Scriptie

Afstudeerstage Topicus Onderwijs – Kotlin all the way

Lawik Ayoub

Hogeschool	Saxion hogeschool Deventer
Opleiding	hbo-ICT (SE)
Kwartiel	4.3-4.4
Datum	15-06-2019
Plaats	Deventer

Versiebeheer

Versie	Datum	Status	Opmerking
0.1	19-02-2019	Concept	• Opzet sjabloon • Over Kotlin
0.2	22-02-2019	Concept	• Kanban epic/story/sub-task informatie toegevoegd • Informatie over Gradle toegevoegd
0.3	08-03-2019	Concept	• Over Kotlin: Properties, Data classes • Onderzoeksfase: Backend technologieën: Hibernate: Entities definieren, CRUD operaties
0.4	10-03-2019	Concept	• Onderzoeksfase: Backend technologieën: Hibernate: Queries opstellen
0.5	13-03-2019	Concept	• Onderzoeksfase: Backend technologieën: JAX-RS: Endpoints definieren, (De)Serialization
0.6	21-03-2019	Concept	• Frontend technologieen: Kotlin sdtlib: Fetch API
0.7	22-03-2019	Concept	• Frontend technologieen: Axios
0.8	28-03-2019	Concept	• Frontend technologieen: OAuth
0.9	04-04-2019	Concept	• Frontend technologieen: React & redux
0.10	05-04-2019	Concept	• Frontend technologieën: React & redux
0.11	08-04-2019	Concept	• Feedback van Rogier Hommels verwerkt
0.12	26-04-2019	Concept	• Code delen tussen frontend en backend (multiplatform project) hoofdstukken: Een multiplatform project, Hoe werkt het, Model classes delen.
0.13	29-04-2019	Concept	• Code delen tussen frontend en backend (multiplatform project) hoofdstukken: Interfaces voor REST-services tussen de frontend en de backend delen, Validatie.
0.14	02-05-2019	Concept	• Code testen
0.15	07-05-2019	Concept	• Extra informatie over HTTP-clients toegevoegd
0.16	26-05-2019	Concept	• Bijlagen toegevoegd • Extra informatie over Kotlin verwijderd
0.17	27-05-2019	Concept	• Prototype: doel, terminologie, requirements, functionele omschrijving & mockups
0.18	28-05-2019	Concept	• Prototype: Technische omscrhijving
0.19	29-05-2019	Concept	• Prototype: Conclusie en aanbevelingen
0.20	31-05-2019	Concept	• Kleine zinsopbouw wijzigingen (taalcheck)
0.21	01-06-2019	Concept	• Bijlagen bijgewerkt • Inleiding bijgewerkt • Samenvatting toegevoegd
0.22	09-06-2019	Concept	• Algemene feedback Rogier Hommels verwerkt
0.23	11-06-2019	Concept	• Feedback Sander Evers verwerkt • Feedback m.b.t. hoofdstuk “Prototype” van Rogier Hommels verwerkt
0.24	12-06-2019	Concept	• Architecture diagram prototype toegevoegd
1.0	15-06-2019	Final	• Mockk populariteit bron toegevoegd • Laatste taalcorrecties

Inhoudsopgave

Inleiding

Als afronding van de opleiding hbo-ICT(Software Engineering) bij Saxion hogeschool te Deventer heb ik, Lawik Ayoub, de afstudeeropdracht “Kotlin all the way” bij Topicus Onderwijs te Deventer vervuld. Topicus Onderwijs levert educatieve IT-oplossingen voor het onderwijs, zij wilden graag weten of men door middel van Kotlin, een moderne programmeertaal, voor zowel de frontend als backend kan ontwikkelen binnen één codebase waarbij mogelijk code tussen de frontend en de backend wordt gedeeld. De stageperiode vond plaats tussen 11-02-2019 en 12-07-2019, een totale omvang van 21 weken. Dit document omschrijft achtergrondinformatie m.b.t. de opdracht, de organisatie, de onderzoekvraag en de bijbehorende deelvragen, de beroepsproducten, de onderzoeksopzet, de onderzoeksfase, het prototype, en de conclusies en aanbevelingen.

Samenvatting

Het doel van de afstudeeropdracht was om de hoofdvraag “Is het voor Topicus realistisch om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen?” te beantwoorden. Dit zou de volgende nadelen van traditionele full-stack applicatie tegen gaan: duplicate code, lastige afstemming tussen frontend en backend, (doorgaans) werken met twee verschillende programmeertalen, en het onderhoud van twee verschillende codebases. Kotlin is een programmeertaal die naar verschillende platformen compileert/transpiled, zoals o.a. de jvm (Java bytecode) en JavaScript, dit waren de platformen welke voor dit onderzoek relevant waren. De hoofdvraag werd onderverdeeld tot deelvragen welke ieder verdiepten op een cruciaal onderdeel van een full-stack applicatie. Per deelvraag zijn er een aantal criteria opgesteld waaraan moest worden voldaan; de deelvragen zijn afzonderlijk onderzocht en door middel van een Proof of Concept per deelvraag getoetst. Nadat alle deelvragen waren beantwoord is de hoofdvraag getoetst door alle losse deelvragen middels de ontwikkeling van een “real world application” prototype samen te brengen. Dit prototype is een full-stack (jvm backend, JavaScript frontend) Kotlin applicatie waarbij code tussen de frontend en backend wordt gedeeld. De applicatie biedt leerkrachten de mogelijkheid om oefeningen m.b.t. rekenen met breuken voor hun groep(en) op te stellen die vervolgens door de leerlingen binnen diens groep(en) konden worden uitgevoerd. Het resultaat van het onderzoek is positief; de losse deelvragen zijn positief beantwoord en het is gelukt om het prototype naar de opgestelde eisen te ontwikkelen. Kotlin biedt vele voordelen voor de ontwikkeling van full-stack applicaties, de eerdergenoemde nadelen van traditionele full-stack applicaties komen niet voor in het opgestelde prototype. Tevens biedt Kotlin als programmeertaal veel voordelen t.o.v. Java en JavaScript die het programmeren makkelijker, overzichtelijker, veiliger (lees: minimalisering van fouten) en naar mijn mening ook leuker maken. Er zijn momenteel echter ook een aantal kleine complicaties, deze liggen vooral op het frontend en multiplatform gedeelte van Kotlin. Dit komt mede doordat deze onderdelen nogal nieuwe ontwikkelingen binnen Kotlin zijn maar hier wordt volgens de ontwikkelaars van Kotlin aan gewerkt. Ondanks de kleine complicaties is het wel gelukt om alle deelvragen positief te beantwoorden en het prototype naar behoren te ontwikkelen.

Op basis hiervan kun je stellen dat het voor Topicus Onderwijs inderdaad realistisch is om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen. Daarnaast zijn er ook (mogelijke) uitbreidingsmogelijkheden voor in de toekomst zoals (onderzoek naar) het toepassen van Kotlin voor niet alleen de JVM backend en de JavaScript frontend maar ook Android en iOS, dit zou dus betekenen dat je met minder code voor nog meer platformen tegelijk zou kunnen ontwikkelen.

1. Achtergrond

1.1 Aanleiding

Topicus Onderwijs levert verschillende IT-oplossingen voor het onderwijs, dit wordt door middel van full-stack applicaties gerealiseerd waarbij er een duidelijke scheiding in de codebase is tussen de frontend en de backend. Vanuit Topicus Onderwijs is de vraag ontstaan of zij d.m.v. een programmeertaal voor zowel de frontend als de backend kunnen schrijven.

1.2 Probleemstelling

De grote scheiding tussen frontend en backend zorgt voor een aantal problemen:

- Het zorgt voor duplicate code
- Het is lastiger om de code van de frontend en de backend op elkaar af te stemmen. Als voorbeeld: wanneer je op de backend de structuur van een model class wijzigt, loop je het risico dat je hiermee de frontend stuk maakt en hier pas achter komt wanneer het (tijdens runtime) fout gaat.
- Je moet (doorgaans) met twee verschillende programmeertalen werken.
- Je moet twee verschillende code bases onderhouden.

1.3 De opdracht

Als mogelijke oplossing voor bovenstaande problemen kwam Topicus Onderwijs met het idee om gebruik te maken van Kotlin. Kotlin is een moderne programmeertaal die kan worden gebruikt op verschillende platformen. De naadloze integratie met de JVM maakt het mogelijk om uitgebreide Java EE applicaties te schrijven. Daarnaast kan Kotlin worden vertaald naar Javascript en dat maakt het mogelijk om Kotlin te gebruiken in de browser.

Tijdens mijn afstudeerstage ga ik de vraag “Is het voor Topicus realistisch om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen?” beantwoorden.

2. Organisatie

Deze afstudeerstage werd uitgevoerd door Lawik Ayoub vanuit de opleiding hbo-ICT (Software engineering) volgend op Saxion Hogeschool te Deventer. De afstudeerstage werd uitgevoerd bij Topicus Onderwijs te Deventer. De bedrijfsbegeleider, verantwoordelijk voor de begeleiding van de afstudeerstage vanuit Topicus Onderwijs is Sander Evers. De afstudeerbegeleider, verantwoordelijk voor de begeleiding van de afstudeerstage vanuit Saxion is Rogier Hommels. Verder was er een tweede bedrijfsbegeleider, Kees van Daalen, hij is de schrijver van de opdracht.

2.1 Contactgegevens

Naam	Rol	Telefoon	E-mail
Lawik Ayoub	Afstudeerder	06-21301778	lawik123@gmail.com
Sander Evers	Bedrijfsbegeleider	06-39860597	sander.evers@topicus.nl
Rogier Hommels	Afstudeerbegeleider	06-10681634	r.m.hommels@saxion.nl
Kees van Daalen	Tweede bedrijfsbegeleider (schrijver van de opdracht)	06-51812183	kees.van.daalen@topicus.nl

Tabel 1 – Contactgegevens van de betrokkenen bij de afstudeerstage.

3. Onderzoeksvragen

Om de vraag “Is het voor Topicus realistisch om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen?” te beantwoorden heb ik in overleg met Topicus Onderwijs de volgende deelvragen opgesteld:

- Hoe zijn de gebruikte backend technologieën (Hibernate en JAX-RS) van Topicus Onderwijs toe te passen met Kotlin?
- Hoe kan Kotlin op de frontend worden toegepast?
- Hoe kan code binnen Kotlin tussen de frontend en de backend worden gedeeld?
- Hoe kan men het beste code binnen Kotlin testen?

3.1 Toetsing

Bovenstaande vragen werden onderzocht en vervolgens op twee manieren getoetst, middels meerdere Proof of Concept applicaties en een prototype applicatie.

3.1.1 Proof of Concepts

De Proof of Concept applicaties dienen als toetsing van de losse onderzoeksvragen en de daarbij behorende criteria.

3.1.2 Prototype

Om alle losse onderzoeksvragen samen te brengen en te valideren of het gebruik van Kotlin voor zowel de frontend als de backend met een gedeelde codebase binnen een “real-world application” is toe te passen is er een prototype applicatie ontwikkeld. Het prototype is een educatieve applicatie waarmee leerkrachten oefening m.b.t. rekenen met breuken kunnen opstellen voor hun leerlingen, die vervolgens de oefeningen kunnen uitvoeren. De ontwikkeling aan het prototype begon nadat alle losse deelvragen waren beantwoord.

3.1.2.1 Requirements

Mijn afstudeerbegeleider gaf mij de volgende beschrijving voor het prototype: “Ik stel voor dat je als prototype een educatieve applicatie bouwt waar leerlingen kunnen oefenen met rekenen met breuken. De applicatie wordt gekoppeld met Wise-r (digitaal leermiddelen platform van Topicus Onderwijs) of toegang.org (ECK licentiekantoor van Topicus Onderwijs) en houdt de resultaten van de leerling bij.” Op basis van deze beschrijving heb ik een lijst met (functionele en technische) requirements opgesteld, deze heb ik na overleg met mijn begeleider nader aangevuld. Nadat de losse onderzoeksvragen waren beantwoord en het tijd werd om aan het prototype te beginnen, hebben wij nogmaals de requirements bijgesteld/aangevuld.

Hieronder de lijst met requirements voor het prototype.

3.1.2.1.1 Functionele requirements

nr	Requirement	Priority
RF.01	Leerlingen en leerkrachten kunnen inloggen via een OAuth2 koppeling (Wise-r) naar het platform.	**MUST**
RF.02	Leerkrachten kunnen oefeningen m.b.t. breuken voor hun groepen opstellen (optellen, aftrekken en gelijknamig maken).	**MUST**
RF.03	Leerlingen kunnen aan hun groep beschikbaar gestelde oefeningen uitvoeren.	**MUST**
RF.04	De resultaten van de leerlingen worden bijgehouden.	**MUST**
RF.05	De leerling kan zijn/haar behaalde score van een oefening inzien.	**SHOULD**
RF.06	De resultaten van de leerlingen zijn door diens leerkracht(en) in te zien.	**COULD**

Tabel 2 – Functionele requirements van het prototype.

3.1.2.1.2 Technische requirements

nr	Requirement	Priority
RT.01	De applicatie bestaat uit een backend en client-side frontend die beide in Kotlin geschreven zijn.	**MUST**
RT.02	Indien mogelijk wordt de code voor validatie tussen de frontend en de backend gedeeld.	**SHOULD**
RT.03	Indien mogelijk wordt de code voor de model classes tussen de frontend en de backend gedeeld.	**SHOULD**
RT.04	Indien mogelijk zijn de REST-endpoints op de frontend en de backend op elkaar afgestemd.	**SHOULD**
RT.05	De applicatie maakt op de frontend gebruik van React.	**MUST**
RT.06	De applicatie implementeert de client-kant van de OAuth2 “implicit” flow.	**MUST**
RT.07	De applicatie heeft middels Hibernate een verbinding met een database.	**MUST**
RT.08	De applicatie beschikt middels JAX-RS over een REST-API welke communiceert met de frontend.	**MUST**
RT.09	De applicatie heeft een koppeling met wise-r.nl.	**MUST**

Tabel 3 – Technische requirements van het prototype.

4. Beroepsproducten

Gedurende de afstudeerperiode zijn er aan de volgende beroepsproducten gewerkt:

- Scriptie
- Adviesrapport
- Proof of Concepts
- Functioneel ontwerp van het prototype
- Prototype

5. Onderzoeksopzet

5.1 Methodiek

Tijdens de afstudeerstage heb ik gebruik gemaakt van twee methodieken. Kanban tijdens het onderzoeken van de losse onderzoeksvragen en Scrum tijdens de ontwikkeling van het prototype. Ik heb in beide fases meegedaan met de daily standup van het team.

5.1.1 Kanban

De onderzoeksvragen bestaan uit meerdere onderdelen waarvan daadwerkelijke softwareontwikkeling slechts een klein deel (Proof of Concept) is, het is niet praktisch om deze nog verder op te delen in userstories. Het gebruik van Scrum zou in deze fase slechts voor veel overhead zorgen wat het onderzoek in de weg kan zitten en vertragen. In deze fase werd daarom gebruik gemaakt van de Kanban-methodiek. Deze methodiek biedt de volgende voordelen:

- Net als Scrum heb je binnen Kanban een projectbord waarmee je het project kan organiseren en visualiseren.
- Geen sprints maar een continu proces, wat beter aansluit op het doorlopend onderzoek naar de losse onderzoeksvragen.
- Er kan continu opgeleverd worden in plaats van aan het einde van een sprint.
- Geen (Scrum) rollen wat voor overhead kan zorgen
- Schatting van tijd in plaats van complexiteit, complexiteit is in deze fase lastig in te schatten aangezien ik onderzoek doe naar nieuwe technieken waar ik nog geen ervaring mee heb.

5.1.1.1 Epics, stories, en sub-taken

Om het overzicht te behouden heb ik de onderzoeksvragen onderverdeeld in epics, stories, en sub-taken. Elke losse onderzoeksvraag is een epic, onder een epic vallen 1 of meerdere stories welke ook sub-taken kunnen hebben.

Als voorbeeld, de onderzoeksvraag "*Hoe zijn de gebruikte backend technologieën (Hibernate en JAX-RS) van Topicus Onderwijs toe te passen met Kotlin*?" is als volgt onderverdeeld:

- Toepassing backend technologieën (epic)
    - Toepassing Hibernate (story)
        - Informatie inwinnen (sub-task)
        - PoC Maken (sub-task)
        - Scriptie bijwerken (sub-task)
        - Adviesrapport bijwerken(sub-task)
    - Toepassing JAX-RS (story)
        - Informatie inwinnen (sub-task)
        - PoC Maken (sub-task)
        - Scriptie bijwerken (sub-task)
        - Adviesrapport bijwerken(sub-task)

Wanneer alle stories binnen een epic af zijn, is de epic, en daarmee de onderzoeksvraag afgerond.

Een epic bevat de volgende informatie (zie Bijlage 1 – Epic voorbeeld voor een voorbeeld):

- Een uitgebreide beschrijven van de epic – Uitzoeken hoe de gebruikte backend technologieen (Hibernate en JAX-RS) van Topicus Onderwijs zijn toe te passen met Kotlin.
- De naam van de epic – Toepassing backend technologieën.
- De issues (stories) die onder deze story vallen – ALA-6 en ALA-11.

De stories bevinden zich in de backlog en worden vandaaruit opgepakt (zie Bijlage 2 – Backlog voor de backlog zoals deze er aan het begin van de onderzoeksfase eruitzag). Stories met > bestaan uit sub-tasks, wanneer hier op wordt geklikt worden de sub-tasks weergegeven (zoals ook bij de eerste story, ALA-6 is te zien). Aan de rechterkant van de story staat uit hoeveel sub-tasks deze bestaat en onder welke epic de story valt.

5.1.2 Scrum

Voor de ontwikkeling van het prototype is gebruik gemaakt van de Scrum-methodiek, dit is de ontwikkelmethodiek die vanuit school is meegegeven voor softwareontwikkeling projecten en is tevens de methodiek die het team bij Topicus Onderwijs gebruikt. In deze fase is het geschikt om Scrum te gebruiken aangezien het hoofddoel van deze fase de ontwikkeling van een applicatie is waarvan requirements zijn opgesteld. De opgestelde requirements zijn vertaald naar userstories welke in iteraties (sprints) werden ontwikkeld, de ervaring die ik tijdens de onderzoeksfase heb opgedaan heeft bijgedragen aan de puntenschatting.

5.1.2.1 *Definition of Ready*

Om de kwaliteit van de userstories te garanderen is de volgende Definition of Ready opgesteld, deze Definition of Ready is een opsomming van regels waar een userstory aan moet voldoen om opgepakt te kunnen worden.

- De acceptatiecriteria is duidelijk.
- Er zijn punten aan de userstory toegewezen.
- De losse onderzoeksvragen zijn beantwoord, waardoor de benodigde (basis)kennis om de userstory uit te werken is vergaard.

5.1.2.2 *Definition of Done*

Om de kwaliteit van het gemaakte werk te garanderen is de volgende Definition of Done opgesteld voor de userstories van het prototype, deze Definition of Done is een opsomming van regels waar een userstory aan moet voldoen om als klaar bestempeld te worden.

- De verwachte functionaliteit is (in een eigen branch) ontwikkeld.
- Helper functions zullen worden voorzien van documentatie, verder zullen onderdelen die niet voor zich spreken worden voorzien van inline documentatie.
- De geschreven tests draaien succesvol.
- De bijbehorende pull-request kan gemerged worden (er zijn geen conflicten).
- Indien van toepassing zijn relevante bevindingen/informatie verwerkt in het afstudeerverslag (adviesrapport/scriptie).

5.2 Tools

Tool	Beschrijving
Intellij IDEA	IDE met ondersteuning voor verschillende talen, waaronder Kotlin, hier werden de Kotlin applicaties mee ontwikkeld.
Git	Versiebeheersysteem, werd gebruikt voor het versiebeheer van de code en tevens als backupsysteem van de code.
Jira	Projectbeheersysteem, werd gebruikt voor het beheer van mijn Kanban/Scrum bord.
PostgreSQL	Databasesysteem
pgAdmin 3	Database administration tool voor PostgreSQL
Insomnia	REST-client, werd gebruikt om REST-API endpoints buiten de applicaties om aan te roepen.
Google Drive	Werd gebruikt voor de backup van de documenten.
Gradle	Build automation tool

Tabel 4 – Te gebruiken tools tijdens de afstudeerstage.

5.2.1 Gradle

Men kan op verschillende manieren Kotlin projecten opzetten/bouwen:

- Command line compiler
- Intellij IDEA
- Ant
- Maven
- Gradle

De meest populaire keuze lijkt Gradle te zijn, deze heeft de meeste plugin support, en wordt binnen Kotlin projecten van JetBrains (makers van Kotlin) en populaire Kotlin libraries gebruikt. (napperley, 2017)

Aangezien Gradle de standaard build tool binnen de Kotlin community lijkt te zijn, heb ik ervoor gekozen om mijn Kotlin projecten gedurende de afstudeerstage ook met Gradle op te zetten.

Gradle is een build automation tool, men kan hier plugins inladen, dependencies opgeven, settings configureren, en tasks definiëren. (Zie Bijlage 3 – Gradle build configuratie voor een (simpel) jvm Kotlin project voor een voorbeeld configuratie.)

5.3 Ontwikkelstraat

Er is geen gebruik gemaakt van een ontwikkelstraat, de applicaties werden lokaal ontwikkeld en gedraaid, de reden hiervoor is het feit dat er enkel werd gewerkt aan Proof of Concept en prototype applicaties met als doel het inzicht geven van de gebruikte technieken.

5.4 Kwaliteitsbewaking

Om de kwaliteit van de op te leveren beroepsproducten te waarborgen werd aan het begin van de afstudeerperiode onderstaande kwaliteitsbewaking opgenomen.

Door systematische controlechecks door de begeleiders zal de kwaliteit van de op te leveren producten gewaarborgd blijven. Belanghebbenden zullen op de hoogte worden gehouden van deze controlechecks en waar nodig, feedback geven. Dit zal gebeuren door de volgende stappen toe te passen:

- Het schrijven van de documenten gebeurt volgens ABN en met correcte grammatica.
- Voor het schrijven van de documenten zal gebruik worden gemaakt van Microsoft Word met ingebouwde spellingscontrole.
- De documenten zullen opgesteld zijn naar de richtlijnen die opgegeven zijn in de handleiding van Saxion.
- Documenten zullen tussentijds bij Saxion worden ingeleverd ter controle.
- Waar nodig zal de documentatie aangepast worden op basis van feedback van de begeleiders.
- Er zal eens per week een voortgangsgesprek met de bedrijfsbegeleider plaatsvinden
- Er zullen unit tests worden geschreven voor het prototype, de wijze waarop dit wordt gedaan zal afhangen van de resultaten uit de deelvraag “Hoe kan men het beste code binnen Kotlin testen?”.
- Codewijzigingen voor het prototype zullen d.m.v. pull-requests worden doorgevoerd.
- Code zal volgens de Kotlin Coding Conventions (https://kotlinlang.org/docs/reference/coding-conventions.html) worden geschreven, de ingebouwde inspecties van IntelliJ IDEA zal hieraan bijdragen.

6. Over Kotlin

Kotlin is een open source *statically typed programming language*, welke voor de volgende platformen integratie biedt: JVM (Java), Android, Javascript, Native platformen (iOS, MacOS, Android, Windows, Linux, WebAssembly). Kotlin is zowel een Object Orientated Programming Language als een Functional Programming Language. De ontwikkeling van Kotlin begon in 2010, de eerste officiële 1.0 release was in februari 2016, de huidige versie op het moment van schrijven is 1.3.21 welke op 6 februari 2019 is uitgebracht.

