SpringBoot

# Auditing in SpringBoot (spring data envers)

3. Annotate entity classes with

@Audited  
@EntityListeners(AuditingEntityListener.**class**)

@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@Entity  
@Table(name = **"employee"**)  
@Accessors(chain = **true**)  
@Audited  
@EntityListeners(AuditingEntityListener.**class**)  
**public class** Employee **extends** BaseEntity{  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private** Long **id**;  
   
 @Column   
 **private** String **name**;  
  
 @Column   
 **private** Double **salary**;  
  
 @Column   
 **private** String **city**;  
  
 @OneToOne(fetch = FetchType.***LAZY***,cascade = CascadeType.***ALL***)   
 **private** Account **account**;  
  
 @OneToMany(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***, orphanRemoval = **true**)  
 @JoinColumn(name = **"employee\_id"**)  
 @NotAudited  
 **private** List<Document> **documentList**;  
  
}

Here

@Audited - When you annotate a JPA entity class with **@Audited**, Hibernate Envers will automatically track changes to instances of that entity, allowing you to maintain a history of modifications over time.

@EntityListeners - These listeners can be useful for tasks like auditing, validation, or custom behavior that needs to be triggered in response to entity state changes

**@NotAudited :** If you dont want the relationship to be audited mark it as *org.hibernate.envers.NotAudited*

4. Extend your Repository with RevisionRepository

**public interface** EmployeeRepository **extends** RevisionRepository<Employee, Long, Long>, JpaRepository<Employee, Long>, EmployeeRepositoryCustom,  
 QuerydslPredicateExecutor<Employee>,  
 QuerydslBinderCustomizer<QEmployee> {  
  
 @Override  
 **default void** customize(QuerydslBindings bindings, QEmployee root) {  
 bindings.bind(String.**class**).first((StringPath path, String value) -> path.containsIgnoreCase(value));  
 }  
}

*RevisionRepository has following methods to get Version history*

Certainly, here are the common methods you might find in a **RevisionRepository** for querying and retrieving historical data using Hibernate Envers, presented in a table format:

| **Method Signature** | **Description** |
| --- | --- |
| **findRevision(Number revision)** | Retrieves a specific revision. |
| **findRevisions(Class<T> entityClass, ID id)** | Retrieves all revisions for a specific entity with the given ID. |
| **findRevisions(Class<T> entityClass)** | Retrieves all revisions for a specific entity class. |
| **findRevisionsBetween(Class<T> entityClass, fromDate, toDate)** | Retrieves all revisions for a specific entity class that occurred between two given timestamps. |
| **findRevisionsByEntityIdAndRevisionNumber(Class<T> entityClass, ID id, Number revision)** | Retrieves a specific revision of an entity with the given ID. |

That’it . once you ran the application you can see both entity tables & audit tables get created in the DB.

Getting History API Example

//Controller

@ApiOperation(**"Get Employee Version History By Id"**)  
@GetMapping(**"/{id}/history"**)  
**public** List<EmployeeDto> getEmployeeHistoryById(@PathVariable(**"id"**) Long id, Pageable pageable) {  
 **return employeeService**.getEmployeeHistoryById(id, pageable);  
}

//service

@Override  
**public** List<EmployeeDto> getEmployeeHistoryById(Long id, Pageable pageRequest) {  
 **if** (!**employeeRepository**.findById(id).isPresent()) {  
 **throw new** EntityNotFoundException(Employee.**class**, id);  
 }  
 List<EmployeeDto> employeeHistoryDtoList = **null**;  
 **try** {  
 Pageable pageable = PageRequest.*of*(pageRequest.getPageNumber(), pageRequest.getPageSize(), RevisionSort.*desc*());  
 Page<Revision<Long, Employee>> employeeRevisions = **employeeRepository**.findRevisions(id, pageable);  
  
 employeeHistoryDtoList = employeeRevisions.stream().map((p) ->  
 **employeeMapper**.toDto(p.getEntity())  
 ).collect(Collectors.*toList*());  
  
 } **catch** (DataAccessException ex) {  
 ex.printStackTrace();  
 }  
 **return** employeeHistoryDtoList;  
}

//Repository

**public interface** EmployeeRepository **extends** RevisionRepository<Employee, Long, Long>, JpaRepository<Employee, Long>, EmployeeRepositoryCustom,  
 QuerydslPredicateExecutor<Employee>,  
 QuerydslBinderCustomizer<QEmployee> {  
  
   
 **default void** customize(QuerydslBindings bindings, QEmployee root) {  
 bindings.bind(String.**class**).first((StringPath path, String value) -> path.containsIgnoreCase(value));  
 }  
}

Errors

org.springframework.dao.InvalidDataAccessApiUsageException: Service is not yet initialized; nested exception is java.lang.IllegalStateException: Service is not yet initialized

* Make sure audit tables are created in DB or not
* Make sure **integration.envers.enabled**: true should be TRUE

# MapStruct – All Secnarios

**package** com.employee.api.v1.model.mapper;  
  
  
**import** com.employee.api.v1.model.dto.AccountDto;  
**import** com.employee.api.v1.model.dto.DocumentDto;  
**import** com.employee.api.v1.model.dto.EmployeeDto;  
**import** com.employee.dao.entity.Account;  
**import** com.employee.dao.entity.Document;  
**import** com.employee.dao.entity.Employee;  
**import** org.mapstruct.\*;  
  
**import** java.util.Date;  
**import** java.util.List;  
  
  
*/\*\*  
 \** ***@author*** *Satya Kaveti  
 \*/*@Mapper(componentModel = **"spring"**, unmappedTargetPolicy = ReportingPolicy.***IGNORE***)  
**public interface** EmployeeMapper **extends** BaseMapper {  
 @Mapping(target = **"createdDate"**, source = **"createdDate"**, qualifiedByName = **"dateToLong"**)  
 @Mapping(target = **"modifiedDate"**, source = **"modifiedDate"**, qualifiedByName = **"dateToLong"**)  
 **void** toEntity(EmployeeDto employeeDto, @MappingTarget Employee target);  
  
  
 @Mapping(target = **"createdDate"**, source = **"createdDate"**, qualifiedByName = **"longToDate"**)  
 @Mapping(target = **"modifiedDate"**, source = **"modifiedDate"**, qualifiedByName = **"longToDate"**)  
 EmployeeDto toDto(Employee entity);  
  
 AccountDto mapEntityToDtoAccount(Account account);  
  
 Account mapDtoToEntityAccount(AccountDto accountDtos);  
  
  
 List<DocumentDto> mapEntityListToDtoListForDocument(List<Document> document);  
  
 List<Document> mapDtoListToEntityListForDocument(List<DocumentDto> documentDtos);  
  
  
 DocumentDto mapEntityToDtoDocument(Document document);  
  
 Document mapDtoToEntityDocument(DocumentDto documentDtos);  
   
 @Named(**"longToDate"**)  
 **default** Date mapLongToDate(Long value) {  
 **if** (value == **null**) {  
 **return null**;  
 }  
 **return new** Date(value);  
 }  
  
 @Named(**"dateToLong"**)  
 **default** Long mapDateToLong(Date date) {  
 **if** (date == **null**) {  
 **return null**;  
 }  
 **return** date.getTime();  
 }  
}

Same Property name mapping

* Same property with different name but same datadtype
* Property with different datatype (Long- Date)
* List <Objects> inside class
* Inner Object level lists

# SpringBoot Querydsl Guide

Done:

<https://medium.com/@satyacodes/querydsl-with-springboot-a7817f45d557>

Querydsl is a framework that enables the construction of statically typed SQL-like queries through its fluent API. Spring Data modules offer integration with Querydsl through QuerydslPredicateExecutor.

