

Nicht in der Aufgabenstellung

<https://www.youtube.com/watch?v=9SGDpanrc8U&t=3982s>

Student Klasse

```
@Entity
@Table
public class Student {
    @Id
    @SequenceGenerator(
        name = "student_sequence",
        sequenceName = "student_sequence",
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "student_sequence"
    )
    private Long id;
    private String name;
    private String email;
    private LocalDate dob;

    @Transient
    private int age;

    public Student(Long id, String name, String email, LocalDate dob) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.dob = dob;
    }

    public Student(String name, String email, LocalDate dob) {
        this.name = name;
        this.email = email;
        this.dob = dob;
    }

    public Student() {
    }
}
```

- **@Entity**
Dient zur Erstellung einer neuen Entität der Domänenklasse. Dabei werden mehrere Datenfelder angegeben wie ID, Name, E-Mail.
- **@GeneratedValue**
`@GeneratedValue(strategy = GenerationType.IDENTITY)`
Dient zum erstellen einer Id für jeden Student in diesem Fall.

Es werden mehrere Konstruktoren erstellt dadurch können wir Daten Mittels einer ConfigKlasse hinzufügen.(einen leere Konstruktor wird in diesem fall benötigt).

- **@Transient**
Wird verwendet für das Alter, da sich das Alter jährlich verändert wird dabei Annotation verwendet.

Zur Studenten Klasse werden noch Getter und Setter- Methoden hinzugefügt und eine toString-Methode.

Diese Klasse wird verwendet zum erstellen einer Entität und um Daten zu bekommen und zu ersetzen.

StudentRepository Interface

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

    @Query("SELECT s FROM Student s WHERE s.email = ?1")
    Optional<Student> findStudentByEmail(String email);
}
```

- **@Repository**
Ist eine spezielle Annotation, mit der alle ungeprüften Ausnahmen aus den DAO-Klassen konvertiert werden können. In diesem Fall ist es unserer Schnittstelle um die verschiedenen Klassen mit einander zu verbinden.
- **@Query**
Dient zu manuellen Definition einer Abfrage der Datenbank. In diesem Fall wird die E-Mail abgefragt, mit einer Methode die in StudentService Klasse verwendet wird. Kann Überprüft das die E-Mail schon vorhanden ist.(Exception Handling)
- **Optional<>**
Ein Containerobjekt, das einen Wert ungleich Null enthalten kann oder nicht. Wenn ein Wert vorhanden ist, gibt `isPresent()` true zurück und `get()` gibt den Wert zurück.

StudentController

```
@RestController
@RequestMapping(path= "api/v1/student")
public class StudentController {

    private final StudentService studentService;

    @Autowired
    public StudentController(StudentService studentService){
        this.studentService = studentService;
    }

    @GetMapping
    public List<Student> getStudent() {
        return studentService.getStudent();
    }

    @PostMapping
    public void registerNewStudent(@RequestBody Student student){
        studentService.addNewStudent(student);
    }

    @DeleteMapping(path = "{studentId}")
    public void deleteStudent(@PathVariable("studentId") Long studentId){
        studentService.deleteStudent(studentId);
    }

    @PutMapping(path = "{studentId}")
    public void updateStudent(@PathVariable("studentId") Long studentId, @RequestParam(required = false) String name,
        @RequestParam(required = false) String email){
        studentService.updateStudent(studentId, name, email);
    }
}
```

