üK 223-Group 5

04.09.2021

Oliver Pettersson & Milena Blaser

Overview

This documentation explains how we as group five have solved the task of implementing several Endpoints to a Spring Boot backend, which has the goal of simulating a blog site. The implementation should reflect the knowledge gained during the ük 223 about multi-user-application, ways of security and the ACID principals.

Each Group has to fulfill the basic requirements of creating your own users, roles and authorities that can be used to test the application. These users, roles and authorities need to be saved persistently in a PostgreSQL database.

The implemented endpoints should only be able to be accessed by users with the necessary authorities.

Goals

1. Standardized Github Repository
2. Authentic Diagrams and Models
3. Even work distribution
4. Working Project
5. Conventional Naming
6. Formatted Code
7. No unused Code

Specific Task

The group-specific task for us was to create a MyListEntry model that includes all the information that belongs to a ListEntry. Those attributes are a title, a text, a user, a creation date as well as a way of determining the importance of the ListEntry. Important is that each ListEntry is owned by a specific user and that each user can have multiple ListEntries.

All CRUD Operations have to be implemented with Endpoints. Requirements for this were that only logged in users are able to look at the lists of other users and that when displaying the list the entries should be sorted by importance. Apart from this the only

users that are allowed to update or delete a ListEntry should be either the creator of the ListEntry or an admin.

MyListEntry Model & DTO'S

MyListEntry Model

```java
@Entity(name = "list_entry")
@Table
@Getter @Setter
@NoArgsConstructor
public class ListEntry {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private UUID id;

    @Column(name = "title")
    private String title;

    @Column(name = "text")
    private String text;

    @Column(name = "creation_date")
    private LocalDate creationDate;

    @Column(name = "importance")
    private int importance;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
    @JoinColumn(name = "user_id", referencedColumnName = "id")
    private User user;
```

We created the model for MyListEntry according to the requirements given in the project task. The table consists therefore of the columns id, title, text, creation date, importance and user. User is a column that we joined with the user table, by referencing the id of the user as a foreign key.

The relationship between those two tables is bi-directional because we save the user in each ListEntry and every ListEntry belongs to a list owned by a user.

We decided to change the FetchType to LAZY since we only need the user_ID and nothing else from the user. To make sure that the ListEntry doesn't reference itself multiple times, we used the JsonIgnoreProperies annotation.

ListEntryDTO

ListEntryDTOForOutput

ListEntryDTOForUpdateUser

ListEntryDTOForUpdateAdmin

Our Conventions

Branch Naming

The first words to the "/" are a description of the branch content, for example "bugfix" or "feature". The next number after the "/" is the number of the issue, which this branch is revolving around, after that two words make a short description of the branch

Commit Messages

Abbreviation	Meaning
DOC: <message>	Directly editing the documentation (Models etc.)
TEST: <message>	Edit/Create tests
WEB: <message>	Adding to a web layer
SERV: <message>	Adding to a service layer
REP: <message>	Adding to a data access layer
SEC: <message>	Adding to Spring Security (security domain)
GEN: <message>	General purpose commit (undefined)