The **Querydsl** is a third-party library that lets us define type-safe queries. It also makes query generation relatively easy through its fluent DSL APIs. In general, the native SQL or the Java Persistence Query Language (JPQL) does not provide any means to validate the correctness of the queries at compile time. Any query syntax or other issues are detected at run time and cause the application to fail.

Maven integration

Add the following dependencies to your Maven project:

<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-apt</artifactId>  
 <version>${querydsl.version}</version>  
 <scope>provided</scope>  
</dependency>  
  
<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-jpa</artifactId>  
 <version>${querydsl.version}</version>  
</dependency>

1.querydsl-apt dependency is the annotation processing tool (APT) that processes the annotations in source files. This tool generates the so-called Q-types.

The Q-type classes are directly related to the entity classes of the application, but these classes are prefixed with the letter **Q**. For instance, if we have a Users class marked with the @Entity annotation in the application, then the generated Q-type will reside in a QUsers.java source file.

2.querydsl-jpalibrary is the Querydsl which is designed to be used together with the JPA application. Similarly, Querydsl has support for other data stores such as MongoDB withquerydsl-mongodbmodule. Refer to<http://www.querydsl.com/>for more details.

And now, configure the Maven APT plugin:

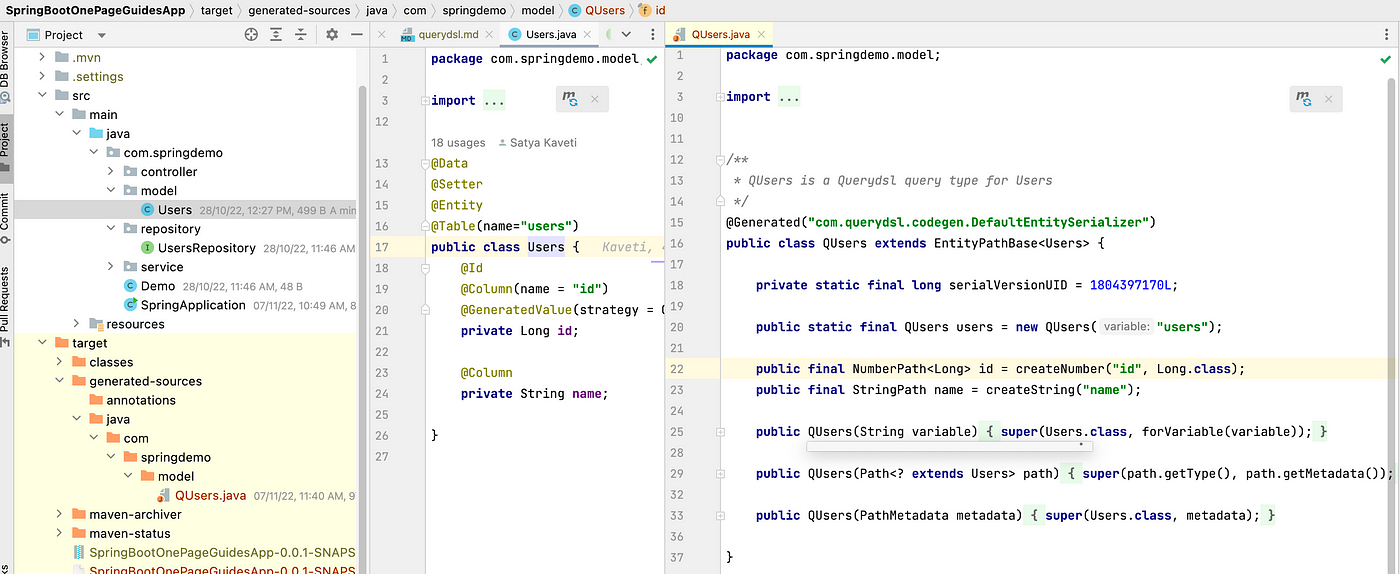
<project>  
 <build>  
 <plugins>  
 ...  
 <plugin>  
 <groupId>com.mysema.maven</groupId>  
 <artifactId>apt-maven-plugin</artifactId>  
 <version>1.1.3</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>process</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>target/generated-sources/java</outputDirectory>  
 <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 ...  
 </plugins>  
 </build>  
</project>

This plugin makes sure that the Q-types are generated automatically during the process goal of Maven build. The output directory is the folder where the the Q-types are generated

The JPAAnnotationProcessor finds domain types annotated with the javax.persistence.Entity annotation and generates query types for them.

If you use Hibernate annotations in your domain types you should use the APT processor com.querydsl.apt.hibernate.HibernateAnnotationProcessor instead.

Run mvn clean install and you will get your Query types generated into target/generated-sources/java.



we have Users.java as Entity class

@Data  
@Setter  
@Entity  
@Table(name="users")  
public class Users {  
 @Id  
 @Column(name = "id")  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 @Column  
 private String name;  
  
 @Column  
 private Double salary;  
}

Let us now create the UsersRepository interface which extends JpaRepository and the QuerydslPredicateExecutor

public interface UsersRepository extends JpaRepository<Users, Long> , QuerydslPredicateExecutor<Users> {  
   
}

TheQuerydslPredicateExecutor interface provides several overloaded methods that lets us allow executing Querydsl predicates. Following are the methods of QuerydslPredicateExecutor:

public interface QuerydslPredicateExecutor<T> {  
 Optional<T> findOne(Predicate predicate);  
 Iterable<T> findAll(Predicate predicate);  
 Iterable<T> findAll(Predicate predicate, Sort sort);  
 Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);  
 Iterable<T> findAll(OrderSpecifier<?>... orders);  
 Page<T> findAll(Predicate predicate, Pageable pageable);  
 long count(Predicate predicate);  
 boolean exists(Predicate predicate);  
}

The methods in the **QuerydslPredicateExecutor** takes a Predicate and also provides an option to use pagination, sorting, ordering and so on.

we have Controller and save call will save 10 Users

@RestController  
@RequestMapping("/api")  
public class UsersController {  
  
 @Autowired  
 private UsersService service ;  
  
 // display list of users  
 @GetMapping("/all")  
 public List<Users> listUserss(Model model) {  
 System.out.println(" ===> listUserss");  
 return service.getAllUserss();  
 }  
  
 @GetMapping("/addall")  
 public String addDummyUsers() {  
 System.out.println(" ===> listUserss");  
 for(int i=0; i<10; i++){  
 Users user = new Users();  
 user.setName(nameGen());  
 user.setSalary(salaryGen());  
 System.out.println("added : "+user);  
 service.saveUser(user);  
 }  
 return "Users added";  
 }  
  
 @GetMapping("/querysln")  
 public String querysln() {  
 System.out.println(" ===> querysln");  
 return service.querySln();  
 }  
 private String nameGen()  
 {  
 String AlphaNumericStr = "ABCDEFGHIJKLMNOPQRSTUVWXYZSATYA";  
 StringBuilder s = new StringBuilder(6);  
 int i;  
 for ( i=0; i<8; i++) {  
 int ch = (int)(AlphaNumericStr.length() \* Math.random());  
 s.append(AlphaNumericStr.charAt(ch));  
 }  
 return s.toString();  
 }  
 private Double salaryGen()  
 {  
 Random r = new Random();  
 int low = 5000;  
 int high = 90000;  
 int result = r.nextInt(high-low) + low;  
 return Double.parseDouble(result+"");  
 }  
}

