Neue Serie: Flutter und Dart für Java-Entwickler S. 43
Neue Serie: Go in der Praxis S. 53
Neue Serie: Erfolgreiches Arbeiten im Brownfield S. 68

Java™ magazin
Java | Architektur | Software-Innovation

JAVA 23

MEHR SEIN ALS SCHEIN

entwickler.de
Ausgabe 11.2024

Java und Angular unter einem Dach mit Nx

One Repo to Rule Them All

Die Zeiten der Monolithen sind vorbei. Heute hat man meist ein Backend und ein Frontend, oft auch mehrere Microservices und -frontends – und für jedes ein eigenes Repository. Diese Trennung bringt aber auch einige Probleme mit sich. Nx verspricht, die meisten davon durch seinen Ansatz eines integrierten Monorepos zu lösen und darüber hinaus einen Mehrwert zu liefern. Schauen wir uns einmal genauer an, was dahintersteckt.

von Marco Sieben

Die Strategie „Ein Repo für ein Projekt“ ist weit verbreitet und hat sich über viele Jahre bewährt. Warum also etwas daran ändern? Wenn wir ehrlich sind, fallen uns wahrscheinlich ohne groß nachdenken zu müssen sofort einzelne Fälle ein, in denen wir uns über die verschiedenen Repos geärgert haben und unsere Arbeit dadurch ineffizienter wurde. Anhand einer kleinen Beispielanwendung wollen wir uns das genauer anschauen und einen Finger auf die Probleme des Multi-Repo-Ansatzes legen. Anschließend werden wir sehen, wie wir zu einem Monorepo migrieren können, das mit Nx verwaltet wird, und welche Verbesserungen wir dadurch erreichen können.

Die minimale Beispielanwendung

Wir haben ein Java-Backend mit Spring-Boot, das über einen Controller einen Benutzer zurückgeben kann. Der folgende Code zeigt das *User*-Modell im Backend, Listing 1 den Controller im Backend:

```
public record User(String id, String userName) {
}
```

Das Frontend besteht aus einer Angular-Anwendung, die diesen Benutzer abruft und darstellt. Der folgende Code zeigt das *User*-Modell im Frontend, Listing 2 die Komponente im Frontend.

```
export type User = {
  id: string
  userName: string
}
```

Kein gemeinsames Projekt-Set-up

Aufgrund der unterschiedlichen Programmiersprachen zeigt sich hier bereits die erste kleine Hürde: Das Backend wird mit Gradle gebaut, das Frontend mit npm. Je nach Projekt müssen also unterschiedliche Befehle zum lokalen Starten (*./gradlew bootRun vs npm run start*) oder zum Ausführen von Tests (*./gradlew test vs npm run test*) verwendet werden. Hätten wir noch weitere Java- oder TypeScript-Projekte, müssten wir unsere IDE und Linter für jedes dieser Projekte separat konfigurieren und bei Änderungen darauf achten, immer alles synchron zu halten. Gleiches gilt für die Versionen der verwendeten Bibliotheken oder gar der Programmiersprache. Verwenden wir hingegen überall die gleiche Version, erhöht das die Kompatibilität, reduziert möglicherweise die Größe des Bundles und vereinfacht Dinge, wie z. B. Open-Source-Software-Clearing-Prozesse.

Das Problem mit der Konsistenz

Wenn wir nun in unserer Anwendung eine Änderung vornehmen, die sowohl das Backend als auch das Frontend betrifft, treten Probleme auf, die wohl jeder Entwickler kennt. Angenommen, wir wollen unser Datenmodell anpassen und das Attribut *userName* in *name* umbenennen. Damit Backend und Frontend weiterhin miteinander kommunizieren können, muss diese Umbenennung in beiden Projekten gleichzeitig vonstattengehen. Ansonsten versucht das Frontend, in der Anzeige auf ein Attribut zuzugreifen, das vom Backend unter einem anderen Namen gesendet wird.

„Gleichzeitig“ ist in zwei verschiedenen Repositories aber nicht möglich. Und da macht es auch keinen Unterschied, ob man als Fullstack-Entwickler beide Änderungen sofort selbst durchführen kann. Sie müssen notgedrungen in zwei einzelne Commits aufgeteilt werden – einen im Backend und einen im Frontend. Und plötzlich entstehen Probleme: Bei einem Kollegen, der nur eines der Projekte lokal aktualisiert hat, funktioniert plötzlich etwas nicht mehr. Die End-to-End-Tests in der Pipeline schlagen unweigerlich beim Push der ersten Änderung fehl. Beim Debuggen möchte man noch einmal eine ältere Version der Projekte auschecken, um zu sehen, ob der Bug dort auch schon vorhanden war – hat aber Schwierigkeiten, miteinander kompatible Versionen zu finden. Das alles ist nicht das Ende der Welt und für einen erfahrenen Entwickler lediglich lästig. Aber es kostet Zeit und Aufmerksamkeit, die man besser in die Weiterentwicklung der Anwendung gesteckt hätte.

Aber nicht nur das lokale Set-up ist davon betroffen: Auch beim Deployment muss immer darauf geachtet werden, kompatible Backend- und Frontend-Stände zu deployen. Da diese in unterschiedlichen Repositories liegen und daher jeweils eigene Pipelines haben, muss für das Deployment eine dritte Pipeline eingerichtet werden, die die Artefakte der beiden anderen verwendet und gleichzeitig deployt. Das erschwert zwangsläufig die Nachvollziehbarkeit des Build-Prozesses.

Geteilter Code? Einfach nur umständlich!

