295 - Backend

Auftrag Fallstudie Teil 2 - Item

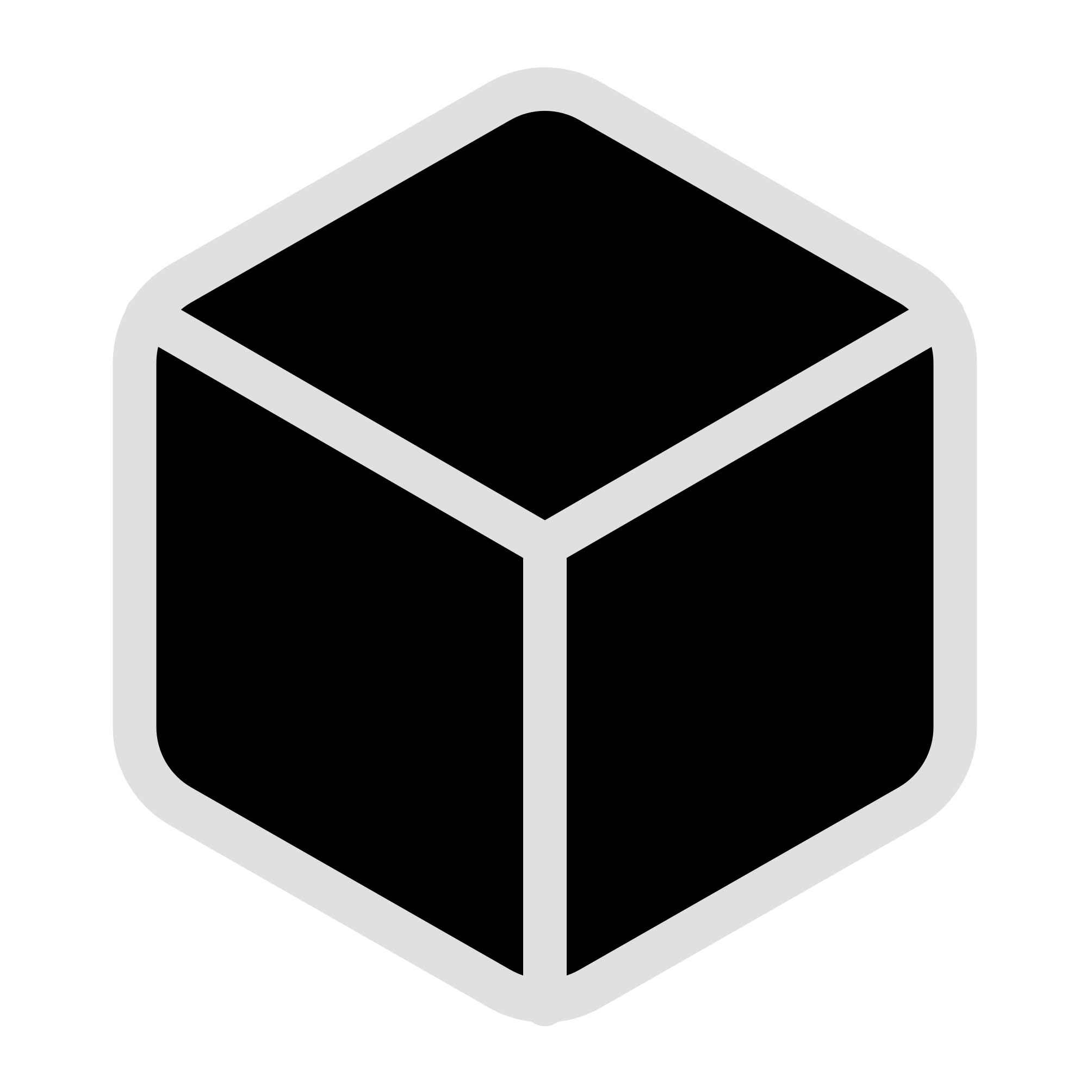
|  |  |
| --- | --- |
| Richtzeit  2h | Sozialform  Einzelarbeit |

Nachdem wir im vorherigen Auftrag das Setup gemacht haben, beginnen wir nun mit dem Item.