Kotlin biedt veel voordelen, over het algemeen hoeft men (vergeleken met Java) gemiddeld 40% minder code te schrijven om hetzelfde resultaat te krijgen. Kotlin is ook meer type-safe, men kan type-safe voor het web programmeren (wat binnen vanilla Javascript niet mogelijk is) en er is ondersteuning voor non-nullablility waardoor de applicatie minder foutgevoelig is voor NullPointerException. Kotlin biedt vele moderne features die het programmeerwerk makkelijker en overzichtelijker maken zoals o.a. smart casting, higher-order functions, extension functions, lambdas met receivers, etc.

(Kotlin Programming Language, z.d.)

Wanneer je Kotlin voor de JVM of JavaScript toepast hoef je geen afscheid te nemen van bestaande code/libraries/frameworks. Kotlin is 100% interoperable met Java, wat wil zeggen dat je Java code binnen Kotlin kan aanroepen en Kotlin code binnen Java kan aanroepen (Kotlin Programming Language, z.d.). Ook is er interoperability tussen Kotlin en JavaScript mogelijk, dit ligt echter iets gecompliceerder dan bij Java, aangezien JavaScript een *dynamicically-typed language* is, zijn er geen type checks tijdens het compileren; men kan daarom JavaScript binnen Kotlin op twee manieren toepassen, via zogenoemde *dynamic* types (hiermee verlies je wel de type-safety die Kotlin beidt), of door het gebruik van zogenoemde header files, een header file bestaat uit functie/class declaraties voor de betreffende JavaScript library, dit zorgt er voor dat men binnen Kotlin type-safe met bestaande JavaScript libraries kan werken (Kotlin Programming Language, z.d.).

7. Onderzoeksfase

Het doel van de onderzoeksfase is het beantwoorden van de losse onderzoeksvragen. Voordat ik met een onderzoeksvraag aan de slag ging, heb ik eerst met mijn begeleider een aantal criteria voor de vraag opgesteld, dit zorgt ervoor dat de vraag concreter is en maakt het duidelijk wat er precies onderzocht moet worden. Per vraag is er als toetsing een Proof of Concept opgesteld. Dit hoofdstuk omschrijft de bevindingen en resultaten.

7.1 Backend technologieën

Dit hoofdstuk omschrijft de resultaten van de onderzoeksvraag “Hoe zijn de gebruikte backend technologieën (Hibernate en JAX-RS) van Topicus Onderwijs toe te passen met Kotlin?”

7.1.1 Hibernate

Hibernate is een ORM (object-relational mapping) tool voor Java, waarmee Java classes met database tabellen ge-mapped kunnen worden (Hibernate, z.d.).
Men wil graag weten hoe Hibernate is toe te passen met Kotlin en welke Kotlin features het gebruik van Hibernate kan ondersteunen. Voor dit onderdeel zijn de volgende criteria opgesteld:

- Het moet mogelijk zijn om binnen Kotlin entitites te kunnen definiëren.
- Mogelijkheid tot CRUD operaties.
- Indien mogelijk, op een minder verbose manier (dan de standard manier) queries kunnen opstellen (eventueel door middel van een library).

7.1.1.1 Entities definiëren

Men wil graag binnen Kotlin entities kunnen definiëren/mappen, dit is binnen Kotlin mogelijk door een aantal technologieën toe te passen. Om de entities te definiëren heb ik gebruik gemaakt van *data classes*, de JPA annotaties kunnen aan de gedefinieerde properties worden gebonden. Dit ziet er als volgt uit:

```
@Entity
@Table(name = "person")
data class Person(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null,

    @Column(nullable = false)
    var name: String
)
```

Dit zal echter niet gaan werken zonder de Kotlin JPA gradle plugin toe te passen, dit zorgt er voor dat Hibernate de class kan instantiëren, ondanks dat deze geen default constructor heeft. Onder water maakt deze plugin een default constructor voor classes met @Entity, @Embeddable en @MappedSuperclass annotaties beschikbaar, deze constructor kan niet direct door de programmeur vanuit de Kotlin code worden aangeroepen maar wordt middels reflectie aan Hibernate beschikbaar gesteld.

Een belangrijk punt om op te letten is de door de data class gegenereerde toString function in het geval van bidirectional JPA mappings. Indien je gebruik maakt van een bidirectional mapping, is het verstandig om de toString function van (één van) de classes te overriden en daarbij te zorgen dat de toString van de mapped class niet wordt aangeroepen. De default toString roept namelijk de toString function van beide classes recursief aan wat resulteert in recursie die niet stopt en dat uiteindelijk resulteert in een StackOverflow error.

7.1.1.2 Mogelijkheid tot CRUD operaties

Kotlin voert geen belemmering voor het uitvoeren van standaard Hibernate CRUD operaties zoals load, multiload, loadAll, count, save, delete, etc en biedt naast de mogelijkheid tot *default parameters* bij sommige functions geen enorme voordelen tegenover Java.

7.1.1.3 Queries opstellen

De JPA CriteriaQuery API is krachtig en type-safe, de API heeft echter een nadeel: hij is *zeer* verbose, een relatief simpele use case ziet er met de JPA CriteriaQuery API (in Kotlin) als volgt uit:

```
fun findAbove18AndNameContains(pattern: String): List<Person> {
    val builder = session.criteriaBuilder
    val criteriaQuery = builder.createQuery(Person::class.java)
    val root = criteriaQuery.from(Person::class.java)

    criteriaQuery.where(
        builder.like(root.get(Person::name.name), "%$pattern%"),
        builder.ge(root.get(Person::age.name), 18)
    )

    criteriaQuery.orderBy(
        builder.asc(root.get<Int>(Person::age.name)),
        builder.asc(root.get<String>(Person::name.name))
    )

    return session.createQuery(criteriaQuery).resultList
}
```

Bij complexere queries is het probleem evidenter, vooral wanneer je meerdere (optionele) argumenten meegeeft welke naar predicates vertaald moeten worden of wanneer je gebruik maakt van or -en and clausules.

Als onderdeel van het onderzoek heb ik uitgezocht of door middel van Kotlin (eventueel middels een third party library/framework) het schrijven van queries versimpeld kan worden. Kotlin biedt de mogelijkheid om middels *Function literals with receiver* DSLs (Domain Specific Language) te schrijven, dit leek mij een perfect middel voor het versimpelen van de CriteriaQuery API. Op het internet kwam ik een bestaande oplossing tegen welke ook een DSL had gemaakt, echter maakte deze gebruik van een externe library: Spring Data. Zowel ik als Topicus Onderwijs hadden de voorkeur voor een oplossing zonder de afhankelijk van een externe library waardoor de Hibernate implementie zo flexibel mogelijk is in te zetten.

Ik heb besloten om zelf een DSL voor de CriteriaQuery API te schrijven, deze DSL maakt het mogelijk om bovenstaande query op de volgende manier op te stellen:

```
fun findAbove18AndNameContains(pattern: String): List<Person> {
    return session.createQuery<Person> {
        where {
            Person::name.like("%$pattern%")
            Person::age.ge(18)
        }
        orderBy {
            asc(Person::age)
            asc(Person::name)
        }
    }.resultList
}
```

Zoals je ziet is bovenstaande query veel simpeler opgesteld en is deze makkelijker te lezen (nogmaals, het verschil is veel evidenter bij complexere queries).

De DSL maakt het ook mogelijk om gemakkelijk or/and clausules aan te maken, deze kunnen direct in de where clausule worden aangeroepen:

```
where {
    or {
        Person::name.like("%$pattern%")
        Person::age.ge(18)
    }
}
```

Dit creëert een nieuwe scope waarbij alle daarbinnen gedefinieerde predicaten aan een or clausule worden toegevoegd welke vervolgens weer aan de where clausule wordt toegevoegd. Dit is vergeleken met de gebruikelijke wijze van het aanmaken van or/and clauses met de CriteriaQuery API veel makkelijker en overzichtelijker.

De or/and clausules kunnen tevens ook binnen andere or/and clausules worden aangeroepen:

```
where {
    or {}
    and {
        or {}
        or {}
    }
}
```

De DSL is mogelijk gemaakt door een reeks aan Kotlin functionaliteiten toe te passen. De definitie van de `session.createQuery<Person>` function ziet er als volgt uit (de function uit het voorbeeld heeft 1 type parameter, deze roept onderstaande function aan waarbij de meegegeven paramater voor zowel T als X wordt meegegeven.):

```
inline fun <reified T, reified X> Session.createQuery(
    init: QueryContext<T, X>.() -> Unit
): Query<T> {
    val builder = this.criteriaBuilder
    val criteriaQuery = builder.createQuery(T::class.java)
    val root = criteriaQuery.from(X::class.java)
    val context = QueryContext<T, X>(builder, criteriaQuery, root)
    context.init()
    return createQuery(criteriaQuery)
}
```

Bovenstaande function is een *inline extension function* voor Session (dit is de Hibernate Session class, de *this* binnen deze function verwijst dus naar Session) waarbij twee reified type parameters (de eerste is de CriteriaQuery type en de tweede de Root type) aan worden meegegeven, hierdoor kan Kotlin de class uit de type parameter ophalen.

De function maakt op basis van de type parameters de benodigde onderdelen: een CriteriaBuilder, een CriteriaQuery, en een Root, deze worden vervolgens aan een QueryContext class toegevoegd (dit is een wrapper voor de benodigde onderdelen).

Vervolgens wordt de “init” parameter, welke van het type *function literal with receiver* is op de QueryContext toegepast. Dit zorgt ervoor dat men in de body van de meegegeven “init” function bij de functions van QueryContext kan. QueryContext kent de where en orderBy functions welke op hun beurt ook weer gebruik maken van een soortgelijke constructie, de “where” function ziet er als volgt uit:

```
fun <T, X> QueryContext<T, X>.where(setup: WhereBuilder<T, X>.() -> Unit) {
    val whereBuilder = WhereBuilder(this)
    whereBuilder.setup()
    this.criteriaQuery.where(whereBuilder.build())
}
```

Zoals je ziet heeft dit een soortgelijke constructie, men kan in de body van de where function bij de functions van de WhereBuilder class, deze class ziet er als volgt uit (ingekort):

```
class WhereBuilder<T, X>(queryContext: QueryContext<T, X>) : BaseQueryBuilder<T,
X>(queryContext) {
    private val predicates = mutableListOf<Predicate>()

    // equal
    @JvmName("equalByProp")
    fun <R> KProperty1<X, R?>.equal(value: R) = equal(this, value)

    fun <R> equal(property: KProperty1<X, R?>, value: R) =
equal(queryContext.root.get(property), value)

    fun <T> equal(expression: Expression<T>, value: T) =
predicates.add(queryContext.builder.equal(expression, value))

    // ...overige predicate functions (ge/le/like/etc)

    fun build(): List<Predicate> {
        return predicates
    }

}
```

In deze class zijn alle predicate functions gedefinieerd, deze maakt door middel van de benodigde classes uit de QueryContext (welke aan de constructor is meegegeven), en de meegegeven parameter(s) predicates aan en houdt deze bij in een list. De build function returned deze list, deze function wordt in de where function aangeroepen en voegt de predicates toe aan de where van de CriteriaQuery.

Note: de eerste equal function heeft een @JvmName annotatie omdat deze en de function daaronder, wanneer zij eenmaal gecompileerd zijn, zijn deze exact hetzelfde op de jvm (java virtual machine) de @JvmName annotatie geeft de functie op de jvm de naam van de meegegeven parameter.

Nadat de body van de QueryContext is verwerkt, wordt er een Query aangemaakt en ge-returned, hier kan men bijvoorbeeld de resultList function op aanroepen.

7.1.2 JAX-RS

Java API for RESTful Web Services, oftewel JAX-RS, is een Java specificatie voor het creëren van REST APIs (JCP, z.d.). Voor de implementatie van de specificatie heb ik gekozen voor het gebruik van RESTEasy, deze wordt binnen Topicus Onderwijs ook gebruikt.

Voor dit onderdeel werden de volgende criteria opgesteld:

- Mogelijkheid tot het definiëren van GET, POST, PUT, en DELETE endpoints, waarbij middels annotaties gebruik kan worden gemaakt van de volgende parameters:
  - Query parameters
  - Header parameters
  - Path parameters
- Kotlin objecten kunnen serializeren naar JSON en JSON kunnen deserializen naar Kotlin objecten.

7.1.2.1 Endpoints definiëren

Het definiëren van JAX-RS endpoints kan binnen Kotlin op de gebruikelijke wijze die men ook van Java gewend is. Hieronder een voorbeeld van een endpoint:

```
@Path("/person")
@Produces(MediaType.APPLICATION_JSON)
class PersonEndpoint {

    @GET
    @Path("/{id}")
    fun personByIdGET(@PathParam("id") id: Int): Response {
        val person: Person? = Data.findById(id)
        return if (person != null) {
            Response.ok().entity(person).build()
        } else {
            Response.status(Response.Status.NOT_FOUND).build()
        }
    }

    // ...overige endpoints

}
```

Zoals je ziet is de syntax voor het definiëren van endpoints gelijkmatig met Java, dit geldt ook voor het gebruik van parameters, Kotlin voert verder ook geen belemmering en men hoeft binnen Kotlin geen speciale stappen te ondernemen om dit mogelijk te maken.

7.1.2.2 (De)Serialization

Voor (de)serialization binnen JAX-RS maakt men over het algemeen gebruik van de Jackson(2) library. Jackson kan binnen Kotlin out-of-the-box voor *serialization* worden gebruikt. Echter loop je (bij het gebruik van Kotlin data classes) met *deserialization* wel tegen problemen aan, dit komt omdat een data class geen default constructor aanmaakt. Gelukkig hebben de makers van Jackson een module ontwikkeld die deserialization middels Jackson binnen Kotlin wel mogelijk maakt (deze module is alleen beschikbaar voor Jackson 2). Om de module toe te passen moet je de module binnen de ObjectMapper ContextResolver registreren, dit kan op verschillende manieren, bijvoorbeeld middels de JAX-RS @Provider annotatie (zie Bijlage 4 – JAX-RS Jackson 2 object mapper voor Kotlin registeren).

7.2 Frontend technologieën

Dit hoofdstuk beschrijft de resultaten van het frontend gedeelte, dit onderdeel was eerst opgesplitst in twee onderdelen: “Welke frontend frameworks/libraries kunnen binnen Kotlin gebruikt worden, en hoe kunnen deze worden toegepast?” en “Hoe kan men zelf ondersteuning voor frontend frameworks/libraries ontwikkelen? Tijdens het opstellen van de criteria en tijdens het verkennen van wat Kotlin op de frontend kan realiseerde ik mij dat er hier en daar overlap tussen de twee onderdelen zit. Om de resultaten overzichtelijk te houden en dubbele resultaten te voorkomen is ervoor gekozen om deze onderdelen samen te voegen tot 1 frontend onderdeel.

Er is samen met Topicus Onderwijs een lijst met criteria opgesteld over wat Topicus Onderwijs op de frontend met Kotlin wil kunnen doen:

- Mogelijkheid tot het maken van een Single Page Application (SPA)
- Mogelijkheid tot routing met history API
- Makkelijk HTTP-calls kunnen doen. (Bij het uitvoeren van een HTTP-call als programmeur weinig code te hoeven schrijven.)
- Mogelijkheid tot het definiëren van herbruikbare componenten
- Mapping tussen state en view (automatische update van de view bij state wijzigingen).
- Browser API’s kunnen aanroepen (LocalStorage)
- Ondersteuning voor de laatste versie van: Chrome, Edge, Firefox en Safari
- Ondersteuning voor OAuth (implicit flow)
- Ondersteuning voor bestaande JavaScript libraries (npm)

Op basis van de criteria is er een lijst met interessante technologieën opgesteld:

- Kotlin standard library (stdlib)
    - Browser API’s kunnen aanroepen
    - Ondersteuning voor OAuth (implicit flow)
    - Makkelijk HTTP calls kunnen doen (Fetch API)
- React + Redux
    - Mogelijkheid tot het maken van een SPA
    - Mogelijkheid tot routing met history API (middels react-router)
    - Mogelijkheid tot het definiëren van herbruikbare componenten
    - Mapping tussen state en view
    - Ondersteuning voor bestaande JavaScript libraries
- Axios
    - Makkelijk HTTP calls kunnen doen
    - Ondersteuning voor bestaande JavaScript libraries
- Ktor
    - Makkelijk HTTP calls kunnen doen

7.2.1 HTTP-clients

In hedendaagse JavaScript webapps maken HTTP-requests een groot onderdeel uit van de applicatie. Men wil zo makkelijk mogelijk (zonder veel boilerplate code) HTTP-requests kunnen uitvoeren, tijdens mijn afstuderen heb ik 3 verschillende technologieën voor het uitvoeren van HTTP-requests onder de loep genomen. Het betreft de volgende 3 technologieën:

- Fetch API
- Axios
- Ktor

In dit hoofdstuk zullen alle 3 de technologieën en het gebruik daarvan worden uitgelicht.

7.2.1.1 Fetch API

Dit onderdeel heeft betrekking tot de criteria “Makkelijk HTTP calls kunnen doen”, verder wordt er ingegaan op (de)serialization op de frontend. De fetch API zit in de Kotlin stdlib, deze kan gebruikt worden om HTTP calls te doen. Om het gebruik van de fetch API binnen Kotlin te demonsteren is er gebruik gemaakt van een online placeholder API, de endpoint is binnen het project als volgt gedefinieerd:

```
private const val API_URL = "https://jsonplaceholder.typicode.com/todos"
```

Het gebruik van de fetch API is vergelijkbaar met JavaScript:

```
fun fetchGET(): Promise<Any> {
    return window.fetch("$API_URL/1", RequestInit(method = "GET")).then {
        it.json()
    }
}
```

Deze aanpak heeft echter wel een groot probleem, de return type is Any in plaats van Todo, hierdoor kan er dus geen gebruik worden gemaakt van Todo functions/properties. De oplossing is om de body te deserializen, hiervoor is gebruik gemaakt van de kotlinx.serialization library en plugin, deze maken het mogelijk om op de frontend Kotlin objecten te serializen naar json en json te deserializen naar Kotlin objecten. De plugin en library bieden de @Serializable en @Optional annotaties (Zie Bijlage 5 – Kotlinx-Serialization voorbeeld). @Serializable geeft aan dat de class te (de)serializen is en @Optional geeft aan dat de property nullable/optional is. Wanneer de class de @Serializable annotatie heeft, kan de static serializer() function van de class worden aangeroepen, deze kan worden gebruikt om middels de Json.stringify() Kotlin objecten te serializen naar json en Json.parse() json te deserializen naar Kotlin objecten:

```
fun fetchGET(): Promise<Todo> {
    return window.fetch("$API_URL/1", RequestInit(method = "GET")).then {
        it.text()
    }.then {
        Json.parse(Todo.serializer(), it)
    }
}
```

Nu is de return type een valide Todo object waarmee men type-safe op de frontend te werk kan gaan.

Er is een helper extension function geschreven om bovenstaande iets te versimpelen (Zie Bijlage 6 – Fetch helper extension function):

Door middel van de extension function kan de request als volgt worden geschreven:

```
fun fetchGET(): Promise<Todo> {
    return window.fetch("$API_URL/1", RequestInit(method = "GET"),
Todo.serializer())
}
```

7.2.1.2 Axios

Dit onderdeel heeft betrekking tot de volgende criteria:

- Makkelijk HTTP calls kunnen doen
- Ondersteuning voor bestaande JavaScript libraries

Verder zijn er in dit onderdeel ook de volgende punten meegenomen:

- Integratie van Webpack met Kotlin
- Het verwijderen van ongebruikte code om de grootte van het uiteindelijke JavaScript bestand te verkleinen. De Kotlin stdlib is in totaal 1.7MB, en deze moet met de applicatie worden bijgeleverd, door ongebruikte onderdelen uit de stdlib te verwijderen kan de totale grootte van het uiteindelijke JavaScript bestand verkleind worden.

Axios is een JavaScript library welke een HTTP client levert waarmee HTTP calls gedaan kunnen worden (Axios, z.d.). Binnen JavaScript is het gebruikelijk om libraries middels npm op te halen, middels de kotlin-frontend plugin is het mogelijk om ook binnen Kotlin gebruik te maken van het npm ecosysteem. Door middel van de plugin is het mogelijk om vanuit de build.gradle npm (dev) dependencies te definiëren:

```
kotlinFrontend {
    npm {
        dependency "axios", "0.18.0" // versie meegeven is optioneel
        // ... overige (dev)depedencies
    }
    // ... overige kotlinFrontend settings
}
```

In plaats van dependency kan ook gebruik worden gemaakt van devDependency om aan te geven dat het om een dev dependency gaat.

De depedencies kunnen vervolgens via de npm task geinstalleerd worden (deze wordt echter al automatisch bij de build task aangeroepen, je hoeft deze dus niet handmatig aan te roepen).

De JavaScript library is nu dus toegevoegd als dependency en zal tijdens het build proces geïnstalleerd worden, maar deze kan nog niet (type-safe) binnen Kotlin worden toegepast. De compiler(transpiler) kan namelijk niet direct (type-safe) overweg met de JavaScript library, middels zogenoemde *external modifier* moet aan de compiler duidelijk worden gemaakt dat de implementatie van de betreffende class/function/property extern is gedefinieerd. Er wordt dus een *definitie* opgegeven (de implementatie is de JavaScript library zelf), dit is vergelijkbaar met de *.d.ts bestanden van TypeScript (er is zelfs een tool die *.d.ts bestanden om kan zetten tot Kotlin code met external modifiers, daar is in dit project echter geen gebruik van gemaakt).

Hieronder de definities van de Axios library:

```
@JsModule("axios")
private external fun <T> axios(config: AxiosConfigSettings):
Promise<AxiosResponse<T>>
```

Zoals je ziet heeft deze external function ook een @JsModule annotatie, deze annotatie geeft aan dat de function een module is welke geïmporteerd moet worden. In andere woorden: “axios” wordt ergens geëxporteerd middels “module.exports.axios” en wordt middels de @JsModule annotatie geïmporteerd, de JavaScript equivalent hiervan is:

```
import axios from 'axios'
```

Zie Bijlage 7 – Axios external definitions voor de rest van de Axios external definitions en Bijlage 7 – Axios TypeScript definitions om te zien hoe deze vergelijkt met de .d.ts definitions.

Er is een function geschreven die het configureren van de request makkelijker maakt en de response deserialized. (Voor (de)serialization is er gebruik gemaakt van dezelfde technologie als tijdens het fetch API onderdeel, dit zal daarom niet opnieuw worden toegelicht). Zie Bijlage 8 – Axios helper function

De function heeft twee parameters, de deserializationStrategy voor het deserializen van json naar Kotlin objecten en de config welke de axios configuratie bevat. Omdat de config parameter van het type *function literal with receiver* is en deze de laatste parameter is kan deze buiten de ronde haakjes worden meegegeven en kan de configuratie gemakkelijk worden opgesteld.