Zugegeben, in diesem Beispiel ist es eher abwegig, Code zwischen dem Backend in Java und dem Frontend in TypeScript zu teilen. Hätten wir stattdessen ein NestJS-Backend, sähe die Sache ganz anders aus, genauso bei Microservices und Microfrontends. Der übliche Weg, Code in einem solchen Fall zu teilen, besteht darin, eine eigene Bibliothek zu erstellen, den geteilten Code darin abzulegen und die Bibliothek in eine Artifactory zu laden. Die eigentlichen Projekte verwenden die Bibliothek dann als Abhängigkeit. Änderungen am gemeinsam genutzten Code ziehen jedoch neue Versionen der Bibliothek, Aktualisierungen der Abhängigkeiten und damit viel zusätzlichen Aufwand nach sich. Die Alternative,

Listing 1: Controller im Backend

```
@RestController()
@RequestMapping("/api/user")
public class UserController {
  @GetMapping(value = "/current", produces = "application/json")
  public User getCurrentUser() {
    return new User("01", "admin");
  }
}
```

Listing 2: Komponente im Frontend

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [AsyncPipe],
  template: `
    @if (user$ | async; as user) {
      <h1>Welcome, {{ user.userName }}</h1>
    }
  `,
})
export class AppComponent {
  private httpClient = inject(HttpClient)

