VERZEKER JE NU VAN EEN PLEKJE IN ONZE EERSTVOLGENDE KENNISSESSIE!

www.chilit.nl/spring-boot-3-workshop

Chilit B.V.
Utrechtseweg 75, Zeist

088 224 5480, hello@chilit.nl
Check onze beloningsstructuur op chilit.nl

V

COLUMN

Mindcraft modding

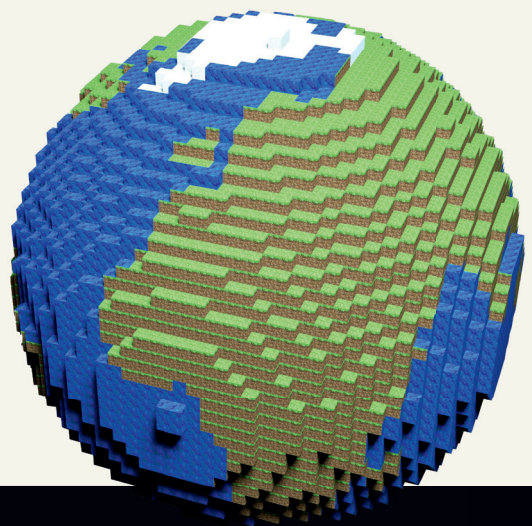
Maja Reißner *is software developer bij SIDN.*

Mindcraft is all about Java. Or at least it is until you mod yourself out to another island. You probably have heard of or played Minecraft (also Java) before. Mindcraft is its VR (visual reality) sibling. Easy to set up, addictive to play in both single player and multiplayer mode.

So how does Mindcraft work? Close your eyes and visualize yourself being on Java. The more experienced you are with Java and/or Mindcraft, the better your visual resolution will be. As a bloody beginner, you will walk around on a green, mountain-rich island and a couple of Indonesian take-away restaurants. You can chill out in a bar, hit on the Swedish tourist or go snorkeling in the deep blue water. After a while, you may stumble on Jakarta on the north-western coast. There's also the Buddhist temple Borobudur that is worth a bit of time. This is the original Mindcraft world. You decide where you go, you decide what you do. You could of course dig around a bit in the temple, but it's really not the purpose of the game to mine volcanic rock. I mean, why would you want to do so anyway? It's Mindcraft, not Minecraft.

The core of Mindcraft is pure Java and a widely loved feature of the game is its sheer amount of possibilities for modding. The game is built with an absurd flexibility in mind. If you like you can use the existing modification and play on Kotlin. This mod compiles much faster (because there's less to do on Kotlin, obviously) but then again, you're stuck on Kotlin and I personally don't see why you would prefer the cold Gulf of Finland to the beautiful Java sea. You can, of course, also build your own mod and introduce features such as flying or teleportation.

Mindcraft is free, the Mindcraft world is huge, full of adventures and just absolutely stunning. I also love the fact that you have an unlimited amount of lives there. When you're stuck, you can just restart at any point in time and place you like. Please be aware that there's a paid rip off of Mindcraft which is called 'RealLife'. It is of most importance to realize that you only have one life there. You start the game by buying a plane ticket to Java. The graphics are wonderful but you're bound to lots of rules, starting with gravity but also possibly a controversial new Indonesian crime code that includes outlawing sex outside marriage. Make sure to read the fine print first or simply stick with Mindcraft.



VAN HET BESTUUR

Met dit Java Magazine is ook 2022 tot een einde gekomen. We kunnen terugblikken op een prachtig post-Corona jaar, waarin we weer volop kennis en plezier hebben gedeeld in en rondom het Java-ecosysteem. Hoewel digitaal samenkomen in dit hybride tijdperk niet meer weg te denken is, merk ik juist hoeveel energie er ontstaat als we als community fysiek bij elkaar komen tijdens conferenties en Meetups.

{ J-FALL }

Daarover gesproken. Het voelt inmiddels alweer lang geleden, en tegelijkertijd als gisteren, dat we de Pathé in Ede met onze kenmerkende J-Fall vibe hebben gevuld. En wat een editie was het weer. Dank aan alle (keynote) sprekers die J-Fall weer een weergaloos inhoudelijk sterk karakter hebben gegeven, de exposanten die op de vloer hun verhaal hebben verteld, en uiteraard ook dank aan de sponsoren, die deze J-Fall weer mede mogelijk hebben gemaakt.

{ INNOVATION AWARD WINNER 2022 }

Via deze weg wil ik ook graag Omoda feliciteren als de door het publiek uitgeroepen winnaar van de NLJUG Innovation Award 2022 voor hun project 'PhotoChopper: automating photo adjustments for an online clothing shop'. Met een zinderende finale op het podium zijn zij er met de prijs vandoor gegaan. Ik heb uit betrouwbare bron begrepen dat de award inmiddels op een mooi plekje prijkt.