Door middel van de function kan de request als volgt worden geschreven:

```
fun axiosGET(): Promise<Todo> {
    return axios(Todo.serializer()) {
        url = "$API_URL/1"
        method = "GET"
    }.then { it.data }
}
```

Het zou veel werk zijn om alle JavaScript bestanden aan de HTML toe te voegen, binnen het JavaScript landschap wordt er veel gebruik gemaakt van Webpack om (o.a.) alle gebruikte modules te bundelen tot 1 bestand. De kotlin-frontend-plugin biedt naast ondersteuning voor npm ook ondersteuning voor Webpack. Deze is vanuit de build.gradle te configureren (zie Bijlage 9 – Webpack Gradle configuratie)

Het is ook mogelijk om middels JavaScript bestanden de Webpack configuratie nog verder te configureren, deze JavaScript bestanden moeten in de “webpack.config.d” map in de root van het project geplaatst worden, de inhoud van de JavaScript bestanden zullen (op alfabetische volgorde, maak gebruik van nummer prefix om de volgorde naar wens aan te passen) aan het einde van het uiteindelijk gegenereerde webpack.config.js configuratie bestand worden toegevoegd. Als voorbeeld wordt de naam van de uiteindelijke bundle middels een JavaScript bestand, filename.js, overschreven. Dit is slechts een simpel voorbeeld (de bundle naam kan al via de build.gradle configuratie worden geconfigureerd), het gebruik van JavaScript bestanden maken het mogelijk om Webpack volledig naar wens te configureren:

```
config.output.filename = "kotlin-poc-frontend-axios.bundle.js"
```

Bovenstaande regel (uit filename.js) wordt dus onderaan de gegenereerde webpack.config.js toegevoegd, zie Bijlage 10 – Gegenereerde webpack.config.js voor het uiteindelijke webpack.config.js bestand.

Wanneer men de volgende command uitvoerd:

```
gradlew clean bundle -Pprod
```

zal er middels Webpack een production bundle worden gemaakt, deze bundle is minified maar alsnog 955KB (ondanks dat dit een klein project was), dit komt mede door de Kotlin standard library. Er is een Dead Code Elimination plugin welke ongebruikte code verwijderd, deze tool wordt automatisch bij het bouwen van de bundle toegepast. Dankzij de tool gaat de grootte van de bundle van 955KB naar 241KB, een aanzienlijk verschil.

7.2.1.3 Ktor

Dit onderdeel heeft betrekking tot de criteria “Makkelijk HTTP calls kunnen doen”.

Ktor is een HTTP-library ontwikkeld door JetBrains (het team achter Kotlin). Ktor is volledig in Kotlin geschreven waardoor deze direct binnen Kotlin projecten gebruikt kan worden(i.t.t. bijvoorbeeld Axios waarbij je eerst e.e.a. moet definiëren). Een groot voordeel van Ktor is dat deze *automatisch* de body voor jou kan (de)serializen, hierdoor hoef je niet elke keer de serializer mee te geven wat boiler plate code voorkomt. (ktor, z.d.)

Hiervoor is dezelfde kotlinx.serialization library/plugin die bij Fetch en Axios werden gebruikt nodig, verder moeten de volgende dependencies aan de build.gradle dependencies worden toegevoegd:

```
compile "io.ktor:ktor-client-js:1.1.3"
compile "io.ktor:ktor-client-json-js:1.1.3"
```

De Ktor client moet eerst aangemaakt/geconfigureerd worden alvorens deze kan worden gebruikt, dit ziet er als volgt uit:

```
private const val API_HOST = "jsonplaceholder.typicode.com"
private const val API_PATH = "/todos"

val client = HttpClient {
    install(JsonFeature)
    defaultRequest {
        host = API_HOST
    }
}
```

Zoals je ziet wordt er hier een *JsonFeature* geïnstalleerd, deze feature zorgt ervoor dat de body automatisch kan worden deserialized (indien je response object serializable is, hiervoor moeten dezelfde stappen worden ondernomen als in het hoofdstuk Fetch API worden beschreven). Features binnen Ktor zijn te vergelijken met JAX-RS filters/interceptors die in de pipeline van de requests/responses worden geïnjecteerd, hierdoor kan je bijvoorbeeld ook features maken voor logging, authenticatie, etc. Zoals je ziet wordt de default api host ook geconfigureerd, hierdoor hoef je in de request methods enkel nog de api path te definiëren welke door Ktor automatisch achter de host wordt geplakt.

Het uitvoeren van een (post) request ziet er als volgt uit:

```
suspend fun post(): Todo = client.post {
    url(API_PATH)
    body = Todo(1, "test", false)
    contentType(ContentType.Application.Json)
}
```

Zoals je ziet hoef je in de request niks met serializers te doen, aan de hand van de function return type weet deze de response body automatisch te deserializen naar een Todo object. Veder zie je dat je de body ook niet hoeft te serializen, dit doet Ktor automatisch *wanneer de content-type application/json is.* Omdat in de configuratie een default host is meegegeven hoef je hier enkel de path van de endpoint mee te geven (API_PATH = “/todos”). Tot slot zie je dat dit een *suspend* function betreft, suspend functions zijn onderdeel van Kotlin Coroutines, dit is een (standaard) library binnen Kotlin voor asynchroon programmeren waar Ktor veel gebruik van maakt. De volledige omvang van Coroutines is veel te groot om in zijn volledigheid in dit onderdeel toe te lichten. Wat belangrijk is om te weten is dat suspend functions enkel binnen andere suspend functions of binnen coroutine (scopes) kunnen worden aangeroepen.

Het uitvoeren van de requests (bijvoorbeeld in de main function) ziet er als volgt uit:

```
fun main() {
    GlobalScope.async {
        println(getList())
        println(post())
        println(put())
        delete()
        println("Todo deleted")
    }
}
```

Zoals je ziet wordt de eerdergenoemde *post* function als tweede aangeroepen binnen een GlobalScope.async coroutine, deze voert de lambda asynchroon uit. Het is hierbij belangrijk om te weten dat *de lambda zelf asynchroon wordt uitgvoerd*, dit wil zeggen dat alles binnen de lambda synchroon wordt uitgevoerd, de *post* function wordt pas nadat *println(getList())* is uitgevoerd aangeroepen. Maar als je *na de lambda* een function aanroept zal deze vrijwel direct worden aangeroepen.

Je ziet in de benamingen van de functions in de lambda dat alle namen standaard HTTP methods zijn op *getList* na. De reden hiervoor is het feit dat Ktor een *top level JSON array* niet als valide JSON ziet en deze dus niet deserialized. Als oplossing heb ik helper functions geschreven om alsnog gebruik te kunnen maken van automatische deserialization. De function ziet er als volgt uit:

```
suspend fun getList(): List<Todo> = client.list {
    url(API_PATH)
}
```

Zoals je ziet is deze vergelijkbaar met de post function, je hoeft ook hier niet handmatig te deserializen omdat de list helper function dit voor jou doet. De list (extension) function ziet er als volgt uit:

```
suspend inline fun <reified T : Any> HttpClient.list(noinline block:
HttpRequestBuilder.() -> Unit): List<T> =
    customSerializerRequest(T::class.serializer().list, block)
```

Zoals je ziet heeft deze een function literal with receiver van het type HttpRequestBuilder als parameter (zodat je de request zoals ieder andere ktor request kan configureren). De function roept de customSerializerRequest function aan waarbij de list serializer van de class en de configuratie (block) wordt meegegeven. De customSerializerRequest ziet er als volgt uit:

```
suspend fun <T : Any> HttpClient.customSerializerRequest(
    serializer: KSerializer<T>,
    block: HttpRequestBuilder.() -> Unit
): T {
    return request<HttpResponse> {
        apply(block)
    }.use {
        if (it.status.isSuccess()) {
            return Json.parse(serializer, it.readText())
        } else {
            throw BadResponseStatusException(it.status, it)
        }
    }
}
```

Dit is wat lower-level Ktor waarbij “handmatig” een request wordt uitgevoerd waarnaar wordt gecontroleerd of de response status code in de 200 range zit, indien dit het geval is wordt de body d.m.v. de meegegeven serializer deserialized en wordt vervolgens de deserialized object ge-returned. Indien de response status code niet in de 200 range zit wordt er een BadResponseStatusException ge-throwed (dit gebeurt ook binnen de standaard request functions van Ktor).

7.2.1.4 Vergelijking

Alle drie de technologieën hebben voor- en nadelen, zie de tabel hieronder.

Client	Voordelen	Nadelen
Fetch API	• Geen extra dependencies • Lightweight • Veel vrijheid • Direct Kotlin support (de Fetch API zit in de standard library) • Wanneer je hierop verder bouwt heb je meer controle over welke features je ondersteunt, bij libraries kunnen er features inzitten waar je geen gebruik van gaat maken.	• Veel boilerplate code • Kans dat je het wiel opnieuw aan het uitvinden bent wanneer je hier allerlei helper function omheen gaat bouwen. • Standaard weinig features vergeleken met libraries.
Axios	• Bestaande, vertrouwede library welke door veel mensen wordt gebruikt* • Veel features: Intereceptors, transformers, cancel request/timout, etc.	• Geen direct Kotlin support, het is een JavaScript library waar je external definitions voor moet schrijven/genereren.
Ktor	• Direct Kotlin support (de library is volledig in Kotlin geschreven). • Geschreven door Jetbrains (hogere kans dat het up-to-date met Kotlin blijft). • Weinig boilerplate code • Automatische (de)serialization • Mogelijkheid tot het installeren van features. • Maakt gebruik van coroutines waardoor je een soortgelijke flow als async/await kan toepassen en je niet bij elke request gebruik hoeft te maken van .then() • Naast JavaScript is deze ook beschikbaar voor andere platformen.	• Kan niet alles automatisch deserializen. • Learning curve, je moet ook e.e.a. weten over Kotlin Coroutines om hier goed gebruik van te kunnen maken

Tabel 5 – HTTP clients vergelijking

* ~58.9k stars op GitHub (GitHub, z.d.), ~18m downloads per maand op NPM (npm-stat, z.d.)

Zoals je ziet valt er voor elke technologie wat te zeggen, ik zou ze in de volgende omstandigheden toepassen:

- Fetch API – Wanneer je webapp weinig gebruik maakt van HTTP requests.
- Axios – Wanneer je webapp veel gebruik maakt van HTTP request *en je geen gebruik wil maken van Kotlin Coroutines.*
- Ktor – Wanneer je webapp veel gebruik maakt van HTTP requests, deze library heeft veel features en direct Kotlin support.

7.2.2 OAuth

Dit onderdeel heeft betrekking tot de volgende criteria:

- Browser APIs kunnen aanroepen:
  - LocalStorage
  - window.location
  - window.atob
- Ondersteuning voor OAuth (implicit grant).

Note: Het registeren van de applicatie op de dienst waarop geautoriseerd dient te worden valt buiten de scope van dit onderdeel, dit onderdeel heeft enkel betrekking tot het uitvoeren van een OAuth flow (implicit) binnen Kotlin op de frontend.

OAuth is een protocol waarbij de gebruiker toegang aan een applicatie kan verlenen om namens de gebruiker acties uit te voeren (acces token) of om informatie over de gebruiker op te halen (id token). Het OAuth protocol kent verschillende grant types voor verschillende doeleinden, dit onderdeel heeft enkel betrekking tot de implicit grant. In grote lijnen ziet deze flow er als volgt uit:

1. De gebruiker wordt vanuit de applicatie (client) naar de authorization server gestuurd met alle benodigde query parameters, inclusief een random gegenereerde state string die op de client is opgeslagen.
2. De gebruiker logt in en verleent toestemming aan de applicatie om namens de gebruiker acties uit te voeren.
3. De gebruiker wordt teruggestuurd naar de applicatie (naar de door de applicatie gedefinieerde URI) inclusief een hash fragment in de URI met daarin de token en (als het goed is, de eerder gegenereerde) state string.
4. De state string wordt vergeleken met de eerder opgeslagen state string, indien deze identiek zijn, wordt de token uit de URI gehaald. Deze kan worden gebruikt om namens de gebruiker acties uit te voeren (access token) of kan informatie over de gebruiker bevatten (id token).

Om bovenstaande middels Kotlin te bereiken is er gebruik gemaakt van een aantal browser APIs:

- LocalStorage (voor het opslaan van de state en de jwt)
- window.location (voor het wijzigen van de huidige URL naar de authorization server, en voor het opvragen van de teruggestuurde location hash parameters)
- window.atob (voor het decoderen van de base64 encoded jwt)

Bovenstaande browser APIs kunnen middels de Kotlin standard library worden aangeroepen, het uitvoeren van een OAuth flow kan dus zonder belemmeringen middels Kotlin op de frontend worden uitgevoerd.

Binnen Kotlin kan men op de volgende wijze data uit de localstorage ophalen:

```
var value = localStorage["KEY"]
value = localStorage.get("KEY")
value = localStorage.getItem("KEY")
```

Alle drie notaties hebben hetzelfde effect, het is binnen Kotlin echter wel gebruikelijk om gebruik te maken van de indexing notatie: localStorage[“KEY”].

Op soortgelijke wijze kan men data in de localstorage opslaan:

```
localStorage["KEY"] = "VALUE"
localStorage.set("KEY", "VALUE")
localStorage.setItem("KEY", "VALUE")
```

Ook hier geldt dat de drie notaties hetzelfde effect hebben en de indexing notatie binnen Kotlin gebruikelijk is.

Het veranderen van de huidige URL kan als volgt:

```
window.location.href = "https://example.com"
```

Het ophalen van de location hash kan als volgt:

```
val hash = window.location.hash
```

Het decoderen van base64 strings kan op de volgende wijze:

```
val decodedString = window.atob("SGVsbG8gV29ybGQh") // Hello World!
```

Door het gebruik van bovenstaande functionaliteiten is het mogelijk om binnen Kotlin OAuth toe te passen.

7.2.3 React & Redux

Dit onderdeel heeft betrekking tot de volgende criteria:

- Mogelijkheid tot het maken van een SPA
- Mogelijkheid tot routing met history API (middels react-router)
- Mogelijkheid tot het definiëren van herbruikbare componenten
- Mapping tussen state en view
- Ondersteuning voor bestaande JavaScript libraries

De combinatie van React en Redux maakt het mogelijk om een SPA te maken waarbij er een mapping zit tussen state en view, wat wil zeggen dat de view (in dit geval de DOM) automatisch update wanneer de state wijzigt.

7.2.3.1 Project opzet

Om een volwaardig react, redux project op te zetten moeten de bijbehorende npm libraries (zie Bijlage 11 – Kotlin React Redux NPM libraries) worden ingeladen. Deze kunnen middels de kotlin-frontend-plugin worden ingeladen (welke in het hoofdstuk Axios is behandeld). Zoals eerder gezegd moet je external definitions schrijven om type-safe binnen Kotlin gebruik te maken van JavaScript libraries. Gelukkig is er een github repository (https://github.com/JetBrains/kotlin-wrappers) met Kotlin wrappers voor populaire JavaScript libraries. Deze wrappers kunnen middels gradle als dependencies worden ingeladen (zie Bijlage 12 – Kotlin React Redux wrappers). Om de wrappers op te kunnen halen moet de volgende repository aan de build.gradle worden toegevoegd:

```
maven { url "https://kotlin.bintray.com/kotlin-js-wrappers" }
```

7.2.3.2 React

Binnen JavaScript worden React componenten middels JSX geschreven, binnen Kotlin maak je gebruik van type-safe builders. Dit ziet er als volgt:

```
fun RBuilder.hello = h1 { +"Hello World!" }
```

Je kan HTML elementen ook nesten, wanneer je de + operator met een string gebruikt wordt de string aan de childList van het HTML element toegevoegd.

De component kan vervolgens op de volgende wijze binnen andere componenten worden gebruikt:

```
fun RBuilder.anotherComponent() =
    div {
        hello()
    }
```

Het definiëren van een class component gaat als volgt:

```
interface HelloProps : RProps {
    var name: String
}

class Hello(props: HelloProps) : RComponent<HelloProps, RState>(props) {
    override fun RBuilder.render() {
        h1 { +"Hello ${props.name}!" }
    }
}

fun RBuilder.hello(name: String = "World") = child(Hello::class) {
    attrs.name = name
}
```

De component kan vervolgens op dezelfde wijze als bovenstaande voorbeeld binnen andere componenten worden aangeroepen.

Je kan op de volgende wijze een component met een state definiëren (alhoewel het doorgaans aangeraden is om Redux te gebruiken voor het beheren van je state, het voorbeeld is meegegeven voor de volledigheid van React):

```
interface HelloProps : RProps {
    var name: String
}

interface HelloState : RState {
    var name: String
}

class Hello(props: HelloProps) : RComponent<HelloProps, HelloState>(props) {
    override fun HelloState.init(props: HelloProps) {
        name = props.name
    }

    override fun RBuilder.render() {
        h1 {
            +"Hello ${state.name}!"
            attrs.onClickFunction = {
                setState { name = name.toUpperCase() }
            }
        }
    }
}

fun RBuilder.hello(name: String = "World") = child(Hello::class) {
    attrs.name = name
}
```

Zoals je ziet kan je door de init function van de state te overriden de initiële state van de component definiëren, de state is te wijzigen middels de setState function.

Het is uiteraard ook mogelijk om children mee te geven/op te halen.

Bij het gebruik van een function component:

```
fun RBuilder.hello(children: RBuilder.() -> Unit) =
    h1 {
        children()
    }
```

Bij het gebruik van een class component:

```
class Hello : RComponent<RProps, RState>() {
    override fun RBuilder.render() {
        h1{
            children()
        }
    }
}

fun RBuilder.hello(children: RBuilder.() -> Unit) = child(Hello::class) {
    children()
}
```

children kan vervolgens in beide gevallen als volgt worden meegegeven:

```
hello {
    + "Hello World!"
}
```

React kan op de volgende wijze in de root DOM node worden ingeladen:

```
fun main() {
    val rootDiv = document.getElementById("root")
    render(rootDiv) {
        hello()
    }
}
```

7.2.3.3 Redux

Het gebruik van Redux is binnen Kotlin vergelijkbaar met JavaScript.

7.2.3.3.1 Actions

Binnen Kotlin kan men een action als volgt definiëren (zie Bijlage 13 – JavaScript Redux action, om te zien hoe dit vergelijkt met JavaScript):

```
class ToggleTodo(val id: Int): RAction
```

Als je eerder met Redux hebt gewerkt zul je je nu wellicht afvragen waarom er in de Kotlin code geen action type is opgegeven; de class type is in dit geval het type (dit zal in het hoofdstuk Reducers hieronder nog duidelijker worden), onderwater zet de Redux Kotlin wrapper de class natuurlijk om tot een JavaScript object welke door Redux gebruikt kan worden.

7.2.3.3.2 Reducers

Binnen Kotlin kan men een reducer als volgt definieren (zie Bijlage 14 – JavaScript Redux reducer, om te zien hoe dit vergelijkt met JavaScript):

```
fun todos(state: Array<Todo> = emptyArray(), action: RAction): Array<Todo> = when
(action) {
    is AddTodo -> state + Todo(action.id, action.text, false)
    is ToggleTodo -> state.map {
        if (it.id == action.id) {
            it.copy(completed = !it.completed)
        } else {
            it
        }
    }.toTypedArray()
    else -> state
}
```

Zoals je ziet wordt in de reducer de action type achterhaald door middels de “is” operator naar de class type te kijken.

BELANGRIJK: Gebruik geen lists in de state aangezien de wrapper lists zonder Kotlin’s type systeem omzet tot JavaScript arrays bij het combineren van reducers, gebruik in de state dus altijd arrays in plaats van lists wanneer je van plan bent om reducers te combineren (Heller, 2018).

Het combineren van reducers gaat binnen Kotlin (normaal gesproken) als volgt:

```
data class State(
    val todos: Array<Todo> = emptyArray(),
    val visibilityFilter: VisibilityFilter = VisibilityFilter.SHOW_ALL
)

fun combinedReducers() = combineReducers<State, RAction>(
    mapOf(
        "todos" to ::todos,
        "visibilityFilter" to ::visibilityFilter
    )
)
```

Je geeft dus een map mee met als key een string bestaande uit de naam van de state property en als value de reducer function.

Hier heb ik echter de volgende helper function voor geschreven:

```
fun <S, A, R> combineReducers(reducers: Map<KProperty1<S, R>, Reducer<*, A>>):
Reducer<S, A> {
    return combineReducers(reducers.mapKeys { it.key.name })
}
```

De map van deze function heeft als key een property van de state, dit heeft twee voordelen:

- Het is minder fout-gevoelig, wanneer je de naam van je property wijzigt, compileert je code niet meer als je de bijbehorende key niet wijzigt aangezien je dan refereert naar een property die niet meer bestaat.
- De compiler kan nu ook het type van de S type-parameter afleiden waardoor je niet meer expliciet de twee type-parameters hoeft mee te geven.

Bovenstaande combinedReducers function kan nu dus als volgt worden geschreven (de State data class is hetzelfde gebleven):

```
fun combinedReducers() = combineReducers(
    mapOf(
        State::todos to ::todos,
        State::visibilityFilter to ::visibilityFilter
    )
)
```

7.2.3.3.3 Store

Het aanmaken van de store gaat middels de createStore function:

```
val store = createStore<State, RAction, dynamic>(
    combinedReducers(), State(), compose(
        rEnhancer(),
        js("if(window.__REDUX_DEVTOOLS_EXTENSION__
)window.__REDUX_DEVTOOLS_EXTENSION__ ();else(function(f){return f;});")
    )
)
```

De eerste parameter is de root reducer, de tweede is de initiële state en de derde parameter de enhancer(s). In het geval van meerdere enhancers zet je de enhancers in de compose function, de rEnhancer maakt het mogelijk om Kotlin met Redux te gebruiken (deze zorgt er onder andere voor dat je action class wordt omgezet tot een een object welke Redux kan verwerken), de js regel maakt het mogelijk om de Redux devtools browser extension te gebruiken.

(Binnen de js function kan je JavaScript code inlinen.)

7.2.3.4 React-Redux

Het verbinden van de Redux state met React components gaat enigszins op dezelfde wijze als met JavaScript.

Hieronder de Kotlin equivalent van de react-redux example project (reduxjs, z.d.) connect onderdelen tussen de Link component en de FilterLink container.

Link component (zie Bijlage 15 – JavaScript React-Redux Link.js, om te zien hoe dit vergelijkt met JavaScript).

```
// components/Link.kt

interface LinkProps : RProps {
    var active: Boolean
    var onClick: () -> Unit
}

class Link(props: LinkProps) : RComponent<LinkProps, RState>(props) {
    override fun RBuilder.render() {
        styledButton {
            attrs.onClickFunction = { props.onClick() }
            attrs.disabled = props.active
            css {
                marginLeft = 4.px
            }
            children()
        }
    }
}
```

FilterLink container (zie Bijlage 16 – JavaScript React-Redux FilterLink.js, om te zien hoe dit vergelijkt met JavaScript).

```
// containers/FilterLink.kt

interface FilterLinkProps : RProps {
    var filter: VisibilityFilter
}

private interface LinkStateProps : RProps {
    var active: Boolean
}

private interface LinkDispatchProps : RProps {
    var onClick: () -> Unit
}

val filterLink: RClass<FilterLinkProps> =
    rConnect<State, SetVisibilityFilter, WrapperAction, FilterLinkProps,
LinkStateProps, LinkDispatchProps, LinkProps>(
        { state, ownProps ->
            active = state.visibilityFilter == ownProps.filter
        },
        { dispatch, ownProps ->
            onClick = { dispatch(SetVisibilityFilter(ownProps.filter)) }
        }

    )(Link::class.js.unsafeCast<RClass<LinkProps>>())
```

Zoals je ziet heeft de rConnect function 7 type-parameters en 2 lambda parameters.

