



# Realisaties stage

Renzo Beeckmans onthaalmodule

Renzo Beeckmans

# Inhoud

Introductie.....	2
Realisaties.....	3
Backend.....	3
Datamodel.....	3
Models.....	6
Data ophalen - aanvraag .....	9
Data ophalen - verwerking.....	12
Data ophalen - mapping.....	15
Frontend.....	21
Introductie frontend.....	21
Models.....	21
Services.....	23
Overzicht afspraken.....	24
Cliënt registreren.....	30
Onderzoek kaartlezer .....	34
Implementatie kaartlezer.....	35
Detailpagina persoon .....	43
Overzicht personen .....	51
Conclusie .....	56

# Introductie

Beste lezer, in dit document presenteer ik een overzicht van de resultaten die ik heb behaald tijdens mijn stageperiode van 13 weken bij Cipal Schaubroeck. Tijdens deze stage hebben mijn collega en ik intensief gewerkt aan een uitdagend project, waarin we veel nieuwe zaken hebben geleerd, zowel over frontend als backend, en hebben we bijgeleerd over hoe de functionele kant van een applicatie wordt bekeken. Het functionele doel van dit project houdt in dat we een applicatie hebben die de mensen aan het onthaal van een OCMW kan ondersteunen. Deze applicatie moet in staat zijn om nieuwe mensen aan te maken wanneer zij aan het loket komen, afspraken moeten gepland kunnen worden, er moet een flow doorlopen worden van het begin wanneer een afspraak gemaakt wordt tot deze afspraak afgerond is. Hierna moeten cliënten opgevolgd kunnen worden met de nodige tekstuele mogelijkheden, samen met de mogelijkheid om documenten te uploaden zodat dit later als 1 geheel kan gebruikt worden om naar een grote applicatie van Cipal Schaubroeck te sturen.

Verder in dit document zal ik in detail ingaan op de verwezenlijkingen waaraan ik heb gewerkt in dit project. Dit ga ik doen aan de hand van de resultaten, ondersteund door concrete bewijzen met code waar nodig. Dit alles zodat u als lezer een zo goed mogelijk beeld krijgt over wat ik heb bijgedragen aan dit project.

Het document zal worden opgedeeld in verschillende secties die staan voor de verschillende aspecten van het project. Zo kan ik overal uitleg voorzien om een duidelijk overzicht te geven wat er concreet gemaakt is en een gestructureerde aanpak te handhaven.

Dit verslag dient niet enkel als een showcase van de prestaties die ik heb behaald, maar is een mogelijkheid om de groei en professionele ontwikkeling te tonen die ik tijdens deze stageperiode bereikt heb.

# Realisaties

## Backend

### Datamodel

Het datamodel werd geleidelijk aan opgebouwd, startende vanuit de tabel “Person” omdat dit centraal staat in het gebruik van de applicatie, zeker op het begin omdat we starten met het aanmaken van personen. Om de database aan te maken, gebruikten we Liquibase, een open source library waarmee we aan de hand van XML-bestanden onze database kunnen aanpassen. Een voorbeeld van de bestanden die we met Liquibase hebben gemaakt ziet er zo uit:

```
<changeSet id="OH_17_course_contact_table" author="rbe">
  <createTable tableName="course_contact">
    <column name="id" type="int" autoIncrement="true">
      <constraints primaryKey="true" nullable="false"/>
    </column>
    <column name="created" type="TIMESTAMP" defaultValueComputed="CURRENT_TIMESTAMP">
    </column>
    <column name="last_modified" type="TIMESTAMP" defaultValueComputed="CURRENT_TIMESTAMP">
    </column>
    <column name="name" type="varchar(50)">
    </column>
  </createTable>

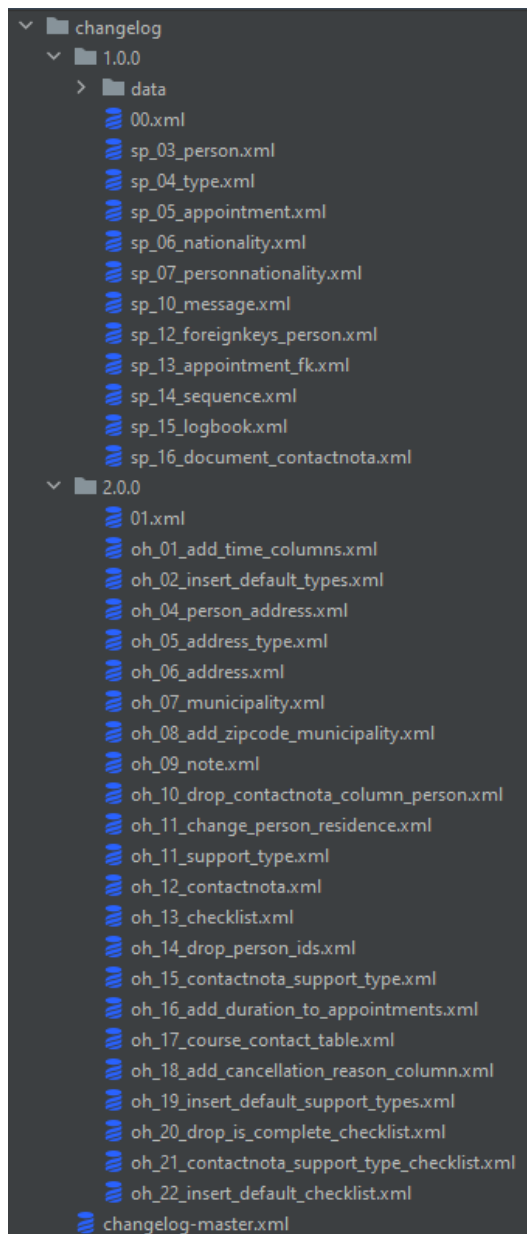
  <addColumn tableName="person">
    <column name="course_contact_id" type="int">
    </column>
    <column name="referred" type="TINYINT(1)">
    </column>
  </addColumn>

  <addForeignKeyConstraint baseTableName="person" baseColumnNames="course_contact_id"
    constraintName="fk_person_course_contact" referencedTableName="course_contact"
    referencedColumnNames="id" onDelete="SET NULL"/>
</changeSet>
```

In een changeSet kunnen verschillende taken meegegeven worden, van het maken van een tabel, tot het aanmaken/verwijderen of aanpassen van kolommen en relaties tussen tabellen.

Dit zorgt ervoor dat we een high level manier hebben om de structuur van de database aan te passen en dat het schrijven van de SQL om dit aan te maken op de achtergrond gebeurt, wat ons heel wat werk bespaart.

Al onze code om de database aan te passen staat in de bestanden in deze afbeelding:



*De bestanden die voornamelijk door mezelf zijn aangepast: sp\_04, sp\_05, sp\_12, sp\_13, sp\_14, oh\_01, oh\_02, oh\_04, oh\_05, oh\_06, oh\_07, oh\_08, oh\_11, oh\_16, oh\_17.*

De bestanden zijn in 2 delen opgesplitst, dit is om aan te duiden waar een grote fase is afgerond in het ontwikkelen.

In de bestanden 00.xml en 01.xml worden alle andere bestanden in hun map geïmporteerd, waarna we ergens anders enkel deze 2 bestanden moeten aanspreken in plaats van elk bestand apart. Dit gebeurt in het onderste bestand, changelog-master.xml. Van hieruit kunnen al deze bestanden dus aangesproken worden, om bij het opstarten na te gaan of deze al zijn verwerkt of niet.

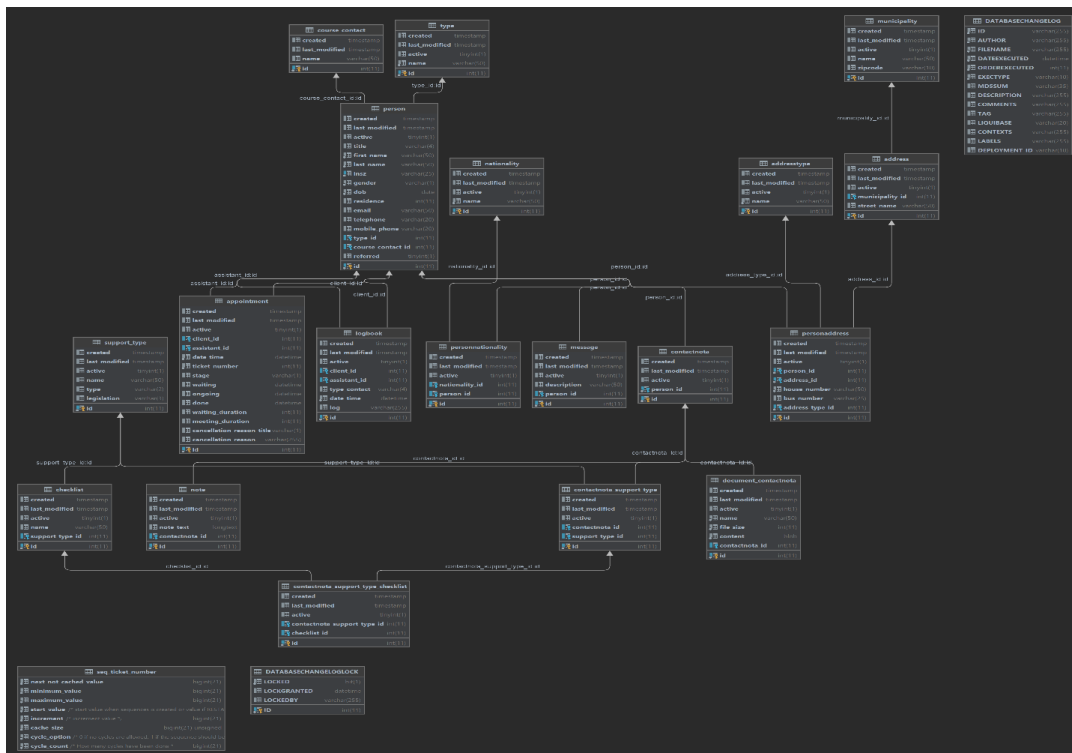
Voor elke aanpassing die we door willen voeren maken we een nieuw bestand aan, zo blijft het ten eerste duidelijk wat er allemaal is aangepast en wanneer dit in de applicatie is

toegevoegd (mits een duidelijke naamgeving), maar ook is dit nodig om problemen met het compileren te vermijden wanneer een collega de applicatie start na het binnenhalen van een nieuwe versie.

De reden hiervoor is dat er een extra databasetabel is genaamd “databasechangelog” die bijhoudt welke XML-bestanden al zijn verwerkt, hierdoor weet de applicatie dat deze bestanden niet nog eens uitgevoerd moeten worden bij het opstarten van de applicatie.

Wanneer we dus verandering zouden uitvoeren in bestaande bestanden en we pushen dit door naar de collega’s, dan zullen deze laatste veranderingen bij hun niet doorgevoerd worden tenzij ze hun databasechangelog leegmaken omdat deze dan wel gestart kunnen worden. Het is dus best practice om elke keer nieuwe veranderingen te maken aan de hand van nieuwe bestanden.

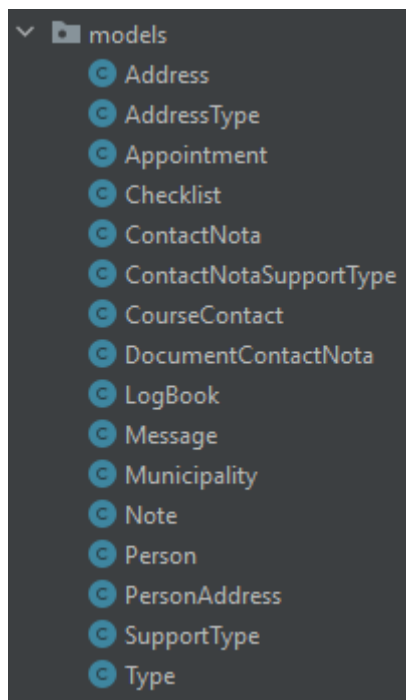
Een overzicht van het volledige databaseschema na het uitvoeren van al deze bestanden ziet er zo uit:



## Models

Om data op te kunnen slaan in de database die nu is aangemaakt, hebben we de models nodig. Dit zijn klassen die we maken in Java, met behulp van het Spring framework. Spring zorgt ervoor dat we met het toevoegen van enkele annotaties veel werk verhuizen naar de achtergrond, zodat wij als developer enkel moeten doen wat nodig is om het te laten werken.

Deze klassen bevatten de data die we ervan kunnen opslaan in de database, maar kunnen ook relaties naar andere klassen bevatten zodat de data uit deze relaties mee kan worden opgehaald wanneer nodig. De klassen die in de applicatie aanwezig waren zijn:



*Klassen die (bijna) uitsluitend door mij zijn aangepast: Address, AddressType, Appointment, CourseContact, Municipality, PersonAddress, Type.*

*Klassen die zowel door mijn collega als mij zijn aangepast: Person.*

*De rest van de klassen zijn dus (bijna) uitsluitend door mijn collega aangepast.*

Een voorbeeld van hoe deze klassen zijn opgebouwd is de klasse “Person”:

```
@Entity
@Getter
@Table(name="person")
@EntityListeners(AuditingEntityListener.class)
@EqualsAndHashCode(callSuper = true)
@Setter
public class Person extends Entry {

    no usages
    @Column(name = "id", unique = true, nullable = false, updatable = false)
    private Long id;
    no usages
    @Column(name = "created", updatable = false)
    private Instant created;
    no usages
    @Column(name = "last_modified")
    private Instant lastModified;

    // @Column(name = "active", length = 1)
    // private Integer active;///  
 type?
    no usages
    @Enumerated(EnumType.STRING)
    private Title title;
    no usages
    @Column(name = "first_name", length = 50, nullable = false)
    private String firstName;
    no usages
    @Column(name = "last_name", length = 50, nullable = false)
    private String lastName;
    no usages
    @Column(name = "insz", length = 25, unique = true, nullable = false)
    private String insz;

    @Enumerated(EnumType.STRING)
    private Gender gender;
    no usages
    @Column(name = "dob")
    private LocalDate dob;
    no usages
    @Column(name = "residence")
    private Integer homeAddressId;
    no usages
    @Column(name = "email", length = 50)
    private String email;
    no usages
    @Column(name = "telephone", length = 50, unique = true)
    private String telephone;
    no usages
    @Column(name = "mobile_phone", length = 50, unique = true)
    private String mobilePhone;
    no usages
    @Column(name = "referred")
    private Boolean referred;
    no usages
    @ManyToOne(targetEntity = Type.class)
    @JoinColumn(name="type_id", referencedColumnName = "id")
    private Type type;
    no usages
    @ManyToOne(targetEntity = CourseContact.class)
    @JoinColumn(name="course_contact_id", referencedColumnName = "id")
    private CourseContact courseContact;
    no usages
    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER)
    private List<Message> messages;
    no usages
    @OneToMany(mappedBy = "personClient")
    @LazyCollection(LazyCollectionOption.FALSE)
    private List<LogBook> logBooks;
    no usages
    @OneToMany(mappedBy = "personClient")
    @LazyCollection(LazyCollectionOption.FALSE)
    private List<Appointment> appointmentsClient;

    @OneToMany(mappedBy = "personAssistant")
    @LazyCollection(LazyCollectionOption.FALSE)
    private List<Appointment> appointmentsAssistant;
    no usages
    @OneToMany(mappedBy = "person")
    @LazyCollection(LazyCollectionOption.FALSE)
    private List<PersonAddress> personAddresses;

    ~ Ruben Tuytens
    public Person() {

    }
}
```

Project Lombok is een Java library die ons in staat stelt om aan de hand van annotaties, veel onnodig werk te besparen. Zo zien we hier boven de klasse Person al meteen 6 annotaties staan die onder andere alle getters en setters genereren, de klasse klaarmaken voor de database of het toevoegen van methodes in de achtergrond. Deze annotaties besparen ons dus enorm veel repetitief werk en zorgt ervoor dat de bestanden minder opgeblazen worden.

Hiernaast zien we bij veel properties de Column-annotatie, die laat weten dat deze properties aan een kolom in de database toebehoren.