{ MASTER OF JAVA 2022 }

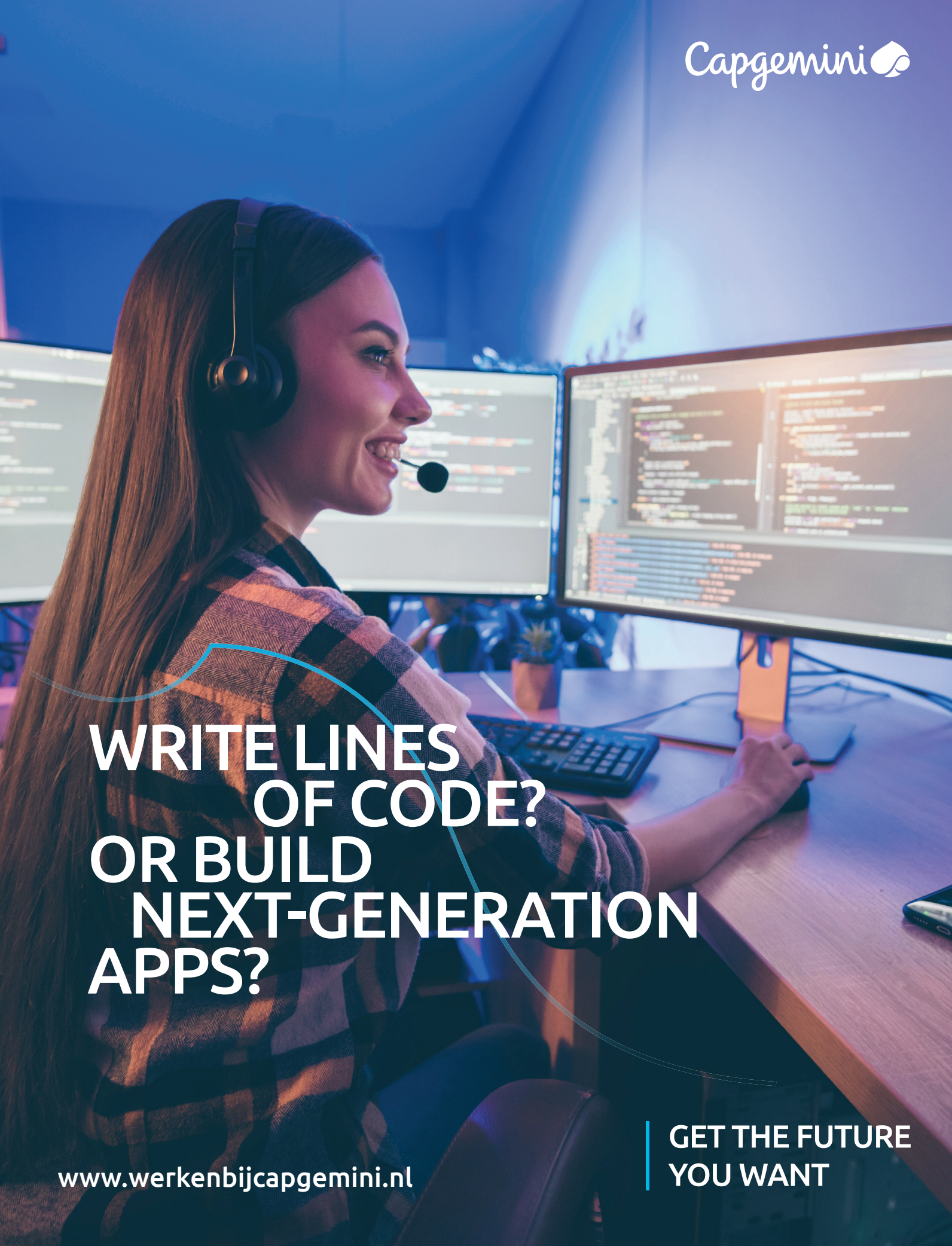
Uiteraard ook stevige felicitaties voor de winnaar van de Masters of Java: Jasper van Merle, meestrijdend als team camel_case! Vorig jaar nog tweede, maar dit jaar heb je in een spannende wedstrijd de competitie achter je weten te laten tijdens dit officieus NK Java Programmeren, congrats!

{ 2023 (EN VERDER) }

Het jaar is voorbij gevlogen en 2023 is alweer van start gegaan. Als bestuur komen wij in januari weer bij elkaar om te bepalen hoe we ook komend jaar weer een mooi NLJUG-jaar kunnen maken, onder andere om een onvergetelijke 20e editie van J-Fall te organiseren. Persoonlijk kijk ik in 2023 uit naar een nog intensievere samenwerking met hogescholen en universiteiten vanuit de NLJUG Academy, en naar het verder ontwikkelen van de Innovation Award. En zoals elk jaar weer kijk ik natuurlijk nu al uit naar de volgende TEQnation, J-Spring en J-Fall, die eigenlijk alweer om de hoek staan...

Namens het hele bestuur hoop ik dat iedereen een bijzonder fijne Kerst en een prachtig uiteinde heeft gehad, en een nog mooier 2023 tegemoet gaat. We zien jullie graag in dit nieuwe jaar op een van de conferenties...wellicht op het podium als spreker?

Namens het hele bestuur,
Bas W. Knopper


Capgemini
WRITE LINES OF CODE?
OR BUILD NEXT-GENERATION APPS?
www.werkenbijcapgemini.nl
GET THE FUTURE YOU WANT

TO THE ROOT

THE PLATFORM FOR **IT PROFESSIONALS**

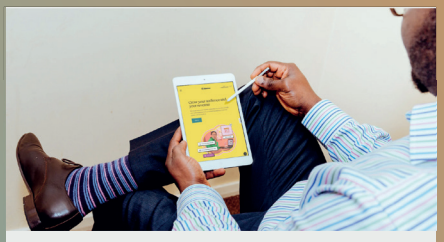
GAIN AND SHARE KNOWLEDGE, INTERACT WITH OUR ACTIVE COMMUNITY

www.totheroot.nl | www.ordina.nl | chat.openai.com | www.stackoverflow.com

/// ARTICLES

DevOps
HOW TO CREATE A PWA WITH THE VUE CLI

BY **MICHEAL AWAD**



DevOps
HOW TO CREATE A PWA WITH THE VUE CLI

BY **MICHEAL AWAD**

DevC
HOW TO CREATE A PWA WITH THE VUE CLI

BY M

SUBJECTS

- Frontend
- Security
- XR
- Data Engineering
- Design
- DevOps
- Agile
- AI/ML
- Cloud

CODE WITH US
READ WITH US
WRITE WITH US.

www.totheroot.nl