A screenshot of a computer

Description automatically generated

We use EntityManager and UsersRepository to create Querydsl queries. ServiceImpl method

public String querySln() {  
 QUsers users = QUsers.users;  
  
 String result = "";  
  
 System.out.println(" -- users having salary greater than 10000 order by salary --");  
 result = result + "-- users having salary greater than 10000 order by salary --";  
 //BooleanExpression implements Predicate  
 //goe='greater than or equals'  
 BooleanExpression booleanExpression = QUsers.users.salary.goe(10000);  
 OrderSpecifier<Double> orderSpecifier = QUsers.users.salary.asc();  
 Iterable<Users> users1 = repository.findAll(booleanExpression, orderSpecifier);  
 for (Users user : users1) {  
 result = result + "\n " + user;  
 }  
  
 System.out.println(" -- users salary between 3000 and 5000 --");  
 result = result + "\n \n \n -- users salary between 3000 and 5000 --";  
 BooleanExpression booleanExpression2 = QUsers.users.salary.between(30000, 50000);  
 Iterable<Users> users3 = repository.findAll(booleanExpression2);  
 for (Users user : users3) {  
 result = result + "\n " + user;  
 }  
 System.out.println(" -- find users YNDFBTUG --");  
 result = result + "\n \n \n-- find users YNDFBTUG --";  
 BooleanExpression booleanExpression3 = QUsers.users.name.eq("YNDFBTUG");  
 Optional<Users> opt = repository.findOne(booleanExpression3);  
 result = result + "\n " + opt.get();  
  
 return result;  
 }

Sample Employee Pagable with SearchCriteria

public Page<EmployeeDto> findEmployee(Pageable pageRequest, SearchCriteriaDTO searchCriteria) {  
 QEmployee qEmployee = QEmployee.Employee;  
   
 BooleanBuilder builder = new BooleanBuilder();  
 builder.and(qEmployee.status.eq("PERMENENT");  
   
 if(!StringUtils.isEmpty(searchCriteria.getEmployeeId())) {  
 builder.and(qEmployee.employeeId.eq(searchCriteria.getEmployeeId()));  
 }  
   
 if(!StringUtils.isEmpty(searchCriteria.getRole())) {  
 builder.and(qEmployee.role.eq(searchCriteria.getRole()));  
 }  
   
 if(!StringUtils.isEmpty(searchCriteria.getName())) {  
 builder.and(qEmployee.name.containsIgnoreCase(searchCriteria.getName()));  
 }  
   
 JPAQuery<Employee> query = new JPAQuery<>(entityManager);  
 query.from(qEmployee);  
 query = query.where(builder);  
 Long totalCount = query.fetchCount();  
   
 query.offset(pageRequest.getOffset());  
 query.limit(pageRequest.getPageSize());  
 query.orderBy(qEmployee.reportId.desc());  
   
 List<Employee> employeeLst = query.fetch();  
 Type listType = new TypeToken<List<EmployeeDto>>() {}.getType();   
   
 ModelMapper modelMapper = mapperUtil.getEmployeeMapper();  
   
 List<EmployeeDto> returnValue = modelMapper.map(EmployeeLst, listType);  
 Page<EmployeeDto> result = new PageImpl(returnValue, pageRequest, totalCount);  
 return result;  
 }

Example Pageable search results with Employee Entity- Complete flow from Controller to Backend Repository

\*/  
//1. Controller   
//==========================================  
public class EmployeeController {  
  
 @Autowired  
 private final EmployeeService employeeService;  
  
 @ApiOperation("Returns a page of all Employee list")  
 @ApiResponses(value = {@ApiResponse(code = 200, message = "Success"), @ApiResponse(code = 400, message = "Bad Request"), @ApiResponse(code = 500, message = "Internal Server Error"), @ApiResponse(code = 403, message = "Unauthorized")})  
 @PostMapping("/search")  
 public Page<EmployeeSearchResultsDto> searchAllEmployee(Pageable pageRequest, @RequestBody EmployeeSearchDto searchCriteria) {  
 return employeeService.searchAllEmployee(pageRequest, searchCriteria);  
 }  
}  
  
  
//==========================================  
//2.Service Class  
//==========================================  
 public Page<EmployeeSearchResultsDto> searchAllEmployee(Pageable pageRequest, EmployeeSearchDto searchCriteria) {  
 return employeeRepository.employeeSearchCriteria(pageRequest, searchCriteria);  
 }  
  
  
//==========================================  
//3.Repository Class which extends EmployeeRepositoryCustom,  
// - QuerydslPredicateExecutor<Employee>,  
// - QuerydslBinderCustomizer<QEmployee>  
//==========================================  
public interface EmployeeRepository extends RevisionRepository<Employee, Long, Long>, JpaRepository<Employee, Long>, EmployeeRepositoryCustom,  
 QuerydslPredicateExecutor<Employee>,  
 QuerydslBinderCustomizer<QEmployee> {  
  
 @Override  
 default void customize(QuerydslBindings bindings, QEmployee root) {  
 bindings.bind(String.class).first((StringPath path, String value) -> path.containsIgnoreCase(value));  
 }  
}  
  
  
  
//==========================================  
//4.EmployeeRepositoryCustom  
//==========================================  
public interface EmployeeRepositoryCustom {  
  
 Page<EmployeeSearchResultsDto> employeeSearchCriteria(Pageable pageRequest, EmployeeSearchDto searchCriteria);  
}  
  
  
  
//==========================================  
//5.Pagable Implementation EmployeeRepositoryCustomImpl  
//==========================================  
@Repository  
@Slf4j  
public class EmployeeRepositoryCustomImpl implements com.employee.dao.repository.EmployeeRepositoryCustom {  
  
 @PersistenceContext  
 private EntityManager entityManager;  
  
 @Autowired  
 private EmployeeSearchResultMapper employeeSearchResultMapper;  
  
  
 @Override  
 public Page<EmployeeSearchResultsDto> employeeSearchCriteria(Pageable pageRequest, EmployeeSearchDto searchCriteria) {  
  
 QEmployee qEmployee = QEmployee.employee;  
 BooleanBuilder builder = new BooleanBuilder();  
 applySearchCriteria(searchCriteria, qEmployee, builder);  
  
 JPAQuery<Employee> query = new JPAQuery<>(entityManager);  
 query = query.from(qEmployee);  
 query = query.where(builder);  
 Long totalCount = query.fetchCount();  
 query.offset(pageRequest.getOffset());  
 query.limit(pageRequest.getPageSize());  
 PathBuilder<Employee> entityPath = new PathBuilder<>(Employee.class, "employee");  
  
 for (Sort.Order order : pageRequest.getSort()) {  
 PathBuilder<Object> path = entityPath.get(order.getProperty());  
 query.orderBy(new OrderSpecifier(Order.valueOf(order.getDirection().name()), path));  
 }  
 List<Employee> result = query.fetch();  
 List<EmployeeSearchResultsDto> employeeDtoResult = employeeSearchResultMapper.toEmployeeSearchResultsDtoList(result);  
 return new PageImpl<>(employeeDtoResult, pageRequest, totalCount);  
  