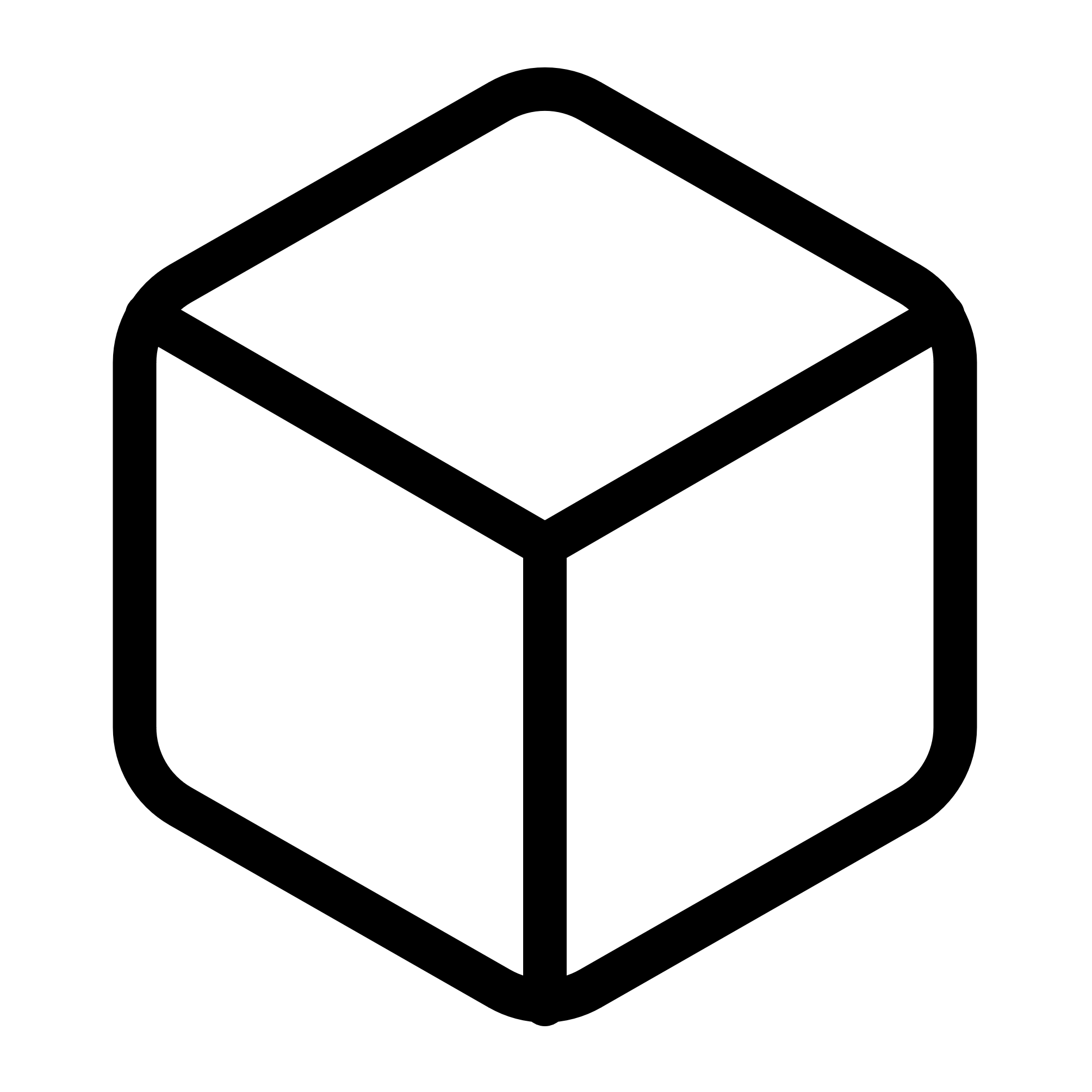
## Item Entity

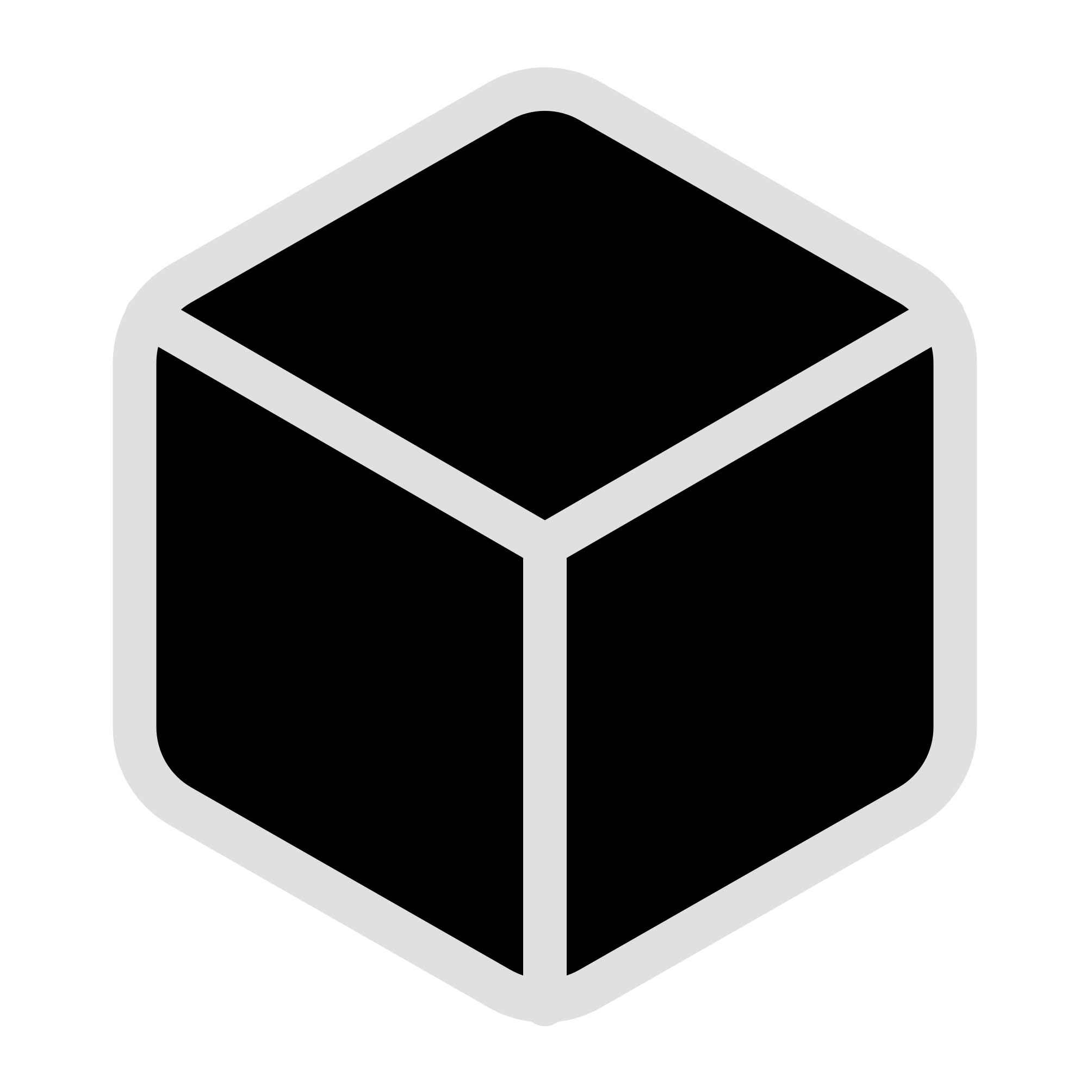
A picture containing graphical user interface

Description automatically generatedIm Package ch.bbcag.backend.todolist.item hat es eine Klasse mit dem Namen Item. Diese Klasse wird unsere ToDo-Items repräsentieren. Füg der Klasse die Felder, analog zum Klassendiagram links, hinzu und lass dir die entsprechenden Getter und Setter generieren. Füg den Feldern auch passenden Annotationen hinzu. Dabei kannst du dich an der Klasse Person orientieren. Achte beim Timestamp darauf, dass du java.sql.Timestamp verwendest.

