**Create Custom Acuator Endpoint**

@Component  
@Endpoint(id = "creators", enableByDefault = true)  
public class CustomActuator {  
 private final List<String> creators = List.*of*("Eduard");  
  
 @ReadOperation  
 public List<String> getCreators(){  
 return creators;  
 }  
 @ReadOperation  
 public boolean checkCreator(@Selector String name){  
 return creators.contains(name);  
 }  
 @WriteOperation  
 public String addCreator(@Selector String name){  
 creators.add(name);  
  
 return name+" was added.";  
 }  
 @DeleteOperation  
 public String deleteCreator(@Selector String name){  
 creators.remove(name);  
  
 return name+" was deleted";  
 }  
  
}

@Endpoint – aici specificam care va fi numele la actuator, si acum va fi /actuator/creators

@ReadOperation – este pentru get, adica cand scriem /actuator/creators

@Selector – e ceea ce e pus dupa /, adica gen /actuator/creators/Eduard

@WriteOperation – e pentru post

@DeleteOperation – pentru delete

Desi enableByDefault e true, tot va trebui asta:

management.endpoints.web.exposure.include=creators

**@PropertySource vs @ConfigurationProperties**

* @PropertySource – adauga proprietiatile in Environment la Spring
* @ConfigurationProperties(prefix = “prefix”)

**@ConfigurationProperties**

* Cand sunt foarte multe fielduri, nu e prea comod sa tot folosim mereu @Value
* Anotatia @ConfigurationProperties ne scapa de a folosi mereu @Value
* Acum, dependenta data e mai mult pentru documentatie

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
</dependency>

Daca vom folosi /\*\* \*/ deasupra la field, dupa ce vom da mvn clean install, vom observa ca daca vom pune mouse deasupra la proprietati din .properties, ele deja vor arata documentatia pusa de noi

* Acum, sa zicem ca avem proprietatile in application.properties:

student.first-name = Eduard  
student.last-name=Mititiuc

Si avem clasa Student:

@Component  
@Getter  
@Setter  
@ConfigurationProperties(prefix = "student")  
public class Student {  
 private String firstName;  
 private String lastName;  
}

Deci, anotatia

@ConfigurationProperties(prefix = "student")

Va face ca sa se ia toate proprietatile ce incep cu student si sa se seteze valorile lor in fieldurile care corespund ca nume.

Atentie! Punem pune ca proprietate firstName sau first-name, acest – va fi interpretat ca litera mare oricum.

Deci: firstName = first-name

**Conditional Configuration(Specific doar lui Spring Boot)**

* @ConditionalOnProperty(name = “property”, havingValue= “value”)

De ex:

@Component

@ConditionalOnProperty(name = "student.create",havingValue = "true")

Beanul dat va fi creat si pus in context doar daca proprietatea student.create are valaorea true in .properties

student.create=true

* ConditionalOnBean(clasa.class)

Beanul se va crea doar daca exista un asa bean de tipul celui specificat in anotatie. De ex:

@RestController  
@ConditionalOnBean(Student.class)  
public class REST {  
 @Autowired  
 private Student student;

Daca un bean de tip Student nu exista, REST nu se va crea ca bean

* ConditionalOnMissingBean(clasa.class)

@ConditionalOnMissingBean(Student.class)

Beanul se va crea doar daca un bean de tip Student nu exista in context

* ConditionalOnJava(JavaVersion.EIGHT) – doar pentru JAVA 8
* ConditionalOnExpression() – combina mai multe properties

@ConditionalOnExpression("${student.create:true} && '${student.first-name}'.equals('Eduard')")

* ConfitionalOnResource(resources={“prop.test”}) – beanul se creaza doar daca exista proprietatea prop.test

**Custom condition annotation**

public class TimeCondition implements Condition {  
 @Override  
 public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
 return LocalTime.*now*().getHour() > 12 && LocalTime.*now*().getHour() < 21;  
 }  
}

Deci, trebuie de implementat Condition

Dupa, folosim clasa noastra cu @Conditional(clasa)

@RestController  
@Conditional(TimeCondition.class)  
public class REST {

**Events**

* **Application Event** – evente care sunt declansate de Spring
* Eventurile sunt folosite in crearea aplicatiilor care trebuie sa fie constiente de schimbarile din environment
* De exemplu, o aplicatie poate sa raspunda cand userul se logheaza ori delogheaza
* Spring Application Event se bazeaza pe Observer design patter
* **Observer** – design pattern in care un obiect este capabil sa observe starea unui alt obiect si sa raspunda corespunzator
* **In cazul la Spring Application Events, observer este aplicatia si obiectul ce este observat e application evironment**
* Spring Application Event ne permite sa cream custom event-driven applications care pot fi refolosite in ale aplicatii fara a fi necesar sa se rescrie iar codul de la 0.
* Pasi de creare:

1. Cream o clasa pe care eventul o va intercepta si o marcam ca@Component:

@Component  
@Getter  
@Setter  
@ToString  
public class Student {  
  
 private String firstName;  
 private String lastName;  
  
}

1. Cream o clasa, ce va fi un event listener pentru diferite evenimente:

@Component  
public class StudentEventListener {  
 @EventListener  
 public void newStudentCreated(Student student){  
 System.*out*.println("A new student was got: \n"+student);  
 }  
}

Anotatia @Component este obligatorie. Totodata, fiecare metoda ce va intercepta un eveniment trebuie notata cu @EventListener, si anume argumentul ei va fi tipul de obiect interceptat de metoda listener

1. Pentru a arunca un eveniment, vom avea nevoie de un ApplicationEventPublisher, si vom folosi metoda .publishEvent(obiect)

@RestController  
public class REST {  
 @Autowired  
 private Student student;  
 @Autowired  
 private ApplicationEventPublisher eventPublisher;  
  
 @GetMapping("/")  
 public Student student(){  
 student.setFirstName("Mititiuc");  
 student.setLastName("Eduard");  
 eventPublisher.publishEvent(student);  
 return student;  
 }

}

**Enumeration in JPA**

DrinkType este o enumeratie, si anume @Enumerated(EnjumType.STRING) va spune ca vrem sa trimitem in tabel valoarea string



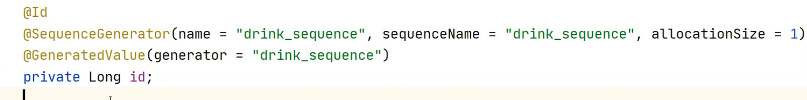
**No Entity in JSON**

Uneori, nu avem nevoie sa trimitem o entitate de raspuns la user, ci doar statusul, si asta o putem face cu @ResponseStatus(code = HttpStatus.TIP)



**Sequence**

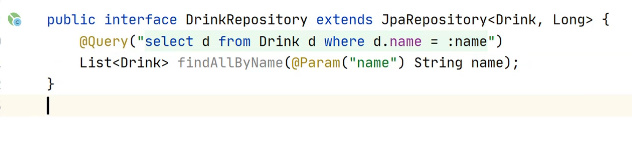
Unele baze de date, ca Oracle, folosesc Sequence pentru a genea key



**Cum scapam de erorile din loguri cand userul face bad request**



**Custom Query Spring Data**



@Param(“nume”) e cel de dupa : din query

**@DynamicPropertySource**

* Anotatia data ne permite sa adaugam in mod dinamic proprietati in Environment la Spring, ce in mod normal ar trebui scrise in application.properties
* Tototdata, va inlocui cele deja existente
* @SpringBootTest   
  public class ArticleLiveTest {  
    
   @DynamicPropertySource  
   static void registerPgProperties(DynamicPropertyRegistry registry) {  
   registry.add("spring.datasource.url",  
   () -> String.format("jdbc:postgresql://localhost:%d/prop", postgres.getFirstMappedPort()));  
   registry.add("spring.datasource.username", () -> "postgres");  
   registry.add("spring.datasource.password", () -> "pass");  
   }  
    
   // tests are same as before  
  }

Metoda trebuie sa fie statica

Metoda trebuie sa ia un argument de tip DynamicPropertyRegistry

**Paginarea**

* Pentru a face mai bine aranjate datele din REST API, putem folosi paginarea
* @RestController  
  public class Rest {  
   @Autowired  
   private StudentDAO studentDAO;  
    
   @GetMapping("/")  
   public Page<Student> getAll(@PageableDefault Pageable pageable){  
   return studentDAO.findAll(pageable);  
   }  
  }
* Page<Student> - Jackson converteste obiectul asta in pagina aramjata
* @PageDefault creaza un obiect de tip Pageable cu nr de pagini default si alte configuratii default
* JPA ofera la repozitorii si metode ce iau un argument de tip Pageable, pentru a utiliza configuratiile din el si sa returneze un Page pe baza lui
* Putem si noi crea un Pageable

@RestController  
public class Rest {  
 @Autowired  
 private StudentDAO studentDAO;  
  
 @GetMapping("/")  
 public Page<Student> getAll(){  
 Pageable pageable = Pageable.*ofSize*(2);

return studentDAO.findAll(pageable);  
 }  
}

Pageable.ofSize(elemente per pagina)

Page are metodele:

- getContent() ce returneaza o lista

- getTotalPages

- getNumber

- getSize

- getTotalElements

.....