Onderaan in de klasse zijn er enkele properties aanwezig met een ManyToOne of OneToMany-annotatie, dit duidt aan dat het om properties gaat met relationele data die dus mee opgehaald kan worden. Bijvoorbeeld bij de eerste ManyToOne-annotatie gaat het over



het type van de persoon waaronder we dan via een JoinColumn-annotatie kunnen zeggen aan welke tabel en kolom dit verbonden moet worden. Wanneer we nu een persoon opvragen zal er in het JSON-object een type-object zitten zonder dat we nog eens extra data moeten opvragen voor het type van de persoon.

In deze klasse zien we bij de kolom gender dat er met een enumeratie gewerkt wordt. Dit houdt in dat we een opsomming aan vaste waarden kunnen opslaan in dit veld. Ik heb een voorbeeld genomen vanuit de klasse Appointment:

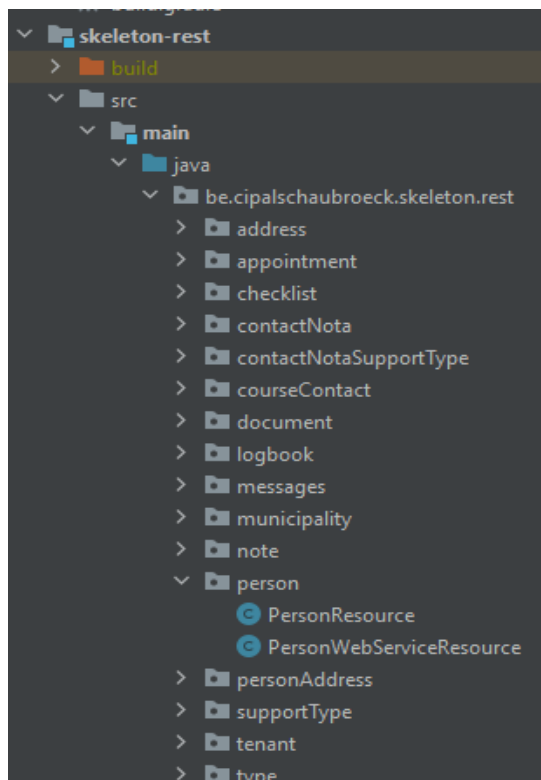
```
public enum StageEnum {  
    no usages  
    U("UPCOMING"), W("WAITING"), P("IN PROGRESS"), D("DONE"), C("CANCELLED");  
    2 usages  
    private final String stage;  
    5 usages 1 rbee  
    StageEnum(String stage) {  
        this.stage = stage;  
    }  
    2 usages 1 rbee  
    public String getStage(){  
        return stage;  
    }  
}
```

Hier is een enumeratie aangemaakt voor de fase waarin de afspraak zich bevindt omdat we hier een vaste lijst aan mogelijkheden willen gebruiken. Op deze manier kunnen we ervoor zorgen dat er geen willekeurige data opgeslagen kan worden in velden waar we voorspelbare data willen hebben.

## Data ophalen - aanvraag

De structuur van de database en de klassen die aangeven welke data hierin zit is nu in orde. Nu moeten we nog een manier voorzien om deze data op te kunnen vragen en over te zetten naar onze frontend. Ook hiervoor gebruiken we het Spring framework om ons te helpen en veel onnodig werk te voorkomen.

Om te starten moeten we een toegangspunt aanmaken dat we kunnen aanroepen vanuit de frontend, waarna de gewenste data opgehaald wordt. Dit gaan we doen aan de hand van REST services die we met behulp van het Spring framework maken. Het project waarin onze REST services staan ziet er zo uit:



*Services die (bijna) uitsluitend door mij zijn aangepast: address, appointment, courseContact, municipality, personAddress, type.*

*Services die zowel door mijn collega als mij zijn aangepast: person.*

*De rest van de services zijn dus (bijna) uitsluitend door mijn collega aangepast.*

We zien hier voor alle klassen waar we data moeten opvragen een folder staan, waarin een bestand staat voor de resource en de webserviceresource. De webserviceresource heeft te maken met autorisatie die buiten onze scope valt, dus hoewel deze aangemaakt is, hebben we hier nog niets mee gedaan en sla ik dit over in de uitleg.

In deze resources kunnen we de annotaties van het Spring framework weer gebruiken om ons een handje te helpen bij het opstellen van deze bestanden, zo beginnen we met een `@RestController`-annotatie die duidelijk maakt dat deze controller zal dienen voor het opvragen van data, en een `@RequestMapping`-annotatie waarin we kunnen meegeven hoe we deze controller moeten aanspreken. Ik zal de `PersonResource` als voorbeeld gebruiken:

```
@RestController
@RequestMapping(value = "${PERSON_BASE_URL}", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
@RequiredArgsConstructor
public class PersonResource extends GenericResource<Person, PersonDTO, PersonService> {

    2 usages
    public static final String PERSON_BASE_URL = AppConstants.API_BASE + "/persons";

    no usages  Ruben Tuytens
    @GetMapping(value = "${}/page/{size}")
    public Page<PersonDTO> getPersonsWithPaging(final @PathVariable int page, final @PathVariable int size) {
        if (size > CSUtil.MAX_PAGE_SIZE) {
            throw new IllegalArgumentException("Max of size is: " + CSUtil.MAX_PAGE_SIZE);
        }
        return this.service.getAllWithPaging(PageRequest.of(page, size, Sort.Direction.ASC, ...properties: "id"));
    }

    no usages  Ruben Tuytens
    @GetMapping(value = "${}/filtered/{page}/{size}")
    public Page<PersonDTO> getPersonsWithPagingAndSearchCriteria(final @PathVariable int page, final @PathVariable int size,
                                                                @RequestParam(value = "filters", required = false)
                                                                @SearchFields final List<SearchCriteria> fields) {
        return this.service.findAllWithFiltersPageable(PageRequest.of(page, size), fields);
    }

    Ruben Tuytens +2
    @GetMapping
    public List<PersonDTO> getAll() {
        List<PersonDTO> personDTOS;
        personDTOS = service.getAll();
        return personDTOS;
    }

    no usages  ipo
    @PostMapping
    @Transactional
    public PersonDTO registerNewPerson(@RequestBody PersonDTO person) {
        return this.service.saveNewPerson(person);
    }
}
```

```
no usages  rbee
@PostMapping("${mainAddress}/")
@Transactional
public PersonDTO registerNewPersonWithMainAddress(@RequestBody PersonDTO person) {
    return this.service.saveNewPersonWithMainAddress(person);
}

no usages  ipo
@DeleteMapping("${insz}")
public void deletePerson(@PathVariable String insz) {
    this.service.deletePerson(insz);
}

no usages  ipo
@GetMapping("${insz}")
@Transactional
public PersonDTO getPersonByInsz(@PathVariable String insz){
    return this.service.getPersonByInsz(insz);
}

no usages  ipo
@GetMapping("${typeName/{type}")
@Transactional
public List<PersonDTO> getPersonyTypeName(@PathVariable String type) {
    return this.service.getPersonByTypeByName(type);
}
}
```

We zien hier dat deze `PersonResource` van een `GenericResource` vertrekt zodat we al wat basisfunctionaliteiten hebben, en we geven mee welke andere bestanden we gebruiken om gebruik te maken van de data die we nodig hebben (`Person`-klasse, `PersonDTO`-klasse en de `PersonService`, hier komen we nog op terug).

Het eerste wat we doen is het adres aanmaken waarop we deze controller kunnen aanspreken, we gebruiken een string uit onze constanten om de API aan te spreken en voegen `"/persons"` hieraan toe om specifiek deze controller te kunnen gebruiken.

Nu moeten we de functies maken die ervoor gaan zorgen dat we de gewenste acties kunnen uitvoeren met de data die we hebben of willen. Hiervoor gebruiken we enkele Spring-annotaties die meteen meegeven wat voor type functie het is.

De annotaties die we hier kunnen gebruiken zijn:

- `@GetMapping`: het ophalen van een object of meerder objecten.
- `@PostMapping`: het opslaan van een nieuw object.
- `@PutMapping`: een bestaand object updaten.
- `@DeleteMapping`: het verwijderen van een object.
- `@PatchMapping`: een deel van een object updaten.

Nu is er meegegeven wat voor type methode het, het volgende dat we moeten meegeven is de logica van wat er met de data moet gebeuren. Zo kunnen we een vrij simpele methode maken zoals de `"getAll"` die dus gewoon alle personen ophaalt. Deze methode spreekt de `personService` aan en haalt daaruit alle objecten op.

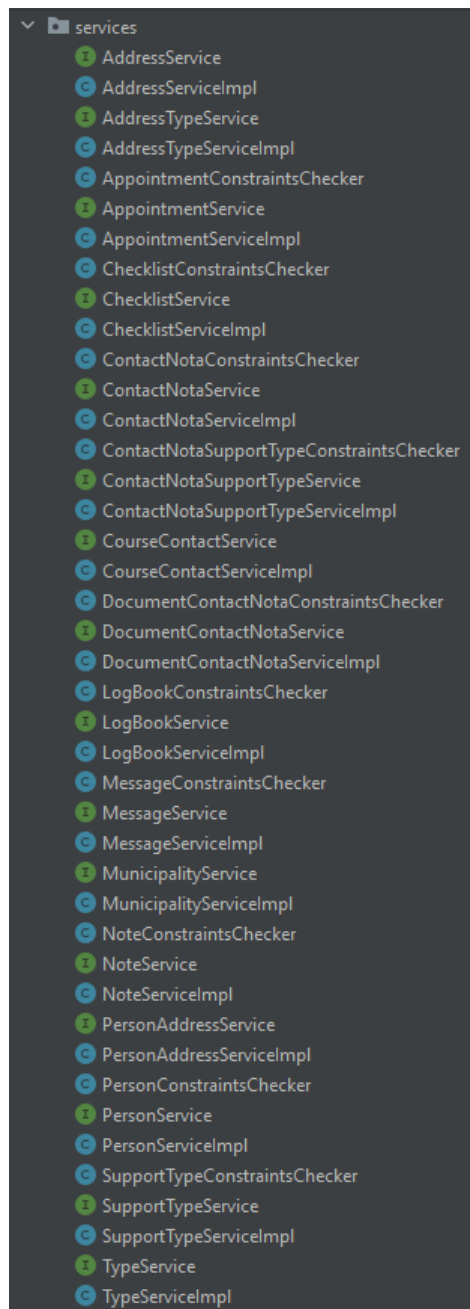
Nog een simpele methode is de `"registerNewPerson"`. Wanneer de methode wordt opgeroepen geven we een `Person`-object mee, deze wordt in de resource opgevangen via de `@RequestBody`-annotatie. Ook hier wordt de `personService` aangesproken om dit object dan op te slaan.

Wanneer we iets extra nodig hebben zoals bij de methode `"deletePerson"`, dan kan dit meegegeven worden via de URL. In dit geval het `insz` (rijksregisternummer). We geven aan dit het `insz` in het pad staat en vangen deze op met annotatie `@PathVariable` waarna we deze waarde kunnen doorgeven naar de `personService` om de persoon te verwijderen.

## Data ophalen - verwerking

We hebben nu via ons toegangspunt in de REST services een aanvraag gedaan en ik heb daarbij uitgelegd hoe deze RestControllers een service hebben die ze aanspreken. Deze service wordt dan opgeroepen in de functies van de controller, daar ga ik nu iets verder op in.

Voor de klassen in onze applicatie hebben we een interface en een implementatie van de interface gemaakt om alles overzichtelijk te houden:



Eerst beginnen we met het maken van de interface, in deze interface zetten we de namen van de methodes, samen met wat ze ontvangen of teruggeven. Een voorbeeld hiervan is de AddressService:

```
public interface AddressService extends BaseService<Address, AddressDTO> {  
  
    1 usage 1 implementation 1 rbee  
    AddressDTO saveAddress(AddressDTO addressDTO);  
  
    1 usage 1 implementation 1 rbee  
    List<AddressDTO> getAddressesByMunicipalityById(Long id);  
}
```

Hier zien we twee methodes staan, één ervan gaat gebruikt worden om een adres op te slaan, de andere om alle adressen op te halen die tot een bepaalde gemeente behoren. Omdat dit een interface is moeten we deze functies ergens anders uitwerken uiteraard, dit gebeurt in de AddressServiceImpl:

```
@Service  
@Slf4j  
@RequiredArgsConstructor  
public class AddressServiceImpl extends GenericServiceImpl<Address, AddressDTO, AddressRepository> implements AddressService {  
  
    2 usages  
    private final MunicipalityService municipalityService;  
    1 usage  
    private final MunicipalityMapper municipalityMapper;  
  
    1 usage 1 rbee  
    public AddressDTO saveAddress(AddressDTO addressDTO) {  
        Address address = saveMunicipalityForAddress(addressDTO);  
        return this.mapper.entityToDto(repository.save(address));  
    }  
  
    1 usage 1 rbee  
    @Override  
    public List<AddressDTO> getAddressesByMunicipalityById(Long id) {  
        MunicipalityDTO municipalityDTO = this.municipalityService.findById(id).get();  
        return this.mapper.entitiesToDtos(repository.findAddressesByMunicipality(municipalityMapper.dtoToEntity(municipalityDTO)));  
    }  
  
    1 usage 1 rbee  
    private Address saveMunicipalityForAddress(AddressDTO addressDTO){  
        MunicipalityDTO savedMunicipalityDTO = this.municipalityService.saveMunicipality(addressDTO.getMunicipality());  
        addressDTO.setMunicipality(savedMunicipalityDTO);  
        return this.mapper.dtoToEntity(addressDTO);  
    }  
}
```

We zien dat deze implementatie naast de klassen Address en AddressDTO, ook nog gebruik maakt van de AddressRepository. In deze repositories gaat de feitelijke actie worden uitgevoerd worden om de data op te slaan. Om deze repositories op te bouwen maken we gebruik van Hibernate, dit is een open source framework waarmee we makkelijk met ORM kunnen werken.

Als voorbeeld zal ik de PersonRepository tonen:

```
public interface PersonRepository extends BaseEntryRepository<Person>, QuerydslPredicateExecutor<Person> {  
    1 usage  ↳ ipo  
    List<Person> findPersonByType (Type type);  
    1 usage  ↳ Ruben Tuytens  
    Person findByInsz(String insz);  
    1 usage  ↳ Ruben Tuytens  
    void deleteByInsz(String insz);  
}
```

In deze repository zien we methodes staan die uit enkele keywords zijn opgebouwd, dit komt omdat Hibernate zelf de mogelijke opties weergeeft tijdens het maken van deze methodes. Neem de methode “findByInsz” even als voorbeeld, wanneer we een object willen zoeken, beginnen we met “findBy”, hierna gaat Hibernate zelf de mogelijke opties geven waarop we kunnen zoeken. Deze opties worden gegenereerd omdat de klasse wordt gebruikt, hierdoor staan alle properties van de klasse tussen de opties om op te zoeken. Zo zal er na het typen van “findBy” voor elke property van de klasse Person een suggestie worden gedaan “findBy<property>”.

Dit zorgt ervoor dat we enorm snel nieuwe methodes kunnen toevoegen zonder dat we zelf de achterliggende code moeten schrijven, een enorme hulp om te hebben dus. Nu we weten hoe onze repositories worden opgebouwd kunnen we terug naar de logica in de ServiceImpl.

Het eerste wat we doen is andere services of mappers aanspreken die we nodig kunnen hebben in onze methodes.

Hierna implementeren we de methodes die in onze interface te vinden zijn. Naast deze twee methodes is er nog een derde te vinden omdat we voor de clean code wat hebben opgesplitst zodat we niet met een te grote functie eindigen die verschillende zaken doet.

De methode “saveAddress” toont ook meteen dat we een soort van ketting kunnen maken tussen verschillende services. De methode wordt zelf al aangeroepen vanuit een andere service (al zien we dat hier niet), en dan wordt in deze methode nog eerst de municipalityService aangesproken. Dit doen we omdat een object van klasse Address een object van klasse Municipality bevat, maar de municipality moet uiteraard eerst bestaan voordat een adres opgeslagen kan worden, dit lossen we dus op deze manier op, door eerst het ene op te slaan vooraleer we met de huidige opdracht verder gaan. Hierna wordt de mapper aangesproken (op de mappers kom ik nog terug) waarna de repository wordt aangesproken om het object uiteindelijk op te slaan.