 }  
  
  
 private void applySearchCriteria(EmployeeSearchDto searchCriteria, QEmployee qEmployee, BooleanBuilder builder) {  
 try {  
  
 if (Objects.nonNull(searchCriteria.getName()) && StringUtils.isNotBlank(searchCriteria.getName())) {  
 builder.and(qEmployee.name.containsIgnoreCase(searchCriteria.getName()));  
 }  
 if (Objects.nonNull(searchCriteria.getSalary())) {  
 builder.and(qEmployee.salary.eq(searchCriteria.getSalary()));  
 }  
 if (Objects.nonNull(searchCriteria.getCity()) && StringUtils.isNotBlank(searchCriteria.getCity())) {  
 builder.and(qEmployee.city.containsIgnoreCase(searchCriteria.getCity()));  
 }  
 } catch (Exception ex) {  
 log.error("Exception Occurred while getting search results", ex);  
  
 }  
 }  
  
}  
  
  
//==========================================  
//6.Search Results Mapper  
//==========================================  
@Mapper(componentModel = "spring")  
public interface EmployeeSearchResultMapper {  
  
 List<EmployeeSearchResultsDto> toEmployeeSearchResultsDtoList(List<Employee> employeeEntityList);  
  
 EmployeeSearchResultsDto toEmployeeSearchResultsDto(Employee employee);  
  
}

# SpringBoot Dynamic Task Scheduling

<https://satyacodes.gitlab.io/SpringBoot-Task-Scheduling-Guide.html>

<https://satyacodes.medium.com/springboot-task-scheduling-guide-dfbce8ae0019>

# Java – Create XML schema from DTO / Entity Classes using JAXB

JAXB provides a fast and convenient way to marshal (write) Java objects into XML and unmarshal (read) XML into objects. It supports a binding framework that maps XML elements and attributes to Java fields and properties using Java annotations.

## Maven Dependencies

<dependency>  
 <groupId>javax.xml.bind</groupId>  
 <artifactId>jaxb-api</artifactId>  
 <version>2.3.1</version>  
</dependency>  
<dependency>  
 <groupId>com.sun.xml.bind</groupId>  
 <artifactId>jaxb-core</artifactId>  
 <version>2.3.0.1</version>  
</dependency>  
<dependency>  
 <groupId>com.sun.xml.bind</groupId>  
 <artifactId>jaxb-impl</artifactId>  
 <version>2.3.3</version>  
</dependency>

## JAXB Annotations

JAXB uses Java annotations to augment the generated classes with additional information. Adding such annotations to existing Java classes prepares them for the JAXB runtime.

Let's take our Employee DTO and understand the JAXB annotations

@XmlRootElement  
@XmlAccessorType(XmlAccessType.***FIELD***)  
**public class** EmployeeDto {  
  
 **private** Long **id**;  
  
 @XmlElement(required = **true**)  
 **private** String **name**;  
  
 @XmlElement(name = **"salary"**, required = **true**)  
 **private** Double **salary**;  
  
 @XmlElement  
 **private** String **city**;  
  
 @XmlElement(required = **true**)  
 **private** AccountDto **account**;  
  
 @XmlElement  
 **private** List<DocumentDto> **documentList**;  
  
  
 **private** String **createdBy**;  
 **private** String **modifiedBy**;  
  
 **private** Date **createdDate**;  
 **private** Date **modifiedDate**;  
  
}

**@XmlRootElement**. This annotation should be applied to the class that you want to be treated as the root element when converting it to XML.

**@XmlAccessorType(XmlAccessType.FIELD)** to specify that JAXB should use field access for XML binding. This means that fields (instance variables) are directly used for reading and writing XML, rather than getter and setter methods

**@XmlType**: define the order in which the fields are written in the XML file

**@XmlTransient** annotation in JAXB is used to mark a class, field, or property as transient, indicating that it should be excluded from XML serialization and deserialization.

**@XmlElement** is an annotation used to indicate that the annotated field or property should be mapped to an XML element.

* **name="salary "** specifies the name of the XML element to which the field or property should be mapped. In this case, it's "**salary**."
* **required=true** indicates that the XML element is required in the XML document. If **required** is set to **true**, it means that the XML element must be present when marshalling (converting Java objects to XML) and unmarshalling (converting XML to Java objects). If the element is missing, it may result in an error.

## JAXB-2 Maven Plugin

This plugin uses the Java API for XML Binding (JAXB), version 2+, to generate Java classes from XML Schemas (and optionally binding files) or to create XML schema from an annotated Java class

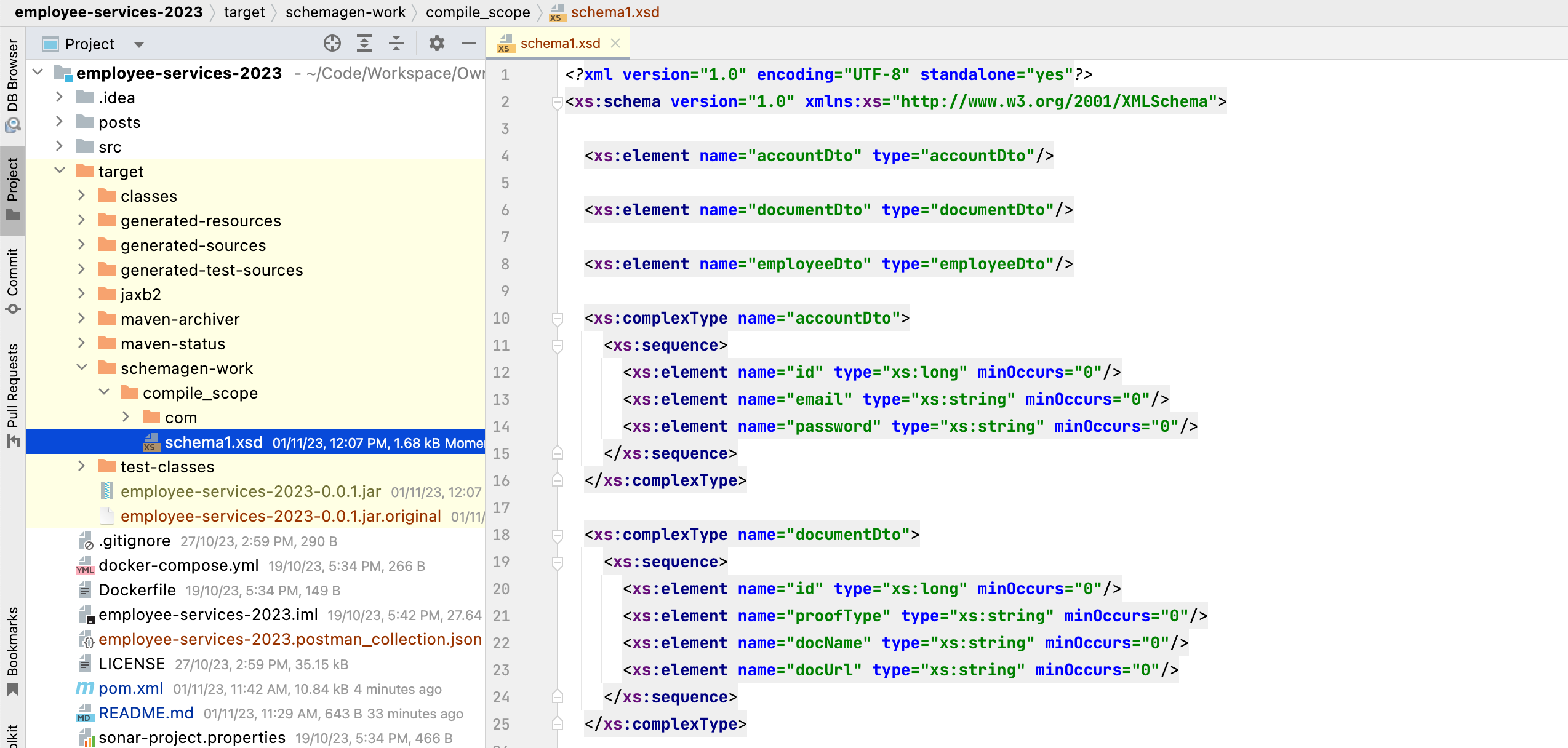
Update maven Build plugin

<**build**>  
 <**plugins**>   
  
 <**plugin**>  
 <**groupId**>org.codehaus.mojo</**groupId**>  
 <**artifactId**>jaxb2-maven-plugin</**artifactId**>  
 <**version**>2.5.0</**version**>  
 <**executions**>  
 <**execution**>  
 <**id**>schemagen</**id**>  
 <**goals**>  
 <**goal**>schemagen</**goal**>  
 </**goals**>  
 </**execution**>  
 </**executions**>  
 <**configuration**>  
 <**sources**>  
 *<!-- Update XML DTOs location here -->* <**source**>src/main/java/com/employee/api/v1/model/dto</**source**>  
 </**sources**>  
 </**configuration**>  
 </**plugin**>  
  