```
}
```

- **@RestController**
`@RestController` is a convenience annotation for creating Restful controllers. It is a specialization of `@Component` and is autodetected through classpath scanning. It adds the `@Controller` and `@ResponseBody` annotations. It converts the response to JSON or XML
- **@RequestMapping(path= "api/v1/student")**
Dient zur Mapping eines HTTP Requests mit path gebe ich die genaue URL an (<http://localhost:8080/api/v1/student>) wenn keine path Angabe gegeben wird bezieht sich das direkt auf <http://localhost:8080>.
- **@Autowired**
`@Autowired` Annotation in Spring Boot wird verwendet, um eine Bean automatisch mit einer anderen Bean zu verbinden. (dependency injection)
- **@GetMapping**
Diese Annotation wird zum lesen verwendet CRUD(Read)
In diesem Fall wird sie zum lesen der StudentenListe verwendet.
- **@PostMapping**
Diese Annotation wird zum hinzufügen verwendet CRUD(Create)(Die POST-Methode hingegen ist eine Prozessoperation, die eine Zielressourcenspezifische Semantik hat, die die Semantik von CRUD-Operationen ausschließt.)
In diesem Fall wird sie verwendet um einen Neuen Studenten hinzuzufügen.
- **@DeleteMapping**
Diese Annotation wird zum löschen verwendet CRUD>Delete)
Zum löschen eines Studentens mittels der selbst erstellten ID.
- **@PutMapping**
Diese Annotation wird zum updaten verwendet CRUD(Update)
- **@RequestBody**
Einfach ausgedrückt, ordnet die Annotation `@RequestBody` den HttpRequest-Hauptteil einem Übertragungs- oder Domänenobjekt zu, wodurch die automatische Deserialisierung des eingehenden HttpRequest-Hauptteils auf ein Java-Objekt ermöglicht wird.
In dieser Methode werden die Daten zum neuen Studenten Objekt hinzugefügt und gleichzeitig in die Datenbank hinzugefügt.
- **@PathVariable**
Ist eine Spring-Annotation, die angibt, dass ein Methodenparameter an eine URI-Vorlagenvariable gebunden werden soll. Wenn der Methodenparameter `Map<String, String>` ist, wird die Map mit allen Pfadvariablenamen und Werten gefüllt.
- **@RequestParam**
Wird verwendet, um die Formulardaten zu lesen und automatisch an den in der bereitgestellten Methode vorhandenen Parameter zu binden. Es ignoriert also die Anforderung des `HttpServletRequest`-Objekts, die bereitgestellten Daten zu lesen.

StudentConfig

```

@Configuration
public class StudentConfig {

    @Bean
    CommandLineRunner commandLineRunner(StudentRepository repository){
        return args -> {
            Student mariam = new Student(
                "Mariam",
                "maria.jamal@gmail.com",
                LocalDate.of(2000, Month.NOVEMBER, 15));

            Student alex = new Student(
                "Alex",
                "alex@gmail.com",
                LocalDate.of(2000, Month.NOVEMBER, 15));

            repository.saveAll(List.of(mariam, alex));
        };
    }
}

```

In dieser Klasse werden die verschiedenen Studenten erstellt.

StudentService

```

@Service
public class StudentService {

    private final StudentRepository studentRepository;

    public StudentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    public List<Student> getStudent(){
        return studentRepository.findAll();
    }

    public void addNewStudent(Student student) {
        Optional<Student> studentOptional = studentRepository.findStudentByEmail(student.getEmail());
        if (studentOptional.isPresent()){
            throw new IllegalStateException("email taken");
        }
        studentRepository.save(student);
    }

    public void deleteStudent(Long studentId) {
        boolean exists = studentRepository.existsById(studentId);
        if (!exists){
            throw new IllegalStateException( "student with id" + studentId + "does not exists");
        }
        studentRepository.deleteById(studentId);
    }

    @Transactional
    public void updateStudent(Long studentId, String name, String email) {
        Student student = studentRepository.findById(studentId).orElseThrow(() -> new IllegalStateException("student with id" + studentId +

        if(name != null && name.length() > 0 && !Objects.equals(student.getName(), name)){
            student.setName(name);
        }
    }
}

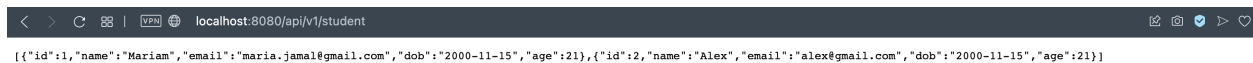
```

```

        if(email != null && email.length() > 0 && !Objects.equals(student.getEmail(), email)){
            Optional<Student> studentOptional = studentRepository.findStudentByEmail(email);
            if(studentOptional.isPresent()){
                throw new IllegalStateException("email taken");
            }
            student.setEmail(email);
        }
    }
}

```

In dieser Klasse wird CRUD eingesetzt Create, Read, Update und Delete. Dazu werden Abfragen gestellt und falls diese zutreffen bekommen wir eine Exception zurück. Dadurch werden die Fehler schneller gefunden.



```

[{"id":1,"name":"Marian","email":"maria.jamal@gmail.com","dob":"2000-11-15","age":21},{ "id":2,"name":"Alex","email":"alex@gmail.com","dob":"2000-11-15","age":21}]

```