Hieronder een toelichting op de type-parameters:

- State – Dit is de root State class.
- SetVisibilityFilter – Dit is het type action die deze container kan dispatchen, maak gebruik van inheritance om meerdere action types te ondersteunen (of vul hier RAction in).
- WrapperAction – Interface welke wordt gebruikt om de action te wrappen tot een object waar Redux mee om kan gaan, je kan hier in principe altijd WrapperAction invullen.
- FilterLinkProps – De props van de *container*, dus in dit geval de props van FilterLink.
- LinkStateProps – De props van de te verbinden component (dus in dit geval Link) welke aan de State moeten worden gemapped. *
- LinkDispatchProps - De props van de te verbinden component (dus in dit geval Link) welke aan dispatch moeten worden gemapped. *
- LinkProps – De props van de te verbinden component (dus in dit geval Link).

* De combinatie van deze twee interfaces vormen meestal de props van de te verbinden component, zoals je ziet heeft LinkProps de properties “active” en “onClick” welke ook in LinkStateProps en LinkDispatchProps zitten.

De eerste lambda parameter is het equivalent van mapStateToProps en de tweede lambda parameter is het equivalent van mapDispatchToProps.

mapStateToProps:

```
{ state, ownProps ->
    active = state.visibilityFilter == ownProps.filter
}
```

- this – Refereert naar LinkStateProps.
- state – Refereert naar State.
- ownProps – Refereert naar FilterLinkProps.

mapDispatchToProps:

```
{ dispatch, ownProps ->
    onClick = { dispatch(SetVisibilityFilter(ownProps.filter)) }
}
```

- this – Refereert naar LinkDispatchProps.
- dispatch – Dispatcher welke gebruikt kan worden om actions van het meegegeven van action type (in dit geval SetVisibilityFilter) te dispatchen.
- ownProps – Refereert naar FilterLinkProps.

De aangemaakte store (zie Store) kan in de root render function middels de provider worden meegegeven:

```
fun main() {
    val rootDiv = document.getElementById("root")
    render(rootDiv) {
        provider(store) {
            app()
        }
    }
}
```

7.2.3.5 React-Router

De react-router Kotlin wrapper biedt ondersteuning voor BrowserRouter en HashRouter. De router is gebouwd d.m.v. de type-safe builder welke gebruikt wordt om componenten te definiëren waardoor je op een gelijkmatige wijze als in JavaScript routes kan definiëren.

```
fun RBuilder.app() =
    hashRouter { // of browserRouter
        switch {
            route("/", Foo::class)
            route("/helloworld") {
                h1 {
                    +"Hello World!"
                }
            }
        }
    }
```

Zoals je ziet kan je de inhoud van de route zowel middels een class component als door het gebruik van de type-safe builder definiëren.

7.3 Code delen tussen frontend en backend (multiplatform project)

Dit hoofdstuk heeft betrekking tot het delen van code tussen de frontend (JS) en backend (JVM) binnen Kotlin (een multiplatform project). Voor dit onderdeel werden de volgende criteria opgesteld:

- Model classes tussen de frontend en de backend kunnen delen.
- Interfaces voor REST-services tussen de frontend en de backend kunnen delen.
- Validatie regels tussen de frontend en de backend kunnen delen.

7.3.1 Een multiplatform project

Om een multiplatform op te zetten zullen onderstaande stappen ondernomen moeten worden.

De multiplatform gradle plugin moet worden toegevoegd:

```
id 'org.jetbrains.kotlin.multiplatform' version '1.3.21'
```

Vervolgens moet de kotlin stdlib (en indien van toepassing overige dependencies) per platform (common, JS en JVM) als dependency worden toegevoegd, dit gaat binnen de build.gradle op de volgende wijze:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation "org.jetbrains.kotlin:kotlin-stdlib-common"
                // overige common dependencies hier
            }
        }
        jvmMain {
            dependencies {
                implementation "org.jetbrains.kotlin:kotlin-stdlib"
                // overige JVM dependencies hier
            }
        }
        jsMain {
            dependencies {
                implementation "org.jetbrains.kotlin:kotlin-stdlib-js"
                // overige JS dependencies hier
            }
        }
    }
}
```

Tot slot moet er in de src directory de volgende folder structuur worden aangemaakt (resources is optioneel):

```
src
│
├───commonMain
│   ├───kotlin
│   └───resources
│
├───jsMain
│   ├───kotlin
│   └───resources
│
└───jvmMain
    ├───kotlin
    └───resources
```

Nu kan je binnen elk onderdeel in de kotlin folder de desbetreffende packages/Kotlin bestanden aanmaken.

7.3.1.1 Hoe werkt het?

Code welke in het common onderdeel is geschreven kan door de andere platformen worden aangeroepen. Het is dan net of alles wat in het common onderdeel is geschreven ook in de andere platformen zit, (dit zie je ook terug in de import statements). Code in het common onderdeel wordt dus per platform voor het desbetreffende platform gecompileerd (Zie Bijlage 17 – Multiplatform project (JVM & JS) compilation diagram). Wanneer je binnen platform specifieke code in een package iets definieert wat al in de common code binnen dezelfde package is gedefinieerd krijg je dan ook een compile error. Je kan binnen de common code geen platform specifieke code aanroepen, enkel wat in de common stdlib of libraries die daarop zijn gebouwd zit.

7.3.2 Model classes delen

Je kan zonder moeite model classes tussen de frontend en de backend delen door deze binnen de common code te definiëren, deze zijn dan binnen de platform specifieke code te benaderen alsof deze ook daarbinnen zijn gedefinieerd.

7.3.3 Interfaces voor REST-services tussen de frontend en de backend delen

Het doel van dit onderdeel is om te kijken in hoeverre op het gebied van REST-services code gedeeld kan worden tussen de frontend en backend om daarbij de server kant en de client kant zo goed mogelijk op elkaar aan te sluiten. Op de backend is er gebruik gemaakt van JAX-RS en op de frontend de ktor client.

7.3.3.1 Paths

De paden naar de endpoints kunnen in de common code worden gedefinieerd waardoor je bij het configureren van de endpoints op de frontend en de backend van dezelfde constants gebruik kan maken en deze altijd op elkaar aansluiten.

Dit ziet er bijvoorbeeld als volgt uit:

```
object PersonPaths {
    const val ROOT = "/person"
    const val GET_BY_ID = "/{id}"
    const val RESULTS_LIST_PATH = "/resultslist"

    fun getByIdPath(id: Long) = "/$id"
}
```

Common code waarbij de paden voor de person endpoint zijn gedefinieerd.

```
@Path(PersonPaths.ROOT)
@Produces(MediaType.APPLICATION_JSON)
class PersonEndpoint : Endpoint() {

    @GET
    fun getAll(): List<PersonDTO> = openAndCloseSession {
        val genericDaoImpl = GenericDaoImpl<Person, Long>(Person::class.java, it)
        genericDaoImpl.loadAll()
    }.map { it.dto }

}
```

JAX-RS endpoint waarbij de path uit de common code wordt gehaald.

Op het JavaScript heb ik een abstract Endpoint class gemaakt waaraan je een root path aan meegeeft, wanneer een endpoint deze class extend is het mogelijk om gemakkelijk HTTP requests te configureren. Deze class ziet er als volgt uit:

```
actual abstract class Endpoint(rootPath: String) {
    private val apiPath = "$API_PATH$rootPath"

    protected fun HttpRequestBuilder.setPath(path: String? = null, query:
List<Pair<String, String?>> = listOf()) {
        url {
            var url = apiPath

            path?.let { url += it }

            if (query.isNotEmpty()) {
                url += "?${query.formUrlEncode()}"
            }
            encodedPath = url
        }
    }
}
```

Zoals je ziet wordt op basis van de rootPath en de api path een path voor de endpoint gecreëerd, deze wordt in de HTTP-client geconfigureerd wanneer de setPath function wordt aangeroepen, waar eventueel ook een extra path wat achter de rootPath moet komen kan worden meegegeven. Een voorbeeld waar dit gebruikt wordt is te zien in het hoofdstuk REST-services afstemmen hieronder.

Wanneer men een path binnen PersonPaths wijzigt zal de wijziging dus op zowel de frontend als de backend worden doorgevoerd.

7.3.3.2 REST-services afstemmen

Dit onderdeel heeft betrekking tot het afstemmen van de REST-services tussen de frontend en de backend, wat hiermee wordt bedoeld is het volgende: Afdwingen dat een endpoint die op de frontend wordt aangeroepen ook daadwerkelijk op de backend is gedefinieerd *en* dat bijbehorende functies op zowel de frontend als de backend dezelfde parameters verwachten en dezelfde class returnen. Dit is bereikt door gebruik te maken van de *expect* en *actual* functionaliteit van Kotlin multiplatform projecten.

De expect en actual functionaliteit heeft de volgende functie: je schrijft in de common code *definities* (dit kan alles zijn, een functie, een class, een annotatie, etc) en markeert deze met de expect keyword, dit geeft aan dat *elke platform* (behalve common) zelf de *implementatie* hiervan levert. Een andere optie is om in de common code voor elke endpoint een interface te maken welke je op de frontend en de backend implementeerd, het voordeel van de expect/actual methode is echter dat de compiler je *verplicht* om de expect classes (en overige expect declaraties) op iedere platformen te implementeren.

Hieronder een voorbeeld.

Common code:

```
expect class PersonEndpoint : Endpoint {
    suspend fun getById(id: Long): PersonDTO
    suspend fun getAll(): List<PersonDTO>
    suspend fun getAllResultsList(): ResultsList<PersonDTO>
    suspend fun create(personDTO: PersonDTO): Long
}
```

JVM:

```
@Path(PersonPaths.ROOT)
@Produces(MediaType.APPLICATION_JSON)
actual class PersonEndpoint : Endpoint() {
    @GET
    @Path(PersonPaths.GET_BY_ID)
    actual suspend fun getById(@PathParam("id") id: Long): PersonDTO =
openAndCloseSession {
        val genericDaoImpl = GenericDaoImpl<Person, Long>(Person::class.java, it)
        genericDaoImpl.load(id)
    }?.dto ?: throw WebApplicationException(Response.Status.NOT_FOUND)

    @GET
    actual suspend fun getAll(): List<PersonDTO> = openAndCloseSession {
        val genericDaoImpl = GenericDaoImpl<Person, Long>(Person::class.java, it)
        genericDaoImpl.loadAll()
    }.map { it.dto }

    @GET
    @Path(PersonPaths.RESULTS_LIST_PATH)
    actual suspend fun getAllResultsList(): ResultsList<PersonDTO> =
ResultsList(openAndCloseSession {
        val genericDaoImpl = GenericDaoImpl<Person, Long>(Person::class.java, it)
        genericDaoImpl.loadAll()
    }.map { it.dto })


    @POST
    @Status(201)
    actual suspend fun create(personDTO: PersonDTO): Long = openAndCloseSession {
        val genericDaoImpl = GenericDaoImpl<Person, Long>(Person::class.java, it)
        genericDaoImpl.save(personDTO.entity)
    }
}
```

JavaScript:

```
actual class PersonEndpoint : Endpoint(PersonPaths.ROOT) {
    actual suspend fun getById(id: Long): PersonDTO = client.get {
        setPath(PersonPaths.getByIdPath(id))
    }

    actual suspend fun getAll(): List<PersonDTO> = client.list {
        setPath()
    }

    actual suspend fun getAllResultsList(): ResultsList<PersonDTO> =
client.resultsList {
        setPath(PersonPaths.RESULTS_LIST_PATH)
    }

    actual suspend fun create(personDTO: PersonDTO): Long = client.post {
        setPath()
        json(personDTO)
    }
}
```

Zoals je ziet hebben de implementaties op de JVM en JavaScript dezelfde benaming/parameters/return type als in de common code. Dus je weet zeker dat de class die door de endpoint op de backend wordt ge-returned ook door de HTTP-client op de frontend wordt ge-returned. Verder heeft dit als voordeel dat zodra je de methode naam/parameters/return type op een plek wijzigt je een compiler error krijgt omdat de endpoints niet overal op elkaar aansluiten, waarmee je dus runtime fouten kan voorkomen.

Zoals je ziet zijn de functies als suspend functies gemarkeerd, dit komt omdat een request vanuit de frontend asynchroon is en de HTTP-client gebruik maakt van Kotlin's coroutines (dit zijn asynchroon functionaliteiten binnen Kotlin).

Omdat de functions in de expect common code als suspend functions zijn gemarkeerd moeten deze op de JVM platform ook als suspend functions worden gemarkeerd, dit zorgt echter wel voor complicaties. Wanneer de code voor de JVM eenmaal gecompileerd is, wordt aan elke suspend function een parameter van het type Continuation toegevoegd, als voorbeeld de getById function:

```
public final Object getById(@PathParam("id") final long id, @NotNull Continuation
$completion)
```

Het probleem hierbij is het feit dat JAX-RS de Continuation parameter als body parameter ziet en deze gaat proberen te deserializen wat zal leiden tot een server error. Als workaround heb ik een JAX-RS MessageBodyReader geschreven die deze parameter negeert (zie Bijlage 18 – JAX-RS suspend keyword workaround).

Verder zie je dat de create function een @Status annotatie heeft, de reden hiervoor is dat je binnen JAX-RS doorgaans een Response object returned waarin je ook je HTTP response status code kan aanpassen. Binnen mijn code wordt er geen gebruik gemaakt van de Response object maar wordt de daadwerkelijke body ge-returned (om zo de frontend en de backend op elkaar aan te sluiten). De @Status annotatie geeft aan welke status de response moet krijgen indien de request succesvol verloopt, dit wordt op de volgende wijze mogelijk gemaakt:

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.TYPE)
annotation class Status(val statusCode: Int)
```

De annotatie zelf.

```
@Provider
class StatusFilter : ContainerResponseFilter {

    @Throws(IOException::class)
    override fun filter(
        containerRequestContext: ContainerRequestContext,
        containerResponseContext: ContainerResponseContext
    ) {
        if (containerResponseContext.status == 200) {

containerResponseContext.entityAnnotations?.filterIsInstance<Status>()?.firstOrNu
ll()?.let { containerResponseContext.status = it.statusCode }
        }
    }
}
```

JAX-RS filter die de status code wijzigt indien de request succesvol is verlopen en deze de @Status annotatie bevat.

7.3.4 Validatie

Voor validatie heb ik slechts 1 library kunnen vinden die multiplatform projecten ondersteunt, namelijk de Konform library. Deze library maakt het mogelijk om middels typesafe builders validatie regels voor classes te schrijven. Hieronder een voorbeeld uit de website van Konform (Lochschmidt, z.d.):

```
data class UserProfile(
    val fullName: String,
    val age: Int?
)
```

Een UserProfile data class.

```
val validateUser = Validation<UserProfile> {
    UserProfile::fullName {
        minLength(2)
        maxLength(100)
    }

    UserProfile::age ifPresent {
        minimum(0)
        maximum(150)
    }
}
```

De validatie regels voor UserProfile welke je als function kan aanroepen waarbij een UserProfile als parameter meegeeft, dit zal een Valid of Invalid object returnen.

```
val invalidUser = UserProfile("A", -1)
val validationResult = validateUser(invalidUser)
```

Het aanroepen van de validator met invalide data, validationResult is in dit geval een Invalid object met daarin de errors.

```
validationResult[UserProfile::fullName]
// returned listOf("must be at least 2 characters")

validationResult[UserProfile::age]
// returned listOf("must be equal or greater than 0")
```

Het ophalen van specifieke errors door middel van de class property.

Ik heb een Validateable interface gemaakt welke aangeeft dat een class te valideren is, deze ziet er als volgt uit:

```
interface Validateable<T> {
    fun validate(): ValidationResult<T>
    fun validateCreate(): ValidationResult<T>? = null
    fun validateUpdate(): ValidationResult<T>? = null
}
```

Zoals je ziet moet je een validate function implementeren en optioneel validateCreate en validateUpdate functions implementeren (omdat het mogelijk is dat je bij POST en PUT requests andere regels wil toepassen). Een voorbeeld waarbij deze interface is geïmplementeerd:

```
data class PersonDTO(val id: Long? = null, val name: String, var age: Int) :
    Validateable<PersonDTO> {
    override fun validate() = validator(this)
    override fun validateCreate() = createValidator(this)
    override fun validateUpdate() = updateValidator(this)
}

object PersonDTOValidator {
    val validator = Validation<PersonDTO> {
        PersonDTO::age{
            minimum(0)
            maximum(200)
        }
        PersonDTO::name{
            notBlank()
            minLength(2)
        }
    }
    val createValidator = Validation<PersonDTO> {
        PersonDTO::id {
            isNull()
        }
        run(validator)
    }
    val updateValidator = Validation<PersonDTO> {
        PersonDTO::id required {
            minimum(1)
        }
        run(validator)
    }
}
```

Zoals je ziet zijn de validators zelf binnen een object gedefinieerd. Dit is gedaan zodat deze niet voor elke instance van de class opnieuw worden aangemaakt. Hier is ook te zien dat validators zijn te hergebruiken middels de run function, de createValidator en updateValidator hebben dus alle regels van de validator plus hun eigen regels. isNull en notBlank zijn custom validatie regels die ik zelf heb geschreven, dit kan op de volgende wijze:

```
fun ValidationBuilder<String>.notBlank() = addConstraint(
    "may not be empty"
) { it.isNotBlank() }
```

Je schrijft een extension function voor ValidationBuilder welke addConstraint aanroept waar je een error message en een lambda welke een boolean returned aan meegeeft (it binnen de lambda verwijst naar de property waar de validatie regel op wordt toegepast).

De reden waarom ik ervoor heb gekozen om een Validateable interface te gebruiken is zodat op de backend requests automatisch gevalideerd kunnen worden voordat deze in de betreffende request function terechtkomen, je hoeft in de request function dus niet handmatig te valideren. Ik heb dit gedaan door een JAX-RS Interceptor te schrijven (zie Bijlage 19 – JAX-RS validation interceptor).

De interceptor controleert eerst of de deserialized body-object de Validateable interface implementeert, zo niet wordt de body ge-returned en zal de request verder worden afgehandeld door de desbetreffende request function. Indien de body wel Validateable implementeert, wordt op basis van de request HTTP method en de geïmplementeerde validators gekeken welke validator gebruikt moet worden en wordt deze toegepast. Indien het object valide is wordt deze ge-returned en wordt de request verder afgehandeld door de desbetreffende request function. Indien het object niet valide is wordt er een HTTP-response gestuurd met status code 422 (Unprocessable Entity) met als body de errors uit de invalid object.

Binnen de JavaScript code is de validator ook aan te roepen (in mijn proof of concept wordt deze binnen een formulier binnen de onChange van input velden en na het submitten aangeroepen), het is dus mogelijk om op 1 plek validatie regels te definiëren (in de common code) en deze op de verschillende platformen toe te passen.

7.4 Code testen

Dit onderdeel heeft betrekking tot het testen van APIs (dit was het test type waar Topicus geïnteresseerd in was). Men kan gebruik maken van de volgende tools om tests te schrijven:

- JUnit5 – Test framework voor de JVM.
- Mockk – (o.b.v. GitHub stars) Meest populaire mocking library voor Kotlin. (GitHub, z.d.)
- UndertowJaxrsServer – Embedded container voor de rest API.
- REST-assured – Library voor het aanroepen/testen van de endpoints.

7.4.1 Mocks

De Mockk library maakt het mogelijk om middels een DSL mocks op te stellen, hieronder een aantal voorbeelden.

Variable mock:

```
private val session: Session = mockk()
```

Constructor mock:

```
mockkConstructor(GenericDaoImpl::class)
```

Function mock waarbij het eerste argument wordt gebruikt d.m.v. any() en firstArg():

```
every { anyConstructed<GenericDaoImpl<Person, Long>>().load(any()) } answers {
persons[firstArg()] }
```

Static function mock:

```
mockkStatic("nl.lawik.poc.test.util.HibernateUtilKt")
every { openAndCloseSession<Any?>(captureLambda()) } answers { lambda<(Session) -
> Any?>().invoke(session) }
```

7.4.2 JAX-RS server

Ik heb een wrapper class voor UndertowJaxrsServer geschreven waar je de te testen resource class aan meegeeft en de server geconfigureerd wordt wanneer de deploy() function wordt aangeroepen:

```
class Server(val resource: KClass<*>) : UndertowJaxrsServer() {

    @ApplicationPath(API_PATH)
    private inner class App : Application() {
        override fun getClasses(): MutableSet<Class<*>> {
            return mutableSetOf(resource.java,
ContinuationMessageBodyReader::class.java,
                CorsFilter::class.java, JSONConsumer::class.java,
                MismatchedInputExceptionMapper::class.java,
                MissingKotlinParameterExceptionMapper::class.java,
                StatusFilter::class.java, ValidationInterceptor::class.java)
        }
    }

    fun deploy() {
        this.deploy(App())
    }
}
```

7.4.3 Het schrijven van tests

De tests zijn per endpoint te groeperen, dit wordt mogelijk gemaakt door binnen de root test class een inner class voor elke endpoint te maken en deze te annoteren met de @Nested annotatie. Binnen deze inner class kan je vervolgens de relevante test functions met daarbinnen je assertions schrijven en de functions annoteren met de @Test annotatie.

De root test class is te annoteren met de @TestInstance(TestInstance.Lifecycle.PER_CLASS) annotatie, dit zorgt er voor dat de root test class eenmaal wordt gemaakt en voor elke test wordt hergebruikt. Het kan zijn dat je voor en na het uitvoeren van elke test function bepaalde acties wil uitvoeren, zoals het resetten van een bijgehouden state, dit kan door functions respectievelijk te annoteren met de @BeforeEach en @AfterEach annotaties.