  user$ = this.httpClient.get<User>('${environment.baseUrl}/api/user/
                                                          current').pipe(
    shareReplay({bufferSize: 1, refCount: true})
  )
}
```

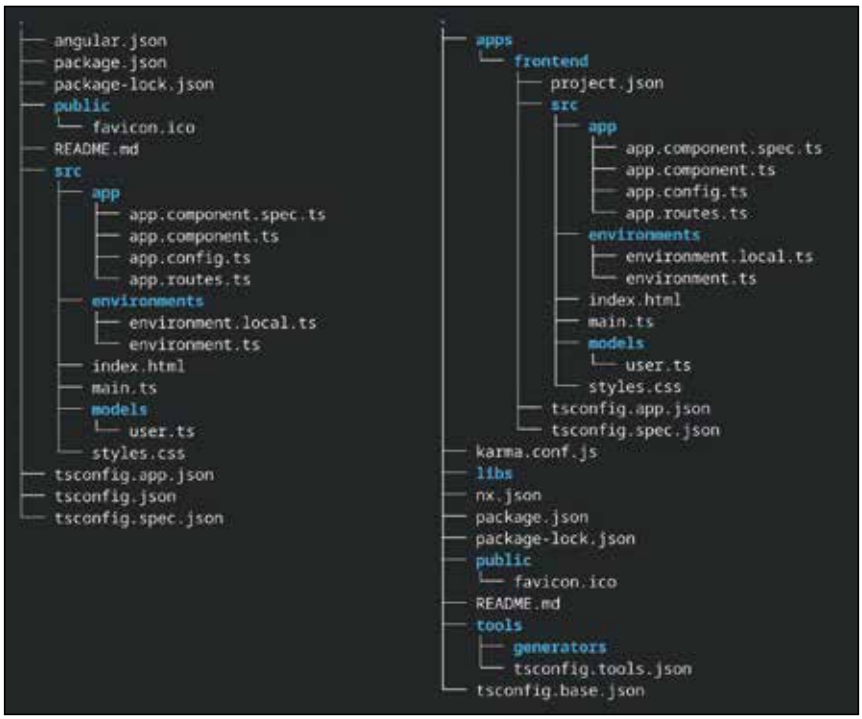

Abb. 1: Die Projektstruktur unseres Frontend-Projekts vor (links) und nach (rechts) der Migration nach Nx

den geteilten Code als Git-Submodule einzubinden, ist leider ähnlich umständlich.

Aber auch zwischen unserem Backend und dem Frontend gibt es Dinge, die wir gern teilen würden. So müssen wir derzeit das *User*-Modell in beiden Projekten definieren und darauf achten, dass die Definitionen konsistent sind. Einfacher und fehlerresistenter wäre es, wenn wir aus unserem Backend eine OpenAPI-Spezifikation generieren könnten, die wir dann zur automatischen Codegenerierung im Frontend verwenden. Das ist aber in der aktuellen Konfiguration mit zwei Repositories nicht ohne Umwege möglich.

Erster Lösungsansatz: ein Monorepo

Als ersten, einfachen Schritt können wir Backend und Frontend einfach in ein gemeinsames Repository verschieben. Damit haben wir bereits die meisten Probleme gelöst: Backend und Frontend sind immer synchron und auch das Teilen von Code oder Ressourcen ist nun einfacher, da wir sie einfach in einen Ordner verschieben können, auf den alle Projekte Zugriff haben. Auch das Deployment verläuft nun automatisch synchron, da sich beide Projekte eine Pipeline teilen. Allerdings haben wir uns damit ein neues Problem ins Haus geholt: Durch die gemeinsame Pipeline bauen wir dort nun immer Backend und Frontend – selbst wenn sich in einem der Projekte gar nichts geändert hat. Auch das Tooling der einzelnen Teilanwendungen ist nach wie vor unterschiedlich und projektübergreifende, voneinander abhängige Aufgaben lassen sich nicht so einfach umsetzen.

Auftritt Nx

Hier kommt Nx (Kasten: „Was ist Nx?“) ins Spiel und bietet uns die Möglichkeit, unser Set-up weiter zu optimieren. Im weiteren Verlauf des Artikels werden wir die beiden Projekte in ein mit Nx verwaltetes Monorepo umwandeln, dieses konfigurieren und nach und nach die Vorteile dieses Set-ups kennenlernen.

Wir könnten nun einen komplett neuen Nx Workspace anlegen und unsere beiden Projekte nacheinander in diesen integrieren. Nx unterstützt auch nativ die Migration eines Angular-Projekts in ein integriertes Monorepo, sodass wir hier diesen Schritt wählen und uns etwas Arbeit sparen können.

Dazu führen wir im Frontend-Repository einfach *npx nx@latest init --integrated* aus und beantworten die Frage nach dem Remotecache mit *skip* – diese Option werden wir uns später genauer ansehen. Nx installiert zusätzliche Abhängigkeiten und passt anschließend die Projektstruktur etwas an. So wurde unser Frontend in einen Nx Workspace umgewandelt, der bereits eine Angular-App enthält. In **Abbildung 1** sehen wir, wie sich die Projektstruktur dadurch geändert hat: Der Anwendungscode wurde in den Unterordner *apps/frontend* verschoben, *angular.json* wurde zu *project.json* und *tsconfig.json* wurde in einen projektspezifischen Teil (*tsconfig.app.json* und *tsconfig.spec.json*) sowie einen übergreifenden Teil (*tsconfig.base.json*) aufgeteilt. Hinweis: Es gibt weiterhin nur eine *package.json* auf Root-Ebene, Abhängigkeiten werden also global verwaltet.

Nach der Migration starten wir das Frontend nicht mehr mit *npm run start*, sondern mit *npx nx serve frontend*, Tests werden mit *npx nx test frontend* ausgeführt, gebaut wird mit *npx nx build frontend*. Um uns das *npx* in den Befehlen zu sparen, installieren wir Nx noch global: *npm i -g nx@latest*. Die Struktur der Nx-Befehle legt nahe, dass sie später ebenso für das Backend (*nx test backend*) aufgerufen werden können. Bevor wir uns also die Vorteile ansehen, die wir allein im Frontend durch

Was ist Nx?

Nx [1] ist ein Open-Source-Build-System, das Werkzeuge und Techniken für die effiziente Verwaltung von Multi-Projekt-Repositorys bereitstellt. Ursprünglich für Angular-Anwendungen konzipiert, bietet es mittlerweile über entsprechende Plug-ins auch Unterstützung für andere Frontend-Frameworks wie React sowie für gänzlich andere Programmiersprachen wie Java oder C#.

das Hinzufügen von Nx erhalten haben, fügen wir das Backend hinzu und schauen uns das Ergebnis an.

Für Java gibt es leider noch keinen automatischen Konverter, daher fügen wir stattdessen eine neue Gradle-Applikation zu Nx hinzu und kopieren dann den Quellcode des alten Backends in die neue Applikation. Zuerst müssen wir das Nx-Plug-in hinzufügen, das die Java-Unterstützung mitbringt (hier sogar konkrete Spring-Boot-Unterstützung). Dazu verwenden wir hier im Root-Verzeichnis des Nx Workspace *npm install @nxrocks/nx-spring-boot --save-dev*. Mit *nx g @nxrocks/nx-spring-boot:project backend* erzeugen wir eine neue Spring-Boot-Applikation in unserem Workspace und beantworten anschließend die Fragen des Wizard nach unseren Anforderungen (in unserem Beispiel Gradle und Java). Schauen wir uns die Projektstruktur an, so finden wir nun unter *apps* ein Java-Gradle-Projekt. Hier können wir nun den Code unseres alten Backend-Projekts einfügen und schon haben wir beide Projekte in einem gemanagten Monorepo.

Das Backend kann nun ebenfalls mit *nx build backend* gebaut oder mit *nx serve backend* gestartet werden. Mit *nx run-many --target=test* oder *nx run-many --target=serve* können wir auch Backend und Frontend gleichzeitig testen oder starten. Nx führt diese Befehle dann für alle Projekte parallel aus. Als Entwickler hat man es also ab jetzt nicht mehr mit verschiedenen Tools und Syntaxen zu tun, sondern kann alle Projekte einheitlich testen, bauen und starten.

Mit dem jetzigen Set-up sind wir auf dem Weg zum mühelosen Entwickeln schon ein gutes Stück vorangekommen. Was jetzt noch fehlt, ist geteilter Code bzw. geteilte Dateien und eine effiziente CI Pipeline.

Eine Single Source of Truth für unser API

Sehen wir uns zunächst an, wie wir die beiden Projekte effektiv miteinander verbinden können. Die Kommunikation läuft über den Spring Controller im Backend, der vom Frontend aufgerufen wird. Damit wir immer sicher sein können, dass diese Schnittstelle einheitlich implementiert ist, wollen wir versuchen, im Backend automatisch eine OpenAPI-Spezifikation aus dem Controller zu generieren und diese im Frontend zu verwenden, um den aufrufenden Code aktuell und kompatibel zu halten.

Dazu fügen wir im Backend das Plug-in *id("org.springdoc.openapi-gradle-plugin") version "1.9.0"* und die Abhängigkeit *implementation("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.6.0")* zu *build.gradle.kts* hinzu. Jetzt können wir im Backend *./gradlew generateOpenApiDocs* aufrufen, um unsere API-Definition als JSON im *build*-Verzeichnis zu erhalten. Aber wollten wir nicht eigentlich auf den manuellen Aufruf von Gradle verzichten? Kein Problem, dazu fügen wir in *apps/backend/project.json* ein neues Nx Target hinzu (Listing 3).

Und schon können wir die Spezifikation als JSON auch über *nx generateApiSpec backend* erhalten.

Im nächsten Schritt wollen wir aus dieser Datei einen Angular Service generieren. Dazu installieren wir zunächst (im Root-Verzeichnis) die Abhängigkeit *npm install @openapitools/openapi-generator-cli --save-dev*. Danach können wir im Root-Verzeichnis mit *npx openapi-generator-cli generate -i apps/backend/build/openapi.json -g typescript-angular -o apps/frontend/src/app/generated/openapi --additional-properties fileNaming=kebab-case,withInterfaces=true --generate-alias-as-model* den Code generieren, den wir anschließend unter *apps/frontend/src/app/generated/openapi* vorfinden.

Das erzeugt unter anderem den *User*-Typ und einen *UserControllerService*, der unser Backend aufruft. Passen wir also unseren Frontend-Code so an, dass er diesen neuen Service verwendet, und löschen wir unser manuell erstelltes Modell:

```
user$ = this.userService.getCurrentUser().pipe(
  shareReplay({bufferSize: 1, refCount: true})
);
```

Zusätzlich müssen wir den URL des Backend explizit zur Verfügung stellen. Dazu fügen wir den Provider *{provide: BASE_PATH, useValue: environment.baseUrl}* in unsere *app.config.ts* ein und sind fertig. Wenn wir die gesamte Anwendung starten (*nx run-many --target=serve*), sehen wir, dass alles weiterhin funktio-

Listing 3: Nx Target zum Generieren der OpenAPI-Spezifikation im Backend

```
"targets": {
  "generateApiSpec": {
    "executor": "nx:run-commands",
    "options": {
      "cwd": "apps/backend",
      "command": "./gradlew generateOpenApiDocs"
    }
  }
}
```

Listing 4: Nx Target zum Generieren der API Services im Frontend

```
"targets": {
  ...
  "generateApiServices": {
    "executor": "nx:run-commands",
    "options": {
      "command": "npx openapi-generator-cli generate -i apps/backend/
          build/openapi.json -g typescript-angular -o apps/frontend/src/
        app/generated/openapi --additional-properties fileNaming=kebab-
                    case,withInterfaces=true --generate-alias-as-model"
    }
  }
}
```

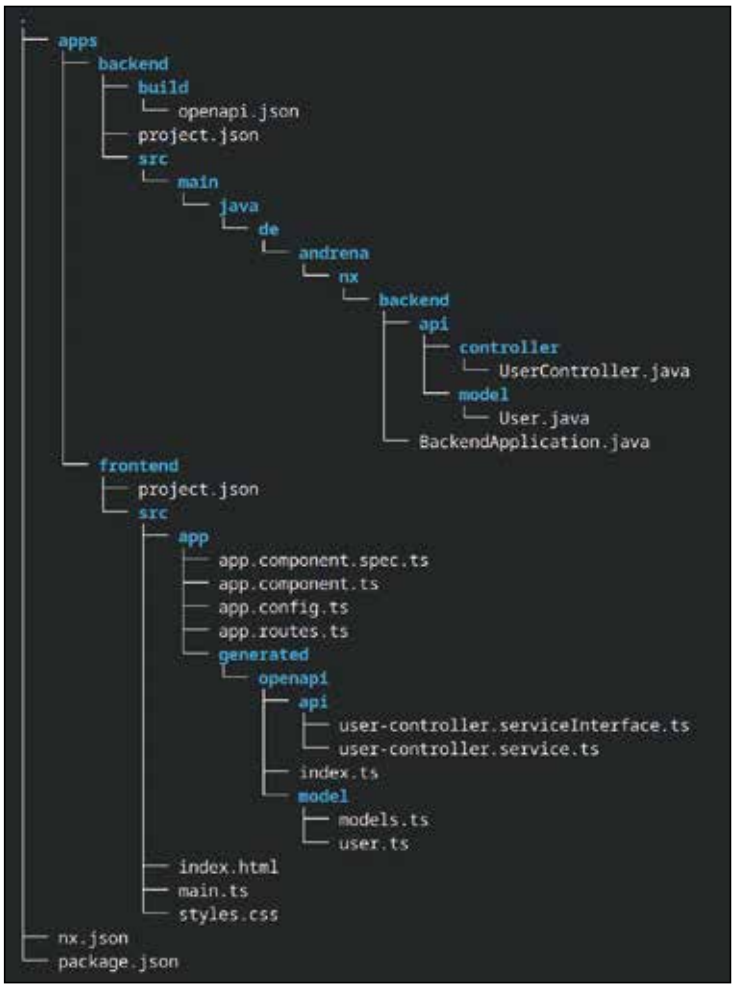

Abb. 2: Die Projektstruktur unseres Monorepos, nachdem beide Generator-Targets ausgeführt wurden (gekürzt)

niert. Fügen wir also auch hierfür ein Nx Target hinzu, um den Befehl über *nx generateApi frontend* aufrufen zu können. Das geschieht in der *project.json* des Frontends (Listing 4).

Vorsicht bei targetDefaults

Nx erlaubt es, in der *nx.json* sogenannte *targetDefaults* zu definieren. Hier können allgemeine Einstellungen für Targets vorgenommen werden, um diese nicht in jedem Projekt einzeln definieren zu müssen. Für das Target *build* wurden in unserem Projekt z. B. bereits Werte für *dependsOn*, *inputs* und *cache* definiert. Gibt es nun in einem Projekt ein Target namens *build*, so gelten hier automatisch die unter *targetDefaults* definierten Werte. Wenn wir nun aber einzelne davon überschreiben, weil wir z. B. ein weiteres *dependsOn* hinzufügen wollen, müssen wir darauf achten, dass wir auch hier die Werte aus *v* übernehmen. In unserem konkreten Fall müssen wir also für das Target *build* im Frontend-Projekt *"dependsOn": ["frontend:generateApi", "^build"]* setzen. Das wird auch später im Abschnitt über Caching relevant und sollte daher beachtet werden, auch wenn wir es im Folgenden nicht mehr explizit erwähnen.

Jetzt haben wir nur noch ein Problem: Sobald sich im Backend etwas ändert, z. B. die Eigenschaft *userName* des *User*-Typs auf *name* umbenannt wird, müssen wir selbst daran denken, beide Generatoren aufzurufen. Aber auch hier lässt sich leicht Abhilfe schaffen, indem wir die einzelnen Tasks als Abhängigkeiten verknüpfen. Zum einen fügen wir zum *generateApiServices* Task im Frontend *"dependsOn": ["backend:generateApiSpec"]* hinzu, zum anderen soll dieser Task auch vor jedem Bauen, Ausführen und Testen des Frontends ausgeführt werden, sodass wir zu den Tasks *build*, *serve* und *test* in *apps/frontend/project.json* noch *"dependsOn": ["frontend:generateApiServices"]* hinzufügen. (Kasten: „Vorsicht bei *targetDefaults*“) Wenn wir nun *nx serve frontend* ausführen, werden zuerst die OpenAPI-Spezifikation im Backend und dann der Frontend-Service generiert und erst danach wird das Frontend gestartet. So können wir immer sicher sein, dass Frontend und Backend zusammenpassen und mögliche Inkompatibilitäten schon beim Bauen auffallen.

Wir brauchen Caching

Schnell zeigt sich allerdings, dass wir dadurch oft unnötig Zeit verlieren. Unabhängig davon, ob sich am API etwas geändert hat oder nicht, werden bei jedem *serve* immer erst beide Generierungs-Targets ausgeführt und wir müssen warten. Um dieses Problem zu lösen, aktivieren wir das Caching für diese Targets. Dazu fügen wir in den Targets *generateApiSpec* und *generateApiServices* in der jeweiligen *project.json* einfach die Eigenschaft *"cache": true* hinzu. Nx prüft nun bei jedem Aufruf, ob sich die Dateien geändert haben, und führt das Target nur in diesem Fall tatsächlich aus. Andernfalls spielt es die Logs und Ergebnisse der gecachten Ausführung aus.

Aber was sind „die Dateien“, die sich ändern müssen? Und was sind „die Ergebnisse“? In einigen Fällen, insbesondere bei von Plug-ins bereitgestellten Targets wie *serve* für Angular-Projekte, ist das bereits definiert. Ansonsten werden standardmäßig alle Dateien im entsprechenden Projektverzeichnis für die Berechnung des Cache-Keys verwendet. In unserem Fall müssen wir den Cache also noch etwas genauer spezifizieren, um sicherzustellen, dass die Targets wirklich nur dann ausgeführt werden, wenn es nötig ist – dann aber auf jeden Fall.