Bij de tweede methode “getAddressessByMunicipalityById” zien we ook eerst weer terugkomen dat we een andere service aanspreken om een object aan te vragen, waarna we dit object gebruiken met een methode die in de repository is opgesteld. Met Hibernate

hebben we de methode “findAddressessByMunicipality” kunnen opstellen, wat ons werk hierna enorm veel makkelijker maakt om nu objecten op te halen.

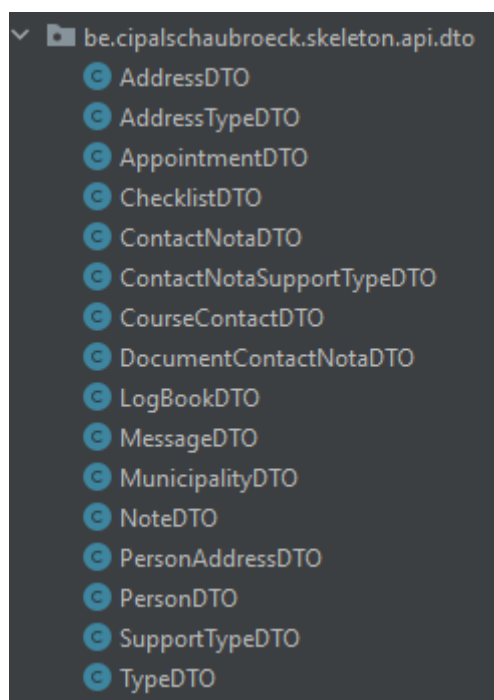
### Data ophalen - mapping

In het vorige deel zagen we telkens dat er een mapper werd aangesproken voordat we iets naar de repository stuurden om op te slaan, maar wat zijn die mappers nu juist? Daar ga ik nu meer duidelijkheid over geven.

Ondertussen heeft u al regelmatig een DTO-klasse gezien waar ook een gewone klasse gebruikt wordt, bijvoorbeeld Person en PersonDTO. De letters DTO staan voor “Data Transfer Object” of met andere woorden, het object dat van client naar server wordt gestuurd of omgekeerd.

De reden dat dit designpatroon heel nuttig is, is omdat we op deze manier kunnen bepalen welke waarden wel of niet worden meegestuurd vanuit de database. Met andere woorden, dit is een kleine tussenstap in de backend waarin een object wordt omgevormd (grotendeels gekopieerd) om naar de frontend te sturen. Dit zorgt er overigens ook voor dat er minder verkeer tussen client en server gebeurt, en we kunnen dit gebruiken om stackoverflows te breken, dit zien we zo meteen terugkomen.

Om dit te gebruiken maken we dus voor elke klasse die we in de database gebruiken, ook een DTO-klasse aan:





Als we PersonDTO openen dan zien we dus al dat de property “id” achterwege wordt gelaten omdat we deze niet nodig hebben, alsook de properties “created” en “last\_updated” want dit is ook nog niet aan de orde gekomen:

```
public class PersonDTO extends EntryDTO {  
    no usages  
    private String title;  
    1 usage  
    private String firstName;  
    1 usage  
    private String lastName;  
    no usages  
    private String insz;  
    no usages  
    private String gender;  
    no usages  
    private LocalDate dob;  
    no usages  
    private Integer homeAddressId;  
    no usages  
    private String email;  
    no usages  
    private String telephone;  
    no usages  
    private String mobilePhone;  
    no usages  
    private TypeDTO type;  
    no usages  
    private Boolean referred;  
    no usages  
    private CourseContactDTO courseContact;  
    no usages  
    private List<MessageDTO> messages;  
    no usages  
    private List<LogBookDTO> logBooks;  
    no usages  
    private List<AppointmentDTO> appointmentsClient;  
    no usages  
    private List<AppointmentDTO> appointmentsAssistant;  
    no usages  
    private List<PersonAddressDTO> personAddresses;  
    no usages  
    Ruben Tuytens
```

Zo ziet de DTO-klasse er dus uit, op zich zit dit enorm simpel in elkaar. Het enige waar goed naar gekeken moet worden is dat de juiste properties meegegeven worden. Nu het principe van de DTO's is uitgelegd, kan ik verder uitleggen hoe de mappers in elkaar zitten.

Wanneer we kijken naar het aanmaken van een mapper, dan zien we dat we overerven van een GenericMapper:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.ERROR)  
public abstract class PersonMapper extends GenericMapper<Person, PersonDTO> {
```

Wanneer we hierop verdergaan dan zien we dat dit gebruik maakt van de GenericIdentifiableMapper:

```
Mapstruct mapper to map Identifiable identities.
Type parameters<ENTITY>
    <DTO>

public abstract class GenericIdentifiableMapper<ENTITY extends Identifiable, DTO extends IdentifiableDTO> {
    public abstract ENTITY dtoToEntity(DTO dto);

    public abstract ENTITY merge(DTO dto, @MappingTarget ENTITY entity);

    public abstract ENTITY clone(ENTITY entity);

    @Mappings({
        @Mapping(target = "id", ignore = true)
    })
    public abstract DTO cloneDTO(DTO entity);

    @InheritInverseConfiguration(name = "dtoToEntity")
    public abstract DTO entityToDto(ENTITY entity);

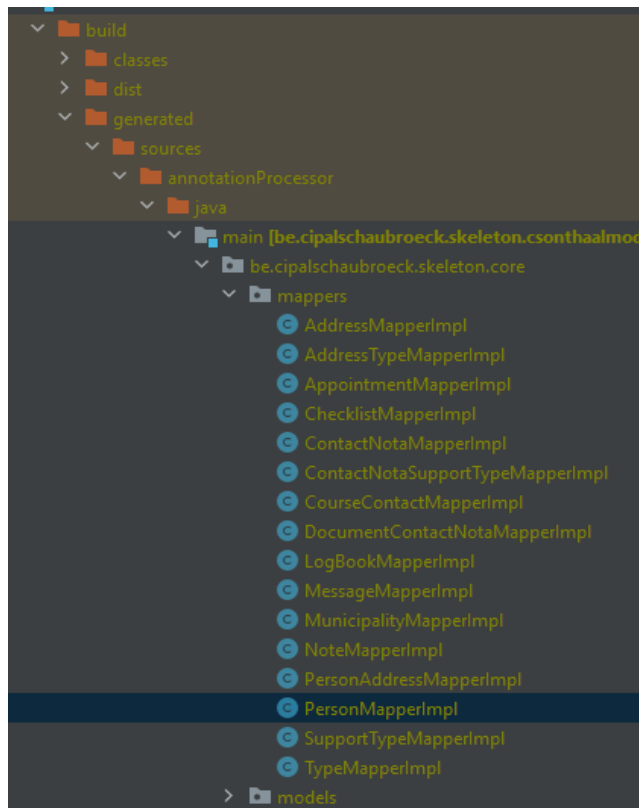
    public abstract List<ENTITY> dtosToEntities(List<DTO> dtos);

    @InheritInverseConfiguration
    public abstract List<DTO> entitiesToDtos(List<ENTITY> entities);
}
```

Dit gebruiken we omdat we met Mapstruct werken. Mapstruct is een library die code genereert, dit wil zeggen dat wanneer dat applicatie compileert, deze mappers automatisch aangemaakt worden en we dit dus niet zelf overal expliciet moeten typen. Dit is enorm handig aangezien bijna alle mappings perfect hetzelfde zijn opgebouwd.

We zien in dit bestand al enkele methodes staan maar het zijn vooral de methodes die entiteiten naar DTO's veranderen of omgekeerd waar wij naar kijken. De namen zeggen al wat deze doen, ze mappen een object van de server (een entiteit) naar een object dat voor de client is (een DTO) of omgekeerd. Deze methodes zagen we al meerdere malen tevoorschijn komen in de serviceImpl, nu zien we dus van waar deze komen.

Na het opstarten kunnen we dus kijken wat er allemaal aangemaakt is op deze manier:



We gaan eens kijken wat er in zulke mappers te vinden is:

```
@Component
public class PersonMapperImpl extends PersonMapper {

    @Override
    public Person dtoToEntity(PersonDTO arg0) {
        if ( arg0 == null ) {
            return null;
        }

        Person person = new Person();

        person.setId( arg0.getId() );
        person.setCreated( arg0.getCreated() );
        person.setLastModified( arg0.getLastModified() );
        if ( arg0.getTitle() != null ) {
            person.setTitle( Enum.valueOf( Title.class, arg0.getTitle() ) );
        }
        person.setFirstName( arg0.getFirstName() );
        person.setLastName( arg0.getLastName() );
        person.setInsz( arg0.getInsz() );
        if ( arg0.getGender() != null ) {
            person.setGender( Enum.valueOf( Gender.class, arg0.getGender() ) );
        }
        person.setDob( arg0.getDob() );
        person.setHomeAddressId( arg0.getHomeAddressId() );
        person.setEmail( arg0.getEmail() );
        person.setTelephone( arg0.getTelephone() );
        person.setMobilePhone( arg0.getMobilePhone() );
        person.setReferred( arg0.getReferred() );
        person.setType( typeDTOToType( arg0.getType() ) );
        person.setCourseContact( courseContactDTOToCourseContact( arg0.getCourseContact() ) );
        person.setMessages( messageDTOListToMessageList( arg0.getMessages() ) );
        person.setLogBooks( logBookDTOListToLogBookList( arg0.getLogBooks() ) );
        person.setAppointmentsClient( appointmentDTOListToAppointmentList( arg0.getAppointmentsClient() ) );
        person.setAppointmentsAssistant( appointmentDTOListToAppointmentList( arg0.getAppointmentsAssistant() ) );
        person.setPersonAddresses( personAddressDTOListToPersonAddressList( arg0.getPersonAddresses() ) );

        return person;
    }
}
```

We zien hier dat dit eigenlijk een vrij simpel concept is. Dit is de functie “dtoToEntity”, de functie krijgt een object van klasse PersonDTO aan, waarna een object van klasse Person wordt aangemaakt en elke property van de DTO wordt overgezet naar de Person. In feite gebeurt hier niet veel meer dan het maken van een kopie van een object.

Nu kan dit ook wel eens mislopen, deze werking neemt enkel een kopie van de data die erin zit, maar dit wil dus ook echt alles overnemen. Onderaan de laatste methode zien we dat de properties appointmentsClient en appointmentsAssistant worden opgevuld in de mapper. Deze appointmentobjecten bevatten ook een persoon, deze persoon bevat weer appointments...

Wanneer we dit laten compileren, krijgen we een stackoverflow omdat dit eindeloos door kan blijven gaan, dit is uiteraard niet wat we willen dus we moeten dit zien op te lossen. Om dit probleem aan te pakken gaan we in onze eigen mapper een methode toevoegen. Deze methode zal de stackoverflow aanpakken door bepaalde gegevens uit het DTO weg te halen zodat de cyclus wordt onderbroken. We starten met dit toe te voegen in de mapper:

```
@IterableMapping(qualifiedByName = "mapWithoutPerson2")
public List<AppointmentDTO> appointmentListToAppointmentDTOList(List<Appointment> list) {
    if (list == null) {
        return null;
    }
    List<AppointmentDTO> list2 = new ArrayList<>(list.size());
    for (Appointment appointment : list) {
        list2.add(mapWithoutPerson2(appointment));
    }
    return list2;
}
```

We willen over de lijst van appointments gaan en voor elke appointment de functie “mapWithoutPerson2” uitvoeren, we willen dus ergens de persoon als property weghalen.

Het probleem bevindt zich dus wanneer we een persoon ophalen, dat de afspraken van de persoon worden opgehaald, waarna de personen in de afspraak weer worden opgehaald. Die tweede keer dat personen worden opgehaald, daar willen we het onderbreken.

Wat we kunnen doen is dus een nieuwe functie schrijven die de gegenereerde code overschrijft, waarin we voor de twee personen in de afspraak zelf een object maken, aan deze objecten geven we de nodige informatie mee, maar wat we zeker moeten weglaten is de lijst van afspraken van deze personen, daar onderbreken we de cyclus dus. Het object van klasse AppointmentDTO geven we nog al zijn properties, behalve dat we de twee personen meegeven die we net hebben aangemaakt in plaats van de standaardobjecten:

```

@Named("mapWithoutPerson2")
public AppointmentDTO mapWithoutPerson2(Appointment appointment) {
    if (appointment == null) {
        return null;
    }
    PersonDTO personClientDTO = new PersonDTO();
    personClientDTO.setId(appointment.getPersonClient().getId());
    personClientDTO.setInsz(appointment.getPersonClient().getInsz());
    personClientDTO.setFirstName(appointment.getPersonClient().getFirstName());
    personClientDTO.setLastName(appointment.getPersonClient().getLastName());
    personClientDTO.setTitle(appointment.getPersonClient().getTitle().name());
    personClientDTO.setDob(appointment.getPersonClient().getDob());

    PersonDTO personAssistantDTO = new PersonDTO();
    personAssistantDTO.setId(appointment.getPersonAssistant().getId());
    personAssistantDTO.setInsz(appointment.getPersonAssistant().getInsz());
    personAssistantDTO.setFirstName(appointment.getPersonAssistant().getFirstName());
    personAssistantDTO.setLastName(appointment.getPersonAssistant().getLastName());
    personAssistantDTO.setTitle(appointment.getPersonAssistant().getTitle().name());
    personAssistantDTO.setDob(appointment.getPersonAssistant().getDob());

    AppointmentDTO appointmentDTO = new AppointmentDTO();

    appointmentDTO.setId(appointment.getId());
    appointmentDTO.setCreated(appointment.getCreated());
    appointmentDTO.setLastModified(appointment.getLastModified());