Hieronder een voorbeeld van de tests voor de endpoint waarbij op basis van een id een PersonDTO wordt opgehaald:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class PersonEndpointTest {
    // ...
    @Nested
    inner class GetByIdTest {
        @Test
        fun `get by id 1 returns status code 200 and PersonDTO`() {
            val personDTO = get("$API_PATH${PersonPaths.ROOT}/1")
                .then()
                .statusCode(200)
                .extract().to<PersonDTO>()

            assertEquals(persons[1]?.dto, personDTO)
        }

        @Test
        fun `get by invalid id returns status code 404`() {
            get("$API_PATH${PersonPaths.ROOT}/-1")
                .then()
                .statusCode(404)
                .body(Matchers.isEmptyString())
        }
    }
    // ...
}
```

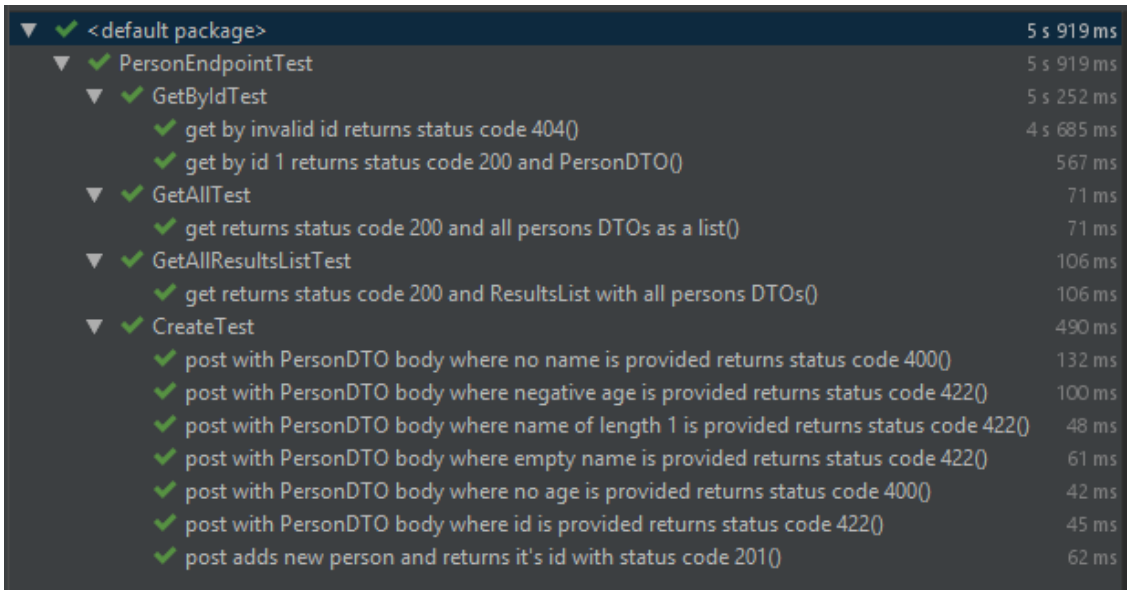
Zoals je ziet zijn beide tests die betrekking hebben op dezelfde endpoint in dezelfde inner class gegroepeerd, dit zorgt voor meer overzicht en leesbaarheid. Wat ook voor meer leesbaarheid zorgt is het feit dat je binnen Kotlin functie namen middels backticks kan definiëren en daarbij gebruik mag maken spaties. `get by id 1 returns status code 200 and PersonDTO` is leesbaarder dan getById1ReturnsStatusCode200AndPersonDTO.

```
val personDTO = get("$API_PATH${PersonPaths.ROOT}/1")
    .then()
    .statusCode(200)
    .extract().to<PersonDTO>()
```

Bovenstaande is mogelijk dankzij de REST-assured library, hier wordt een HTTP-get request naar het meegegeven path uitgevoerd en wordt vervolgens gecontroleerd of de status code 200 is (.statuscode(200) is dus een assertion), tot slot wordt de body middels de .extract().to() function deserialized tot een PersonDTO welke voor een assertEquals gebruikt wordt.
De to(…) function is een extension function, deze zit niet standaard in REST-assured.

```
inline fun <reified T> ResponseBodyExtractionOptions.to(): T {
    return this.jsonPath().getObject("", T::class.java)
}
```

Wanneer je de tests (bijvoorbeeld via IntelliJ) draait krijg je door het gebruik van inner classes en backticks overzichtelijke test resultaten, hieronder een voorbeeld waarin dit is te zien:



Figuur 1 – Test resultaten vanuit Intellij

8. Prototype

Nadat alle deelvragen waren beantwoord was het tijd om de hoofdvraag “Is het voor Topicus realistisch om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen?” te beantwoorden. Ik heb dit gedaan door met behulp van de vergaarde kennis uit de deelvragen een “real world (prototype) application” te ontwikkelen.

8.1 Doel

Het doel van de applicatie is om oefeningen m.b.t. rekenen met breuken aan leerlingen beschikbaar te stellen. De oefeningen worden door de leerkrachten voor diens groep(en) opgesteld. De resultaten van de leerlingen worden bijgehouden, de score is door de leerling in te zien en resultaten zijn door diens leerkracht(en) in te zien. De leerlingen en leerkrachten loggen in met hun eigen school gegevens (OAuth2), dit wordt mogelijk gemaakt door een koppeling met het Wise-r platform.

8.2 Terminologie

Term	Betekenis
Oefening	**Een set aan vragen.**
Vraag	**1 rekenvraag met m.b.t. breuken: optellen, aftrekken en gelijknamig maken.**

Tabel 6 – Prototype terminologie

8.3 Requirements

Voorafgaand de afstudeerperiode werden de requirements voor het prototype opgesteld, deze zijn nadat alle losse deelvragen waren beantwoord bijgewerkt. Zie het hoofdstuk Requirements voor de opgestelde requirements van het prototype.

8.4 Werkwijze

Nadat de requirements duidelijk waren heb ik een Functionele omschrijving en mockups gemaakt om meer duidelijkheid en inzicht te krijgen en om er zeker van te zijn dat ik en mij begeleiders op 1 lijn zijn m.b.t. de te ontwikkelen functionaliteiten. Nadat hier een akkoord op was heb ik de requirements vertaald naar userstories en deze middels de SCRUM methode (zie Scrum) uitgewerkt. Ik heb gewerkt met sprints van 1 week, op de eerste week na, dat was een sprint van een halve week aangezien ik op woensdag begon en onze wekelijkse voortgangsbespreking op vrijdag is.

8.5 Functionele omschrijving & mockups

Om de te ontwikkelen functionaliteit duidelijker te krijgen heb ik een functionele omschrijving voor ieder onderdeel opgesteld en om visueel inzichtelijk te krijgen wat precies op welke pagina komt heb ik voor iedere pagina mockups gemaakt (dit waren enkel mockups om de *inhoud* van de pagina inzichtelijk te maken, geen definitieve designdocumenten, het opgeleverde prototype ziet er qua design enigszins anders uit). Zie Bijlage 20 – Prototype functionele omschrijving & mockups voor de functionele omschrijving en de gemaakte mockups.

8.6 Technische omschrijving

8.6.1 Projectstructuur

De applicatie is volledig in Kotlin ontwikkeld, het project bestaat uit 3 onderdelen:

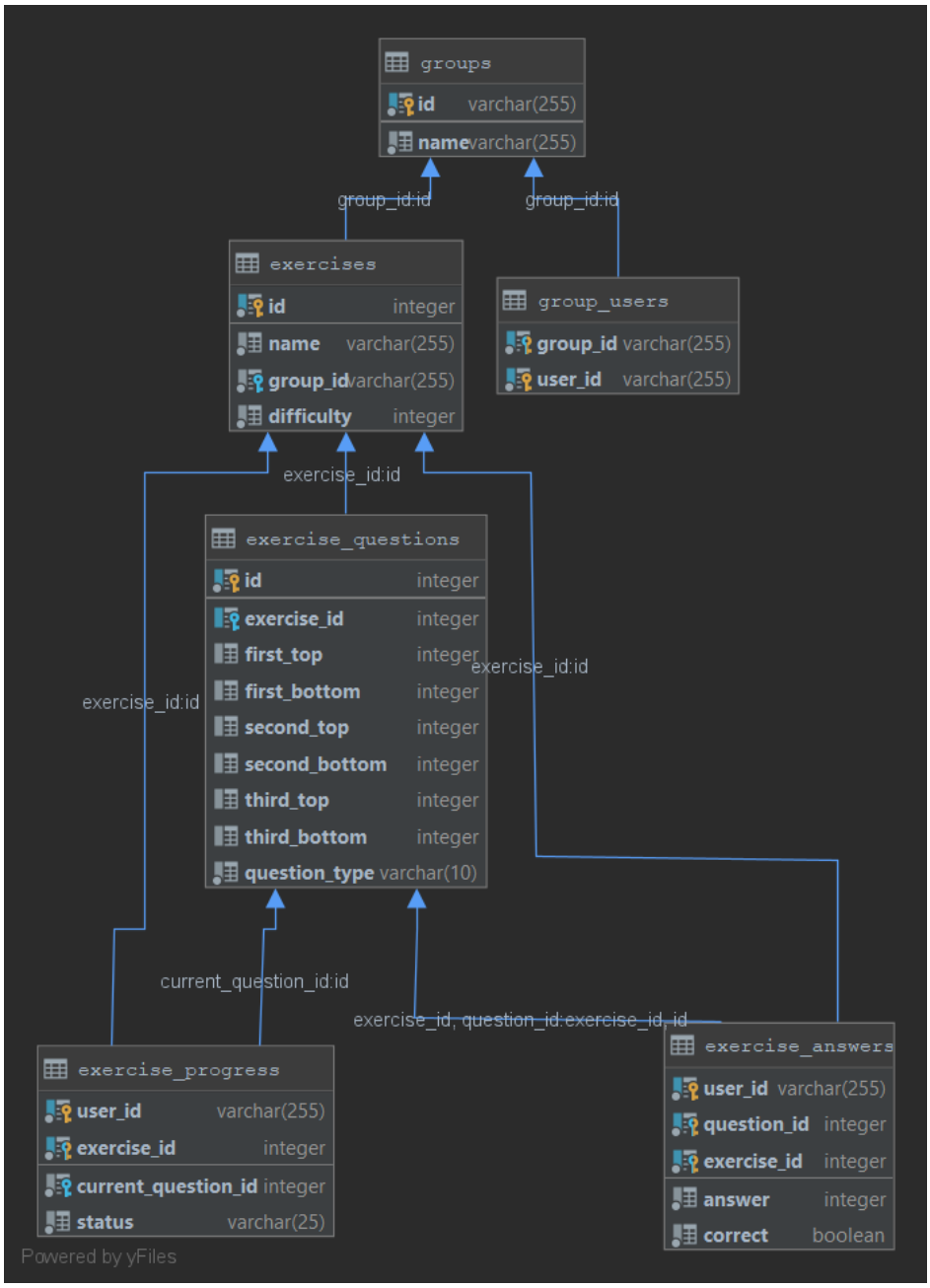
- Backend – Dit is de JVM backend, Kotlin code die hierin wordt geschreven wordt gecompileerd naar Java bytcode.
- Frontend – Dit is de JavaScript frontend, Kotlin code die hierin wordt geschreven wordt gecompileerd/transpiled naar JavaScript.
- Common – Dit het gedeelde onderdeel, Kotlin code die hierin wordt geschreven wordt *zowel* naar Java bytcode *als* JavaScript gecompileerd/transpiled. De JVM en JavaScript onderdelen kunnen van de hierin gedefinieerd code gebruik maken alsof deze in hun eigen onderdeel zijn gedefinieerd (zie Bijlage 17 – Multiplatform project (JVM & JS) compilation diagram).

Verder beschikt de applicatie (middels een verbinding op de backend) over een database. De database houdt de volgende informatie bij:

- Groepen
- Gebruikers binnen de groepen
- Oefeningen
- Vragen behorende tot de oefeningen
- De voortang van de oefening van leerlingen
- De door de leerlingen gegeven antwoorden op vragen van oefeningen

De database houdt alleen informatie bij wat daadwerkelijk nodig is voor de werking van de applicatie.

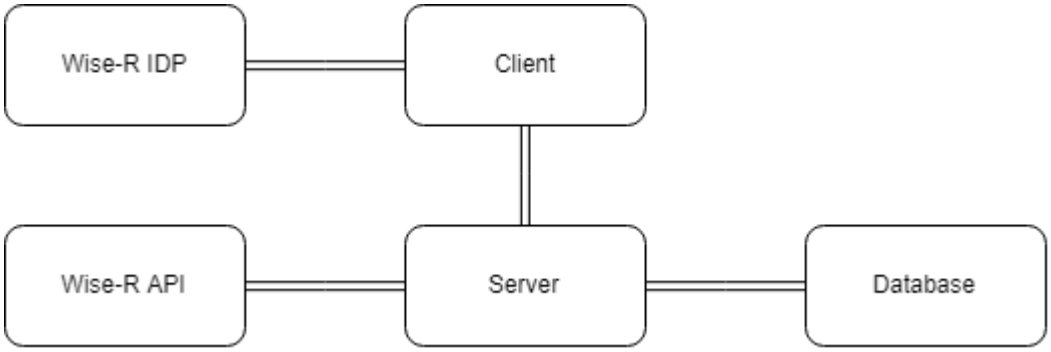
Zie het database diagram hieronder voor meer details.



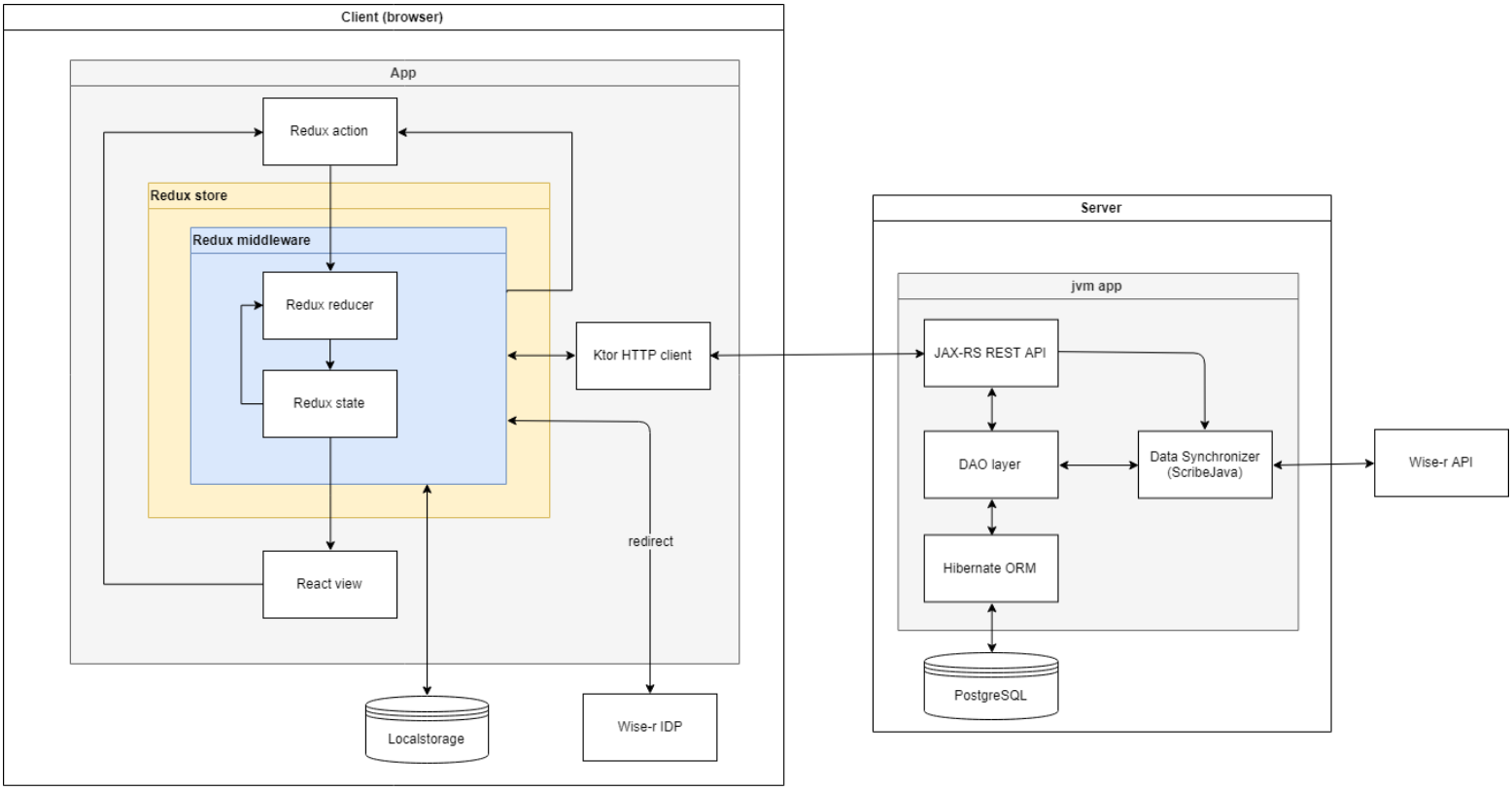
Figuur 2 – Prototype: Database diagram

Verder communiceert de client met de Wise-r IDP (Identity Provider) voor de authenticatie en communiceert de server met de Wise-r API om benodigde informatie van de gebruikers op te halen en te synchroniseren met de database.

Het communication diagram hieronder weergeeft welke onderdelen met elkaar communiceren. Het architecture diagram hieronder geeft een gedetailleerde weergave van de architectuur weer.



Figuur 4 – Prototype Communication diagram



Figuur 3 – Architecture diagram

Bovenstaande onderdelen zullen in de hoofdstukken hieronder worden toegelicht.

8.6.1.1 Backend

De backend is verantwoordelijk voor de volgende onderdelen:

- Authenticatie en Autorisatie afhandelen
- Data ontsluiting middels een JAX-RS REST-API
- Database afhandelingen middels Hibernate ORM
- Data uit Wise-r middels de Wise-r API (d.m.v. de ScribeJava OAuth library) synchroniseren met de database (dit betreft de groepen van leerkrachten en de bijbehorende leerlingen)

De backend heeft de volgende package structuur:

- dao – Hier zitten alle dao's in die communiceren met de database, per dao een package.
- endpoint – Hier zitten alle REST-endpoints in.
- entity – Hier zitten alle database entiteiten met ORM middels Hibernate.

- hibernate – Hier zitten hibernate utilities en de hibernate query DSL in.
- jaxrs – Hier zitten JAX-RS gerelateerde packages in.
    - annotations
    - consumers
    - filters
    - interceptors
    - messagebodyreaders
    - util

8.6.1.2 Frontend

De frontend is verantwoordelijk voor de functionele weergave van de applicatie aan de gebruikers. De frontend is opgebouwd middels React en Redux (en daarbij behorende libraries/wrappers). De state van de applicatie wordt middels redux bijgehouden en de views van de applicatie zijn middels React in (herbruikbare) componenten opgebouwd, er is een mapping tussen de Redux state en de React view waardoor state wijzigingen automatisch in de view worden doorgevoerd. De frontend communiceert middels de Ktor client met de REST-API welke op de backend draait. Deze client is gekozen omdat deze direct Kotlin ondersteuning biedt, automatisch responses deserialized en vele features biedt (zie het hoofdstuk HTTP-clients).

De frontend heeft de volgende package structuur:

- client – Hier zit de client (welke met de backend API communiceert) en de daarbij behorende utilities in.
- config – Hier zitten configuratie welke relevant zijn voor de frontend in, denk hierbij aan OAuth parameters, localstorage keys en de routes binnen de applicatie.
- endpoint – Hier zitten alle beschikbare REST-endpoints welke de frontend kan aanroepen.
- model – Hier zitten model classes in welke enkel voor de frontend relevant zijn.
- react – Hier zitten alle React gerelateerde bestanden in.
    - components – herbruikbare componenten, indien een component uit meerdere sub-componenten bestaat zitten de relevante bestanden voor de desbetreffende componenten in een eigen package.
    - containers – De pagina's binnen de applicatie zelf, (voornamelijk) opgebouwd uit de herbruikbare componenten.
- redux – Hier zitten alle Redux gerelateerde bestanden in.
    - action – Hier zitten alle redux actions in.
    - middleware – Hier zit custom Redux middleware in.
    - reducer – Hier zitten alle Redux reducers en de daarbijhorende state classes.
- util – Hier zitten algemene util functions in.

8.6.1.3 Common

Binnen common zit code waar zowel de frontend als de backend gebruik van maken. Er worden op 4 fronten code gedeeld tussen de frontend en de backend:

- Model classes (DTO's)
- Validatie voor de DTO's
- Paths naar de REST-endpoints
- (Door de compiler) afgedwongen structuren voor de REST-endpoints.

Common heeft de volgende package structuur:

- endpoint – Hier zitten de structuren voor de REST-endpoints en de paths naar de bijbehorende endpoints.
- model – Hier zitten alle models/DTO's en de bijbehorende validatie functions.
    - exercise – model/DTO classes voor oefeningen.
        - question – Model/DTO classes voor vragen.
    - group – Hier zit momenteel enkel een DTO voor groepen in.
    - jwt – Hier zit momenteel alleen een UserType enum in welke zowel door de frontend als de backend wordt gebruikt bij autorisatie.

Door de werking van Kotlin multiplatform projecten zit bovenstaande package structuur dus ook in de frontend en de backend. Als voorbeeld zie je hier de import statement voor de ExerciseDTO binnen ExerciseEndpoint op de backend (nl.lawik.prototype.endpoint.ExerciseEndpoint).