Speziell soll das Feld createdAt mit @Column(insertable = false) annotiert werden, sodass es nicht von unserer Applikation her in der Datenbank verändert werden kann. Ausserdem braucht das Feld person folgende Annotationen: (Darauf wird in der Präsentation zu Assoziationen noch mehr eingegangen)

@ManyToOne  
@JoinColumn(name = "person\_id")

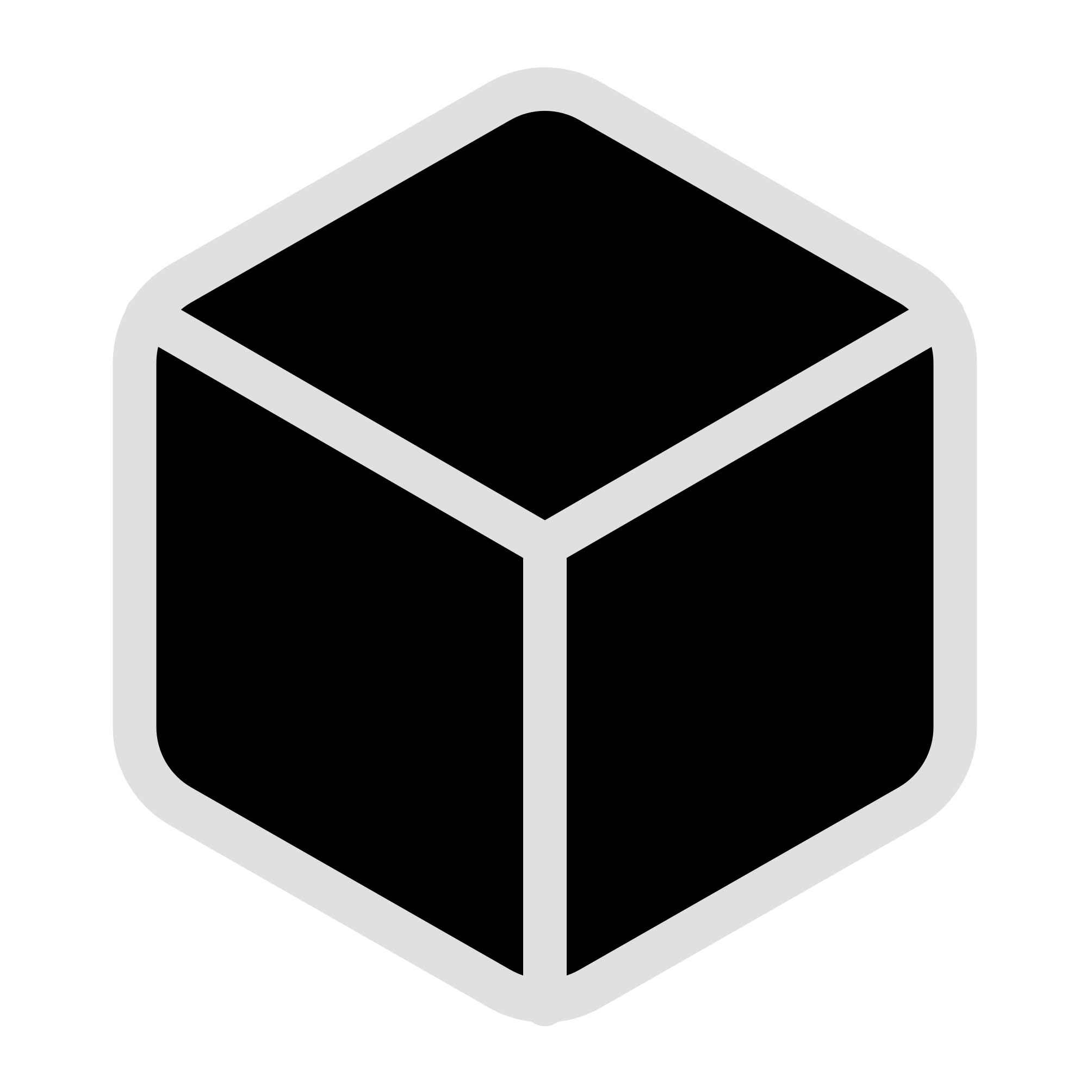
Überschreib zusätzlich noch die equals **und hashCode Funktionen, oder lass sie dir von der IDE generieren. Verwende für** die beiden Methoden nur das Feld id.

Auf der anderen Seite der Relation füg die folgenden Zeilen der Person-Klasse hinzu:

@OneToMany(mappedBy = "person", fetch = FetchType.*LAZY*)  
private Set<Item> items = new HashSet<>();

Generiere auch hierfür Getter und Setter.

Auch im PersonResponseDTO müssen die Items hinzugefügt werden. Erstell dafür in der PersonResponseDTO-Klasse eine neue Instanzvariable mit dem Typ List<Integer> und generier getter und setter dazu.