    appointmentDTO.setPersonClient(personClientDTO);
    appointmentDTO.setPersonAssistant(personAssistantDTO);
    appointmentDTO.setDateTime(appointment.getDateTime());
    appointmentDTO.setTicketNumber(appointment.getTicketNumber());
    if (appointment.getStage() != null) {
        appointmentDTO.setStage(appointment.getStage().name());
    }
}

```

Nu hebben we alle stappen overlopen om data op te halen uit de backend wanneer we dit nodig hebben. De database is gemaakt, de klassen zijn aangemaakt en de nodige relaties zijn aanwezig. De vereiste services en methodes om deze data op te vragen zijn aanwezig en de data wordt hierna omgevormd naar DTO's zodat deze op het einde via de REST services eindelijk terug naar de frontend gestuurd kunnen worden.

Nu dat dit allemaal is uitgelegd hoe ik het gemaakt heb, kan ik overschakelen naar de frontend om u te tonen wat we daar hebben verwezenlijkt.

## Frontend

### Introductie frontend

De technologie die we gebruiken voor de frontend is Angular. Angular is een framework dat gebouwd is op JavaScript waardoor we dynamische webapplicaties kunnen maken. Door het gebruik van componenten kunnen we hier gestructureerd mee te werk gaan waardoor het ook goed onderhoudbaar is.

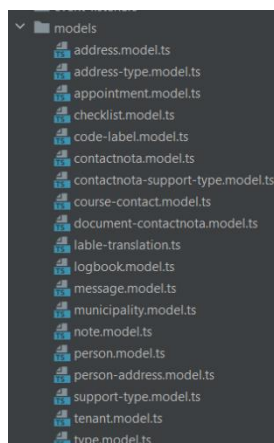
Naast de componenten die we zelf maken, hebben we gebruik gemaakt van het PrimeN framework. Dit is een UI-framework dat vele componenten bevat die we zo kunnen implementeren in onze applicatie door het schrijven van enkele lijnen code.

Voor het tonen van de realisaties die we in de frontend hebben gemaakt, ga ik de applicatie afbakenen in bepaalde modules en deze één voor één uitleggen aan de hand van afbeeldingen hoe het component eruit ziet, alsook wat code die voor dat deel belangrijk was om het te laten werken. Om te starten begin ik met het tonen van de klassen in de frontend en toon ik ook nog hoe de backend wordt aangesproken vanuit de frontend. Hierna zijn we klaar om de rest van de applicatie te tonen.

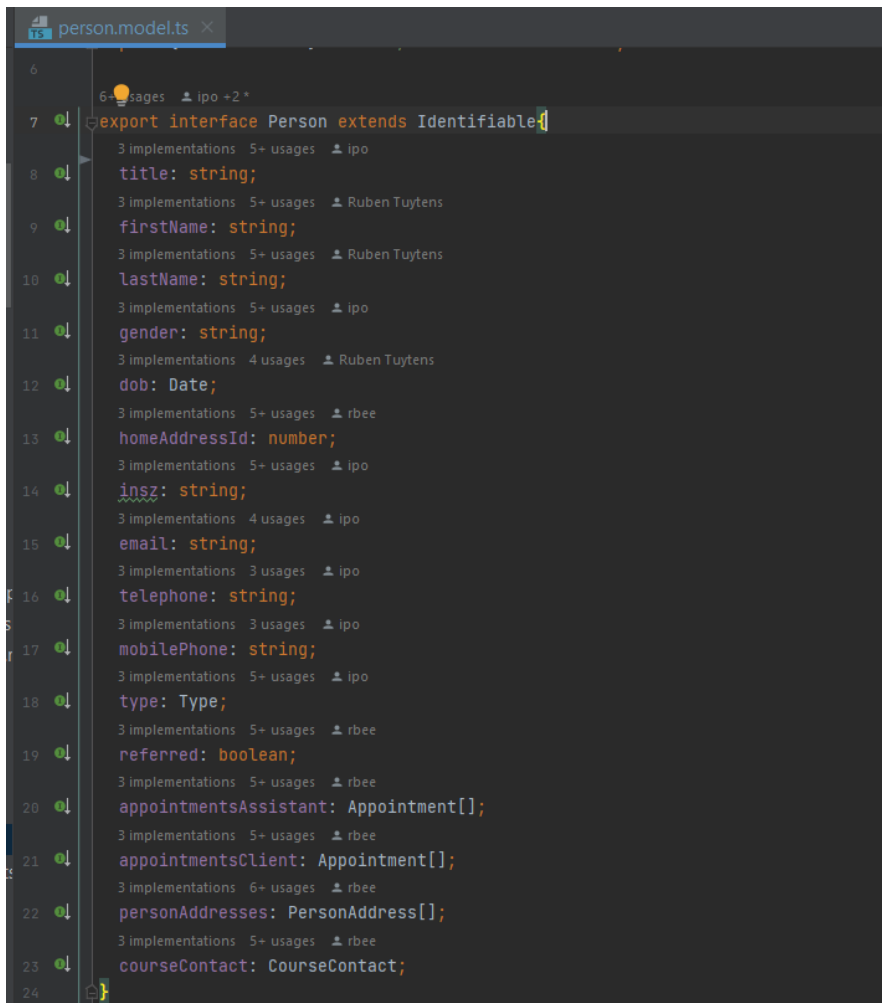
### Models

Angular is het meest krachtig wanneer we correct aanduiden met wat voor data we zijn aan het werken, net zoals we bij gewone variabelen aanduiden of het bijvoorbeeld een nummer of een string is, kunnen we zeggen dat een variabele bijvoorbeeld van de klasse Person is. Als deze klasse correct is opgesteld kunnen we makkelijk aan de properties van een object. Stel dat we een variabele “person” hebben van klasse Persoon, dan kunnen we snel aan de properties van het object door “person.firstName” te typen.

Dit houdt dus in dat we in de frontend dezelfde klassen nodig zullen hebben als in de backend:



Laten we even kijken wat er in de klasse Person te vinden is:



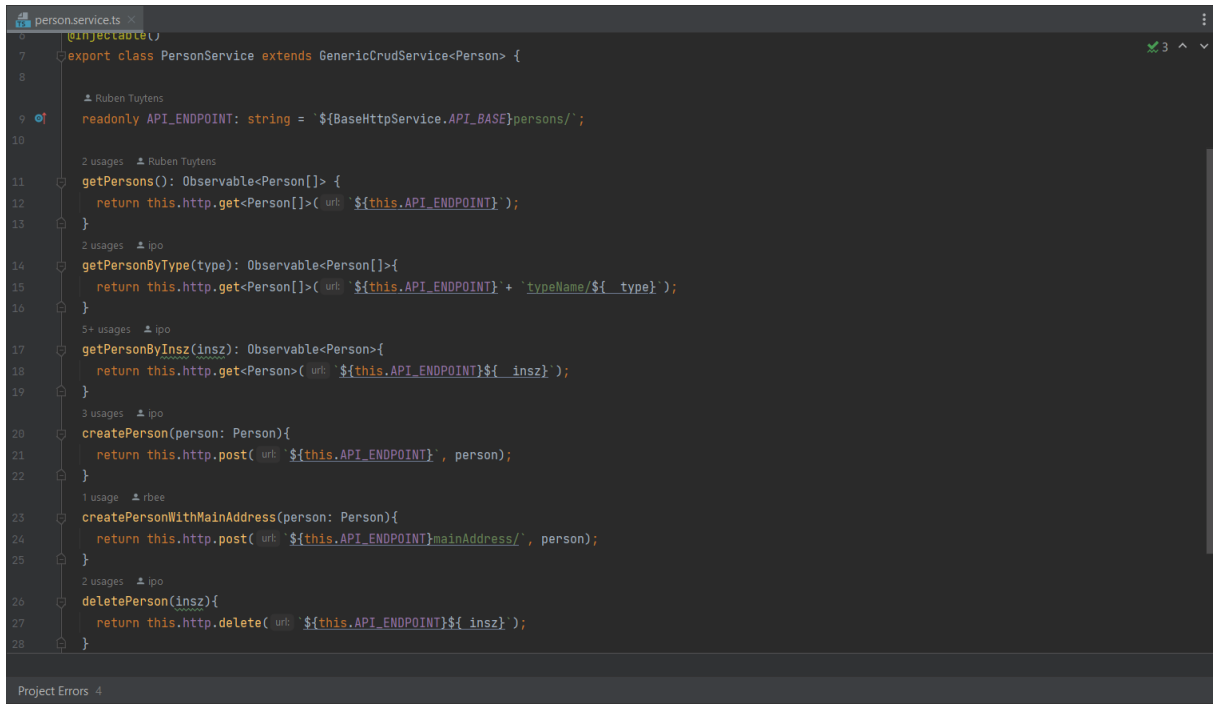
```
6
7 export interface Person extends Identifiable {
8   title: string;
9   firstName: string;
10  lastName: string;
11  gender: string;
12  dob: Date;
13  homeAddressId: number;
14  insz: string;
15  email: string;
16  telephone: string;
17  mobilePhone: string;
18  type: Type;
19  referred: boolean;
20  appointmentsAssistant: Appointment[];
21  appointmentsClient: Appointment[];
22  personAddresses: PersonAddress[];
23  courseContact: CourseContact;
24 }
```

The screenshot shows a code editor with a file named 'person.model.ts'. The code defines an interface 'Person' that extends 'Identifiable'. The interface has the following properties: 'title' (string), 'firstName' (string), 'lastName' (string), 'gender' (string), 'dob' (Date), 'homeAddressId' (number), 'insz' (string), 'email' (string), 'telephone' (string), 'mobilePhone' (string), 'type' (Type), 'referred' (boolean), 'appointmentsAssistant' (Appointment[]), 'appointmentsClient' (Appointment[]), 'personAddresses' (PersonAddress[]), and 'courseContact' (CourseContact). Each property is followed by a tooltip showing the number of implementations and usages, along with the name of the user who made the change.

We zien dat hier dezelfde properties zijn aangemaakt die we in PersonDTO hebben, zo is het duidelijk dat we deze klasse maken om de DTO-objecten op te vangen, waarna we deze doorheen de frontend kunnen gebruiken.

## Services

Om de data op te halen die in onze database zit, moeten we nog iets voorzien om de juiste eindpunten aan te spreken. Dit doen we aan de hand van services. Zo een service bouwen we op een `GenericCrudService` waardoor we toegang krijgen tot de standaard acties die we willen uitvoeren, dit kan gaan om bijvoorbeeld “get” om data te halen, “post” om iets te sturen, of “delete” om iets te verwijderen. Een voorbeeld van deze services ziet er zo uit:



```
6  @injectable()
7  export class PersonService extends GenericCrudService<Person> {
8
9      Ruben Tuytens
10     readonly API_ENDPOINT: string = `${BaseHttpService.API_BASE}persons/`;
11
12     2 usages Ruben Tuytens
13     getPersons(): Observable<Person[]> {
14         return this.http.get<Person[]>({ url: `${this.API_ENDPOINT}` });
15     }
16
17     2 usages ipo
18     getPersonByType(type): Observable<Person[]> {
19         return this.http.get<Person[]>({ url: `${this.API_ENDPOINT} + 'typeName/${type}'` });
20     }
21
22     5+ usages ipo
23     getPersonByInsz(insz): Observable<Person> {
24         return this.http.get<Person>({ url: `${this.API_ENDPOINT}${insz}` });
25     }
26
27     3 usages ipo
28     createPerson(person: Person) {
29         return this.http.post(`${this.API_ENDPOINT}`, person);
30     }
31
32     1 usage rbee
33     createPersonWithMainAddress(person: Person) {
34         return this.http.post(`${this.API_ENDPOINT}mainAddress/`, person);
35     }
36
37     2 usages ipo
38     deletePerson(insz) {
39         return this.http.delete(`${this.API_ENDPOINT}${insz}`);
40     }
41 }
```

Project Errors 4

Bovenaan stellen we een string in als variabele, het eindpunt dat we achteraan toevoegen moet overeen komen met die van de resource in de backend. Op deze manier kunnen we de connectie maken tussen de acties die we hier versturen en waar ze moeten aankomen op de backend.

Wanneer de methodes hier overeenkomen op vlak van pad en de data die ze eventueel ontvangen/versturen, dan zijn deze methodes klaar om gebruikt te worden doorheen onze applicatie.

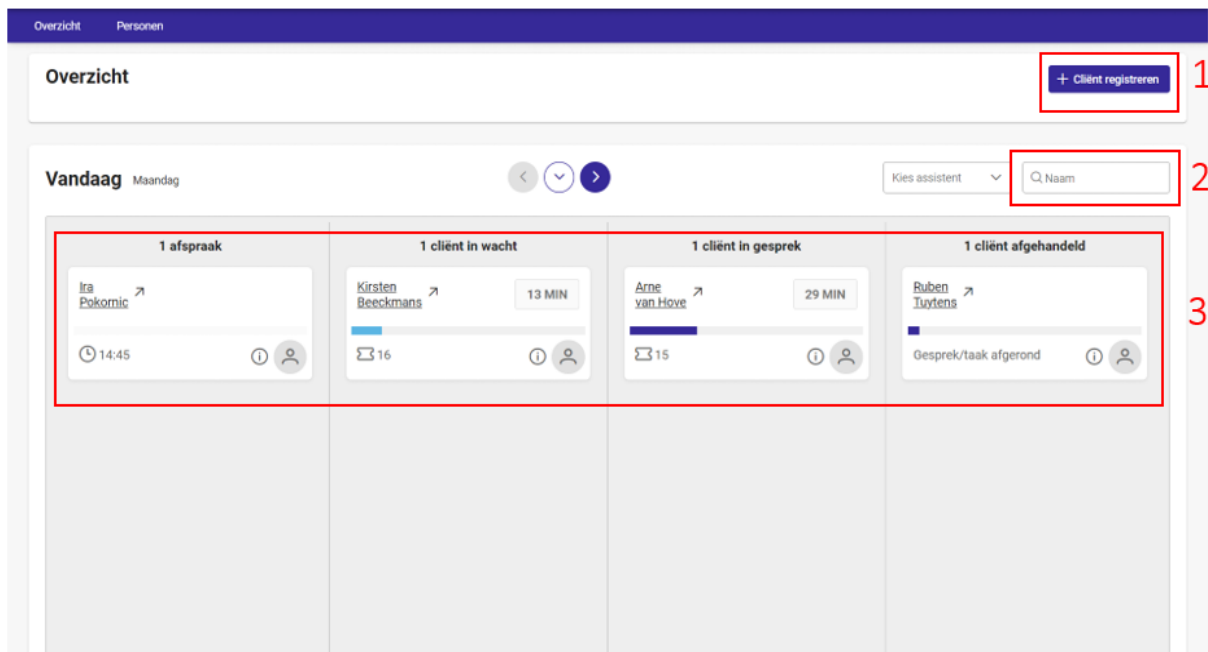
Nu zijn we eindelijk helemaal rond met het klaarzetten van alles wat we nodig hebben om de componenten op te bouwen die onze frontend gaan vormen. Ik ga deel per deel van de applicatie overlopen wat er functioneel moest gebeuren, wat ik hierin heb gemaakt en ik ga de belangrijke code tonen die het allemaal laat werken.



## Overzicht afspraken

Wanneer een gebruiker de applicatie start krijgen zij een overzicht te zien waar alle afspraken van vandaag op te zien zijn. Op dit overzicht kan de gebruiker de planning van de afspraken bekijken en beheren. Deze zal vooral gebruikt worden om de flow te doorlopen waarin een cliënt een afspraak heeft, waarna deze aankomt, hierna in gesprek gaat met een maatschappelijk assistent waarna de afspraak is afgerond.

De algemene opbouw van de pagina is door mezelf voorzien, alsook de items die in het rood zijn aangeduid. De filter op assistent en de pijltjes om van dag te wisselen zijn niet door mij gemaakt.



**1:** Aangezien de gebruiker vaak op deze pagina werkt, willen we een snelle manier voorzien om nieuwe cliënten aan te maken. Deze knop herleidt de gebruiker naar de pagina om nieuwe personen aan te maken.

**2:** Wanneer de applicatie in gebruik wordt genomen kan het zijn dat er heel veel afspraken op een dag gepland zijn. Via dit inputveld willen we kunnen filteren op de naam van een cliënt om een afspraak snel terug te vinden.

Opbouw inputveld:

```
<div class="p-float-label my-3 pr-2">
  <input id="float-input" type="text" pInputText [(ngModel)]="filterText" (keyup)="filterAppointments()">
  <label for="float-input">
    <i class="pi pi-search"></i> {{'DASHBOARD.GRID.Name' | translate}}
  </label>
</div>
```

Functies worden uitgevoerd na het ingeven van tekst:

```
filterAppointments() {  
  this.emptyAllColumns();  
  this.sliceAppointmentsByFilterText();  
  this.splitAppointmentsByStage(this.filteredAppointments);  
}
```

Deze functie wordt opgeroepen bij elke input op het toetsenbord. Eerst wordt een functie opgeroepen die de bestaande lijsten leegmaakt. Daarna wordt de filtering toegepast, hier ga ik zo meteen op verder. Als laatste stap worden de gefilterde afspraken verdeeld over de kolommen door `splitAppointmentsByStage()`.

Functie die de filtering toepast:

```
sliceAppointmentsByFilterText() {  
  if (this.filterText && this.allAppointmentsForAssistants) {  
    const tempList = this.allAppointmentsForAssistants.slice();  
    this.filteredAppointments = tempList.filter(item => item.personClient.firstName.toLowerCase().includes(this.filterText.toLowerCase()));  
  
    return;  
  }  
  this.filteredAppointments = this.allAppointmentsForAssistants;  
}
```

**3:** De verschillende fases van een afspraak. Ik heb met behulp van Drag&Drop-functionaliteiten een scherm gemaakt waarin we de afspraken kunnen beheren. Deze functionaliteit is gemaakt met de Angular CDK waarvan Drag&Drop een onderdeel is.

Als eerste stap wilden we een Drag&Drop module van PrimeNG gebruiken maar deze bleek al snel te beperkt voor onze doeleinden. Hier was geen verkeer in beide richtingen mogelijk en dit leek vooral visueel te werken, niet met functies die wij voor ogen hadden. Ik heb dus nog wat onderzoek gedaan en zo zijn we bij de Angular CDK terecht gekomen die veel meer mogelijkheden heeft.

Om deze te gebruiken maken we lijsten aan op deze manier:

```
<div cdkDropList #listOne="cdkDropList" [cdkDropListConnectedTo]="[listTwo]" [cdkDropListData]="columnAppointments" class="example-list"  
  (cdkDropListDropped)="drop($event, constants.Stages.Upcoming)">  
  <app-overview-card class="my-2 mb-1" *ngFor="let item of columnAppointments" cdkDrag [cdkDragData]="item" [appointment]="item"></app-overview-card>  
</div>
```

Dit staat ons toe om deze te verbinden met andere lijsten om de componenten te kunnen slepen van de ene naar de andere. Wanneer dit gebeurt wordt het event genaamd “`cdkDropListDropped`” opgeroepen waarna de volgende functie wordt uitgevoerd:

```

drop(event: CdkDragDrop<Appointment[], any>, listName: string) {
  if (event.previousContainer === event.container) {
    moveItemInArray(event.container.data, event.previousIndex, event.currentIndex);

    return;
  }
  transferArrayItem(
    event.previousContainer.data,
    event.container.data,
    event.previousIndex,
    event.currentIndex
  );
  const cardWithChangedStage = this.changeAppointmentStage(listName, event.item.data);
  this.saveAppointmentAndDuration(cardWithChangedStage, event.previousContainer.id);
}

```

Deze functie is grotendeels al door de CDK voorzien van code om de verplaatsing van elementen te faciliteren (van het begin tot na `transferArrayItem`) waarna ik zelf nog wat logica heb toegevoegd. Eerst maken we een nieuw object waarin we de fase gaan aanpassen naargelang waar het element naartoe is gesleept:

```

changeAppointmentStage(listName: string, draggedCard: Appointment) {
  switch (listName) {
    case CONSTANTS.Stages.Upcoming:
      draggedCard.stage = CONSTANTS.Stages.Upcoming;
      break;
    case CONSTANTS.Stages.Waiting:
      draggedCard.stage = CONSTANTS.Stages.Waiting;
      draggedCard.waiting = new Date();
      break;
    case CONSTANTS.Stages.Ongoing:
      draggedCard.stage = CONSTANTS.Stages.Ongoing;
      draggedCard.ongoing = new Date();
      break;
    case CONSTANTS.Stages.Done:
      draggedCard.stage = CONSTANTS.Stages.Done;
      draggedCard.done = new Date();
      break;
    default:
      draggedCard.stage = CONSTANTS.Stages.Upcoming;
  }

  return draggedCard;
}

```

Hierna veranderen we nog wat properties die te maken hebben met de moment waarop ze zijn versleept omdat we dit voor andere functionaliteiten kunnen gebruiken:

```

saveAppointmentAndDuration(draggedCard: Appointment, listName: string){
  const updatedAppointment = this.setDurationProperty(draggedCard, listName);
  this.appointmentService.updateAppointment(updatedAppointment).subscribe();
}

1 usage  rbee
setDurationProperty(draggedCard: Appointment, listName: string){
  if(listName === CONSTANTS.DropDownLists.List2){
    draggedCard.waitingDuration += this.getTimeInStage(draggedCard.waiting);
  }
  if(listName === CONSTANTS.DropDownLists.List3){
    draggedCard.meetingDuration += this.getTimeInStage(draggedCard.ongoing);
  }

  return draggedCard;
}

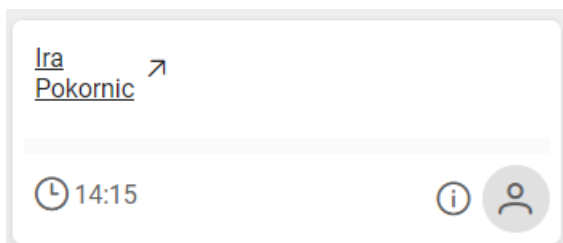
2 usages  rbee
getTimeInStage(stageEntered: Date){
  const currentTime = new Date();

  return Math.round((currentTime.getTime() - stageEntered.getTime()) / 60000);
}

```

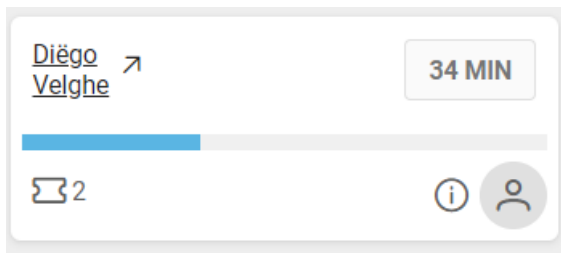
Nu hebben we de logica besproken die de verplaatsing van de elementen mogelijk maakt en kunnen we de afspraken in de verschillende fases bespreken.

3a: Opkomende afspraak:



Bij een opkomende afspraak zien we eerst de naam van de cliënt, die ook als link dient om naar de detailpagina van deze persoon te gaan. Hieronder staat een lege voortgangsbalk die bij de volgende stappen opgevuld zal worden. Links onderaan zien we nog het geplande tijdstip van de afspraak en rechts onderaan zien we een eventuele foto van de maatschappelijke assistent voor deze afspraak, het i-icoontje ernaast is een link naar de detailpagina van de assistent.

### 3b: Afspraak voor persoon in wachtzaal:



Bij een afspraak in de wachtzaal zien we dat de voortgangsbalk opvult met de lichtblauwe kleur. Links onderaan zien we nu een ticketnummer in plaats van de tijd, alsook een timer rechts bovenaan zodat er zowel via timer als visueel kan gezien worden hoe lang deze persoon in de wachtzaal zit. Dit wordt gedaan door elke minuut een functie aan te roepen die dit opnieuw berekent:

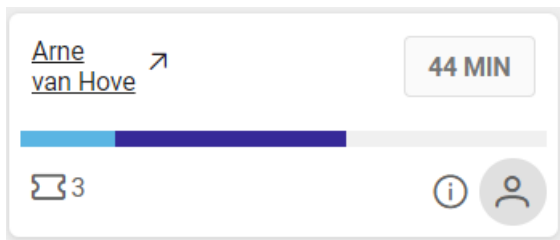
```
setInterval() {  
  if (this.appointment.stage === CONSTANTS.Stages.Waiting || this.appointment.stage === CONSTANTS.Stages.Ongoing) {  
    this.timer = setInterval( handler: () => {  
      this.calculateDifferenceInMinutes();  
      this.setMinutesToDisplay();  
    }, CONSTANTS.MillisecondsToMinutes);  
  }  
}  
}  
  
2 usages  rbee  
calculateDifferenceInMinutes() {  
  const currentTime = new Date();  
  this.differenceInMinutes = this.getDifferenceInMinutes(currentTime, this.timeToCountFrom);  
}  
  
1 usage  ipo +1  
getDifferenceInMinutes(time1: Date, time2: Date): number {  
  if (time1 && time2) {  
    return Math.round( (time1.getTime() - time2.getTime()) / CONSTANTS.MillisecondsToMinutes);  
  }  
}  
  
2 usages  rbee  
setMinutesToDisplay() {  
  if(this.appointment.stage === CONSTANTS.Stages.Waiting){  
    this.timeInStageToDisplay = this.appointment.waitingDuration + this.differenceInMinutes;  
  }  
  if(this.appointment.stage === CONSTANTS.Stages.Ongoing){  
    this.timeInStageToDisplay = this.appointment.meetingDuration + this.differenceInMinutes;  
  }  
}
```

We zorgen er ten eerste voor dat er op een vast interval iets wordt uitgevoerd, dit doen we met `setInterval` waardoor we elke minuut de functies `calculateDifferenceInMinutes` en `setMinutesToDisplay` aanspreken.

In de functie `calculateDifferenceInMinutes` gebruiken we de properties van de afspraak om te berekenen hoeveel minuten deze al in deze fase zit.

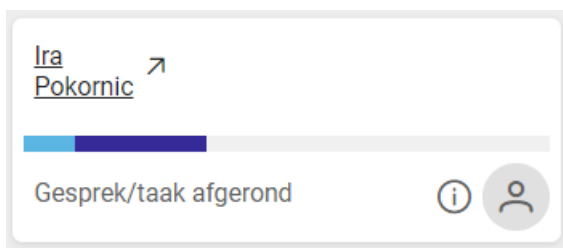
In de functie `setMinutesToDisplay` gebruiken we deze waarde om de tijd op de kaart te tonen.

3c: Cliënt in gesprek met assistent:



De wachttijd van de afspraak wordt behouden in de lichtblauwe kleur en er worden nu nieuwe variabelen aangesproken om te berekenen hoe lang de persoon in gesprek zit. De nieuwe tijd wordt in de donkere kleur getoond zodat er een duidelijk verschil te zien is.

3d: Afspraak is afgerond:



Het bericht links onderaan komt tevoorschijn en maakt duidelijk dat de afspraak afgesloten is. De voortgangsbalk houdt bij hoe lang beide fases hebben geduurd.

Deze voortgangsbalken zijn gemaakt door verschillende elementen naast elkaar te plaatsen en deze breedtes aan te passen naargelang de waarden die te berekenen zijn vanuit de afspraken:

```
<div *ngIf="appointment.stage === constants.Stages.Ongoing || appointment.stage === constants.Stages.Done" class="progress-bar-container mb-2 mx-2">
  <div class="progress-bar-section progressBarWaitingBg" [style.width]="appointment.waitingDuration + '%"></div>
  <div class="progress-bar-section progressBarMeetingBg"
    [style.width]="((appointment.waitingDuration + appointment.meetingDuration + differenceInMinutes) > constants.ProgressbarMaxAmount) ? (constants.ProgressbarMaxAmount -
    appointment.waitingDuration) + '%' :
    (appointment.meetingDuration + differenceInMinutes) + '%"></div>
  <div class="progress-bar-section width-100 progressBarEmptyBg"></div>
</div>
```

Extra's op de pagina:

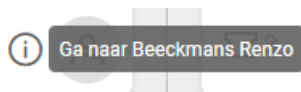
Geen afspraken	1 cliënt in wacht	1 cliënt in gesprek	2 cliënten afgehandeld
----------------	-------------------	---------------------	------------------------

Tellers boven elke kolom die aangeven hoeveel afspraken er in elke lijst aanwezig zijn.

```

<p class="swinlaneHeader" *ngIf="columnWaiting.length === 1">1 {{DASHBOARD.SWIMLANES.OneClientWaiting | translate}}</p>
<p class="swinlaneHeader" *ngIf="columnWaiting.length !== 1">{{columnWaiting.length}} {{DASHBOARD.SWIMLANES.ClientsWaiting | translate}}</p>

```



Tooltip bij het hoveren over het icoon om naar de pagina van de assistent te gaan.

```

<p><i (click)="toPersonDetails(appointment.personAssistant.insz)"
  pTooltip="{{ACTION.GoTo | translate}} {{appointment.personAssistant.lastName}} {{appointment.personAssistant.firstName}}"
  class="pi pi-info-circle fs-large"></i>
  <p-avatar label="{{getInitials(appointment.personAssistant)}}" styleClass="mx-2 mb-2" size="large" shape="circle"></p-avatar>
</p>

```

## Clïënt registreren

Wanneer een nieuwe clïënt zich aanmeldt aan de balie is er de mogelijkheid dat de gegevens van deze persoon nog niet in het systeem zitten, we moeten dus een manier voorzien om een persoon aan te maken.

Om het overzichtelijk te houden heb ik dit in twee componenten opgesplitst, één voor de algemene gegevens en één voor de adresgegevens van de persoon vast te leggen. De overschakeling van de ene naar de andere gebeurt met de steps component van PrimeNG waar we op kunnen klikken om van pagina te veranderen, dit wordt wel enkel toegelaten wanneer de vereiste gegevens van een persoon zijn ingevuld, aangezien we eerst een persoon moeten hebben vooraleer we er een adres aan vast kunnen hangen.

Deze pagina bestaat vooral uit standard inputvelden die met Angular te maken zijn, gemengd met een PrimeNG dropdown element voor de titel van een persoon, alsook tweemaal een PrimeNG selectButton element voor het geslacht en de rol van de persoon. De gegevens worden aan een ngModel gekoppeld. De code voor enkele van deze velden ziet u op de volgende pagina, waarin u op de eerste afbeelding de velden zelf ziet, en op de tweede afbeelding de tekst die erbij komt wanneer de validatie faalt, een kleine rode tekst onder de velden.

```
<fieldset class="border-round border-gray-300 border-2 my-2 width-100">
  <legend>{{'TITLES.Salutation' | translate}}</legend>

  <div class="col-12 grid mx-auto">
    <span class="mb-2 mt-4 p-float-label col-2 field">
      <p-dropdown [options]="titles" [(ngModel)]=newPerson.title [ngModelOptions]={standalone: true}
        optionLabel="translation" optionValue="name" id="float-input-title">
      </p-dropdown>
      <label for="float-input-title">{{'title' | translate}}</label>
    </span>
    <span class="mb-2 mt-4 p-float-label col-5 field">
      <input id="float-input-firstName" formControlName="firstname" type="text" pInputText [(ngModel)]=newPerson.firstName">
      <label for="float-input-firstName">{{'firstName' | translate}}</label>
    </span>
    <span class="mb-2 mt-4 p-float-label col-5 field">
      <input id="float-input-lastName" formControlName="lastname" type="text" pInputText [(ngModel)]=newPerson.lastName">
      <label for="float-input-lastName">{{'lastName' | translate}}</label>
    </span>
  </div>
</fieldset>
```

```
<div class="col-12 formgrid grid mx-auto">
  <div class="col-2">
  </div>
  <div class="col-5">
    <small *ngIf="infoPersonForm.get('firstname').touched && infoPersonForm.get('firstname').errors?.['required']">
      <i class="pi pi-exclamation-circle colorError"></i>
      {{'firstName' | translate}} {{'VALIDATION.Required' | translate}}
    </small>
    <small *ngIf="infoPersonForm.get('firstname').errors?.['pattern'] && infoPersonForm.get('firstname').touched">
      <i class="pi pi-exclamation-circle colorError"></i>
      {{'firstName' | translate}} {{'VALIDATION.NoNumbers' | translate}}
    </small>
  </div>
  <div class="col-5">
    <small *ngIf="infoPersonForm.get('lastname').touched && infoPersonForm.get('lastname').errors?.['required']">
      <i class="pi pi-exclamation-circle colorError"></i>
      {{'lastName' | translate}} {{'VALIDATION.Required' | translate}}
    </small>
    <small *ngIf="infoPersonForm.get('lastname').errors?.['pattern'] && infoPersonForm.get('lastname').touched">
      <i class="pi pi-exclamation-circle colorError"></i>
      {{'lastName' | translate}} {{'VALIDATION.NoNumbers' | translate}}
    </small>
  </div>
</div>
</fieldset>
```

Wanneer de vereiste persoonsgegevens allemaal zijn ingevuld, krijgen we wel de mogelijkheid om naar de adresgegevens te gaan. Het idee hierachter is dat een persoon meerdere adressen kan hebben, en dat er ook nog een verschil is tussen verblijfsadressen of verzendadressen. In de database wil dit zeggen dat we dus niet gewoon een één op één relatie hebben, maar nog een tussentabel. Het backend gedeelte was het moeilijkste bij dit deel omdat dit mijn eerste ticket was waar ik zelf veel moest opbouwen in de backend van de applicatie.

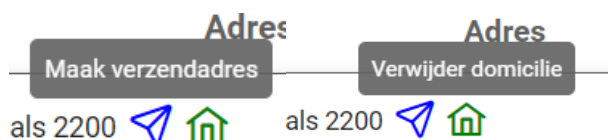


Het uitdagende was ook dat ik moest zorgen dat een adres correct werd opgeslagen met nieuwe gegevens, dit werd moeilijk omdat ik eigenlijk van achter naar voor moest opslaan. Normaal zou je denken dat je eerst een persoon kan opslaan, en daar een adres aan kan koppelen, maar wanneer een gemeente nog niet in het systeem zit, dan gaat dat niet, dus ik moest eerst van achter naar voor de gegeven opslaan vooraleer de persoon met adressen opgeslagen kon worden.

De pagina om de adressen op te slaan ziet er redelijk gelijkaardig uit qua elementen als de persoonsgegevens, met het verschil dat ik hier twee velden met filtering heb toegevoegd, dit gebeurt op het veld voor de gemeente en voor de straatnaam. Dit helpt de gebruiker bij het kiezen van een adres dat al in het systeem aanwezig is.

Wanneer de vereiste velden voor een adres zijn ingevuld kan dit opgeslagen worden en komt het onder het formulier bij in één van de twee lijsten te staan. Bij de verblijfsadressen is er dan nog de mogelijkheid om een adres als hoofdadres aan te duiden door op de knop te drukken, als er al een hoofdadres is dan zal dit vervangen worden. Ook kan een adres overgezet worden als verzendadres door op het blauwe icoon te drukken, hierdoor moet de gebruiker niet tweemaal hetzelfde adres invullen.

The screenshot displays a web interface for managing addresses. At the top, there are two tabs: '1 Algemene gegevens' and '2 Adresgegevens', with the second tab being active. A user profile icon is in the top right corner. The 'Adresgegevens' section contains input fields for 'Gemeente' (with a dropdown arrow), 'Postcode', 'Straat' (with a dropdown arrow), 'Huisnummer', and 'Bus'. Below these are two buttons: 'Adres' and 'Verzendadres', followed by a '✓ Adres opslaan' button. The 'Opgegeven adressen' section is divided into two columns: 'Adres' and 'Verzendadres'. The 'Adres' column lists two addresses: 'Musketstraat 93, Herentals 2200' and 'Molenbergstraat 489, Turnhout 2300'. Each address has a blue location pin icon, a green house icon, and a red 'X' delete button. Below the second address is a 'Maak hoofdadres' button with a house icon. The 'Verzendadres' column shows one address: 'Musketstraat 93, Herentals 2200', with a red 'X' delete button.



Bij het zoeken naar een gemeente of straatnaam zitten we dus met een filtering, ik heb ervoor gekozen deze niet bij elke toets te laten filteren maar om te wachten tot er een halve seconde geen toets meer is ingedrukt. Dit heb ik bereikt door met een Subject te werken waar deze code belangrijk in was:

```
addDebounceTimerToMunicipality() {  
  this.municipalityChangeSubject  
    .pipe(debounceTime( dueTime: 500))  
    .subscribe( next: () => {  
      this.handleMunicipalityChange();  
    });  
}
```

Wanneer er aan deze conditie voldaan was, dan ging ik op zoek naar de gemeentes die in aanmerking komen om getoond te worden. Dit doe ik door eerst te kijken of de gemeente al in de lijst van bestaande gemeentes aanwezig is, zo niet dan wordt dit een null-waarde. Dit wordt dan gevolgd door de functie die de gemeente toevoegd aan het adres:

```
handleMunicipalityChange() {  
  const foundObject = this.municipalities.find(  
    obj => obj.name === this.newAddressMunicipality.name || obj.name === this.newAddressMunicipality  
  );  
  
  this.insertMunicipalityIntoNgmodels(foundObject);  
}
```

Als het adres gevonden is, dan vullen we dit in in het object dat we willen aanmaken, dit wordt dan gevolgd door het opzoeken van de straten die bij deze gemeente horen, zo blijft het andere veld waar in gefilterd kan worden altijd actueel met de gemeente die aangeduid is. Wanneer de gemeente een null-waarde is, dan vullen we een nieuw object in met de huidige waarden, hier moeten geen straten bij opgezocht worden want de gemeente bestaat nog niet dus hier hangt nog geen data aan vast:

```
insertMunicipalityIntoNgmodels(municipality){  
  if (municipality) {  
    this.setNewAddressPropertiesWithMunicipality(municipality)  
    this.getStreetsBelongingToMunicipality(municipality.id)  
  
    return;  
  }  
  this.newAddress.address.municipality = {id: null, name: this.newAddressMunicipality, zipcode: null};  
}
```

Dit alles leidt tot het veld van de adressen waar we kunnen filteren op de straten die tot de huidige gemeente behoren als deze al aanwezig zijn, als een nieuwe straat in het systeem moet komen kan deze straatnaam gewoon getypt worden en dan wordt deze mee opgeslagen, het veld voor de gemeentes ziet er hetzelfde uit:



The image shows a web form for selecting a street. At the top, there is a label 'Straat' in blue text. Below it is a light gray rectangular box with a downward-pointing chevron icon on the right side. Underneath this box is a search input field with a blue border and a magnifying glass icon on the right. Below the search field, two street names are listed: 'Kruisbergstraat' and 'Molenbergstraat'. The list is enclosed in a light gray box with a thin border.

### Onderzoek kaartlezer

Wat ook op de planning stond was het implementeren van een kaartlezer aangezien dit vaak gebruikt wordt om de identiteit van een cliënt na te gaan of om deze persoon in het systeem te plaatsen. We merkten zelf ook al dat het niet echt gebruiksvriendelijk is om elke keer opnieuw alle data van een persoon te moeten typen om iemand te registreren.

We wisten dat er in een andere applicatie van Cipal Schaubroeck al een functionaliteit bestaat met een kaartlezer, maar dit is een totaal andere applicatie, gemaakt met JavaScript en Struts. We wilden dus wel eens vragen aan de andere mensen van onze dienst wat zij ons hierover konden vertellen om te kijken of we daar iets mee konden doen.

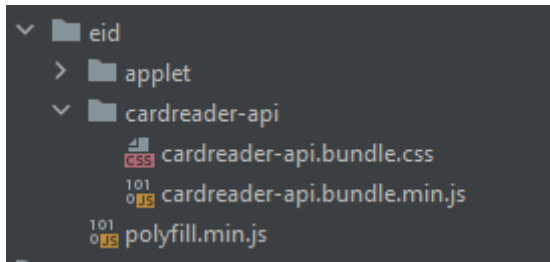
Voordat deze taak plaatsvond hadden we nog nergens echt onderzoek naar moeten doen, dus dit vond mijn mentor ook een zeer interessante vaardigheid om eens iets mee te doen aangezien dit voor ons ook nog vrij nieuw is. Zowel ik als de andere stagair kregen de taak om te zoeken naar methodes om een kaartlezer te implementeren in een Angular applicatie, dit eens te bekijken, te bundelen en daarna ook te rapporteren aan de mentor. Hierna zou dan beslist worden welke routes we als eerste zouden uitproberen.

Wat we als eerste stap wel deden was het installeren van de DioSS-middleware. Dit is nodig omdat een browser zelf niet aan een kaartlezer kan, we hebben dus iets van middleware nodig die kan communiceren tussen de applicatie en de kaartlezer. Hierna moesten we op zoek naar een manier om in de applicatie te bepalen wanneer en hoe dit moet gebeuren.

Ik ben enkele libraries tegengekomen waarvan men zei dat dit zou kunnen werken, maar heel vaak liep het vast met TypeScript of konden we de library niet installeren op het project door configuraties van Cipal Schaubroeck. Na veel zoeken en proberen ben ik met mijn

mentor overeengekomen dat ik eens onderzoek moest doen naar het gebruik van JavaScript in TypeScript, om te kijken of het gebruik van de bestaande functionaliteit voor kaartlezers overgenomen zou kunnen worden vanuit de andere applicatie. Hij heeft me een bestand doorgestuurd met de code van één pagina waar de kaartlezer gebruikt wordt, zodat ik de werking kon bekijken.

Mijn mentor is daarna nog in de code van de andere applicatie gaan kijken en kwam de volgende bestanden tegen:



Het bestand dat de doorslag heeft gegeven is de cardreader-api.bundle. Dit is een bestand van bijna 12000 lijnen JavaScript code vol met methodes om gebruik te maken van de kaartlezer. Heel veel hiervan was moeilijk of niet te begrijpen maar dit is voornamelijk een library met methodes die op de andere pagina die me werd gegeven worden gebruikt.

Hiermee heb ik de achtergrond gegeven over hoe ik ben gestart met de kaartlezer, nu kan ik tonen wat ik met deze bestanden heb gedaan om dit in onze applicatie te gebruiken.

### Implementatie kaartlezer

Ik ben dus aan de slag gegaan met het bestand van de andere applicatie, en de eerste taak was te proberen deze code te begrijpen. Hier stond heel veel in waar wij niets aan hebben omdat de andere applicatie met Struts werkt, dus het was soms wel heel onoverzichtelijk. Ik ben dan begonnen met de functies over te nemen tot het moment waarop ik logs te zien kreeg en dus wist dat er iets gebeurde.

Vanaf dat ik dit had kon ik beginnen met de code aan te passen zodat het voor onze applicatie zou werken en vooral wanneer het moest werken. Aangezien dit toch een belangrijk en interessant deel van de stage is geworden zal ik even de volledige code tonen van mijn JavaScript die ervoor heeft gezorgd dat ik met de kaartlezer kon werken:

```

var cardReaderApi;
5+ usages  ▲ rbee
export var clientData;
4 usages  ▲ rbee
export var personAddressData;
5+ usages  ▲ rbee
export var personPhoto;
2 usages  ▲ rbee
export var readerDisconnected;
5+ usages  ▲ rbee
export var triggerEidPopup;
var globalLatestCard;
var currentClientObjectId;
var $j = jQuery.noConflict();

3 usages  ▲ rbee
export function prepareEid() {
    let compatibleApis = IntoitCardReaderApiFactory.getCompatibleApiNames();
    if (compatibleApis.indexOf("middleware") >= 0) {
        cardReaderApi = IntoitCardReaderApiFactory.getMiddlwareApi();
        initializeApi();
    }
}

2 usages  ▲ rbee
function clearData() {
    currentClientObjectId = undefined;
    clientData = null;
    personAddressData = null;
    personPhoto = null;
}

```

Eigenlijk is dit een bestand dat tussen de applicatie en de cardreader-api (het bestand van 12000 lijnen) zit, dus ik moest beginnen met enkele variabelen die ik nodig had om te communiceren tussen de bestanden.

Methode prepareEid zorgt ervoor dat we de cardreader-api gaan kunnen gebruiken door deze methodes in een object te steken.

Met de clearData methode haal ik gegevens weg als er een nieuwe uitlezing gebeurt.

```

function initializeApi() {
  var statusOk = $j.Deferred();

  cardReaderApi.checkStatus().done(function () {
    statusOk.resolve();
  });

  statusOk.done(cardReaderMonitorSetup);
}

1 usage  rbee
function cardReaderMonitorSetup() {
  var cardReaderMonitor = cardReaderApi.getCardReaderMonitor();
  var latestInsertedEidCard = null;
  cardReaderMonitor.addListener({

    correctCardInserted: function (eidCard) {
      latestInsertedEidCard = eidCard;
      globalLatestCard = eidCard;
      readCard();
      triggerEidPopup = true;
    },

    correctCardRemoved: function () {
      clearData();
      triggerEidPopup = false;
    },

    cardReaderDisconnected: function () {
      triggerEidPopup = false;
      readerDisconnected = true;
    }
  })
}

```

initializeApi is de laatste stap waardoor de applicatie weet dat de api klaar is om te gebruiken. Wanneer dit in orde is begint de cardReaderMonitorSetup zodat deze begint te lezen. Door enkele eventListeners kan ik op bepaalde events logica toevoegen voor wanneer een kaart wordt gelezen of verwijderd.

Voor nu leek het mij het belangrijkste dat we iets doen wanneer een kaart wordt ingestoken, verwijderd of wanneer de kaartlezer niet meer gevonden wordt zodat we op deze momenten actie kunnen ondernemen.

```

export function readCard() {
  clearData();
  if (globalLatestCard == null) {
    return;
  }

  $.when(readRnData())
    .then(readAddressData)
    .then(readPhoto);
}

5+ usages  rbee
export function readRnData() {
  var success = $.Deferred();
  if (globalLatestCard) {
    globalLatestCard.readRnData(function (rnData, error) {
      if (error != null) {
        success.reject(false);
        return;
      }
      success.resolve(true);
      clientData = rnData;
    });
  }

  return success.promise();
}

```

De functie readCard wordt uitgevoerd wanneer een kaart in de kaartlezer komt, daarna worden 3 verschillende functies uitgevoerd, één voor de persoonlijke gegevens, één voor de adresgegevens en één voor de string die de pasfoto voorstelt.

```

export function readAddressData() {
  var success = $j.Deferred();
  if (globalLatestCard) {
    globalLatestCard.readAddressData(function (addressData, error) {
      if (error != null) {
        success.reject(false);
        return;
      }
      success.resolve(true);
      personAddressData = addressData;
    });
  }

  return success.promise();
}

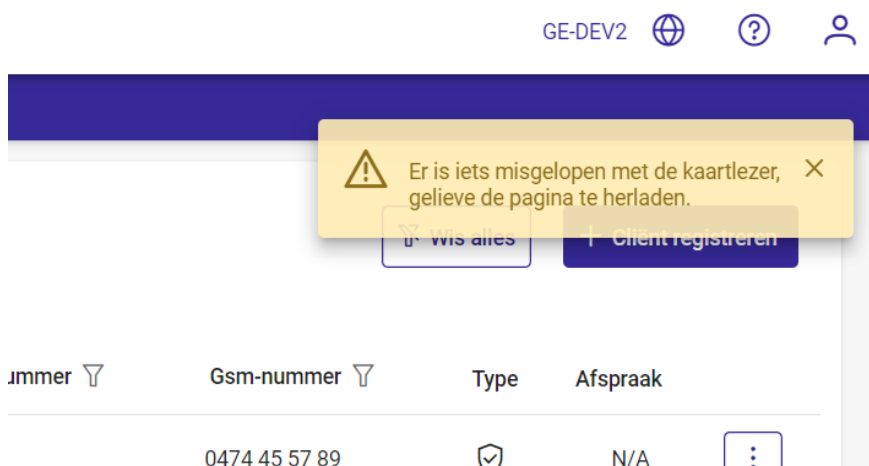
5+ usages  rbee
export function readPhoto() {
  var success = $j.Deferred();
  if (globalLatestCard) {
    globalLatestCard.readPhoto(function (data, error) {
      if (error != null) {
        success.reject(false);
        return;
      }
      success.resolve(true);
      personPhoto = data;
    });
  }

  return success.promise();
}

```

Deze functies zijn niet door mezelf geschreven, ik heb enkel variabelen aangepast (zoals globalLatestCard om de connectie tussen bestanden te maken) of extra variabelen toegevoegd om de gewenste logica uit te voeren (zoals personPhoto).

Na een tijd merkte ik op dat de kaartlezer soms geen gewenst gedrag vertoonde, na veel te testen en debuggen ben ik erachter gekomen dat de kaartlezer na een tijd in een soort van standby gaat waardoor deze niet meer wil activeren. Ik ben gaan zoeken in de code waar dit werd opgevangen en heb een connectie tussen mijn TypeScript en de Javascript van Dioss kunnen leggen om de volgende notificatie te tonen:






```
showRefreshToast(){
  this.notifyUserFromJs = notifyUser;
  if(this.notifyUserFromJs && !this.eidNotificationShown){
    this.eidNotificationShown = true;
    this.messageService.add({
      severity: 'warn', detail: `${this.translate.instant( key: 'NOTIFICATION.DETAIL.RefreshForEid')}`, life: 30000
    });
  }
}
```

Het uitdagende om dit voor elkaar te krijgen lag hem vooral in het vinden van waar het misloopt in de hele api, en dan deze data kunnen doorgeven wanneer dit wel of niet nodig was.

Wanneer de kaartlezer wel werkt zoals gewenst, dan krijgen we het volgende scherm als pop-up tevoorschijn wanneer we een identiteitskaart insteken:

### Gegevens identiteitskaart

Voornaam:	Renzo
Achternaam:	Beeckmans
Rijksregisternummer:	
Geboortedatum:	1999-06-05
Geslacht:	Man



Deze persoon zit nog niet in het systeem. Druk op 'Naar opslaan' om deze persoon aan te maken.

✓ Naar opslaan
✗ Annuleren

Dit komt omdat ik een eventListener heb gemaakt in Angular die regelmatig de waarden van de JavaScript blijft volgen want het is in JavaScript dat de kaart wordt uitgelezen. Op deze manier kan ik deze verandering opvangen in de TypeScript en een functie aanspreken:

```
2 usages  rbee
eventListener = new InsertEidListener( callback: (value : boolean ) => {
  this.showEidModal = value;
});

1 usage  rbee
trackBooleanChanges() {
  let previousValue = this.rootFunctionsService.showEidModal;

  setInterval( handler: () => {
    if (this.rootFunctionsService.showEidModal !== previousValue) {
      previousValue = this.rootFunctionsService.showEidModal;
      this.eventListener.listen(this.rootFunctionsService.showEidModal);
    }
  }, timeout: 1000);
}
```

De kaart wordt automatisch uitgelezen en er wordt wat data getoond om na te gaan of alles correct is verlopen. Wanneer de persoon al in het systeem zit krijgen we de optie om naar de detailpagina van die persoon te gaan, zo niet, dan kunnen we naar het registratiescherm gaan om deze persoon toe te voegen.

Op dit registratiescherm heb ik een knop toegevoegd waarmee we heel makkelijk alle gegevens die nodig zijn uit de kaart lezen en in het formulier invullen, inclusief het adres dat op de kaart staat:

Registratieformulier [Ga terug](#) [Opslaan](#) [Annuleren](#)

1 Algemene gegevens 2 Adresgegevens

Aanspreking  
Titel: Mevr. Voornaam: Achternaam:

Algemene informatie  
Rijksregisternummer: Man Vrouw Ander  
Geboortedatum: Client Medewerker

Contact  
Email: Gsm-nummer: Telefoonnummer:

Cipal Schaubroeck NV - © 2023

Wanneer dit knop wordt gebruikt dan wordt de kaart uitgelezen en moet de data in het formulier gebruikt worden:

```
fillData(eidObject, addressObject){  
  this.fillPersonForm(eidObject);  
  this.makeAddressFromEid(addressObject);  
}
```

```
fillPersonForm(eidObject){  
  this.newPerson.firstName = eidObject.firstName;  
  this.newPerson.lastName = eidObject.name;  
  this.newPerson.gender = this.getCorrectGender(eidObject);  
  this.newPerson.insz = eidObject.nationalNumber;  
  this.newPerson.dob = new Date(eidObject.birthDate);  
  this.newPerson.type = {id: CONSTANTS.ClientTypeId, name: this.translate.instant( key: 'PERSON.Client')};  
  this.fillTitleFieldFromEid(eidObject);  
  console.log(this.newPerson);  
}
```

Eerst gebruik ik de waarden van de persoon om de persoonsgegevens in ons model in te vullen.