 </**plugins**>  
</**build**>

**<source> - here we need to update Dto classes location**

At last, do mvn clean install

Now, we can see generated schema document inside target/



# SpringBoot – Custom exception handling with @ControllerAdvice

Before going further, Let’s see how we do validation in spring boot.

In Spring Boot, you can perform validation using the Java Bean Validation framework, which is based on the Java Validation API (JSR-380) and the **javax.validation** package

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

To perform validation, we need to annotate the fields of the bean/dto class with validation annotations from the **javax.validation.constraints** package. For example, you can use **@NotNull**, **@Size**, **@Email**, and more to specify validation constraints.

@Data  
@JsonIgnoreProperties(ignoreUnknown = **true**)  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@XmlRootElement  
@XmlAccessorType(XmlAccessType.***FIELD***)  
**public class** EmployeeDto {  
  
 @XmlElement(required = **true**)  
 **private** Long **id**;  
  
 @XmlElement(required = **true**)  
 @NotNull  
 @Size(min = 4, max = 10)  
 **private** String **name**;  
  
 @XmlElement(name = **"salary"**, required = **true**)  
 @Digits(integer = 10, fraction = 2)  
 **private** Double **salary**;  
  
 @XmlElement  
 @Size(max = 20)  
 **private** String **city**;  
  
 @XmlElement(required = **true**)  
 **private** AccountDto **account**;   
}

To perform validation whenever user submits employee payload to API call, we need to annotate @Valid annotation against the dto parameter like below.

@ApiOperation(**"Create a new Employee or update exist "**)  
@PostMapping  
**public** EmployeeDto save(@RequestBody @Valid EmployeeDto employeeDto) {  
 **return employeeService**.save(employeeDto, id);  
}

Ok, lets run & check the validation messages.

Payload:

{

"name": "HYDGHAGHGHAHGAHGAHGAHGHYDGHAGHGHAHGAHGAHGAHG",

"salary": 25000,

"city": "HYDGHAGHGHAHGAHGAHGAHG",

"account": {

"email": "satya@gmail.com",

"password": "password",

"dob": "1990-10-27"

},

Response:

{

"exception": "org.springframework.web.bind.MethodArgumentNotValidException",

"fingerprint": "036d809a-175d-4db1-ba77-9c5258c1e244",

"errors": [

{

"code": "javax.validation.constraints.Size.message",

"arguments": {

"min": 0,

"max": 20,

"invalid": "HYDGHAGHGHAHGAHGAHGAHG",

"property": "city"

},

"message": "size must be between 0 and 20"

}

],

"status": "Bad Request"

}

Okay, city allows max of 10 char, buts its more than that – and we got the error.

But, This is NOT READABLE, the end user don’t want this technical details. Just want what’s the field & what he need to pass.

This we can achive by implementing Custom exception handling with **@ControllerAdvice**

## Custom exception handling with @ControllerAdvice

**@ControllerAdvice** is an annotation in Spring Boot that allows you to define global exception handling for your controllers. You can use it to handle exceptions that occur in multiple controllers or to provide a centralized place for handling exceptions throughout your application.

1. Create a custom exception class. I want to display HTTP Status with all the error messages.

**public class** ErrorDetails {  
  
 **private** HttpStatus **code**;  
  
 **private** List<String> **message**;  
  
}

2. Create a **CustomExceptionHandler** class & annotate with **@ControllerAdvice.**  It contains methods for handling specific exceptions. You can specify the exception types to handle in the **@ExceptionHandler** annotation on each method.

Iam also extending  **ResponseEntityExceptionHandler &** overridinghandleMethodArgumentNotValid()**.** This is usefull in case of Rest API, because It provides a way to handle exceptions and customize the HTTP response entity that is returned to the client.

@ControllerAdvice  
**public class** CustomExceptionHandler **extends** ResponseEntityExceptionHandler {  
  
 @ExceptionHandler(EntityNotFoundException.**class**)  
 **public final** ResponseEntity<Object> handleDBEntityNotFound(EntityNotFoundException ex, WebRequest webRequest) {  
 ErrorDetails error = **new** ErrorDetails();  
 error.setCode(HttpStatus.***INTERNAL\_SERVER\_ERROR***);  
 List<String> message = **new** ArrayList<>();  
 message.add(ex.getMessage());  
 error.setMessage(message);  
 **return new** ResponseEntity<Object>(error, error.getCode());  
 }  
  
  
 @Override  
 **protected** ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {  
 ErrorDetails error = **new** ErrorDetails();  
 error.setCode(HttpStatus.***BAD\_REQUEST***);  
 List<String> message = **new** ArrayList<String>();  
  
 List<String> collect = ex.getBindingResult().getFieldErrors().stream().filter(Objects::*nonNull*)  
 .map(m -> (m.getField() + **" "** + m.getDefaultMessage())).collect(Collectors.*toList*());  
 message.addAll(collect);  
 error.setMessage(message);  
 **return new** ResponseEntity<Object>(error, HttpStatus.***BAD\_REQUEST***);  
 }  
}

That’s it now run the application again with the same payload.

{

"code": "BAD\_REQUEST",

"message": [

"city size must be between 0 and 20",

"name size must be between 4 and 10"

]

}

That’s all on a high level.

# SpringBoot — export Java Objects to JSON & JSON to Object

Its very simple tutorial, wrote for reference purpose.

Im using Jackson ObjectMapper to read and write JSON data into Java Objects

Let’s add dependency

<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.13.0</version>   
</dependency>

using ObjectMapper class for marshalling and unmarshalling the JSON.