Pentru a modifica pagina:

[localhost:8080/?page=2](http://localhost:8080/?page=2)

Pentru a modifica size:

[localhost:8080/?page=2](http://localhost:8080/?page=2)&size=5

Pentru a sorta

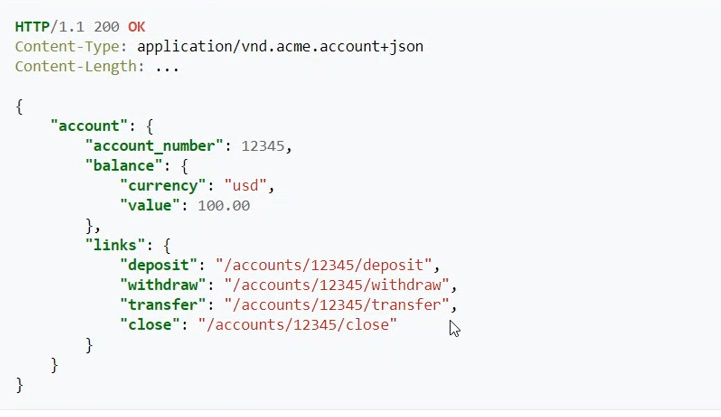
[localhost:8080/?page=2](http://localhost:8080/?page=2)&sort=id

H2 cu Spring Boot application

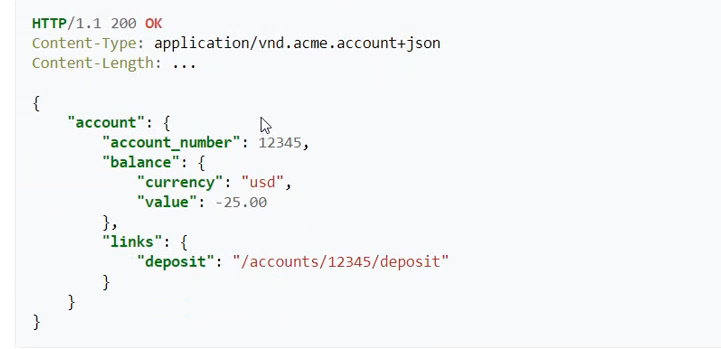
spring.jpa.hibernate.ddl-auto=update

**HateOAS**

* REST Api facut cu paginare este in mare parte suficient.
* Totusi, uneori e necesar ca in rest api sa specificam si linkuri cu operatiile ce pot fi efectuate cu datele returnate



HATEOAS se bazeaza anume pe aceea ca nu vor aparea anume toate linkurile posibile, ci anume acelea care sunt disponibile. De ex, in imagine se vede ca in cont la user sunt 100$, deci se poate deschide un depozit, face transfer etc. Dar, daca contul ar fi gol sau in minus, unele operatii nu ar mai fi disponibile



* Principiul la HATEOS este ca API trebuie sa ghideze clientul prin aplicatie returnand informatii relevante despre potentialii urmatori pasi.

Pasi de creare:

1. Dependeneta:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-hateoas</artifactId>  
</dependency>

1. Clasele @Entity trebuie sa extinda clasa RepresentationModel<Entitate>

@Entity  
@Table(name = "Student")  
@Data  
public class Student extends RepresentationModel<Student> {  
 @Id  
 @Column(name = "id")  
 private Long id;  
 @Column(name = "first\_name")  
 private String firstName;  
 @Column(name = "last\_name")  
 private String lastName;  
}

1. Clasa RepresentationModel ofera metoda add()
2. @GetMapping("/{id}")  
   public Student getById(@PathVariable Integer id){  
    Student student = studentDAO.findById(id).get();  
    student.add(*linkTo*(*methodOn*(Rest.class).getAll()).withSelfRel());  
     
    return student;  
   }

Ea se va asigura sa adaugae fielduri, sa le prelucreze etc.

add() – adauga un Link in JSON, ca si cum un field inca la lista de linkuri

linkTo() – creaza propriu zis linkul pe baza la o metoda dintr-un obiect

methodOn(Clasa.class) – specifica clasa de la care se ia metoda

withSelfRel() – spuen ca relatia cu linkul e self, adica acel link e legat de resursa itself

* Pentru a obtine lista de useri:
* @GetMapping("/")  
  public List<Student> getAll(){  
   List<Student> list = studentDAO.findAll();  
   for(Student s : list){  
   s.add(*linkTo*(*methodOn*(Rest.class).getById(s.getId())).withSelfRel());  
   }  
    
   return list;  
  }

Dar, vedem ca la urma nu e link propriu zis pentru lista:



Putem rezolva problema asa:

@GetMapping("/")  
public CollectionModel<Student> getAll(){  
 List<Student> list = studentDAO.findAll();  
 for(Student s : list){  
 s.add(*linkTo*(*methodOn*(Rest.class).getById(s.getId())).withSelfRel());  
 }  
 CollectionModel<Student> collectionModel = CollectionModel.*of*(list);  
 collectionModel.add(*linkTo*(*methodOn*(Rest.class).getAll()).withSelfRel());  
  
 return collectionModel;  
}

Si vedem la urma:

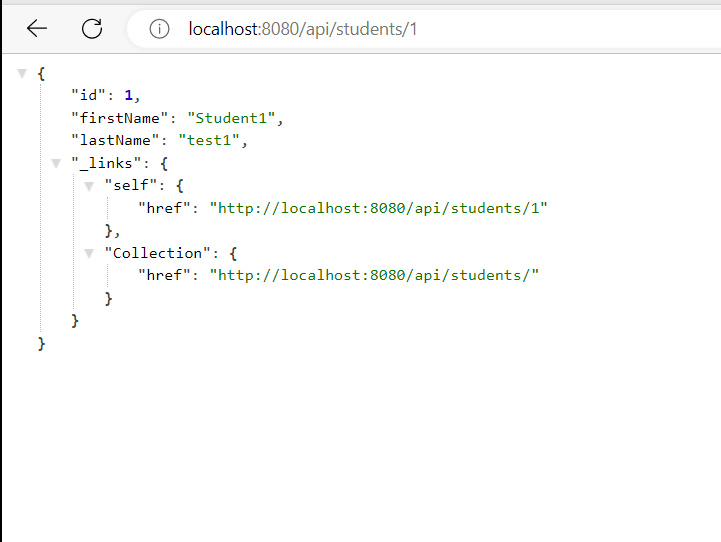


CollectionModel este pentru a stoca o lista anume.

Acum, poate vrem ca atunci cand accesam datele unui user, sa arate ceva vre-un link custom, setat de noi, cu nume ales de noi. De ex, sa fie un link catre el insusi, si unul catre toti studentii, si sa fie numi Collection:

@GetMapping("/{id}")  
public Student getById(@PathVariable int id){  
 Student student = studentDAO.findById(id).get();  
 student.add(*linkTo*(*methodOn*(Rest.class).getById(id)).withSelfRel())  
 .add(*linkTo*(*methodOn*(Rest.class).getAll()).withRel("Collection"));  
  
 return student;  
}

Observam ca mai adaugam un .add si apoi withRel(“Nume la relatie”) si obtinem:



withSelfRel doar pune “self” ca nume la relatie si atat

* Avem niste cod care se tot repeta. Putem evita asta asa:

@Component  
public class StudentModelAssembler implements RepresentationModelAssembler<Student, EntityModel<Student>> {  
 @Override  
 public EntityModel<Student> toModel(Student student) {  
 EntityModel<Student> entityModel = EntityModel.*of*(student);  
 entityModel.add(*linkTo*(*methodOn*(Rest.class).getById(student.getId())).withSelfRel())  
 .add(*linkTo*(*methodOn*(Rest.class).getAll()).withRel("Collection"));  
  
 return entityModel;  
 }  
}

Clasa trebuie sa implementeze RepresentationModelAssembler

@RestController  
@RequestMapping("/api/students")  
public class Rest {  
 @Autowired  
 private StudentDAO studentDAO;  
 @Autowired  
 private StudentModelAssembler assembler;  
  
  
 @GetMapping("/")  
 public CollectionModel<Student> getAll(){  
 List<Student> list = studentDAO.findAll();  
 for(Student s : list){  
 s.add(*linkTo*(*methodOn*(Rest.class).getById(s.getId())).withSelfRel());  
 }  
 CollectionModel<Student> collectionModel = CollectionModel.*of*(list);  
 collectionModel.add(*linkTo*(*methodOn*(Rest.class).getAll()).withSelfRel());  
  
 return collectionModel;  
 }  
 @GetMapping("/{id}")  
 public ResponseEntity<EntityModel<Student>> getById(@PathVariable int id){  
 Student student = studentDAO.findById(id).get();  
  
 return new ResponseEntity<>(assembler.toModel(student), HttpStatus.*ACCEPTED*);  
 }  
 @PostMapping("/")  
 public ResponseEntity<EntityModel<Student>> save(@RequestBody Student student){  
 Student newStudent = studentDAO.save(student);  
  
 return new ResponseEntity<>(assembler.toModel(newStudent),HttpStatus.*OK*);  
 }  
 @DeleteMapping("/{id}")  
 public ResponseEntity<Student> delete(@PathVariable int id){  
 studentDAO.deleteById(id);  
 return ResponseEntity.*noContent*().build();  
 }  
}

EntityModel – anume ea pasteraza entitatea ce va fi transormaat in JSON cu HATEAOS