```
import nl.lawik.prototype.model.exercise.ExerciseDTO
```

Zoals je ziet staat hier niks over "common" o.i.d., in de ogen van de backend (en frontend) bestaat deze class simpelweg binnen zijn eigen packagestructuur ondanks dat deze buiten de backend om (in het common gedeelte) is gedefinieerd.

8.6.2 Authenticatie

Er is een OAuth koppeling met het Wise-r platform van Topicus Onderwijs waardoor de gebruikers middels hun schoolgegevens op de applicatie kunnen inloggen. Authenticatie gebeurt op de volgende happy flow (foutafhandelingen zitten tevens ook in de applicatie verwerkt):

1. De gebruiker opent de applicatie en ziet het login scherm.
2. De gebruiker klikt op de "Login" knop.
3. Er wordt een random state string gegenereerd welke in de localstorage wordt opgeslagen, de gebruiker wordt doorgestuurd naar de Wise-r omgeving waarbij de random state string wordt meegestuurd.
4. De gebruiker logt in en wordt terugverwezen naar de applicatie met een id token en een state string in de location hash (we gaan er in deze happy flow van uit dat de state in de location hash gelijk is met de state die in de localstorage staat opgeslagen, *er is echter ook error handling ontwikkeld!)*.
5. Er wordt een request naar de backend gestuurd met de id token als body.
6. De backend controleert middels de Wise-r public key de signature van de id token (we gaan er in deze happy flow van uit dat deze correct is, *er is echter ook error handling ontwikkeld!)*
7. De relevante claims worden uit de id token gehaald, dit betreft de volgende gebruikersgegevens: UUID, gebruikerstype (leerling/leerkracht), volledige naam en organisatie (school).
8. Er wordt een JWT voor de applicatie gemaakt/signed o.b.v. de opgehaalde claims uit de Wise-r id token.
9. De JWT wordt middels een response teruggestuurd naar de frontend waar deze de JWT zal opslaan in de localstorage/huidige state van de applicatie.
10. OPTIONEEL: Deze stap geldt enkel wanneer een *leerkracht* zich heeft geauthentiseerd, deze stap gebeurt *asynchroon op de backend* voordat de response naar de frontend wordt verstuurd, *de gebruiker hoeft hier dus niet op te wachten*.
    a. Er wordt middels de client id en client secret een nieuwe id token gemaakt voor de *Wise-r API* (dit is de API waarmee gegevens over leerling/leerkrachten/scholen/groepen/etc opgehaald kan worden).
    b. De groep-data van de leerkracht wordt opgehaald.

c. De relevante data wordt opgeslagen in de database (Groep ID, naam en bijbehorende gebruikers).

De reden waarom voor bovenstaande constructie is gekozen is omdat de id token welke Wise-r terugstuurt na 5 minuten verloopt, deze dient enkel om iemand te *authentiseren*. Dankzij bovenstaande constructie hoeft de gebruiker niet iedere 5 minuten opnieuw in te loggen en blijft het authenticatieproces veilig, aangezien de Wise-r id token op de backend wordt gevalideerd en wordt omgezet tot een nieuwe signed JWT (met langere geldigheidsduur) welke vanaf dat moment zal dienen voor authenticatie/autorisatie, wanneer deze verloopt zal bovenstaande flow opnieuw door de gebruiker doorlopen moeten worden.

8.6.3 Autorisatie

Autorisatie is een van de onderdelen waarbij code tussen de frontend en de backend wordt gedeeld.

8.6.3.1 Common

In de common code zit een enum welke de usertype omschrijft (de huidige usertype zit in de JWT). Deze enum ziet er als volgt uit:

```
enum class UserType {
    TEACHER,
    STUDENT
}
```

De enum is zowel door de frontend als de backend te gebruiken.

8.6.3.2 Backend

Om de autorisatie zonder veel boilerplate code op de backend mogelijk te maken is er een @Secured annotatie geschreven welke op JAX-RS endpoints (zowel classes als functions) kan worden toegepast (zie Bijlage 21 – @Secured annotatie). De @Secured annotatie heeft een varargs parameter van het type UserType, er kunnen dus 0..* UserTypes worden meegegeven.

Er is een JAX-RS filter geschreven welke de autorisatie voor requests afhandelt (zie Bijlage 22 – AuthorizationFilter), zonder deze filter doet de @Secured annotatie helemaal niks. De filter controleert of de authorization header van de request een *geldig* JWT bevat. Indien dit het geval is controleert de filter of de UserType in de JWT ook bij de endpoint mag (@Secured annotatie is leeg of bevat de UserType uit de JWT).

Indien bovenstaande allemaal goed gaat wordt de gebruiker toegevoegd aan de securitycontext van de JAX-RS request (zodat de request function de ID van de huidige geautoriseerde gebruiker kan ophalen) en wordt de bijbehorende request function uitgevoerd. Indien er iets verkeerd gaat (request bevat geen (geldig) JWT of type gebruiker mag niet bij de endpoint) wordt de request afgekapt en wordt er een response gestuurd met de status unauthorized (401) of forbidden (405), respectievelijk.

De reden waarom voor bovenstaande constructie is gekozen is zodat men niet per endpoint handmatig hoeft te controleren of de gebruiker geautoriseerd is maar dit automatisch wordt gedaan d.m.v. de @Secured annotatie en de bijbehorende JAX-RS filter.

8.6.3.3 Frontend

Op de frontend staat er autorisatie op de pagina's, leerlingen kunnen enkel bij pagina's bestemd voor leerlingen en leerkrachten kunnen enkel bij pagina's bestemd voor leerkrachten. Dit wordt mogelijk gemaakt door een zelfgemaakte PrivateRoute component, deze is voortgebouwd op de react-router Route component en heeft exact dezelfde API, op een ding na, je kan ook een array van UserType meegeven.

De PrivateRoute component controleert eerst of de huidige gebruiker geauthentiseerd is (is er een geldig JWT), zo niet dan wordt de gebruiker doorgestuurd naar de login pagina. Indien de gebruiker wel geauthentiseerd is, wordt er gecontroleerd of de gebruiker bij de route mag (is de opgegeven UserType array leeg of bevat deze de UserType van de huidige gebruiker), indien de gebruiker bij de pagina mag wordt de pagina weergegeven, indien de gebruiker niet bij de pagina mag wordt de gebruiker doorgestuurd naar de algemene homepagina waar hij/zij o.b.v. diens UserType zal worden doorgestuurd naar de homepagina die voor de gebruiker relevant is.

De reden waarom voor bovenstaande constructie is gekozen is zodat je op dezelfde manier als react-router routes kan definiëren met als toevoeging dat je routes kan beschermen tegen ongeautoriseerde gebruikers.

8.6.4 Validatie

Voor validatie is dezelfde strategie als in het hoofdstuk Validatie aangehouden. De backend en de frontend maken beiden gebruik van dezelfde validatie regels die in de common code voor de desbetreffende model classes zijn gedefinieerd. De JAX-RS filter op de backend valideert model classes welke de Validateable interface implementeren voordat deze in de desbetreffende endpoint functions terechtkomen. De frontend controleert bij het submitten van formulieren of de invoer valide is. Verder is er ook een validatie function geschreven welke het antwoord op een vraag berekent, deze wordt naast de automatische validatie, op de backend ook handmatig in een request function aangeroepen om te controleren of het gegeven antwoord van een leerling correct is.

8.6.5 Endpoints

Voor het definiëren van endpoints is dezelfde strategie als in het hoofdstuk Interfaces voor REST-services tussen de frontend en de backend delen aangehouden, de endpoints sluiten op de frontend en de backend op elkaar aan en de paths zijn op 1 plek gedefinieerd. Verder beschikken de endpoints op de backend over autorisatie beveiliging zoals in het hoofdstuk Autorisatie hierboven staat beschreven.

8.6.6 React

Op de frontend heb ik waar mogelijk componenten beperkt tot een kleine set (meestal 1) aan doeleinden. Dit zorgt ervoor dat de applicatie bestaat uit herbruikbare componenten welke op meerdere plekken van de applicatie kunnen worden gebruikt. Een mooi voorbeeld is de tabel component welke op meerdere plekken wordt gebruikt (2 maal op de overzichtspagina van de leerling en 1 maal op de beheerpagina van de leerkracht). Deze tabellen maken alle 3 gebruik van hetzelfde component wat code duplicatie voorkomt.

Dit component zorgt ervoor dat bijvoorbeeld de tabel op de beheerpagina (Zie Bijlage 20 – Prototype functionele omschrijving & mockups: Figuur 9 – Prototype: Leerkracht beheer overzicht mockup) als volgt kan worden gedefinieerd:

```
props.exercises.isNotEmpty() -> table(
    props.exercises,
    ExerciseMetadataDTO::id,
    "Naam" to ExerciseMetadataDTO::name,
    "Groep" to ExerciseMetadataDTO::group.nested(GroupDTO::name),
    "Niveau" to ExerciseMetadataDTO::difficulty
)
```

Zoals je ziet wordt er een tabel function (component) aangeroepen met de volgende meegegeven parameters:

- props.exercise – Dit is een array met data die te tabel moet weergeven, in dit geval een array van ExerciseMetadataDTO.

- ExerciseMetadataDTO::id – Dit geeft aan welke property van de array uniek is. Indien je een onRowClick lambda meegeeft (wat hier niet het geval is) wordt de waarde van deze property uit de regel waar je op hebt geklikt meegegeven als lambda parameter.
- Tot slot zie je 3 (vararg) Pair parameters, deze geven aan welke properties van de class in de array met data weergegeven moet worden. De eerste waarde uit de pair is de header *tekst* en de tweede waarde is een lambda met als parameter de class in de array met data welke Any? (elke waarde) returned (String, Boolean, Int, etc kunnen dan allemaal weergegeven worden).

Zoals je ziet zorgt dit ervoor dat je makkelijk, typesafe tabellen kan bouwen welke makkelijk zijn uit te breiden. Stel je wilt in bovenstaande tabel ook de ID weergeven, dan hoef je enkel de volgende regel toe te voegen:

```
"id" to ExerciseMetadataDTO::id
```

Bovenstaande wordt mogelijk gemaakt door de table helper function (zie Bijlage 23 – Table helper function), welke onderwater de daadwerkelijke table function aanroept (zie Bijlage 24 – Table function).

Dit was slechts een enkel voorbeeld, zo bestaat een groot deel van de applicatie uit meerdere hergebruikte componenten.

8.6.6.1 CSS

Dankzij de kotlin-styled en kotlin-css libraries heb ik de CSS ook middels Kotlin kunnen definiëren. Dit heeft een paar voordelen, ten eerste kan je nu typesafe CSS opstellen en ten tweede kan je CSS-regels ook hergebruiken. Een mooi voorbeeld hiervan zijn kleuren, binnen de applicatie worden op verschillende plekken dezelfde kleuren gebruikt. In plaats van deze elke keer opnieuw te definiëren heb ik een Colors object gemaakt waarin de kleuren van de applicatie zijn gedefinieerd (zie Bijlage 25 – Colors object). Deze wordt bijvoorbeeld bij de Button component gebruikt, de button component heeft o.a. een parameter welke aangeeft wat voor buttontype het is (een ButtonType enum), op basis van de buttontype heeft de button een andere kleur. De ButtonType enum ziet er als volgt uit:

```
enum class ButtonType(val backgroundColor: Color, val hoverBackGroundColor:
Color){
    WARNING(Colors.orange, Colors.orange2),
    ERROR(Colors.red, Colors.red2),
    SUCCESS(Colors.green, Colors.green2),
}
```

Zoals je ziet worden de kleuren voor de button uit de Colors object gehaald.

De button component weergeeft vervolgens op basis van de meegegeven ButtonType de juiste kleur.

```
fun RBuilder.button(text: String, block: Boolean = false, buttonType: ButtonType
= ButtonType.SUCCESS, onClick: () -> Unit = {}) =
    styledButton {
        +text
        attrs.onClickFunction = { onClick() }
        css {
            +ButtonStyles.main
            if (block) {
                width = 100.pct
            }
            backgroundColor = buttonType.backgroundColor
            hover{
                backgroundColor = buttonType.hoverBackGroundColor
            }
        }
    }
```

Dit zorgt ervoor dat je makkelijk een nieuwe type knop kan toevoegen, je hoeft dan enkel een nieuwe waarde aan de ButtonType enum toe te voegen waarbij je de daarbij horende kleuren mee geeft.

Dit was slechts 1 onderdeel waar de Colors object wordt gebruikt, deze wordt echter ook op andere onderdelen van de applicatie gebruikt. Het voordeel hiervan is dat je middels 1 wijziging de styling op meerdere plekken kan aanpassen.

8.6.6.2 Libraries

De applicatie maakt gebruik van 2 React libraries, het gebruik van React libraries binnen Kotlin is misschien niet direct duidelijk (al bestaat het voornamelijk uit definieren van external definitions zoals in het hoofdstuk Axios is beschreven), ik zal dit daarom d.m.v. de react-toastify library toelichten (dit is een library voor het weergeven van toast messages). In principe moeten er altijd 2 bestanden worden aangemaakt 1 bestand bevat de external defintions (Toast.kt) en 1 bestand bevat de RBuilder extension functions waarmee de component aangeroepen kan worden (ToastDSL.kt).

Toast.kt:

```
@file:JsModule("react-toastify")

package nl.lawik.prototype.react.components.external.toast

import react.*

@JsName("ToastContainer")
external class ToastContainerComponent: Component<RProps, RState> {
    override fun render(): ReactElement?
}

@JsName("toast")
external val toast: dynamic
```

Zoals je ziet bevat Toast.kt een @file:JsModule annotatie, dit komt omdat react-toastify meerdere exports heeft, dit bestand heeft nu de context van deze library. Middels de @JsName annotatie kunnen de individuele exports aan external declarations gebonden worden.

ToastDSL.kt (imports en package weggelaten)

```
fun RBuilder.toastContainer() = child(ToastContainerComponent::class) {}
```

Zoals je ziet wordt er hier een RBuilder extension function gemaakt voor een class component zoals dat ook bij andere componenten is gedaan, het enige verschil hier is dat de ToastContainerComponent een external class is. Nu is de react-toastify library te gebruiken (echter wel zonder de configuraties die de library biedt, hier zouden ook external declarations voor geschreven moeten worden, ik had echter alleen de standaard configuratie nodig).

8.6.7 Redux

Voor (React-)Redux heb ik voornamelijk de werkwijze uit het hoofdstuk uit het hoofdstuk Redux kunnen aanhouden. Echter liep ik wel tegen een probleem aan, Redux werkt met simpele synchrone actions. Dit levert problemen op wanneer je bijvoorbeeld: een loader spinner wilt laten zien → een request naar de backend wilt sturen → De request wilt afhandelen (data in de store updaten of error property setten indien er iets misgaat) → de loader spinner wilt verbergen.

Binnen het JavaScript landschap wordt vaak gebruik gemaakt van redux-thunk om bovenstaand probleem op te lossen, dit is een middleware voor Redux waarmee je actions ook functions kunnen zijn/bevatten, d.m.v. deze middleware kan je het dispatchen van actions uitstellen. Ik heb als oplossing voor mijn probleem deze middleware enigszins binnen Kotlin nagebouwd. Ik heb een ThunkAction class gemaakt met een function literal van het type MiddleWareApi.

```
class ThunkAction(val middleware: MiddlewareApi<State, RAction, WrapperAction>.()
-> Unit) : RAction
```

Dit zorgt ervoor dat je binnen je action bij de state en dispatch kan.

Hieronder een voorbeeld van een action die gebruikt maakt van de ThunkAction.

```
fun fetchAvailableExercises() = ThunkAction {
    dispatch(OverviewAction.SetAvailableExercisesLoading(true))
    GlobalScope.launch {
        try {
            getState().auth.jwt?.let {
                val exerciseEndpoint = ExerciseEndpoint()
                val exercises = exerciseEndpoint.getAvailableExercisesMetadata()

dispatch(OverviewAction.SetAvailableExercises(exercises.toTypedArray()))
            }
        } catch (e: Throwable) {
            toast.error("Er ging iets mis.")
            dispatch(OverviewAction.SetAvailableExercisesError(true))
        }
        dispatch(OverviewAction.SetAvailableExercisesLoading(false))
    }
}
```

Zoals je ziet kan je binnen de action bij de state en kan je aanvullende actions dispatchen.

Hieronder de middleware die dit mogelijk maakt.

```
fun thunkMiddleware(): Middleware<State, RAction, WrapperAction, RAction, Any> =
{ middlewareApi ->
    { next ->
        {
            if (it is ThunkAction) {
               it.middleware(middlewareApi)
            } else {
                next(it)
            }
        }
    }
}
```

Deze middleware controleert of de binnengekomen action van het type ThunkAction is, indien dit het geval is wordt de lambda uitgevoerd, indien dit niet het geval is wordt de action zoals gewoonlijk uitgevoerd.

8.6.8 Backend

De backend is naast de eerder genoemde onderdelen (authenticatie, autorisatie en synchronisatie) en de gedeelde onderdelen op dezelfde wijze gebouwd zoals in de hoofdstukken: Hibernate, JAX-RS en Code delen tussen frontend en backend (multiplatform project) staan beschreven.

8.6.9 Testen

Er was vooraf wel nagedacht over het testen van de REST-API (zie Code testen), echter was het door de authenticatie en autorisatie niet mogelijk om hier unit tests voor te schrijven. Een mogelijk oplossing is om (een deel van) de autorisatie onderdelen te mocken/uit te zetten binnen tests, dit is echter een nice-to-have voor in de toekomst.

8.7 Conclusie en aanbevelingen

Het resultaat van het project is een full-stack multiplatform (jvm en JavaScript) applicatie welke volledig in Kotlin is gebouwd en code tussen de twee verschillende platformen deelt. Alle deelvragen zijn positief beantwoord en het prototype voldoet aan alle opgestelde requirements (op 1 functionele COULD requirement na welke wegens gebrek aan tijd en lage prioriteit niet is ontwikkeld), en werkt naar behoren. Echter zijn er wel een aantal complicaties waar rekening mee gehouden dient te worden:

- Learning curve (ervan uit gaande dat niet iedereen in een team eerder met Kotlin heeft gewerkt) – Net als elke nieuwe taal/technologie is er een learning curve (dit is een tijdsinvestering). De learning curve voor de syntax van de taal zelf valt mee, ik doel hier voornamelijk op de toepassing van Kotlin op de frontend.
- Informatie is niet altijd te vinden – Dit heeft vooral betrekking tot de frontend, veel informatie is niet altijd (direct) te vinden, hierbij doel ik met name over informatie over library wrappers en hoe deze toegepast kunnen worden. Desondanks is het wel gelukt een volwaardig prototype te bouwen, de Kotlin community is erg actief en bij vragen krijg je vaak (snel) antwoord.
- Kotlin/JS nog niet volledig stabiel – Hierbij doel ik op de *ontwikkeling* van Kotlin/JS door JetBrains, deze heeft momenteel de status Additional in Incremental Releases wat wil zeggen dat er bij nieuwe releases *mogelijk* onderdelen verwijderd kunnen worden (hoewel JetBrains dit zoveel mogelijk proberen te vermijden). *
- Kotlin Multiplatform nog niet volledig stabiel – Hierbij doel ik op de *ontwikkeling* van Kotlin Multiplatform door JetBrains, deze heeft momenteel de status Moving Fast, wat wil zeggen dat het mogelijk is dat nieuwe releases niet comptabel zijn met voorgaande releases, dit onderdeel is dus nog zeer experimenteel. *
- JavaScript interoperability – Hoewel JavaScript interoperability mogelijk is, is dit naar mijn mening nog niet perfect. Het schrijven van external declarations kan tijd in beslag nemen (hoewel dit ook een learning curve betreft waar je steeds beter in kan worden). De ts2kt tool welke typescript defintions naar Kotlin externals vertaald zorgt ook niet voor 100% correcte externals maar het is een stap in de goede richting waarbij je waar nodig dingen kan aanpassen. JetBrains gaf aan bezig te zijn met betere JavaScript interoperability waarbij mogelijk Kotlin direct met .d.ts bestanden te werk kan gaan en externals niet meer nodig zijn (hoewel het nog niet duidelijk is wanneer dit beschikbaar zal zijn). **
- Bundle size – Ondanks de DCE plugin en de minifier van webpack zijn bundle sizes nog steeds aan de hoge kant. De bundle size van het prototype is 1978KB (450KB gzip). Een (leeg) JavaScript React project *met soortgelijke dependencies* is ~200KB (~61KB gzip). Jetbrains gaf aan bezig te zijn met *veel* betere DCE (hoewel het nog niet duidelijk is wanneer dit beschikbaar zal zijn). **

* (Kotlin Programming Language, z.d.)
** (Bannykh, Re: Hi guys, what are the current plans for KotlinJS? Like what are the primary features, changes, etc. the KotlinJS team are working on for this year? [Online Chat Bericht], 2019)
(Bannykh, Re: Hi guys, what are the current plans for KotlinJS? Like what are the primary features, changes, etc. the KotlinJS team are working on for this year? - 2 [Online Chat Bericht], 2019)

Desondanks de kleine complicaties is het wel gelukt om alle deelvragen positief te beantwoorden en het prototype naar behoren te ontwikkelen, op basis hiervan kun je stellen dat het voor Topicus Onderwijs inderdaad realistisch is om Kotlin voor frontend en backend applicaties binnen één codebase toe te passen.

Kotlin is een hele mooie ontwikkeling, ik raad ten zeerste aan om in ieder geval Kotlin te leren, gezien het feit dat je middels 1 taal voor verschillende platformen kan ontwikkelen (JVM, JS, iOS, Android, Native) en daarbij meerdere platformen binnen 1 project kan ontwikkelen en code kan delen. Daarnaast is het naar mijn mening een mooie taal met zeer krachtige features (data classes, extension functions, higher order functions, etc) welke ook nog eens interoperability biedt voor de taal waarmee doorgaans voor het platform wordt ontwikkeld.

Tevens heeft Kotlin een actieve community waar je bij vragen/problemen terecht kunt, denk hierbij aan Slack, youtrack (issue tracker), Kotlin forum, GitHub en StackOverflow.

Tot slot heb ik een lijst met interessante opties voor de mogelijke doorontwikkeling van deze afstudeeropdracht opgesteld:

- Tests voor het prototype (zie Testen)
- Doorontwikkeling van de Hibernate/JPA CriteriaQuery DSL (subquery, in, not, etc)
- Onderzoek naar automatische validatie van formulieren op de frontend d.m.v. de Validateable interface welke ook op de backend wordt gebruikt.
- Onderzoek naar dependency injection
- Onderzoek naar het ontwikkelen van compiler plugins
- Onderzoek naar het testen van common code
- Onderzoek naar het testen van frontend code
- Onderzoek naar volledige Kotlin libraries als (mogelijke) vervanging voor de onderzochte technologieën zoals:
    - KVision – Voor frontend development (als vervanging voor React).
    - Ktor (server) – Voor REST-API’s (als vervanging voor JAX-RS).
    - Exposed – SQL library/ORM (als vervanging voor Hibernate).
- Onderzoek naar de mogelijkheden om naast JavaScript en JVM nog meer platformen binnen 1 project te ontwikkelen, Kotlin biedt ook ondersteuning voor o.a. Android en iOS, het zou interessant zijn om uit te zoeken hoe 3 clients (web/JavaScript, Android en iOS) en de server (jvm) binnen 1 project ontwikkeld kunnen worden.

9. Bibliografie

Axios. (z.d.). *Promise based HTTP client for the browser and node.js*. Opgeroepen op mei 30, 2019, van Axios GitHub: https://github.com/axios/axios

Bannykh, A. (2019, april 10). *Re: Hi guys, what are the current plans for KotlinJS? Like what are the primary features, changes, etc. the KotlinJS team are working on for this year? - 2 [Online Chat Bericht]*. Opgeroepen op mei 29, 2019, van https://kotlinlang.slack.com/archives/C0B8L3U69/p1554889572042200?thread_ts=1554827960.039400&cid=C0B8L3U69

Bannykh, A. (2019, april 10). *Re: Hi guys, what are the current plans for KotlinJS? Like what are the primary features, changes, etc. the KotlinJS team are working on for this year? [Online Chat Bericht]*. Opgeroepen op mei 29, 2019, van https://kotlinlang.slack.com/archives/C0B8L3U69/p1554889049041700?thread_ts=1554827960.039400&cid=C0B8L3U69

GitHub. (z.d.). *Axios Stargazers*. Opgeroepen op mei 05, 2019, van Axios GitHub: https://github.com/axios/axios/stargazers

GitHub. (z.d.). *Search Mock*. Opgeroepen op juni 06, 2019, van GitHub: https://github.com/search?l=Kotlin&o=desc&q=mock&s=stars&type=Repositories

Heller, A. (2018, juli 24). *kotlin-redux wrapper README*. Opgeroepen op april 05, 2019, van kotlin-wrappers Github: https://github.com/JetBrains/kotlin-wrappers/blob/master/kotlin-redux/README.md#limitations

Hibernate. (z.d.). *Your relational data. Objectively*. Opgeroepen op maart 08, 2019, van Hibernate ORM.

JCP. (z.d.). *JSRs: Java Specification Requests - Detail JSR# 311*. Opgeroepen op maart 13, 2019, van Java Community Process: https://jcp.org/en/jsr/detail?id=311

Kotlin Programming Language. (z.d.). *Calling JavaScript from Kotlin*. Opgeroepen op februari 19, 2019, van https://kotlinlang.org/docs/reference/js-interop.html

Kotlin Programming Language. (z.d.). *FAQ*. Opgeroepen op februari 19, 2019, van https://kotlinlang.org/docs/reference/faq.html

Kotlin Programming Language. (z.d.). *Stability of Different Components*. Opgeroepen op mei 29, 2019, van https://kotlinlang.org/docs/reference/evolution/components-stability.html

ktor. (z.d.). *asynchronous Web framework for Kotlin*. Opgeroepen op mei 30, 2019, van ktor: https://ktor.io/

Lochschmidt, N. (z.d.). *Portable validations for Kotlin*. Opgeroepen op april 29, 2019, van Konform: https://www.konform.io/

napperley. (2017, juli 25). *Re: What's the recommended build tool to use with kotlin: maven or gradle? [Online Forum Bericht]*. Opgeroepen op februari 22, 2019, van https://discuss.kotlinlang.org/t/whats-the-recommended-build-tool-to-use-with-kotlin-maven-or-gradle/3873/2

npm-stat. (z.d.). *Download statistics for package axios*. Opgeroepen op mei 05, 2019, van npm-stat: https://npm-stat.com/charts.html?package=axios&from=2018-05-06&to=2019-05-06

reduxjs. (z.d.). *todos source*. Opgeroepen op april 05, 2019, van Redux GitHub: https://github.com/reduxjs/redux/tree/master/examples/todos/src

10. Bijlagen

Bijlage 1 – Epic voorbeeld

Bijlage 2 – Backlog

Bijlage 3 – Gradle build configuratie voor een (simpel) jvm Kotlin project

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.21" // Kotlin gradle plugin
}

group 'kotlin-jvm' // group
version '1.0-SNAPSHOT' // huidige versie van het project

repositories { // repositories waar dependencies vandaan gehaald kunnen worden
    mavenCentral() // maven central is een repository waar dependencies vandaan
worden gehaald
}

dependencies { // de dependencies (libraries) die het project nodig heeft
    implementation "org.jetbrains.kotlin:kotlin-stdlib" // de Kotlin standard
library dependency
}
```