Im nächsten Auftrag wird genauer auf den Mapper und die DTOs eingegangen, für die Person existiert der bereits, jedoch noch ohne das Item. Füg die Items mit folgendem Code der PersonMapper.toResponsDTO()-Methode hinzu:

if (person.getItems() != null) {  
 List<Integer> itemIds = person  
 .getItems()  
 .stream()  
 .map(Item::getId)  
 .toList();  
  
 personResponseDTO.setItemIds(itemIds);  
}

Das Projekt sollte nun wieder kompillieren. Sende mit einem http-Client ein GET auf /persons und überprüfe, dass die Item-Id’s aus der DB gesetzt sind.

## ItemRepository

Unsere Datenbank ist erstellt, befüllt mit Daten und die Verbindung vom Backend zur Datenbank besteht. Jedoch haben wir noch keinen Code, welcher Abfragen für die Datenbank behandelt. Dafür brauchen wir das Repository. In diesem Interface werden der Code bzw. die SQL-Queries definiert. Schauen wir uns ein Beispiel dieses Interfaces an:

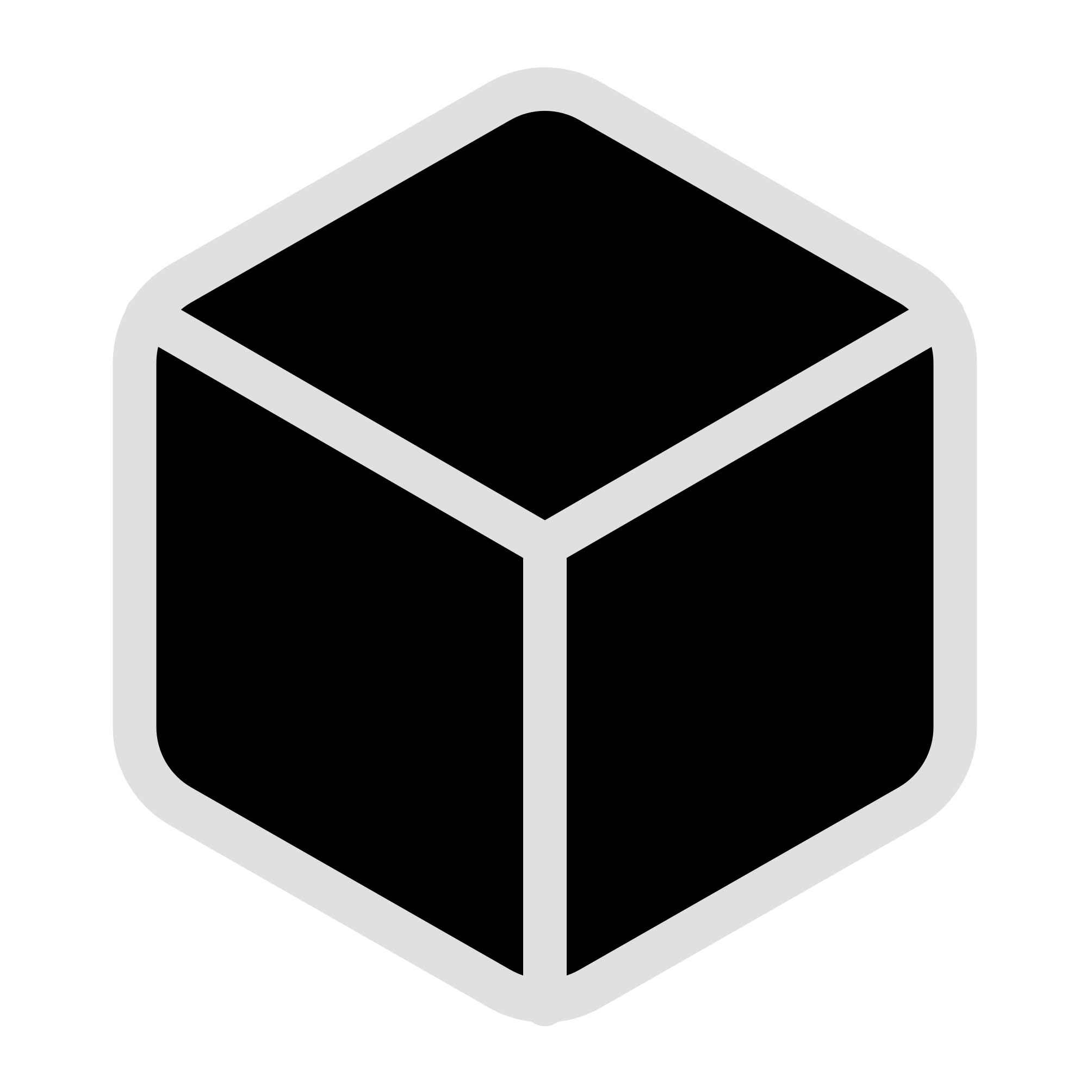
|  |
| --- |
| public interface ItemRepository extends JpaRepository<Item, Integer> {   @Query("SELECT i FROM Item i WHERE i.name LIKE CONCAT('%', :name, '%')")  List<Item> findByName(String name);  } |