Nx bietet die Möglichkeit, Inputs und Outputs für Targets zu definieren. Inputs sind die Dateien und Ordner, die für die Ausführung relevant sind und die in die Berechnung des Cache-Keys einfließen. Outputs sind die Dateien und Ordner, die durch das Target erzeugt wurden, die gecacht und im Falle eines Cache-Hits wieder ausgeliefert werden sollen.

Betrachten wir dazu noch einmal unsere Projektstruktur, nachdem alle Dateien generiert wurden (**Abb. 2**): Die für die Generierung der OpenAPI-Spec notwendigen Dateien befinden sich alle in einem gemeinsamen Package im Backend. Die generierte Spezifikation wird in *build/openapi.json* abgelegt. Diese wird dann zur Generierung des Frontend-Codes verwendet, der in *apps/frontend/src/app/generated/openapi* landet.

Für unser *generateApiSpec*-Target können wir sehr einfach Input und Output angeben: Wir wollen das Target immer dann neu ausführen, wenn sich eine Datei im Package *de.andrena.nx.backend.api* geändert hat. Das Ergebnis dieses Tasks ist dann die Datei *apps/backend/build/openapi.json*. Indem wir den Input genauer spezifizieren, werden alle anderen Dateien, die nicht in dieses Muster passen, ignoriert. Insgesamt sieht unsere Target-Definition also so aus wie in Listing 5.

Für das Target *generateApiServices* gestaltet sich der Input etwas komplizierter. Wir wollen dieses Target immer dann neu ausführen, wenn sich *openapi.json* ändert – was der Output des unter *dependsOn* angegebenen Targets ist. Die fertige Konfiguration sehen wir in Listing 6.

Zwei Besonderheiten fallen sofort auf: Ein Input ist als *dependentTasksOutputFiles* definiert. Das bedeutet, dass die Outputs der *dependsOn* Targets auf dieses Muster überprüft und alle Dateien, die übereinstimmen, als Input für die Cache-Key-Berechnung verwendet werden. Diese Methode erlaubt es uns, auch Dateien zu überprüfen, die sich in einem Verzeichnis befinden, das durch *.gitignore* ignoriert wird. Solche Dateien werden normalerweise in den Inputs ignoriert, selbst wenn sie explizit angegeben werden. Wenn sie jedoch über *dependentTasksOutputFiles* referenziert werden, können sie trotzdem verwendet werden.

Zusätzlich gibt es den Eintrag *"!{projectRoot}/**/*"*, der explizit alle Dateien im Projektverzeichnis ausschließt. Wie schon erwähnt, werden standardmäßig alle Dateien aus dem Projektverzeichnis für den Cache-Input verwendet. Die Angabe eines *dependentTasksOutputFiles* überschreibt dieses Standardverhalten nicht, sondern erweitert es nur, sodass wir es explizit ausschalten müssen. Als Output wurde das Verzeichnis angegeben, in das die fertigen TypeScript-Dateien generiert werden.

Zuletzt müssen wir noch die Inputs der *build*- und *test*-Targets aktualisieren und hier ebenfalls *dependentTasksOutputFiles* hinzufügen, damit auch neu generierte API Services zu einem Cache-Hit führen. Dazu fügen wir bei beiden Targets *"dependentTasksOutputFiles": "**/*"* zu den Inputs hinzu. Bei *serve* ist der Cache deaktiviert, sodass wir es hier nicht benötigen.

Was haben wir nun erreicht? Wir können sicherstellen, dass die Schnittstelle zwischen Backend und Frontend immer konsistent ist. Sobald sich im Backend etwas ändert, werden diese Änderungen automatisch im Frontend gespiegelt und führen bei Inkonsistenzen zu Kompilierfehlern. Durch effizientes Caching geht das nicht zulasten unserer Entwicklungszeit.

Unser Set-up in CI/CD

Um dem Set-up wirklich seine Tauglichkeit für Projekte bescheinigen zu können, müssen wir uns abschließend noch anschauen, wie es sich in einer Build Pipeline verhält. An dieser Stelle sei gleich gesagt, dass wir nur eine von vielen möglichen und sinnvollen Build Pipelines betrachten. Um einen Eindruck von den Möglichkeiten von Nx zu bekommen, reicht das aber aus; weitere Optimierungen oder ein ganz anderer Aufbau der Pipelines sind natürlich auch möglich. Beginnen wir also mit einem ersten Entwurf einer Pipeline (hier für GitLab) und optimieren sie anschließend (Listing 7).

Zum Ausführen der Pipeline benötigen wir ein Image, das sowohl Node als auch Java installiert hat. Man könnte speziellere, kleinere Images für die verschiedenen Jobs verwenden, aber hier haben wir uns dafür entschieden, keine separaten Test- und Build-Schritte für Backend und Frontend zu erstellen, sondern einen gemeinsamen Job für beide zu definieren. Das hat den Vorteil, dass das Image und der Cache, den wir später noch hinzufügen, nur einmal geladen werden müssen.

Listing 5: Target generateApiSpec in der project.json im Backend