ObjectMapper mapper = new ObjectMapper();  
  
//Java Object -> JSON  
String json = mapper.writeValueAsString(employeeDto);  
  
//JSON -> Java Object  
EmployeeDto dto = mapper.readValue(json, EmployeeDto.class);

Export Java Objects to JSON file

@GetMapping("/json/export")  
public ResponseEntity<Resource> exportEmployeesToJson() {  
 try {  
 List<EmployeeDto> employees = employeeService.getAllEmployees();  
 String jsonData = objectMapper.writeValueAsString(employees);  
 ByteArrayResource resource = new ByteArrayResource(jsonData.getBytes());  
 String fileName = "employees\_" + LocalDateTime.now() + ".json";  
 return ResponseEntity.ok().header(HttpHeaders.CONTENT\_DISPOSITION, "attachment; filename=\"" + fileName + "\"").contentLength(resource.contentLength()).contentType(MediaType.APPLICATION\_JSON).body(resource);  
 } catch (Exception e) {  
 log.error(e.getMessage(), e); }  
}

Import JSON file and convert in to Java Entity

@PostMapping("/json/import")  
public EmployeeDto importEmployees(MultipartFile file) {  
 try {  
 String jsonContent = new String(file.getBytes());  
 EmployeeDto employeeDto = objectMapper.readValue(jsonContent, new TypeReference<>() {  
 });  
 return employeeService.save(employeeDto, employeeDto.getId());  
 } catch (Exception e) {  
 log.error("error : {}", e.getMessage());  
 }  
}

//Service Class

public class EmployeeServiceImpl implements EmployeeService {  
  
 @Autowired  
 private EmployeeRepository employeeRepository;  
  
 @Autowired  
 private EmployeeMapper employeeMapper;  
  
 private final ObjectMapper objectMapper;  
  
 @Override  
 public EmployeeDto save(EmployeeDto employeeDto, Long employeeId) {  
 log.info("save start :::: employeeId:{}", employeeId);  
 try {  
 Boolean createRequest = StringUtils.isEmpty(employeeId) ? Boolean.TRUE : Boolean.FALSE;  
 Employee employee = getEmployee(employeeId, createRequest);  
 employeeMapper.toEntity(employeeDto, employee);  
 Employee entity = employeeRepository.save(employee);  
 return employeeMapper.toDto(entity);  
 } catch (Exception ex) {  
 log.error("Error while saving employee", ex);  
 throw new BusinessException("Save Failed");  
 }  
 }  
  
  
 @Override  
 public List<EmployeeDto> getAllEmployees() {  
 List<Employee> employees = employeeRepository.findAll();  
 return employeeMapper.mapEntityListToDtoListForEmployee(employees);  
 }  
  
}

# Spring Boot Role-based Authentication with Keycloak

## Keycloak Intro

Keycloak is an open-source identity and access management (IAM) tool. Keycloak implements almost all standard IAM protocols like OAuth 2.0, OpenID, and SAML.

Keycloak provides the following features:

* OpenID Connect support.
* OAuth 2.0 support.
* SAML support.
* Single-Sign On and Single-Sign Out for browser applications.
* Social Login - Enable login with Google, GitHub, Facebook, Twitter, and other social networks.
* User Federation - Sync users from LDAP and AD servers & [many more(Read More).](https://www.keycloak.org/docs/latest/server_admin/index.html) [**Cloud Doc**](https://documentation.cloud-iam.com/?pk_vid=b1c0510047749b951699432592034cf4)

In our application, we are using an OAuth library to integrate Keycloak with our application. 

## Setting up a Keycloak server

Setting up a Keycloak server Locally

* Download Distribution powered by Quarkusfrom here : <https://www.keycloak.org/downloads>
* Extract, go to bin run ./kc.sh start-dev
* The default port is 8080. Open <http://0.0.0.0:8080/> in browser

Setting up a Keycloak Online Cloud server Free

<https://www.cloud-iam.com/>

* Set up a free cluster, you will get credentials via email.
* Login to the admin account & update the password.

## Configuring Keycloak server

**1. Creating a Realm**

From the Master drop-down menu, click Add Realm and typespringboot-realm in the name field and click Create button.

**2. Creating a client**

A client represents a resource that can be accessed by certain users.

Provide ClineID & keep the default configs and save.

Then we need to provide a **Valid Redirect URIs**– for this add springboot service url



**3. Creating Roles**

Mainly there are 2 role types in Keycloak.

1. **Realm Role:** It is a global role, belonging to that specific realm. This can access from any client and map it to any user. Global Admin and Admin roles can be considered examples of this.
2. **Client Role:** It is a role which belongs only to that specific client. These roles cannot be accessed from a different client. This can only map to the users from that client. *Example Roles:* Employee, User etc

**Client Role:** : select the client springboot-client and click Roles Press the Create role button. Create admin, user roles here.

A screenshot of a computer

Description automatically generated

**Realm roles:**  Navigate to the *Realm Roles* page to create roles.(app\_user, app\_admin) A screenshot of a computer

Description automatically generated

Now, add clinet roles to releam roles. For that Click one of the roles and select Add associated roles in the Action drop down.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**4. Creating Users**

Create users & assign roles to them.

Set password for the users Click the **Credentials** tab and press **Set password**button. turn off the **Temporary** switchA screenshot of a computer

Description automatically generated

## Generating Access tokens with Keycloak API

You can send a post request to this token endpoint to get a token. You can use one of the users we have already created ( user1 , user2 or admin ) as credentials.

A screenshot of a computer

Description automatically generated



Use Postman to get the token with below payload.

curl --location 'http://localhost:8080/realms/springboot-realm/protocol/openid-connect/token' \

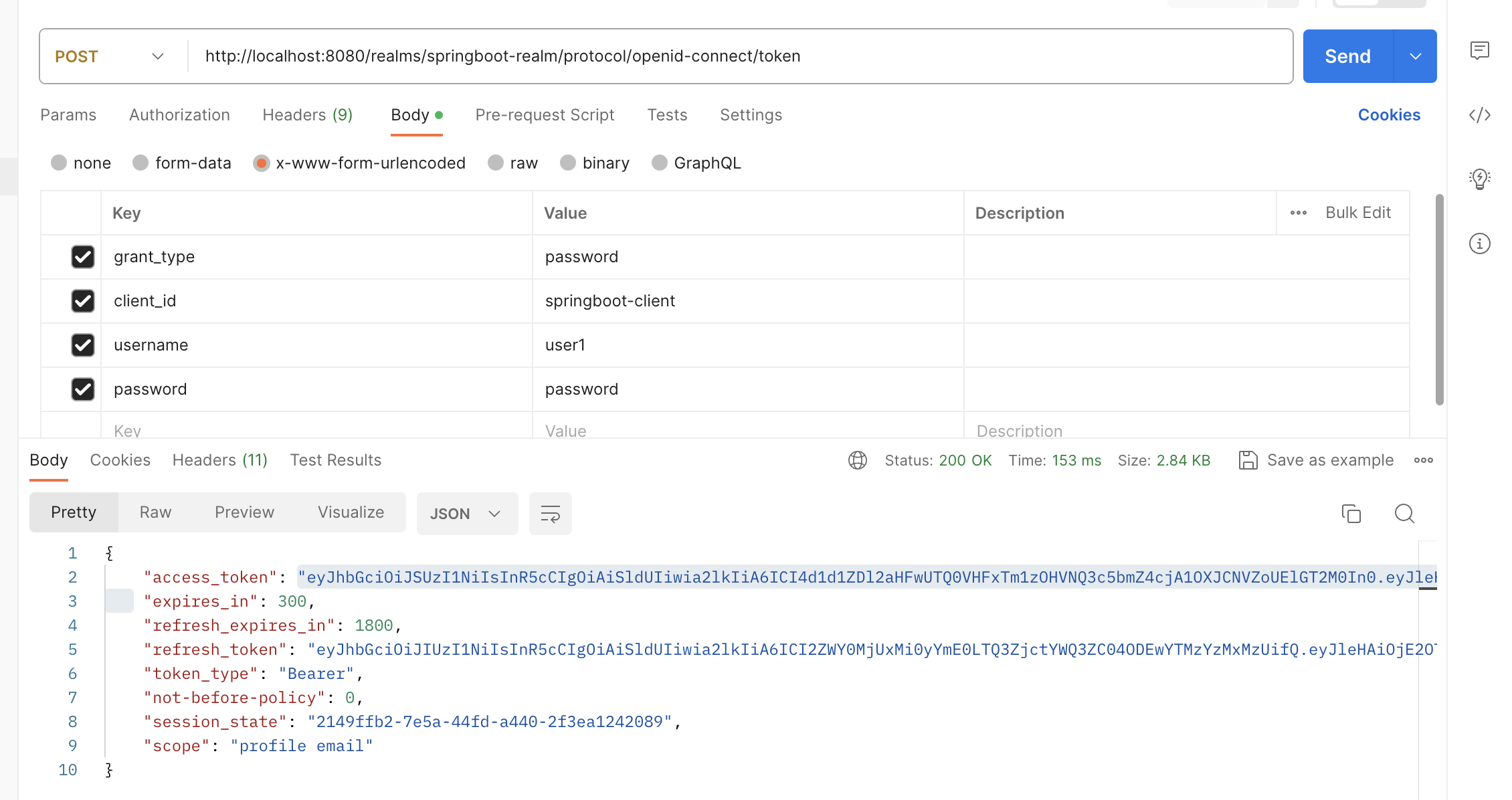
--header 'Content-Type: application/x-www-form-urlencoded' \