```
makeAddressFromEid(addressObject){
  ⚡ rbee
  const fullAddressObject: PersonAddress = new class implements PersonAddress {
    no usages ⚡ rbee
    address: Address;
    no usages ⚡ rbee
    addressType: AddressType;
    no usages ⚡ rbee
    busNumber: string;
    no usages ⚡ rbee
    houseNumber: string;
    1 usage ⚡ rbee
    id: number;
    no usages ⚡ rbee
    person: Person;
  };
  fullAddressObject.addressType = {id: CONSTANTS.DomicileAddressId, name: this.translate.instant( key: 'ADDRESS.MainAddress')};
  fullAddressObject.busNumber = addressObject.numberExtra;
  fullAddressObject.houseNumber = addressObject.number;
  fullAddressObject.address = {id: null, name: this.onlyStreet(addressObject.street), municipality: {id: null, zipcode: addressObject.zipCode, name: addressObject.municipality}};
  fullAddressObject.person = { ...this.newPerson };
  fullAddressObject.person.personAddresses = [];

  if (!this.checkIfAddressInList(fullAddressObject)){
    this.newPerson.personAddresses.push(fullAddressObject);
  }
}
```

Op de kaart staat ook één adres, dit gebruik ik om al meteen een adres in de lijst bij te plaatsen, dat scheelt ook al een hoop werk voor de gebruiker.

Het zal u ook wel opgevallen zijn dat er bij de pop-up een foto te zien is. Ook deze wordt van de kaart gelezen, op de kaart is een base64 string te zien. Om deze te kunnen gebruiken moeten we het decoderen en kunnen we er een afbeelding van maken. Na wat onderzoek heb ik het op deze manier kunnen doen:

```
getPersonDataFromJs() {
  if(clientData && personPhoto){
    this.clientData = clientData;
    this.photoData = personPhoto;
    this.getImgFromBase64();
    this.getPersonFromInsz();
  }
}

1 usage new *
getImgFromBase64(){
  if(this.photoData){
    const blob = this.decodeBase64ToBlob(this.photoData);
    this.photoDataToImg = this.sanitizeImageUrl(blob);
  }
}

1 usage new *
decodeBase64ToBlob(base64: string): Blob {
  const byteCharacters = atob(base64);
  const byteNumbers = new Array(byteCharacters.length);

  for (let i = 0; i < byteCharacters.length; i++) {
    byteNumbers[i] = byteCharacters.charCodeAt(i);
  }

  const byteArray = new Uint8Array(byteNumbers);
  return new Blob( [blobParts: [byteArray], options: { type: 'image/jpeg' }]);
}

1 usage new *
sanitizeImageUrl(blob: Blob): string {
  const objectUrl = URL.createObjectURL(blob);
  return this.sanitizer.bypassSecurityTrustUrl(objectUrl) as string;
}
```

Wat hier vooral gebeurt is dat de string eerst “opgekuist” moet worden zodat we met het juiste type data bezig zijn en er geen foute karakters meer aanwezig zijn. Daarna moet de methode sanitizImageUrl er nog voor zorgen dat de nieuwe waarde geaccepteerd wordt in onze applicatie waarna we dit als image kunnen gebruiken om de foto te tonen.

## Detailpagina persoon

Wanneer al de gewenste data is ingevuld bij het aanmaken van een person kan de persoon opgeslagen worden. Na het opslaan wordt de gebruiker meteen herleidt naar de detailpagina van deze nieuwe persoon. Deze pagina ziet er voor een cliënt zo uit:

The screenshot shows the 'Detailpagina persoon' for 'Tuytens Werner'. The page is divided into several sections, each highlighted with a red box and a number:

- 1**: A red box highlights the 'Algemene gegevens' section, which includes a 'Verloop contact' dropdown menu and a 'Doorverwijzing' dropdown menu.
- 2**: A red box highlights the 'Persoonlijke gegevens' section, which includes fields for 'Titel', 'Voornaam', 'Achternaam', 'Rijkregisternummer', 'Geslacht', 'Geboortedatum', 'Woonplaats', 'Email', 'Telefoonnummer', 'Gsm-nummer', and 'Type'.
- 3**: A red box highlights the 'Afspraak' section, which includes a date and time '30 mei 2023 11:45' and a status 'RT'.
- 4**: A red box highlights the 'Berichten' section, which includes a 'Bericht toevoegen' button and a message 'Er zijn nog geen berichten over deze persoon bijgehouden.'
- 5**: A red box highlights the 'Logboek' section, which includes a '+' button and a message 'Er is nog niets aan het logboek van deze persoon toegevoegd.'
- 6**: A red box highlights the 'Logboek' section, which includes a '+' button and a message 'Er is nog niets aan het logboek van deze persoon toegevoegd.'

**1:** In deze blok kan aangeduidt worden of de cliënt via doorverwijzing aan de balie is terechtgekomen en van waar deze doorverwijzing komt. Dit zijn twee dropdowns waarvan de vraag of het een doorverwijzing is enkel een ja/nee-optie heeft, en de andere heeft een lijst aan opties:

The close-up shows the 'Verloop contact' dropdown menu, which is currently set to 'Verloop contact'. The dropdown list includes the following options: 'Rechterlijk', 'Loketbezoek', 'E-mail', 'Telefonisch', and 'Politie'. The 'Doorverwijzing' dropdown menu is currently set to 'Ja'.

Wanneer de bestaande opties niet volledig zijn en er dus iets nieuw nodig is, kan er in het vak getypt worden, waarna de waarde mee wordt opgeslagen in de database zodat deze optie voor andere cliënten ook in de lijst aanwezig is.

Er staat nergens een knop om deze waarden op te slaan, dit komt omdat ik gebruik heb gemaakt van de Angular lifecycles. In de meeste gevallen gebruiken we enkel de ngOnInit lifecycle die wordt uitgevoerd wanneer het component aangesproken. Voor deze component gebruik ik de ngOnDestroy zodat volgende functies worden aangesproken wanneer de component weg gaat:

```
ngOnDestroy(): void {  
  if(this.person.courseContact.name !== this.currentCourseContact || this.person.referred !== this.currentReferred){  
    if(this.person.courseContact.name){  
      this.saveWithCourseContact();  
    } else {  
  
      this.saveWithoutCourseContact();  
    }  
  }  
}
```

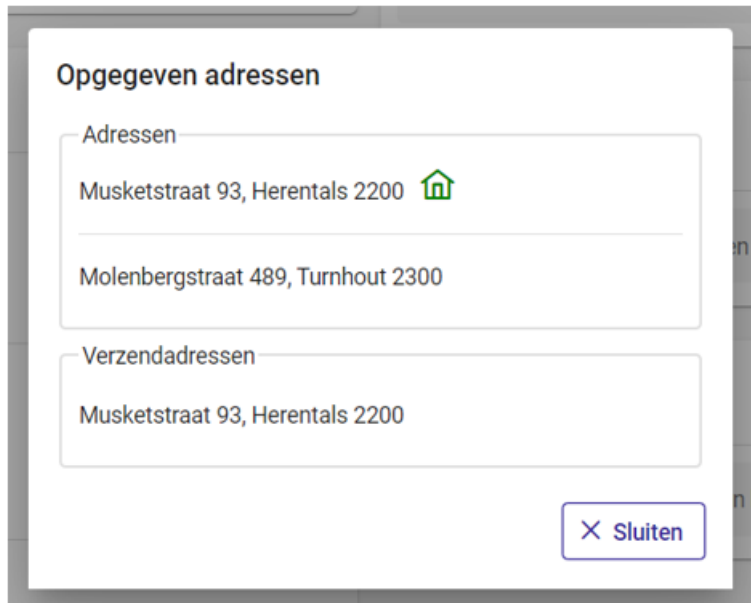
Ik ga kijken of de waarden die aanwezig zijn verschillen van de opgeslagen waarden omdat we de database enkel willen aanspreken als er daadwerkelijk iets iets veranderd en niet elke keer wanneer men de pagina verlaat.

Wanneer we op de pagina van een maatschappelijk assistant kijken is deze blok niet zichtbaar want zij kunnen niet verwezen worden.

**2:** Daaronder hebben we een oplijsting gemaakt van de persoonsgegevens die we bij het registreren hebben ingevuld. De rijen van de tabel zijn niet hard gecodeerd maar worden aangemaakt naargelang de properties van het personenobject dat binnekomt bij het inladen van de pagina.


Wat er gebeurt om dit aan te maken is dat we over alle properties van een persoon gaan, en zolang het niet in een lijst van ongewenste properties zit, maken we er een kolom voor aan, op deze manier tonen we enkel de data die we willen tonen.

Hier zijn twee waarden die iets extra bevatten, die van de woonplaats en die van het emailadres. Wanneer de persoon een hoofdverblijfplaats heeft aangeduid, dan zal de gemeente van dit adres te zien zijn. Hier kan op geklikt worden om dan een volledige lijst van de adressen te tonen:



Opgegeven adressen

Adressen

Musketstraat 93, Herentals 2200 

Molenbergstraat 489, Turnhout 2300

Verzendadressen

Musketstraat 93, Herentals 2200

✕ Sluiten

De adressen worden eerst gesplitst op vlak van type om de twee lijsten te tonen. Hierna wordt de hoofdverblijfplaats op de eerste plaats gezet, dit was een functionele eis omdat het wel proper is om dit altijd bovenaan te hebben staan.

```
splitAddressesByType(){
  const {domicileAddresses, shippingAddresses} = this.functionsService.splitAddressesByType(this.person);
  this.domicileAddresses = domicileAddresses;
  this.shippingAddresses = shippingAddresses;
  this.domicileAddresses = this.moveMainAddressToTop();
}

1 usage  rbee
moveMainAddressToTop(){
  let newAddressList = [];
  for(const address of this.domicileAddresses){
    if(address.id === this.person.homeAddressId){
      const indexToDelete = this.domicileAddresses.indexOf(address);
      newAddressList = [address, ...this.domicileAddresses.slice(0, indexToDelete), ...this.domicileAddresses.slice(indexToDelete+1)];
    }
  }

  return newAddressList;
}
```

Men kan ook op het emailadres klikken waarna je rechtstreeks naar een mailapplicatie kan gaan om deze persoon te contacteren:

Email

[tuytensruben@gmail.com](mailto:tuytensruben@gmail.com)



**3:** Vanop de detailpagina van een cliënt willen we ook de mogelijkheid om de fase van een afspraak te beïnvloeden zodat we niet altijd naar het overzicht moeten gaan. Hier zijn dus knoppen voorzien die ons daarbij helpen zodat we snel van fase kunnen veranderen, zowel naar voor als naar achter:

```
updateAppointmentToWaiting() {  
  this.appointment.stage = CONSTANTS.Stages.Waiting;  
  this.appointment.waiting = new Date();  
  this.appointmentService.updateAppointment(this.appointment).subscribe();  
}
```

Dit is een voorbeeld van wat de knop zou uitvoeren als we iemand van op de planning naar de wachtzaal sturen via de knop.

Onder de knoppen hebben we nog een timer staan ook, deze laat zien hoe lang de persoon al in de huidige fase zit. Zo kunnen maatschappelijke assistenten vanop deze pagina snel zien wat er aan de hand is.

De code hierachter is vrij uitgebreid voor wat het is, maar dat is omdat er eerst moet bepaald worden vanaf wanneer er getelt moet worden, er moet een interval gestart worden en de minuten en seconden zijn twee aparte variabelen, dus ook twee aparte methodes:

```

calculateTimeDifference() {
  const currentTime = new Date();
  this.progressInMinutes = this.getDifferenceInMinutes(currentTime, this.timeToCountFrom);
  this.progressRemainingSeconds = this.getRemainingSeconds(currentTime, this.timeToCountFrom);
}

1 usage  ↳ rbee +1
getDifferenceInMinutes(time1: Date, time2: Date): number {
  return Math.floor( (time1.getTime() - time2.getTime()) / CONSTANTS.MillisecondsToMinutes);
}

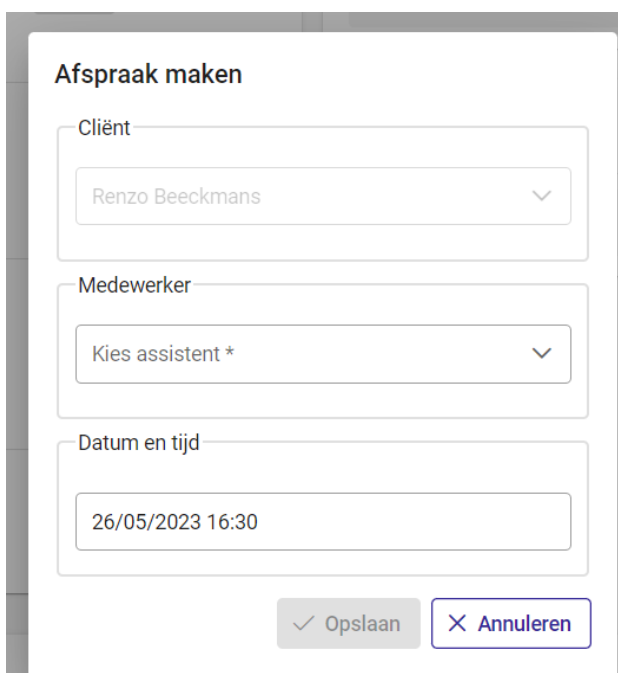
1 usage  ↳ rbee +1
getRemainingSeconds(time1: Date, time2: Date): number {
  return Math.floor( (time1.getTime() - time2.getTime()) / CONSTANTS.MillisecondsToSeconds) % CONSTANTS.SecondsToMinutes);
}

2 usages  ↳ rbee +1
setInterval() {
  if (this.appointment.stage === CONSTANTS.Stages.Waiting || this.appointment.stage === CONSTANTS.Stages.Ongoing) {
    this.timer = setInterval( handler: () => {
      this.calculateTimeDifference();
    }, CONSTANTS.TimeOutAppointmentStageInterval);
  }
}