```
"generateApiSpec": {
  "inputs": ["{projectRoot}/src/main/java/de/andrena/nx/backend/
                                                        api/**/*"],
  "outputs": ["{projectRoot}/build/openapi.json"],
  "cache": true,
  "executor": "nx:run-commands",
  "options": {
    "cwd": "apps/backend",
    "command": "./gradlew generateOpenApiDocs"
  }
}
```

Listing 6: Target generateApiServices in der project.json im Frontend

```
"generateApiServices": {
  "cache": true,
  "inputs": [
    { "dependentTasksOutputFiles": "**/openapi.json" },
    "!{projectRoot}/**/*"
  ],
  "outputs": ["{projectRoot}/src/app/generated/openapi"],
  "executor": "nx:run-commands",
  "options": {
    "command": "[...]"
  },
  "dependsOn": [
    "backend:generateApiSpec"
  ]
}
```

Über *run-many* führt Nx dann den Bau und Test der beiden Projekte parallel aus.

Das Deployment am Ende haben wir in zwei parallele Jobs aufgeteilt, da es von Projekt zu Projekt sehr unterschiedlich sein kann und oft komplexere Logik enthält. In unserem Beispiel geben wir als Platzhalter lediglich die Größe des jeweils erstellten Artefakts auf der Konsole aus.

Im Prinzip können wir unsere Anwendung bereits mit dieser Pipeline bauen, aber es gibt noch viel Raum für Optimierungen. Als Erstes können wir das Caching für unsere Pipeline aktivieren, um nicht in jedem Schritt die Node- und Gradle-Abhängigkeiten herunterladen zu müssen. Dazu fügen wir zunächst in unserem Backend ein Nx Target zum Download der Abhängigkeiten (und implizit auch der richtigen Gradle-Version über den Wrapper) hinzu (Listing 8).

Listing 7: Erster Entwurf der Build Pipeline

```
image: einjavaundnodeimage

stages:
  - test
  - build
  - deploy

before_script:
  - npm ci

test:
  stage: test
  script:
    - npx nx run-many --target=test

build:
  stage: build
  script:
    - npx nx run-many --target=build
  artifacts:
    paths:
      - apps/backend/build/libs
      - dist/apps/frontend/browser

deploy_backend:
  stage: deploy
  dependencies:
    - build
  script:
    - echo "Deploying BE - $(du -sh apps/backend/build/libs | cut -f1)"

deploy_frontend:
  stage: deploy
  dependencies:
    - build
  script:
    - echo "Deploying FE - $(du -sh dist/apps/frontend/browser | cut -f1)"
```

Für unser Frontend benötigen wir das nicht, hier kommen die Abhängigkeiten automatisch durch *npm ci* des Root-Projekts mit. Durch das Muster mit dem separaten Target für das Backend können wir aber in Zukunft mit *nx run-many --target=installDependencies* alle Abhängigkeiten in allen Unterprojekten installieren, was insbesondere dann relevant wird, wenn wir noch weitere hinzufügen.

Danach können wir einen *install_dependencies*-Schritt in unsere Pipeline einfügen und die Verzeich-

Listing 8: Das installDependencies Target in project.json im Backend

```
"installDependencies": {
  "executor": "nx:run-commands",
  "options": {
    "cwd": "apps/backend",
    "command": "./gradlew buildEnvironment"
  }
}
```

Listing 9: GitLab-Cache