--data-urlencode 'grant\_type=password' \

--data-urlencode 'client\_id=springboot-client' \

--data-urlencode 'username=user1' \

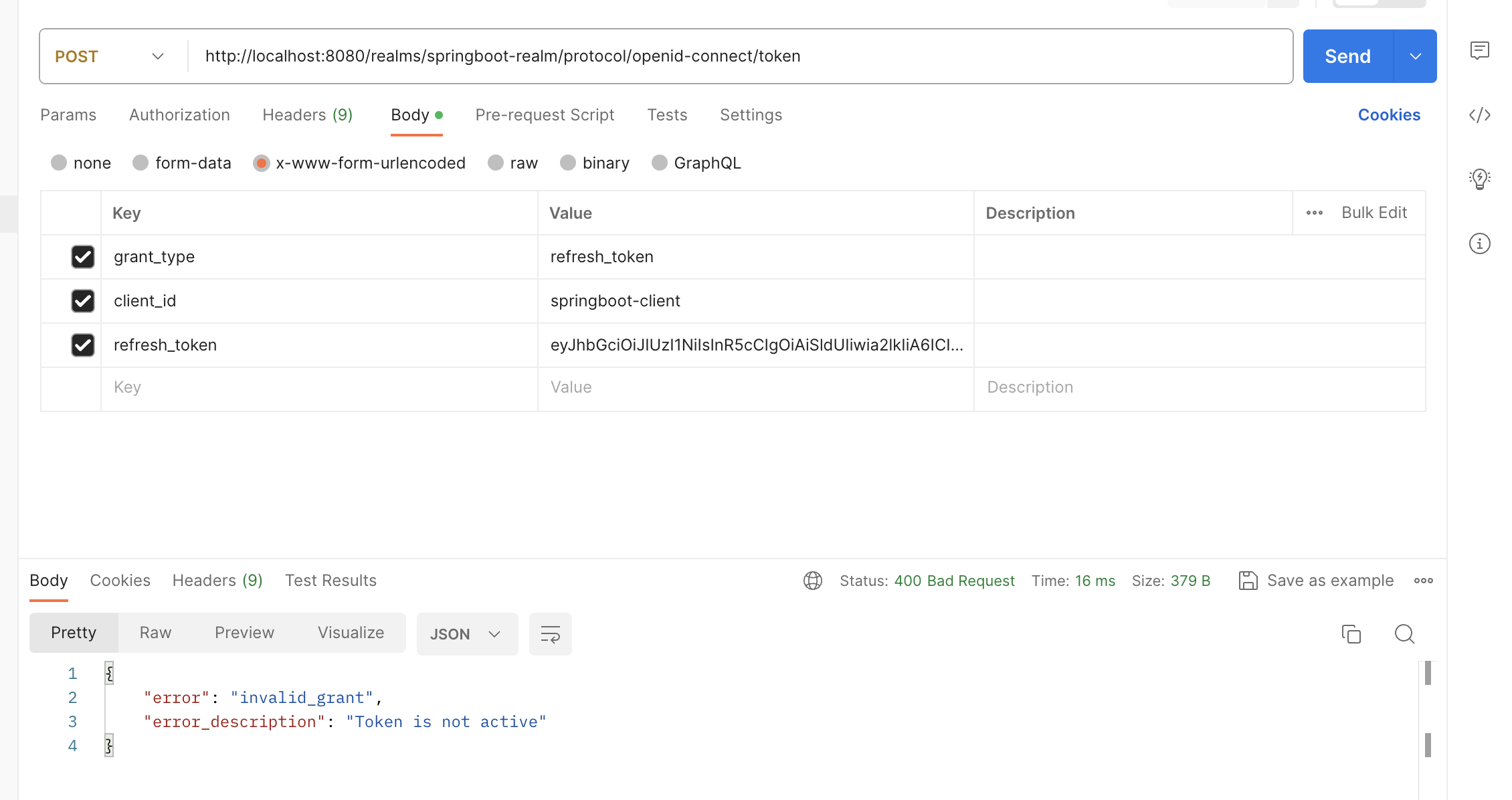
--data-urlencode 'password=password'



Now you can copy that access\_token and decode it using the [jwt.io](https://jwt.io/) website to see the information inside that.

#### Getting refresher token

If the access\_token expired, you can send a request to the same endpoint with refresh\_token & grant\_type as refresh\_token as below.