Bijlage 4 – JAX-RS Jackson 2 object mapper voor Kotlin registeren

```
@Provider
@Consumes("application/json", "application/*+json", "text/json")
class JSONConsumer : ContextResolver<ObjectMapper> {

    private val objectMapper: ObjectMapper = jacksonObjectMapper()

    override fun getContext(type: Class<*>?): ObjectMapper {
        return objectMapper
    }

}
```

Bijlage 5 – Kotlinx-Serialization voorbeeld

```
@Serializable
data class Todo(
    val userId: Long,
    val title: String,
    val completed: Boolean,
    @Optional val id: Long? = null
)
```

Bijlage 6 – Fetch helper extension function

```
fun <T> Window.fetch(
    input: dynamic,
    requestInit: RequestInit,
    deserializationStrategy: DeserializationStrategy<T>
): Promise<T> {
    return this.fetch(input, requestInit).then {
        it.text()
    }.then {
        Json.parse(deserializationStrategy, it)
    }
}
```

Bijlage 7 – Axios external definitions

```
external interface AxiosConfigSettings {
    var url: String
    var method: String
    var baseUrl: String
    var transformRequest: (data: dynamic, headers: dynamic) -> dynamic
    var transformResponse: (data: dynamic, headers: dynamic) -> dynamic
    var headers: dynamic
    var params: dynamic
    var paramsSerializer: (dynamic) -> String
    var data: dynamic
    var timeout: Number
    var withCredentials: Boolean
    var adapter: dynamic
    var auth: dynamic
    var responseType: String
    var xsrfCookieName: String
    var xsrfHeaderName: String
    var onUploadProgress: (dynamic) -> Unit
    var onDownloadProgress: (dynamic) -> Unit
    var maxContentLength: Number
    var validateStatus: (Number) -> Boolean
    var maxRedirects: Number
    var httpAgent: dynamic
    var httpsAgent: dynamic
    var proxy: dynamic
    var cancelToken: dynamic
}

external interface AxiosResponse<T> {
    val data: T
    val status: Number
    val statusText: String
    val headers: dynamic
    val config: AxiosConfigSettings
    val request: dynamic
}
```

Bijlage 7 – Axios TypeScript definitions

```
export interface AxiosRequestConfig {
  url?: string;
  method?: string;
  baseURL?: string;
  transformRequest?: AxiosTransformer | AxiosTransformer[];
  transformResponse?: AxiosTransformer | AxiosTransformer[];
  headers?: any;
  params?: any;
  paramsSerializer?: (params: any) => string;
  data?: any;
  timeout?: number;
  withCredentials?: boolean;
  adapter?: AxiosAdapter;
  auth?: AxiosBasicCredentials;
  responseType?: string;
  xsrfCookieName?: string;
  xsrfHeaderName?: string;
  onUploadProgress?: (progressEvent: any) => void;
  onDownloadProgress?: (progressEvent: any) => void;
  maxContentLength?: number;
  validateStatus?: (status: number) => boolean;
  maxRedirects?: number;
  httpAgent?: any;
  httpsAgent?: any;
  proxy?: AxiosProxyConfig | false;
  cancelToken?: CancelToken;
}

export interface AxiosResponse<T = any>  {
  data: T;
  status: number;
  statusText: string;
  headers: any;
  config: AxiosRequestConfig;
  request?: any;
}
```

Bijlage 8 – Axios helper function

```
fun <T> axios(
    deserializationStrategy: DeserializationStrategy<T>,
    config: AxiosConfigSettings.() -> Unit
): Promise<AxiosResponse<T>> {
    val axiosConfigSettings = axiosConfigSettings()
    axiosConfigSettings.config()
    axiosConfigSettings.transformResponse = { data, _ ->
Json.parse(deserializationStrategy, data as String) }
    return axios(axiosConfigSettings)
}
```

Bijlage 9 – Webpack Gradle configuratie

```
kotlinFrontend {
    webpackBundle {
        bundleName = "this-will-be-overwritten" // NOTE: for example purposes
this is overwritten in `webpack.config.d/filename.js`.
        contentPath = file('src/main/resources/web')
        if (project.hasProperty('prod')) {
            mode = "production"
        }
    }
    // ... overige kotlinFrontend settings
}
```

Bijlage 10 – Gegenereerde webpack.config.js

```
'use strict';

var webpack = require('webpack');

var config = {
    "mode": "production",
    "context": "D:\\Afstuderen\\Code\\poc\\kotlin-poc-frontend-
axios\\build\\kotlin-js-min\\main",
    "entry": {
        "this-will-be-overwritten": "./kotlin-poc-frontend-axios"
    },
    "output": {
        "path": "D:\\Afstuderen\\Code\\poc\\kotlin-poc-frontend-
axios\\build\\bundle",
        "filename": "[name].bundle.js",
        "chunkFilename": "[id].bundle.js",
        "publicPath": "/"
    },
    "module": {
        "rules": [

        ]
    },
    "resolve": {
        "modules": [
            "kotlin-js-min\\main",
            "resources\\main",
            "D:\\Afstuderen\\Code\\poc\\kotlin-poc-frontend-
axios\\build\\node_modules",
            "node_modules"
        ]
    },
    "plugins": [

    ]
};

module.exports = config;

// from file D:\Afstuderen\Code\poc\kotlin-poc-frontend-
axios\webpack.config.d\filename.js
config.output.filename = "kotlin-poc-frontend-axios.bundle.js"
```

Bijlage 11 – Kotlin React Redux NPM libraries

```
dependency "react"
dependency "react-dom"
dependency "react-router-dom"
dependency "react-redux"
dependency "redux"
dependency "core-js"
dependency "inline-style-prefixer" // voor css ondersteuning
dependency "styled-components" // voor css ondersteuning
```

Bijlage 12 – Kotlin React Redux wrappers

```
implementation 'org.jetbrains:kotlin-react:16.6.0-pre.70-kotlin-1.3.21'
implementation 'org.jetbrains:kotlin-react-dom:16.6.0-pre.70-kotlin-1.3.21'
implementation 'org.jetbrains:kotlin-react-redux:5.0.7-pre.70-kotlin-1.3.21'
implementation 'org.jetbrains:kotlin-redux:4.0.0-pre.70-kotlin-1.3.21'
implementation 'org.jetbrains:kotlin-styled:1.0.0-pre.70-kotlin-1.3.21'
implementation 'org.jetbrains:kotlin-react-router-dom:4.3.1-pre.70-kotlin-1.3.21'
```

Bijlage 13 – JavaScript Redux action

```
export const toggleTodo = id => ({
    type: 'TOGGLE_TODO',
    id
})
```

Bijlage 14 – JavaScript Redux reducer

```
const todos = (state = [], action) => {
    switch (action.type) {
        case 'ADD_TODO':
            return [
                ...state,
                {
                    id: action.id,
                    text: action.text,
                    completed: false
                }
            ]
        case 'TOGGLE_TODO':
            return state.map(todo =>
                (todo.id === action.id)
                    ? {...todo, completed: !todo.completed}
                    : todo
            )
        default:
            return state
    }
}

export default todos
```

Bijlage 15 – JavaScript React-Redux Link.js

```
// components/Link.js

const Link = ({active, children, onClick}) => (
    <button
        onClick={onClick}
        disabled={active}
        style={{
            marginLeft: '4px',
        }}
    >
        {children}
    </button>
)

Link.propTypes = {
    active: PropTypes.bool.isRequired,
    children: PropTypes.node.isRequired,
    onClick: PropTypes.func.isRequired
}

export default Link
```

Bijlage 16 – JavaScript React-Redux FilterLink.js

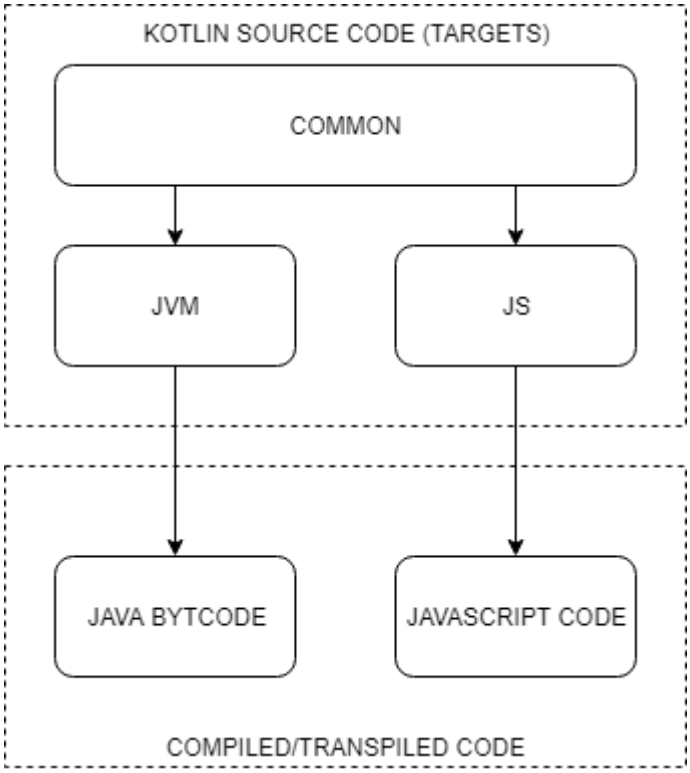
```
// containers/FilterLink.js

const mapStateToProps = (state, ownProps) => ({
    active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
    onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
})

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(Link)
```

Bijlage 17 – Multiplatform project (JVM & JS) compilation diagram



Bijlage 18 – JAX-RS suspend keyword workaround

```
@Provider
@Consumes("*/*")
class ContinuationMessageBodyReader : MessageBodyReader<Continuation<*>> {
    override fun isReadable(
        type: Class<*>?,
        genericType: Type?,
        annotations: Array<out kotlin.Annotation>?,
        mediaType: MediaType?
    ): Boolean {
        return type == Continuation::class.java
    }

    override fun readFrom(
        type: Class<Continuation<*>>?,
        genericType: Type?,
        annotations: Array<out kotlin.Annotation>?,
        mediaType: MediaType?,
        httpHeaders: MultivaluedMap<String, String>?,
        entityStream: InputStream?
    ): Continuation<*>? {
        return null
    }
}
```

Bijlage 19 – JAX-RS validation interceptor

```
@Provider
@Consumes("application/json", "application/*+json", "text/json")
class ValidationInterceptor : ReaderInterceptor {

    @Context
    private lateinit var context: Request

    @Throws(IOException::class, WebApplicationException::class)
    override fun aroundReadFrom(interceptorContext: ReaderInterceptorContext):
Any? {
        val body = interceptorContext.proceed()
        if (body is Validateable<*>) {
            lateinit var validation: ValidationResult<*>
            val method = context.method
            if (method == HttpMethod.POST) {
                validation = body.validateCreate() ?: body.validate()
            } else if (method == HttpMethod.PUT) {
                validation = body.validateUpdate() ?: body.validate()
            }
            when (validation) {
                is Valid -> return body
                is Invalid -> {
                    // hacky but whatever, this should be accessible anyway...
                    val errorsField =
validation::class.java.getDeclaredField("errors")
                    errorsField.isAccessible = true
                    val errors = errorsField.get(validation)

                    throw
WebApplicationException(Response.status(422).entity(mapOf("errors" to
errors)).build())
                }
            }
        }
        return body
    }
}
```

Bijlage 20 – Prototype functionele omschrijving & mockups

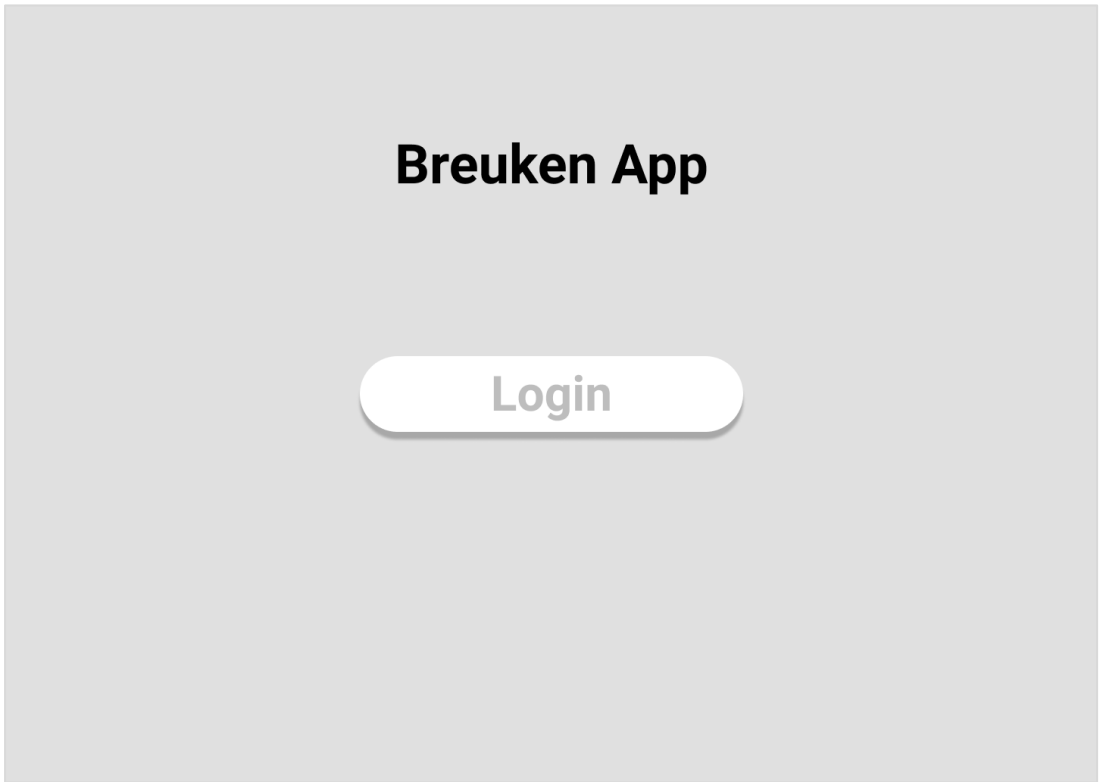
Structuur

De applicatie bestaat uit twee omgevingen, een omgeving voor de leerling en een omgeving voor de leerkracht. Gebruikers kunnen enkel bij de inhoud van hun eigen omgeving. Beide type gebruikers loggen in middels hun eigen schoolgegevens, dit wordt mogelijk gemaakt door een OAuth koppeling met het Wise-r platform van Topicus Onderwijs.

- De omgeving van de leerkracht – Binnen de omgeving van de leerkracht kan de leerkracht oefeningen aanmaken (beheer) en de resultaten van diens groep(en) inzien (resultaten).
- De omgeving van de leerling – Binnen de omgeving van de leerling kan de leerling een overzicht zien van zijn/haar beschikbare oefeningen en zijn/haar voltooide oefeningen met de daarbij behaalde scores. Wanneer de leerling op een beschikbare oefening klikt wordt hij/zij doorverwezen naar de pagina waar hij/zij de oefening kan uitvoeren.

Login

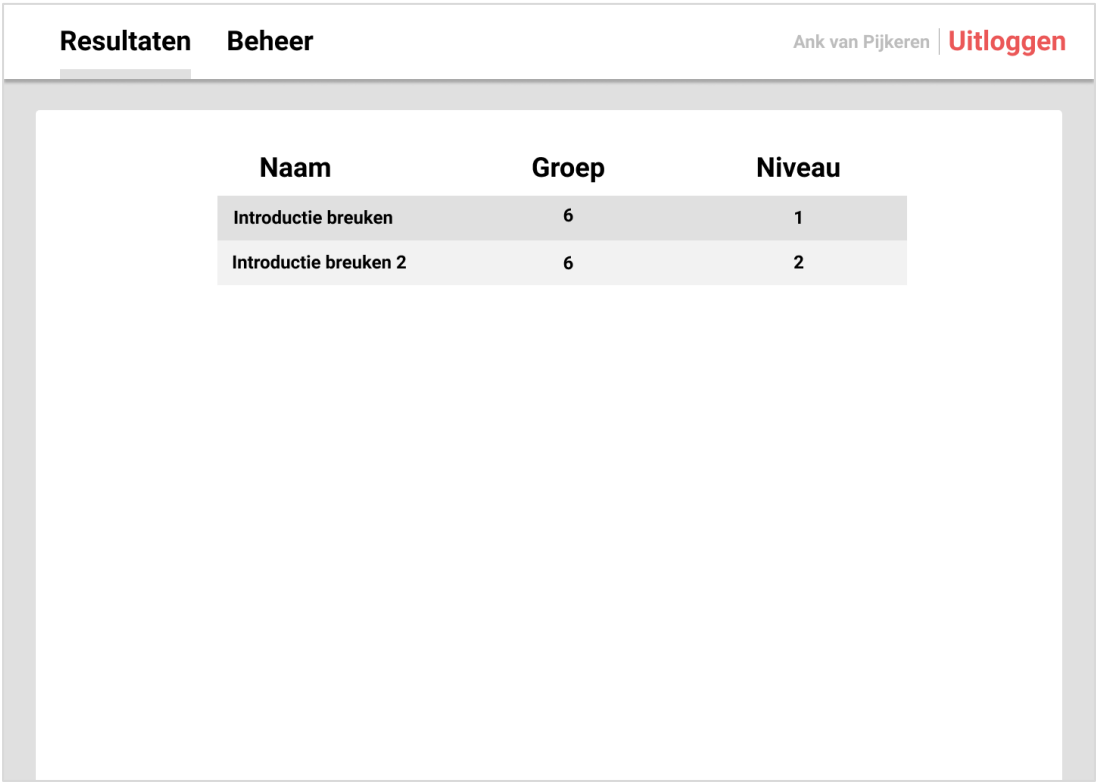
Wanneer de gebruiker niet geauthentiseerd is (de gebruiker heeft geen of een verlopen JWT) dan ziet de gebruiker het login scherm, dit scherm bevat de titel van de applicatie en een login knop wanneer hierop wordt gedrukt wordt er een OAuth implicit flow gestart met het Wise-r platform waar de gebruiker middels zijn/haar schoolgegevens kan inloggen waarnaar hij/zij zal worden terugverwezen naar de applicatie.



Figuur 5 – Prototype: Inlog scherm mockup

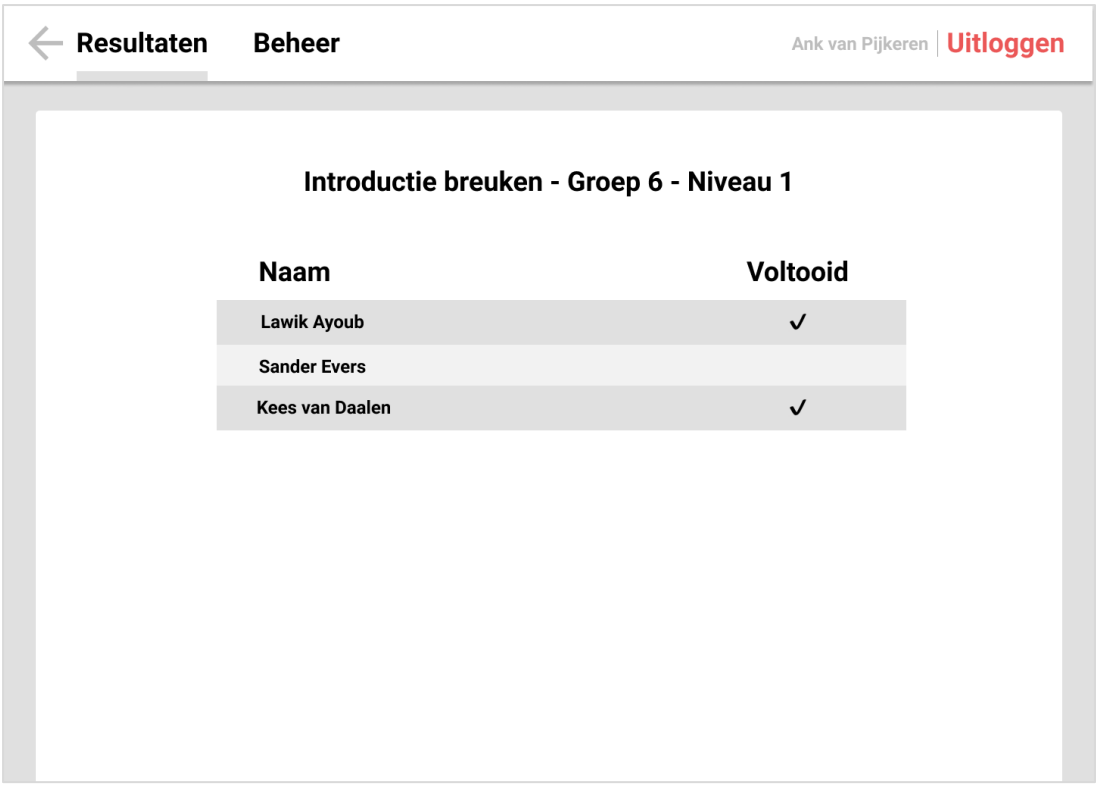
Leerkracht – Resultaten (niet ontwikkeld)

Op de resultaten pagina kan de leerkracht de resultaten van diens leerlingen inzien. Men ziet eerst een overzicht van alle oefeningen, dit is een tabel met de naam, groep en niveau van de oefening.



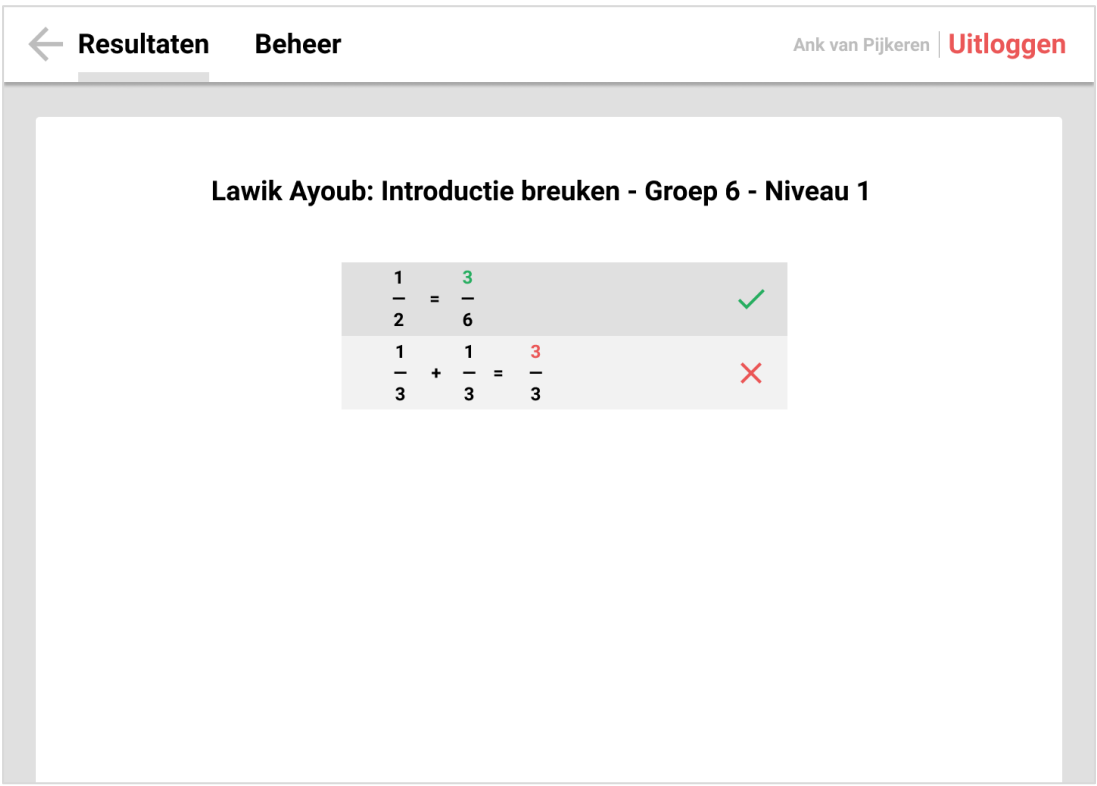
Figuur 6 – Prototype: Leerkracht resultaten overzicht mockup

Wanneer op een oefening wordt geklikt ziet men een lijst met leerlingen die tot de groep van de oefening behoren en of de leerling de oefening heeft voltooid.