1 usage  ↳ rbee
getTimeToCountFrom() {
  if (this.appointment.ongoing) {
    this.timeToCountFrom = this.appointment.ongoing;
    return;
  }
  this.timeToCountFrom = this.appointment.waiting;
}

```

4: Wanneer een cliënt langs moet komen, doen we dit via afspraken. Wanneer er nog geen gepland staat staat er in het vak een tekst dat er nog geen gepland is met een blauwe tekst om dit wel te doen, wanneer men daarop klikt komt het volgende scherm naar boven:



**Afspraak maken**

Cliënt  
Renzo Beeckmans

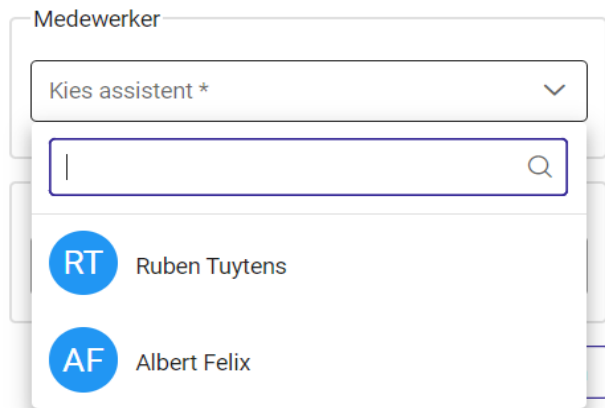
Medewerker  
Kies assistent \*

Datum en tijd  
26/05/2023 16:30

✓ Opslaan    ✕ Annuleren



We krijgen een pop-up waarbij de cliënt niet verandert kan worden aangezien we van deze persoon zijn detailpagina komen. Daaronder kunnen we een maatschappelijke assistent toewijzen uit een lijst met een filter:

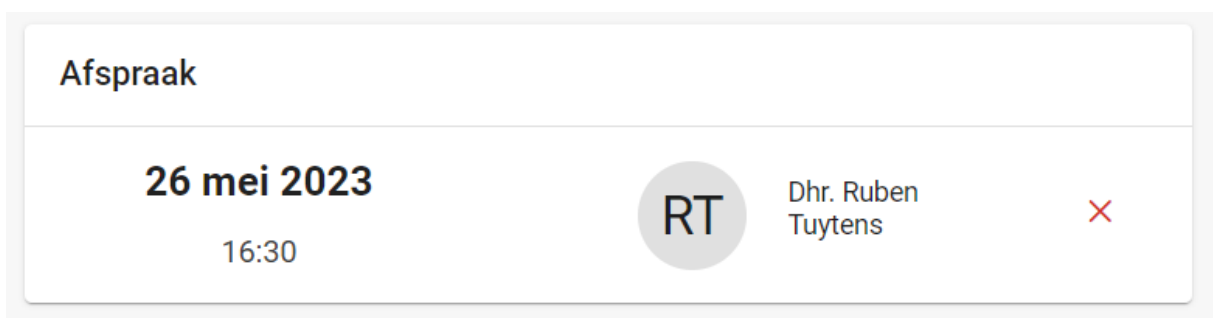


```
<p-dropdown [options]="assistants" [(ngModel)]="newAppointment.personAssistant" required
styleClass="width-100" [filter]="true" filterBy="firstName" [showClear]="true"
id="dropdownAssistant" placeholder="{{ 'APPOINTMENT.PickAssistant' | translate }}" * formControlName="personAssistant">
```

Als laatste moeten we dan nog een datum en tijdstip kiezen, dit heb ik gedaan met een PrimeNG kalender component waarbij de stappen bij de minuten per 15 gebeurt en dagen in het weekend of verleden kunnen ook niet meer aangeduid worden:

```
<p-calendar [(ngModel)]="newAppointment.dateTime" [defaultDate]="minDate" [minDate]="minDate"
[stepMinute]="constants.minuteSteps" [hideOnDateTimeSelect]="false"
styleClass="width-100" [showTime]="true" [required]="true" [disabledDays]="[0,6]"
formControlName="dateTime" inputId="time"></p-calendar>
```

Wanneer we deze afspraak opslaan komt deze bij op de detailpagina te staan en kunnen we er geen nieuwe meer bijmaken tenzij we de huidige afspraak annuleren:



Op de pagina van een maatschappelijk assistent zijn er wel meerdere afspraken te zien aangezien dit hier wel belangrijk is, alsook een knop die voorzien is om er altijd nieuwe bij te maken:

Afspraken

+ Afspraak maken

DagWeek

Vandaag

<⌵>

30 mei 2023 11:30	NM	Mevr. Nelson Mandela	×
30 mei 2023 11:45	WT	Mevr. Werner Tuytens	×
30 mei 2023 12:45	RB	Dhr. Renzo Beeckmans	×

De getoonde lijst aan afspraken en de knop heb ik gemaakt, de filters om te tonen welke afspraken getoond moeten worden komen van mijn collega en kan ik dus niet verder op ingaan.

**5-6:** Deze twee componenten komen van mijn collega, ik zal hier dus ook niet ver op ingaan want behalve wat opmaak hier en daar heb ik hier geen bijdrage gehad aan de functionaliteiten.

Tonen van berichten en items in het logboek:

Berichten

Bericht toevoegen

Huishoudelijk geweld

Armoede

Logboek

+

18/05/2023 10:00

Ruben Tuytens

Telefonisch

⚙

×

Céline kwam binnen met een blauw oog van haar vriend.

Nieuw logboek aanmaken:

The image displays two overlapping screenshots of a web form titled "Logboek aanmaken".

**Left Screenshot (Afspraak tab):**

- Title:** Logboek aanmaken
- Tabs:** Afspraak (selected), Vrij
- Afspraak:** A dropdown menu showing "30 mei 2023 | Ruben Tuytens".
- Log:** An empty text input field.
- Buttons:** Opslaan (grey), X Annuleren (blue outline).

**Right Screenshot (Vrij tab):**

- Title:** Logboek aanmaken
- Tabs:** Afspraak, Vrij (selected)
- Hoe was client gecontacteerd:** A dropdown menu showing "Telefonisch".
- Medewerker:** A dropdown menu showing "Kies assistent".
- Datum en tijd:** An empty text input field.
- Log:** An empty text input field.
- Buttons:** Opslaan (grey), X Annuleren (blue outline).

At the bottom right of the right screenshot, a timestamp "18/05/2023 10:00" is visible.

## Overzicht personen

Tot slot is er nog een functionaliteit die niet meteen betrokken is bij de algemene flow van de rest van de applicatie en dat is het overzicht van al onze gebruikers. Hierin willen we zorgen dat we snel naar een gebruiker kunnen zoeken om naar de detailpagina te gaan, te bewerken of eventueel zelfs te verwijderen.

Een standaardoverzicht was net voorzien voor ik aan mijn stage begon door mijn collega, er is uiteraard wel veel aan aangepast doorheen de stage. Het uiteindelijke overzicht ziet er zo uit:

Naam	Rijksregisternummer	Geslacht	Geboortedatum	Woonplaats	Telefoonnummer	Gsm-nummer	Type	Afspraak
Céline Hachez	99.07.18-416.07	Vrouw	18 jul. 1999	Sterrebeek	–	–	👤	26 mei 2023 20:00
Nelson Mandela	18.07.18-463.05	Man	18 jul. 1918	Turnhout	–	–	👤	
Werner Tuytens	70.05.10-153.58	Man	10 mei 1970	Turnhout	0132 51 68 9	–	👤	29 mei 2023 18:30
Dwight Eisenhower	99.10.14-469.15	Man	14 okt. 1890	Turnhout	–	–	👤	31 mei 2023 16:45
Donald Trump	46.06.14-497.45	Man	14 jun. 1946	Geel	0147 85 96 00	–	👤	

We zien hier een dynamisch gegenereerde tabel die met hetzelfde principe is opgebouwd als de tabel op de detailpagina. De kolommen worden gemaakt op basis van de properties die te vinden zijn in een personenobject. Wat we dan nog moeten doen is de kolommen weghalen die we niet willen zien. Ook de acties om een persoon te verwijderen of bewerken worden hieraan toegevoegd. Zelf heb ik nog een manier gevonden om de voor- en achternaam in één kolom te combineren. De code voor deze kolommen ziet er zo uit:

```

createPersonColumns() {
  if (this.columns.length === 0) {
    this.columns.push({
      field: 'firstName',
      header: 'Name'
    });
    Object.keys(this.allPersons[0]).forEach(r => {
      if (r.toString().includes('created')
        || r.includes('lastModified')
        || r.includes('id')
        || r.includes('messages')
        || r.includes('logBooks')
        || r.includes('title')
        || r.includes('email')
        || r.includes('firstName')
        || r.includes('lastName')
        || r.includes('appointmentsAssistant')
        || r.includes('documentContactNotas')
        || r.includes('notes')
        || r.includes('personAddresses')
        || r.includes('referred')
        || r.includes('courseContact')) {
      } else {
        this.columns.push({
          field: r.toString(),
          header: r.toString()
        });
      }
    });
    this.columns.push({
      field: 'acties',
      header: ''
    });
    this.data = this.allPersons;
    this.isLoading = false;
  }
}

```

Verder in de tabel staat de kolom voor de woonplaats, deze heb ik ook aangepast aangezien de data die hiervoor wordt bijgehouden enkel een “id” is. Ik heb dus de backend hiervoor aangepast zodat ook een object van de woonplaats wordt meegegeven met de persoon en ik daar de naam van kan tonen.

Voor de velden telefoon- en gsm-nummer heb ik zelf een pipe geschreven. De data die wordt bijgehouden is enkel een string van cijfers aan elkaar en dit is niet zo handig om te lezen als gebruiker. Ik heb dus een pipe geschreven zodat ik bij het maken van die kolom heel makkelijker de data kan manipuleren door de volgende code te gebruiken:

```

<p *ngIf="item.header.includes('telephone') || item.header.includes('mobilePhone')">{{item.field | app_addSpaces}}</p>

```

Om een pipe toe te voegen moeten we enkel “| <naam van pipe>” toevoegen achter de variabele en dan wordt deze toegepast. De pipe zelf waarbij ik af en toe een spatie toevoeg ziet er als volgt uit:

```

1 |import {Pipe, PipeTransform} from '@angular/core';
2 |
3 |5+ usages  rbee
4 |@Pipe({name: 'app_addSpaces'})
5 |export class AddSpacesPipe implements PipeTransform {
6 |3 usages  rbee
7 |transform(value: string): string {
8 |    if (value !== null) {
9 |        let firstSpacePassed = false;
10 |        let result = '';
11 |        let count = 0;
12 |        for (let i = 0; i < value.length; i++) {
13 |            if (count === 4) {
14 |                result += ' ';
15 |                firstSpacePassed = true;
16 |            }
17 |            if (count % 2 === 0 && i !== 0 && firstSpacePassed) {
18 |                result += ' ';
19 |            }
20 |            result += value[i];
21 |            count++;
22 |        }
23 |        return result;
24 |    }
25 |}
26 |

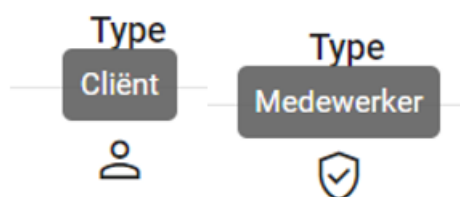
```

Het laatste dat ik aan de tabel zelf heb toegevoegd is bij de kolom van het type persoon dat ze zijn, hier heb ik een icoon toegevoegd met tooltip waarbij de tooltip met een ternary operator wordt toegevoegd. Ik heb zo vaak mogelijk de ternary operator proberen gebruiken omdat dit er toch wel netjes uitziet:

```

<p *ngIf="col.field.includes( value: 'type' )" class="flex pl-1 textAlignCenter">
  <span pTooltip="{{rowData[col.field].name === ('PERSON.Employee' | translate) ? ('PERSON.Employee' | translate) : ('PERSON.Client' | translate)}}"
    tooltipPosition="top"
    class="material-symbols-outlined">
    {{rowData[col.field].name === ('PERSON.Employee' | translate) ? 'verified_user' : 'person'}}
  </span>
</p>

```



Het laatste dat ik op deze pagina nog heb toegevoegd is het filteren op type persoon dat we zien:

[Cliënten](#)
[Medewerkers](#)
[Personen](#)

Dit is gedaan met de `tabView` component van PrimeNG waarbij we via een `onChange` event de index terugkrijgen van de tab die wordt aangeduid. Dit heb ik dan kunnen gebruiken om de correcte data te tonen. Om hiermee te beginnen moeten we eerst alle personen opsplitsen zodat we een lijst met alle personen, een lijst met cliënten en een lijst assistenten hebben:

```
getPersonClients() {
  this.allPersons.forEach(person => {
    if (person.type.id === CONSTANTS.ClientTypeId) {
      this.clients.push(person);
    }
  });
}
```

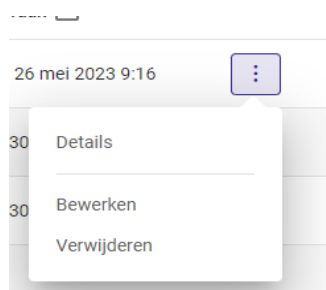
Daarna kunnen we de verandering van tabs opvangen in deze functie, zodat de juiste actie kan ondernomen worden naargelang de tab die wordt aangeklikt:

```
handleChange(e) {
  const index = e.index;
  if (index === 0) {
    this.onPersonClients();
    this.isClientTab = true;
  } else if (index === 1) {
    this.onPersonEmployees();
    this.isClientTab = false;
  } else {
    this.onAllPersons();
    this.isClientTab = false;
  }
}
```

```
onPersonClients() {
  this.personsCurrentTab = this.clients;
  this.filteredUsers = this.clients;
}
```

Zaken die enkel door mijn collega zijn gemaakt:

de menu's voor de acties bij elke persoon.

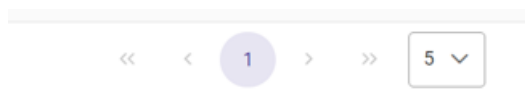


De filters boven elke kolom om met veel opties te kunnen filteren:



A vertical filter panel with a light gray background and rounded corners. It contains two dropdown menus: the first is labeled 'Match alle' and the second 'Begint met', both with a downward arrow icon. Below the second dropdown is an empty rectangular input field. Underneath the input field is a link that says '+ Regel toevoegen'. At the bottom of the panel are two buttons: 'Wissen' (light gray with a dark border) and 'Toepassen' (solid dark blue with white text).

De pagination onderaan de pagina:



A horizontal pagination bar with a light gray background and rounded corners. It features navigation arrows: double left arrows, a single left arrow, a circle containing the number '1', a single right arrow, and double right arrows. To the right of these arrows is a box containing the number '5' and a downward arrow, indicating a dropdown menu for page count.



## Conclusie

Met trots kan ik zeggen dat dit de realisaties zijn die ik tijdens mijn stageperiode heb kunnen verwezenlijken. Zowel frontend als backend heb ik veel bij kunnen dragen in het project en vind het zelf enorm fijn om te zien wat ik hier allemaal in heb bijgeleerd, de sprong die ik heb gemaakt sinds het begin van de stage is enorm.

Naast de realisaties die u hier ziet denk ik dat ik ook voldoende heb bijgedragen op vlak van inbreng voor oplossingen of verbeteringen die konden plaatsvinden, dit was ook mede dankzij de vrijheid en de verantwoordelijkheid die aan ons werd toevertrouwd.

Bij deze wil ik ten eerste mijn mentor, Ruben Tuytens bedanken voor de fantastische begeleiding die ik heb gekregen. Ook over mijn stagebedrijf Cipal Schaubroeck wil ik benadrukken dat het een heel aangename stageplaats is en dat ik van veel geluk mag spreken dat ik hier terecht ben gekomen.

En natuurlijk gaat er ook een bedanking richting Thomas More en de docenten om mij de kennis en tools te geven om tot op dit punt te geraken en mij klaar te stomen voor het werkveld.

Dan kan ik bij deze het document over mijn realisaties tijdens de stage afronden en ik hoop van harte dat alles voldoende duidelijk was voor u, de lezer.