```
variables:
  GRADLE_USER_HOME: "$CI_PROJECT_DIR/.gradle"
  FF_USE_FASTZIP: "true"
  CACHE_COMPRESSION_LEVEL: "fastest"

default:
  cache: &global_cache
    key: $CI_COMMIT_REF_SLUG
    paths:
      - node_modules/
      - .gradle/wrapper/
      - .gradle/caches/

install_dependencies:
  stage: prepare
  script:
    - npm ci
    - npx nx run-many --target=installDependencies
  cache:
    <<: *global_cache
    policy: pull-push
[...]
test:
  [...]
  cache:
    <<: *global_cache
    policy: pull
[...]
build:
  [...]
  cache:
    <<: *global_cache
    policy: pull
```

nisse mit den installierten Abhängigkeiten cachen. In den nachfolgenden Jobs (*test*, *build*) können wir diesen Cache dann mit einer Pull Policy verwenden. Zusätzlich optimieren wir auch die Cacheeinstellungen noch etwas auf Geschwindigkeit (Listing 9).

Diese Änderung hat die Laufzeit unserer Pipeline bereits von ca. sechseinhalb auf ca. dreieinhalb Minuten reduziert. Zugegeben, bei einem größeren Projekt wird der Anteil der Laufzeit, der für das Installieren von Abhängigkeiten benötigt wird, kleiner werden, aber verschwinden wird er nicht.

Caching, die Zweite: verteilter Cache

Bisher war das nur eine allgemeine Optimierung und hatte noch nichts mit Nx zu tun. Wir haben immer noch das Problem, dass immer beide Teile der Anwendung gebaut und getestet werden, auch wenn sich in diesem Commit nur in einem Teil etwas geändert hat. Lokal haben wir dieses Problem nicht mehr, da wir den Cache bereits in Nx konfiguriert haben. Schauen wir uns also an, wie wir ihn in die Pipeline bringen können und wie einfach es ist, einen verteilten Cache in Nx einzurichten.

Nx bringt von Haus aus einen Distributed Cache mit, der die Nx-Cloud verwendet und dort einen Account und ab einer gewissen Projektgröße ein Abonnement erfordert. Es ist aber auch problemlos möglich, den verteilten Cache an anderen Orten zu hosten. In diesem Beispiel konzentrieren wir uns auf die Lösung in AWS S3.

In AWS erstellen wir einen S3 Bucket und einen IAM-Benutzer mit Schreib- und Leserechten auf diesen Bucket. Von diesem Benutzer benötigen wir dann Access Key und Secret Access Key, die wir als Umgebungsvariablen *AWS_ACCESS_KEY_ID* und *AWS_SECRET_ACCESS_KEY* in unserem GitLab Runner zur Verfügung stellen.

Um diesen Bucket nun als Cache für unser Nx-Projekt zu verwenden, installieren wir dort das notwendige Plug-in (*npm install --save-dev @pellegrims/nx-remotecache-s3*) und fügen anschließend die Konfiguration zu *nx.json* hinzu (Listing 10).

Und das war's! Nx durchsucht nun immer zuerst den lokalen Cache nach einem passenden Treffer, und wenn das nicht gelingt, schaut es im S3 Bucket nach. Gibt es keinen Treffer, wird am Ende ein neuer Cacheeintrag für den aktuellen Durchlauf erzeugt und auch wieder auf S3 hochgeladen.

Um das Ganze etwas kostengünstiger (Kasten: Remote-Caching verursacht Kosten“) zu gestalten, können wir mit der Option *"readOnly": "true"* in den *options* des Runners in *nx.json* das Pushen in den Cache deaktivieren, sodass nicht ständig von den Entwicklungsmaschinen in den verteilten Cache geschrieben wird. Damit die CI das aber dennoch tun kann, fügen wir in unserem Runner die Umgebungsvariable *NX-CACHE_S3_READ_ONLY: false* hinzu.

Wenn es nun nur eine Änderung im Frontend gibt, baut Nx dieses in der Pipeline neu, verwendet aber den Cache für Test und Build des Backends. In diesem Fall

Listing 10: Konfiguration unserer verteilten S3-Caches in nx.json