Figuur 7 – Prototype: Leerkracht resultaten oefening details mockup

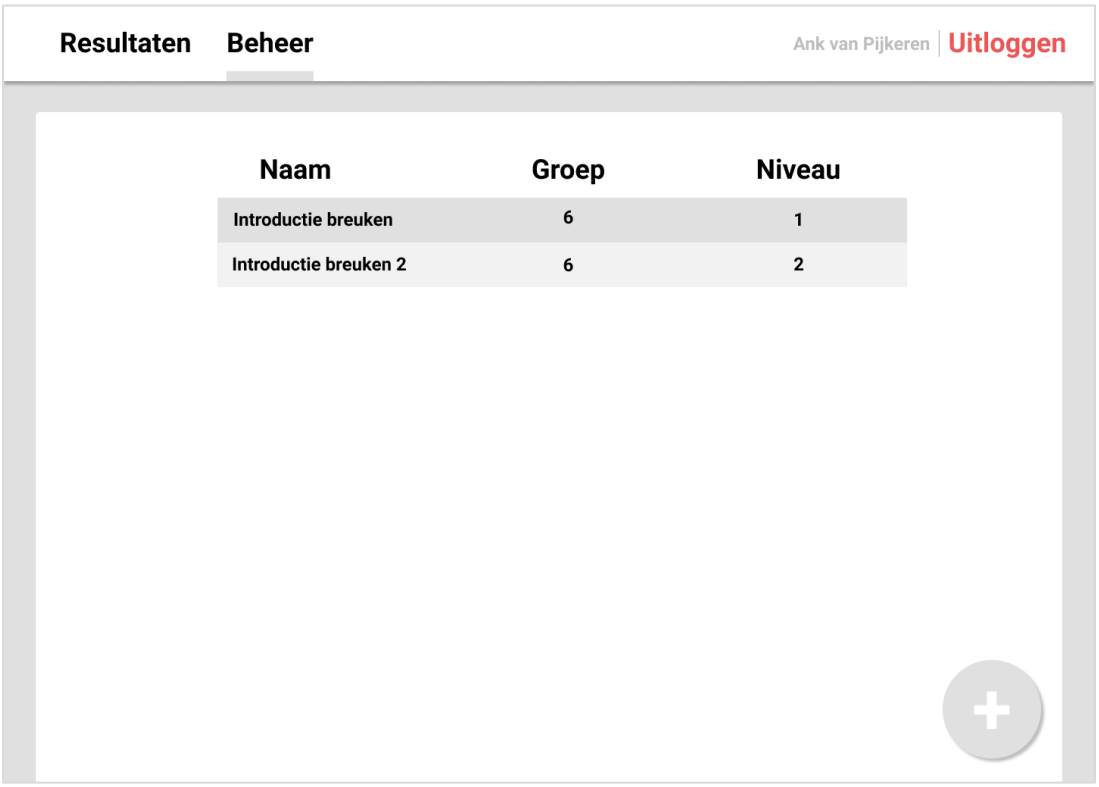
Wanneer op een leerling wordt geklikt worden de resultaten van de desbetreffende leerling per vraag weergegeven.



Figuur 8 – Prototype: Leerkracht resultaten oefening leerling details mockup

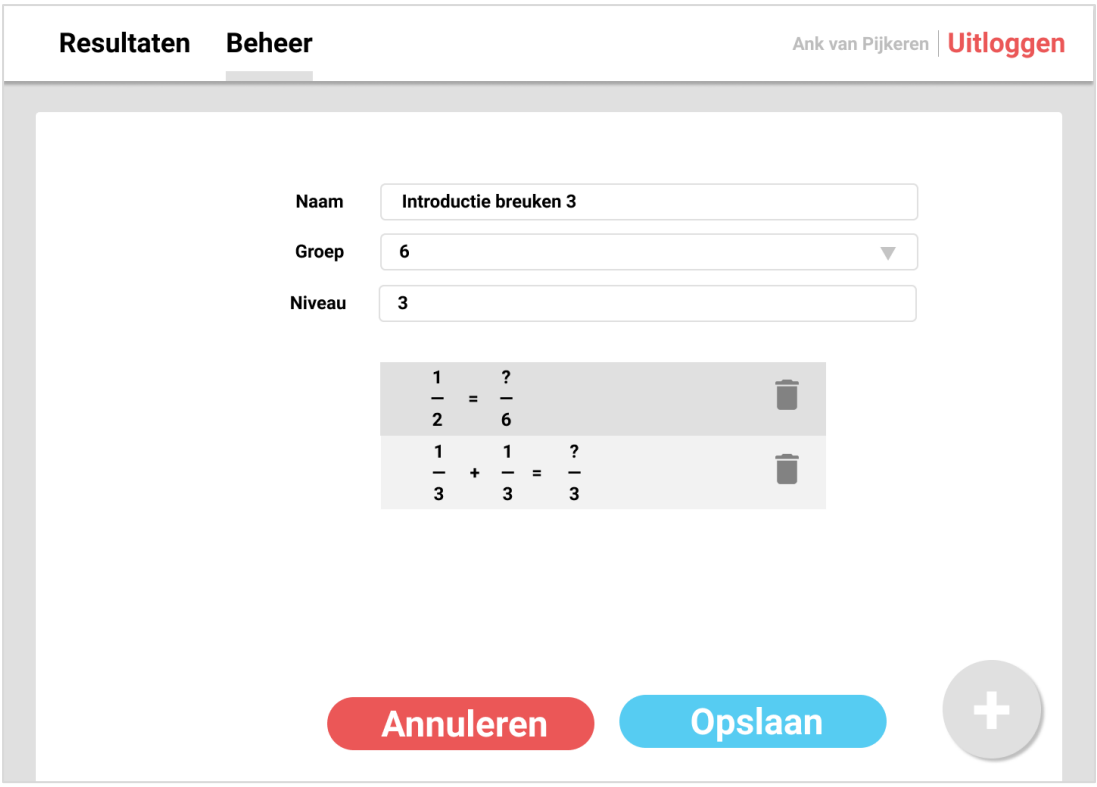
Leerkracht – Beheer

Op de beheer pagina ziet de leerkracht een overzicht van alle opgestelde oefeningen voor diens groep(en), dit is een tabel met de naam, groep en niveau van de oefening.



Figuur 9 – Prototype: Leerkracht beheer overzicht mockup

Onderin het beheerscherm ziet de gebruiker een Floating Action Button, wanneer hierop wordt geklikt wordt de leerkracht doorverwezen naar de pagina waarop hij/zij een nieuwe oefening kan toevoegen.



Figuur 10 – Prototype: Leerkracht beheer nieuwe oefening mockup

Op deze pagina bevinden zich de volgende 3 invoervelden:

- Naam – Tekst, verplicht.
- Groep – Dropdown met de groepen van de leerkracht waarbij 1 groep kan worden gekozen, verplicht.
- Niveau – Nummer, minimaal 0, verplicht.

Verder ziet de leerkracht een overzicht van alle huidige toegevoegde vragen (waarbij bij elke vraag een vuilnis icoon staat waarmee de vraag kan worden verwijderd) en de volgende 3 knoppen:

- Annuleren – Annuleert het toevoegen van de vraag en keert terug naar de vorige pagina.
- Opslaan – Valideert de huidige input van de invoervelden, indien deze valide is wordt de oefening opgeslagen.
- Vraag toevoegen (+) – Opent de modal voor het toevoegen van een nieuwe vraag voor de huidige oefening.

Binnen de modal voor het toevoegen van een nieuwe vraag voor de huidige oefening ziet de leerkracht 4 invoervelden in de vorm van 2 breuken en 3 knoppen: +, - en = waarmee het type van de vraag kan worden opgegeven.

Figuur 11 – Prototype: Leerkracht beheer nieuwe gelijknamig maken vraag mockup

Wanneer het type vraag + of – is verschijnt een = en nog 2 invoervelden in de vorm van 1 breuk.

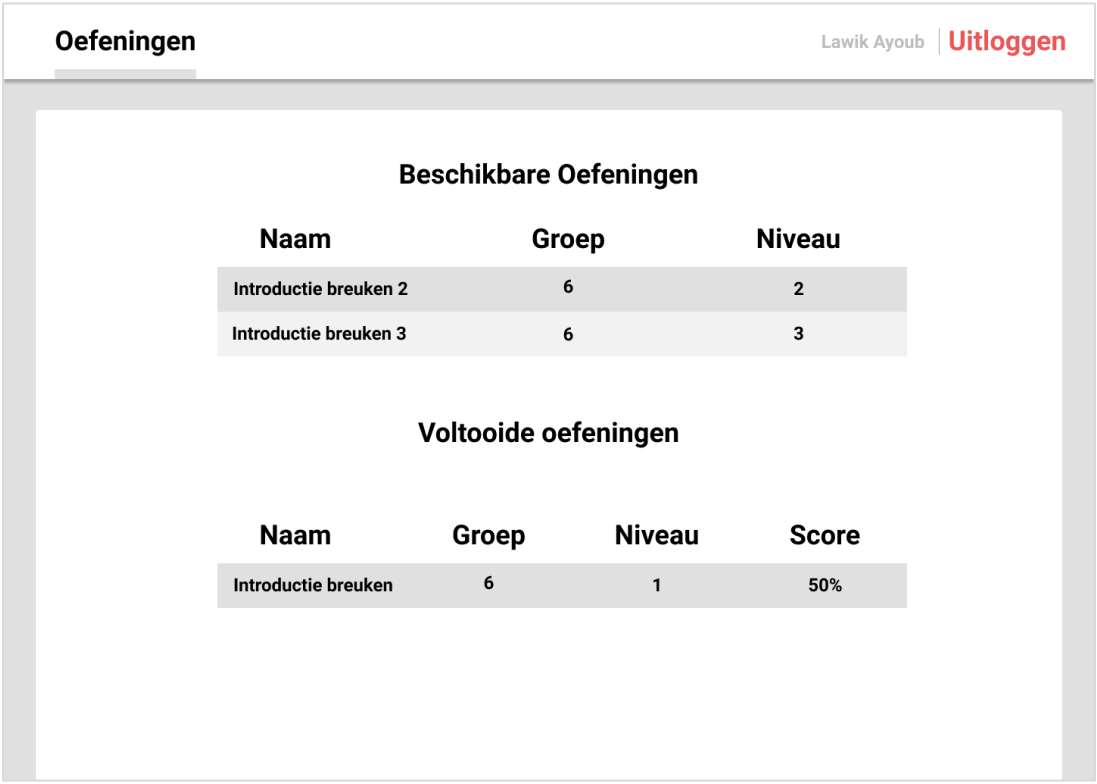
Figuur 12 – Prototype: Leerkracht beheer nieuwe optellen vraag mockup

Alle invoervelden, op 1 na, dienen een getal tussen 1-99 te bevatten, het overige invoerveld moet een vraagteken bevatten, deze moet uiteindelijk door de leerling worden ingevuld.

De leerkracht ziet onderin de modal 2 knoppen, een annuleren knop welke het toevoegen van de vraag annuleert en de modal sluit en een toevoegen knop welke de input valideert (zijn alle velden ingevuld, is er 1 vraagteken aanwezig en levert een geldige vraag op (wat wil zeggen dat het antwoord geen decimaal getal, een getal onder de 0 of boven 99 is)), als de input valide is wordt de vraag toegevoegd aan de lijst met vragen en worden de invoervelden van de modal gereset zodat de leerkracht direct nog een vraag kan toevoegen.

Leerling – Overzicht

Op de overzicht pagina van de leerling ziet de leerling 2 overzichten, een overzicht met beschikbare oefeningen, dit is een tabel met de naam, groep en niveau van de oefening, en een overzicht met voltooide oefening, dit is een tabel met de naam, groep, niveau en behaalde score van de oefening.

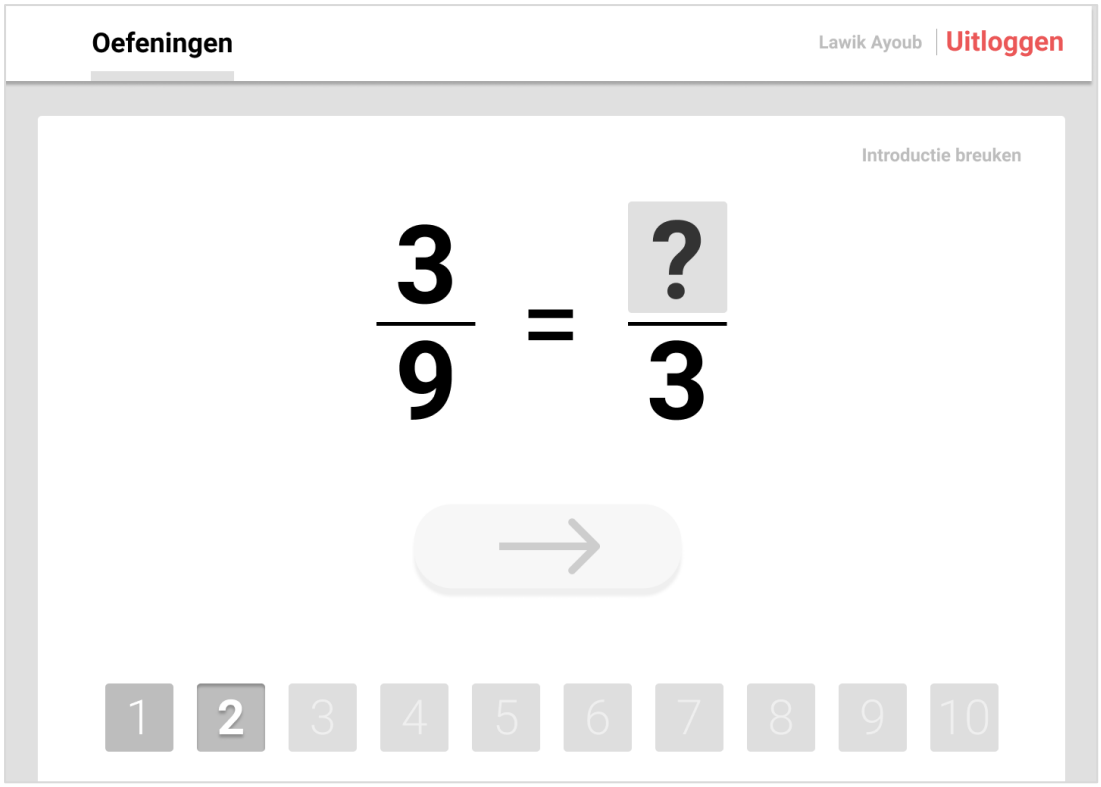


Figuur 13 – Prototype: Leerling oefeningen overzicht mockup

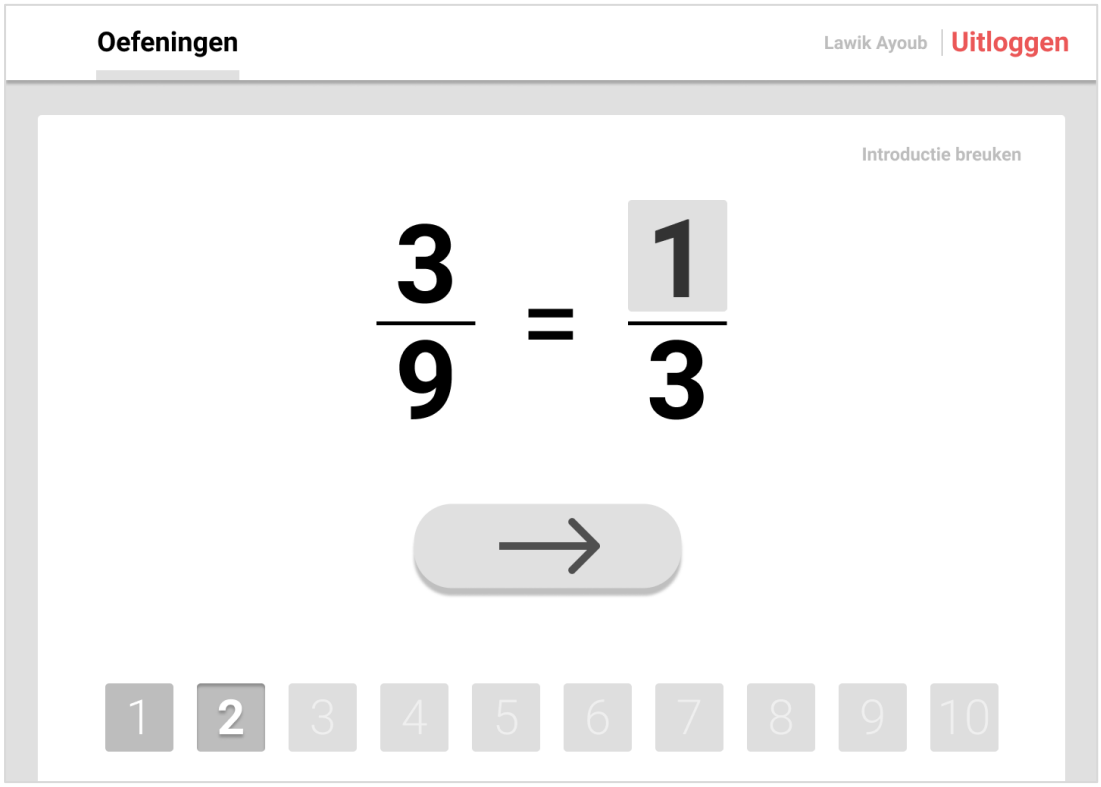
De leerling kan op een beschikbare oefening klikken om de oefening te starten waarnaar de leerling zal worden doorverwezen naar de pagina waarop de oefening kan worden uitgevoerd.

Leerling – Oefening uitvoeren

Op deze pagina ziet de leerling de huidige oefening. Hierbij ziet de leerling rechtsboven de naam van de oefening. In het midden staat de huidige vraag waarbij de door hem/haar in te vullen getal als een vraagteken is gemarkeerd en een knop welke (indien de vraag is beantwoord) naar de volgende vraag zal leiden. Onderin staat de huidige voortgang van de oefening.



Figuur 14 – Prototype: Leerling oefening vraag onbeantwoord mockup



Figuur 15 – Prototype: Leerling oefening vraag beantwoord mockup

Bijlage 21 – @Secured annotatie

```
@NameBinding
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class Secured(vararg val roles: UserType = [])
```

Bijlage 22 – AuthorizationFilter

```
@Secured
@Provider
@Priority(Priorities.AUTHORIZATION)
class AuthorizationFilter : ContainerRequestFilter {

    @Context
    private lateinit var resourceInfo: ResourceInfo

    @Throws(IOException::class)
    override fun filter(requestContext: ContainerRequestContext) {
        val authorizationHeader =
requestContext.getHeaderString(HttpHeaders.AUTHORIZATION)

        if (authorizationHeader == null ||
!authorizationHeader.startsWith("Bearer ")) {
            throw NotAuthorizedException("Authorization header must be provided")
        }

        val tokenString = authorizationHeader.substring("Bearer".length).trim()

        try {
            val jwt = validateAppJWT(tokenString)
            val userType = UserType.valueOf(jwt["userType"].toString())

            val classRoles = extractRoles(resourceInfo.resourceClass)
            val methodRoles = extractRoles(resourceInfo.resourceMethod)


            if (methodRoles.isNotEmpty() && !methodRoles.contains(userType)) {

requestContext.abortWith(Response.status(Response.Status.FORBIDDEN).build())
            }

            if (classRoles.isNotEmpty() && !classRoles.contains(userType)) {

requestContext.abortWith(Response.status(Response.Status.FORBIDDEN).build())
            }

            requestContext.securityContext = object : SecurityContext {
                override fun isUserInRole(role: String?): Boolean {
                    return true
                }

                override fun getAuthenticationScheme(): String {
                    return "Bearer"
                }

                override fun getUserPrincipal(): Principal {
                    return UserPrincipal(jwt.subject, userType,
jwt["organisation"].toString())
                }

                override fun isSecure(): Boolean {
                    return requestContext.securityContext.isSecure
                }

            }

        } catch (e: Exception) {
            when (e) {
                is ExpiredJwtException, is UnsupportedJwtException, is
MalformedJwtException, is SignatureException, is IllegalArgumentException ->

requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED).build())
                else -> throw WebApplicationException(500)
            }
```

```
        }
    }

    private companion object {
        private fun extractRoles(annotatedElement: AnnotatedElement?):
List<UserType> =
            annotatedElement?.getAnnotation(Secured::class.java)?.roles?.toList()
?: emptyList()
    }
}
```

Bijlage 23 – Table helper function

```
fun <T, R> RBuilder.table(data: Array<T>, key: (T) -> R, vararg headers:
Pair<String, (T) -> Any?>, onRowClick: ((R) -> Unit)? = null) =
    table(
        headers.map { it.first }.toTypedArray(),
        data.map{d -> key(d) to headers.map { h->
h.second(d)}.toTypedArray()}.toMap(),
        onRowClick
    )
```

Bijlage 24 – Table function

```
fun <R> RBuilder.table(headers: Array<String>, rows: Map<R, Array<Any?>>,
onRowClick: ((R) -> Unit)? = null) = styledTable {
    thead {
        tr { headers.forEach { th { +it } } }
    }
    tbody {
        rows.forEach { r ->
            styledTr {
                r.value.forEach { td { +"${it ?: ""}" } }
                onRowClick?.let {
                    attrs.onClickFunction = { it(r.key) }
                    css {
                        cursor = Cursor.pointer
                    }
                }
            }
        }
    }
    css { +TableStyles.main }
}
```

Bijlage 25 – Colors object

```
object Colors {
    val gray3 = Color("#828282")
    val gray4 = Color("#BDBDBD")
    val gray5 = Color("#E0E0E0")
    val gray6 = Color("#F2F2F2")
    val orange = Color("#ffb100")
    val orange2 = Color("#ffa700")
    val green = Color("#28a745")
    val green2 = Color("#218838")
    val red = Color("#dc3545")
    val red2 = Color("#c82333")
}
```