# Spring Security

<https://docs.spring.io/spring-security/reference/servlet/getting-started.html>

Add Security Dependency.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

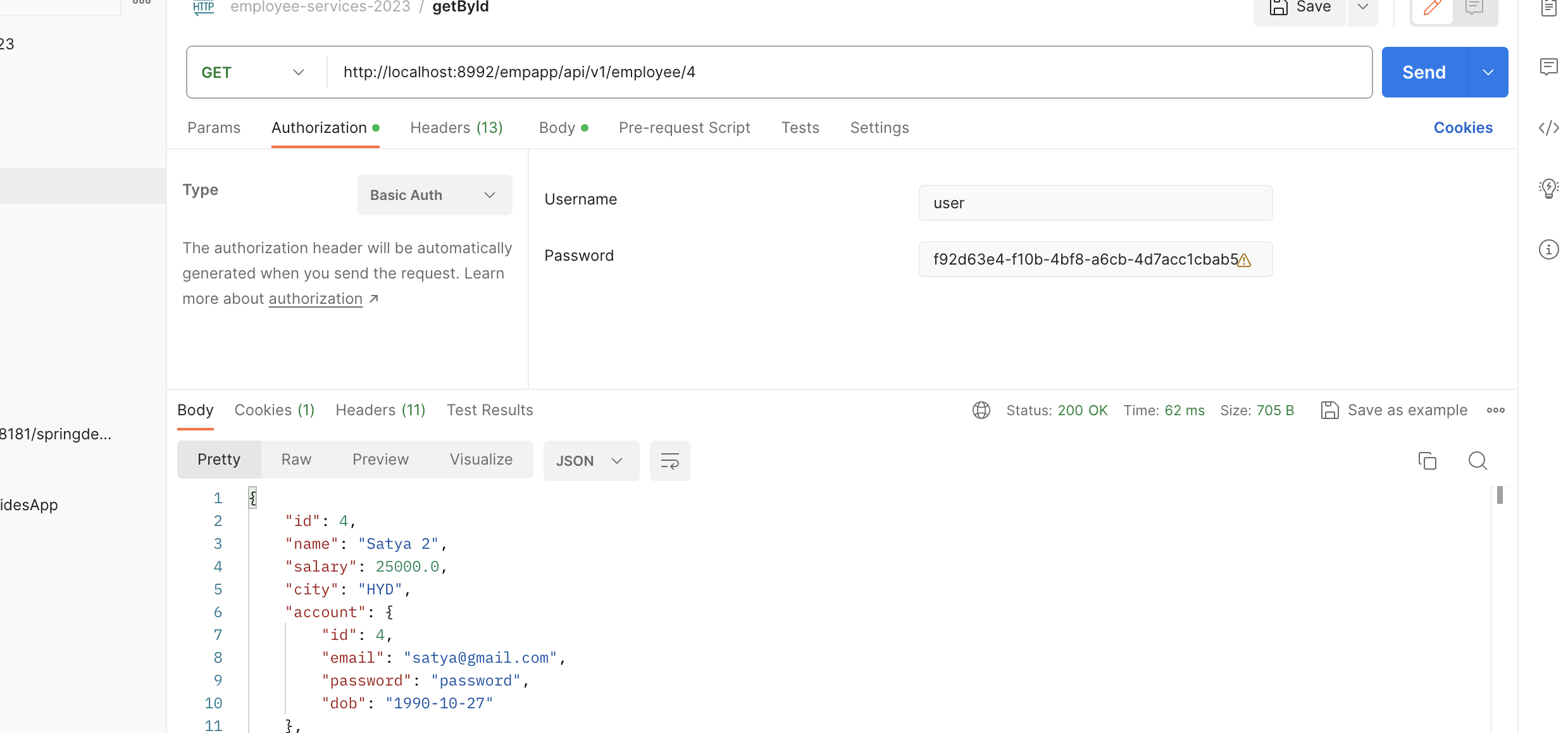
</dependency>

Run the Application & you can see the default security password on the console

security password: a134381d-3db8-4d6c-9f62-8d5010c67a0f

Let’s Hit Any API. You will get **unauthorized** response.

Select Basic Auth from Postman, fill details & hit again



This is the default Spring Security configuration provided by Spring.

@EnableWebSecurity

@Configuration

public class DefaultSecurityConfig {

@Bean

@ConditionalOnMissingBean(UserDetailsService.class)

InMemoryUserDetailsManager inMemoryUserDetailsManager() {

String generatedPassword = *// ...;*

return new InMemoryUserDetailsManager(User.withUsername("user")

.password(generatedPassword).roles("ROLE\_USER").build());

}

@Bean

@ConditionalOnMissingBean(AuthenticationEventPublisher.class)

DefaultAuthenticationEventPublisher defaultAuthenticationEventPublisher(ApplicationEventPublisher delegate) {

return new DefaultAuthenticationEventPublisher(delegate);

}

}

1. Adds the @EnableWebSecurity annotation.
2. Publishes a [UserDetailsService](https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/user-details-service.html) @Bean with a username of user and a randomly generated password that is logged to the console
3. Publishes an [AuthenticationEventPublisher](https://docs.spring.io/spring-security/reference/servlet/authentication/events.html) @Bean for publishing authentication events

## 1.Username/Password Authentication (In Memmory)

Lets implement our own Authentication mechanism.

@Configuration  
@EnableWebSecurity  
**public class** SecurityConfig {  
 @Bean  
 **public** SecurityFilterChain securityFilterChain(HttpSecurity http) **throws** Exception {  
 http.authorizeHttpRequests((authorize) -> authorize.anyRequest().authenticated())  
 .httpBasic(Customizer.*withDefaults*())  
 .formLogin(Customizer.*withDefaults*());  
  
 **return** http.build();  
 }  
  
 @Bean  
 **public** UserDetailsService userDetailsService() {  
 UserDetails userDetails = User.*withDefaultPasswordEncoder*()  
 .username(**"user"**)  
 .password(**"password"**)  
 .roles(**"USER"**)  
 .build();  
  
 **return new** InMemoryUserDetailsManager(userDetails);  
 }  
}

Now check the console, it won't print password now.

Because we override the SecurityConfig, so now instead of generated password , we need to pass our own password, which is **password** in the above code.

A screenshot of a computer

Description automatically generated

This type of Step of authentication also called as in memory authentication. Because we are passing username and password in the code itself.

Now it will register the default username and password with UserDetialsService.

Now We need to implement AuthenticationManager by passing UserDetailsService & PasswordEncoder

The **AuthenticationManager** is a crucial component responsible for authenticating users. When a user attempts to log in, the **AuthenticationManager** uses various authentication providers to validate the user's credentials.

@Configuration  
@EnableWebSecurity  
**public class** SecurityConfig {  
 @Bean  
 **public** SecurityFilterChain securityFilterChain(HttpSecurity http) **throws** Exception {  
 http.authorizeHttpRequests((authorize) -> authorize.anyRequest().authenticated())  
 .httpBasic(Customizer.*withDefaults*())  
 .formLogin(Customizer.*withDefaults*());  
  
 **return** http.build();  
 }  
  
 @Bean  
 **public** UserDetailsService userDetailsService() {  
 UserDetails userDetails = User.*withDefaultPasswordEncoder*()  
 .username(**"user"**)  
 .password(**"password"**)  
 .roles(**"USER"**)  
 .build();  
  
 **return new** InMemoryUserDetailsManager(userDetails);  
 }  
  
 @Bean  
 **public** AuthenticationManager authenticationManager(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {  
 DaoAuthenticationProvider authenticationProvider = **new** DaoAuthenticationProvider();  
 authenticationProvider.setUserDetailsService(userDetailsService);  
 authenticationProvider.setPasswordEncoder(passwordEncoder);  
 **return new** ProviderManager(authenticationProvider);  
 }  
   
 @Bean  
 **public** PasswordEncoder passwordEncoder() {  
 **return** PasswordEncoderFactories.*createDelegatingPasswordEncoder*();  
 }  
}

**DaoAuthenticationProvider** : is one of the authentication providers provided by Spring Security. It's designed to handle authentication using a data access object (DAO) to retrieve user details from a a database/local objects.

**PasswordEncoder:** is an interface in Spring Security that provides a way to encode (hash) and verify passwords.

Steps involved for implementing our own Security Configuration

1. Create **SecurityConfig** class & annotate with @Configuration, @EnableWebSecurity
2. Configure **SecurityFilterChain** with URL’s & its access permissions
3. Configure **UserDetailsService** bean with user deatils
4. Configure **AuthenticationManager** bean with AuthenticationProvider & PasswordEncoder details.
5. Configure **PasswordEncoder** bean

## SpringBoot Example

<https://sultanov.dev/blog/migrate-from-spring-security-oauth-to-keycloak/>

| **Methods** | **Urls** | **Actions** |
| --- | --- | --- |
| POST | /api/auth/signup | signup new account |
| POST | /api/auth/login | login an account |
| GET | **api/v1/employee/search** | public access |
| GET | GET, UPDATE | access User’s content |
| GET | ADD, DELETE | access Admin’s content |

## Ref.

<https://www.bezkoder.com/spring-boot-jwt-authentication/>

<https://javainfinite.com/spring-boot/integrate-spring-boot-with-keycloak-example/>

<https://www.baeldung.com/spring-boot-keycloak>

<https://webcache.googleusercontent.com/search?q=cache:https://medium.com/geekculture/using-keycloak-with-spring-boot-3-0-376fa9f60e0b>