Das Repository erweitert (extends) das Interface JpaRepository. Dieses beinhaltet eine grosse Anzahl vordefinierter Funktionen, welche wir so erben können.  
Weiter sehen wir die Annotation @Query. Dies bedeutet, dass mit dem Aufruf dieser Methode ein SQL-Statement abgesetzt werden soll. Da unsere Methode findByName heisst, macht es Sinn im SQL-Query einen Namen mitzugeben und nach diesem zu suchen. Die Namen des Parameters der Methode und des Parameters im Query müssen identisch sein, damit der mitgegebene String auch im Query verwendet wird.

Erstelle entsprechend das ItemRepository im item Package.

## Item Controller

Schreib nun eine weitere Klasse ItemController im ch.bbcag.backend.todolist.item Package analog zum Diagramm. Ruf dabei die Passenden Methoden auf dem ItemRepository auf. Beachte, dass es im Repository mehrere Arten gibt ein Item mit der Id zu lesen. Verwende da jeweils die Methode

findById(id).orElseThrow(EntityNotFoundException::new).

**Hinweis**

Vergiss nicht, das ItemRepository mittels **@Autowired** einzubinden.



Achte darauf, dass die Funktionen unter folgenden **Pfaden** und **HTTP-Methoden** erreichbar sind:

|  |  |  |  |
| --- | --- | --- | --- |
| Controller-Funktion | Pfad | HTTP- Methode | Repository-Funktion |
| void delete(Integer id) | /items/{id} | DELETE | deleteById(id) |
| Item findById(Integer id) | /items/{id} | GET | findById(id) |
| Item insert(Item item) | /items | POST | save(item) |
| List<Item> findItems(String name) | /items?name={name} | GET | findByName(name)  findAll() |

Die Methode findItems(String name) nimmt einen optionalen Parameter name entgegen. Falls ein Name vorhanden ist soll auf dem Repository die Methode findByName() aufgerufen werden. Falls kein Name vorhanden ist soll auf dem Repository die findAll()-Methode aufgerufen werden.

Wenn du den Controller implementiert hast, starte die Applikation neu und ruf, beispielsweise mit Postman, die verschiedenen URL’s auf. Prüf mit dem Debugger, ob du bei POST und GET / DELETE in die jeweiligen Funktionen gelangst.

Ruf nun mit GET https://localhost:8080/items/1 auf. Dies führt zu einem Fehler. Setze einen Breakpoint im Item.getPerson. Was fällt dir auf?

|  |
| --- |
|  |

## Zirkuläre/Zyklische Referenz

Das Phänomen, dass du in der Antwort beobachten kannst, nennt sich zirkuläre oder zyklische Referenz. Da ein Item eine Person beinhaltet und die Person wiederum ein Item wird diese Beziehung endlos aufgelöst. Im Backend führt das Ganze früher oder später zu einer StackOverflowException, da es sich dabei um eine Endlosschlaufe handelt.

Überleg dir bis zur nächsten Präsentation, wie du dieses Problem lösen könntest.

Icon

Description automatically generated

**Hinweis**

Schau dir das person-Package an und versuch herauszufinden, was dort anders gemacht wurde als in deiner Lösung.

## Neues Item

**Frage: Wie muss das JSON aussehen, wenn du ein Item mit Namen «Testaufgabe» und Beschreibung «Beschreibung der Testaufgabe» erstellen willst.**

|  |
| --- |
|  |

Sende das JSON an deinen Controller und überprüf das Resultat. Schau bei Fehlern den Output deiner Applikation an. Spring Boot wird dir mitteilen, falls du etwas nicht oder falsch mitgeschickt haben solltest.