```
"tasksRunnerOptions": {
  "default": {
    "runner": "@pellegrims/nx-remotecache-s3",
    "options": {
      "cacheableOperations": ["build", "test", "generateApiServices",
                                                    "generateApiSpec"],
      "bucket": "UNSER_BUCKET_NAME",
      "prefix": "javamagdemo/",
      "region": "eu-central-1"
    }
  }
}
```

Remote-Caching verursacht Kosten

Bei der hier vorgestellten Variante des Remote-Caching in S3 sollte man immer auch die Kostenseite im Auge behalten. Remote-Caching spart Entwicklungs- und Pipelinezeit und damit indirekt auch Kosten, verursacht aber an anderer Stelle wiederum Kosten für den Speicherdienst. Hier sollte also eine sinnvolle Balance gefunden werden. Durch geeignete Einstellungen (z. B. automatisches Löschen älterer Cachedateien) können die Kosten gesenkt werden. Neben der vorgestellten Lösung mit S3 gibt es weitere Plug-ins, um den Remote-Cache mit anderen Speicherdiensten zu realisieren. Ebenso ist es möglich, eine komplett eigene Remote-Caching-Lösung für Nx zu implementieren [2].

nx affected

Anstatt unveränderte Teilprojekte aus dem Cache zu laden und damit die Testausführung und Builds zu beschleunigen, gibt es auch die Möglichkeit, Targets nur für Teilprojekte auszuführen, die in diesem Commit Codeänderungen erfahren haben. GitLab bietet diese Möglichkeit beispielsweise mit *rules:changes*. Aber auch Nx selbst bietet mit *nx affected* die Möglichkeit, Targets nur für Teilprojekte mit Änderungen auszuführen (*nx affected --target build* statt *nx run-many --target=build*). Dieser Ansatz hat jedoch zwei Probleme: In unserem Fall würde eine Änderung der *User*-Klasse im Backend keine direkte Dateiänderung im Frontend anzeigen. Sowohl *rules:changes* als auch *nx affected* würden also nur das Backend bauen und somit die Änderung des API-Modells nicht ins Frontend übertragen, wodurch dort durch den Build-Prozess unbemerkt mögliche Inkonsistenzen entstehen könnten. Außerdem benötigen wir für den Deployment-Schritt die Artefakte aller Teilprojekte, sodass wir spätestens dann auf die gecachten Ergebnisse zugreifen müssten. Nichtsdestotrotz gibt es Fälle, in denen diese Vorgehensweise die effizienteste und kostengünstigste ist, sie sollte daher nicht unerwähnt bleiben.

verkürzt sich die Laufzeit der Build Pipeline unseres kleinen Projekts von dreieinhalb auf zwei Minuten.

Ein kleiner Nachteil bleibt: Auch wenn Build und Test aus dem Cache kommen, so wird in unserer Pipeline am Ende ein unverändertes Teilprojekt komplett neu deployt (Kasten: „nx affected“). Theoretisch gibt es Möglichkeiten, diesen Fall zu behandeln. Es ist jedoch nicht ratsam, diese Entscheidung allein von Änderungen im Quellcode in diesem Commit abhängig zu machen. Soll z. B. später wieder eine ältere Version deployt werden, sollten unbedingt Backend und Frontend deployt werden – unabhängig davon, ob es zu diesem Zeitpunkt in der ausführenden Pipeline keine Änderung im Backend gab. Um auch diesen Schritt zu optimieren, sollte man also eher herausfinden, welche Version gerade deployt ist, und diese mit der Version aus der Pipeline vergleichen. Da das aber wiederum keine Besonderheit von Nx ist, lassen wir diese Betrachtung in unserem Beispiel weg.

Was haben wir erreicht?

Sehen wir uns an, was wir erreicht haben, indem wir unsere zwei Repositorys in ein einziges, von Nx verwaltetes Repository migriert haben:

- Zusammenhängende Änderungen im Backend und Frontend können in einem gemeinsamen Commit durchgeführt werden.
- Pro Feature gibt es immer genau einen Branch, sodass beim Mergen in *main* keine Inkonsistenzen entstehen können.
- Wir können unser Backend und unser Frontend mit den gleichen Befehlen bauen und testen (und können dies um weitere Befehle erweitern, z. B. für Linting).
- Unser API ist zwischen Backend und Frontend immer konsistent. Bei Änderungen im Backend wird der entsprechende Code im Frontend automatisch aktualisiert, sodass Probleme sofort bemerkt werden.
- Durch lokales Caching wird auf den Entwicklungsmaschinen immer nur das Nötigste gebaut und getestet.
- Durch Remote-Caching ist unsere Pipeline schnell und effizient.
- Weitere Teilprojekte können schnell und einfach zum Monorepo und zur Pipeline hinzugefügt werden.
- Durch eine globale *package.json* verwenden alle Node-Projekte die gleichen Bibliotheks- und Frameworkversionen. Für Java ist das leider nur mit zusätzlichem Aufwand möglich.

Das haben wir uns durch einige wenige Nachteile erkauft:

- Das initiale Einrichten des Projekts und des Cachings ist etwas aufwendiger als bei einem unmanaged Repo.
- Mit Nx haben wir eine weitere Abstraktionsebene hinzugefügt, die das Debugging erschweren kann. Außerdem kann man gerade am Anfang leichter auf Probleme stoßen, bis man die Eigenheiten von Nx besser kennt (z. B., dass durch *.gitignore* ignorierte Dateien auch für den Cache-Input ignoriert werden).

Fazit und Ausblick

Natürlich haben wir hier mit einem Minimalprojekt gearbeitet. Es gibt kaum Logik in Frontend und Backend und auch Drittdienste wie Datenbanken sind nicht eingebunden. Aber all das wird durch Nx nicht erschwert. Im schlimmsten Fall bringt Nx einfach keinen Mehrwert, im besten Fall können wir die Möglichkeiten von Nx auch dort effektiv nutzen.

Wenn wir weitere Apps hinzufügen, die teilweise voneinander abhängen, bietet uns Nx sogar die Möglichkeit, durch einfache Linter-Regeln Abhängigkeitszyklen zwischen Teilprojekten sofort zu erkennen und zu verbieten [3]. Ebenso kann ein Projektgraph erstellt werden, der die verschiedenen Abhängigkeiten der Targets grafisch darstellt [4]. Aber auch, wenn man diese erweiterten Features nicht nutzt, hat Nx seinen Mehrwert bereits bewiesen – oder hoffentlich zumindest die Lust geweckt, es einmal selbst auszuprobieren. Das Minimalprojekt, das wir hier erstellt haben, findet sich auch auf GitHub, um es sich in Ruhe und im Detail anschauen zu können [5].

Marco Sieben arbeitet seit acht Jahren als Full-Stack-Entwickler bei andrena objects ag. Seine Liebe zu sauberem Code und sauberer Infrastruktur treibt ihn an, bestehende Praktiken immer wieder infrage zu stellen und einfachere, elegantere Lösungen auszuprobieren.

Links & Literatur

[1] https://nx.dev

[2] https://github.com/NiklasPor/nx-remotecache-custom

[3] https://nx.dev/features/enforce-module-boundaries

[4] https://nx.dev/features/explore-graph

[5] https://github.com/andrena/Java-Magazin-Nx