Roles and Authorization

Green: All

Yellow: Only Own

Grey: Nothing

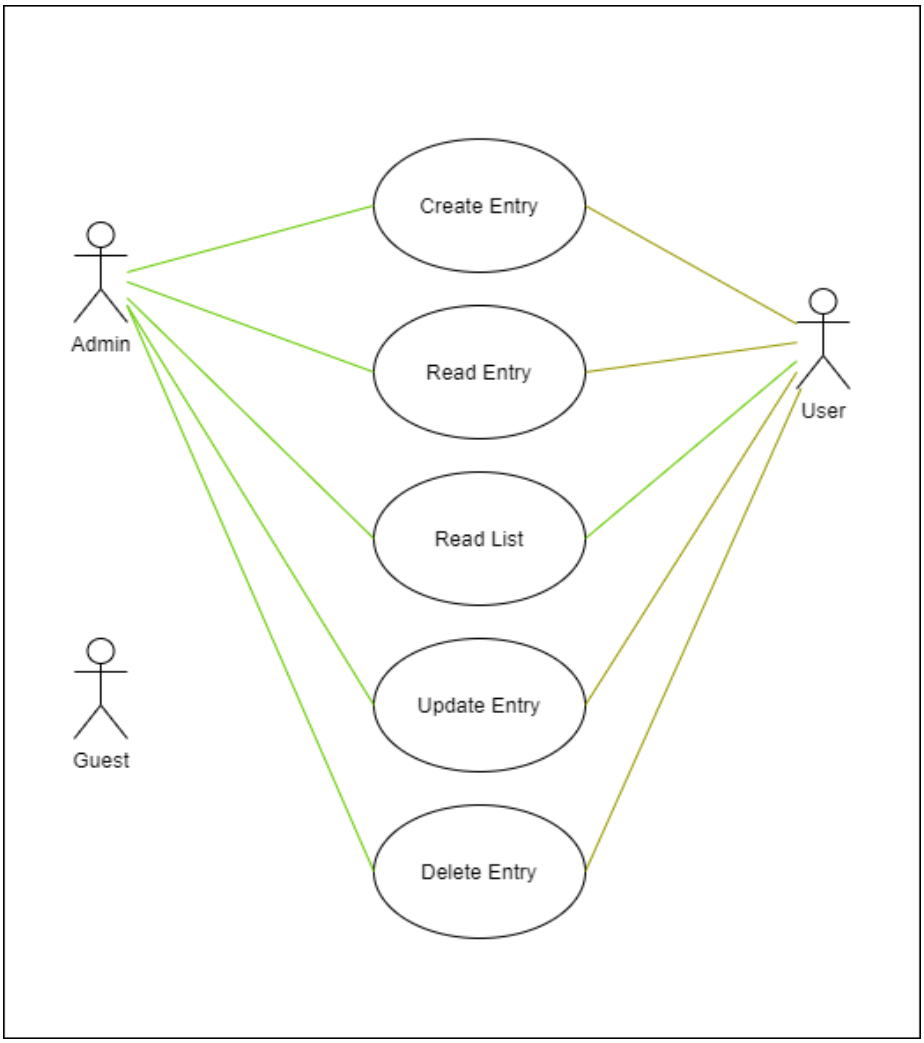
Role	Create ListEntry	Read ListEntry	Update ListEntry	Delete ListEntry
Admin				
User				
Guest*				

*not an actual role, describes the status of “user is not logged in” called “Guest” for the sake of convenience

Use Cases

Green: All

Yellow: Only Own



Successful Use Case

Use Case:	Delete all entries
Use Case ID:	1
Short Description:	All entries belonging to a given user are deleted.

Precondition:	An existing user is logged in. In this case user “james”.
Actor (Primary):	Logged in User
Actor (Secondary):	User(Owner of the List of ListEntries)

MainProcess:
1. The Use Case starts when the currently logged in user accesses the Delete-Endpoint via “http://localhost:8080/list/username”.

2 .The decoded user Credentials from the RequestHeader as well as the

3. In the ListEntryServiceImpl the deleteAllListEntries method checks whether the given username(james) belongs to an existing user.

4.All the ListEntries that belong to the searched user are saved in a List.

5. In the next step we check whether the current user(james) is an admin. If he is not we call the isOwner and hasCertainAuthority method to make sure that James is the owner of the listEntries and if he has the correct authorities to delete the List.

6. FOR each found listEntry:
→ the ListEntry is deleted.

7. The user receives the confirmation that all the entries have been deleted.

Postconditions:	The user “james” has now no ListEntries anymore, because all of them have been deleted

Alternative Flows:

1. The user whose list we tried to delete does not exist.

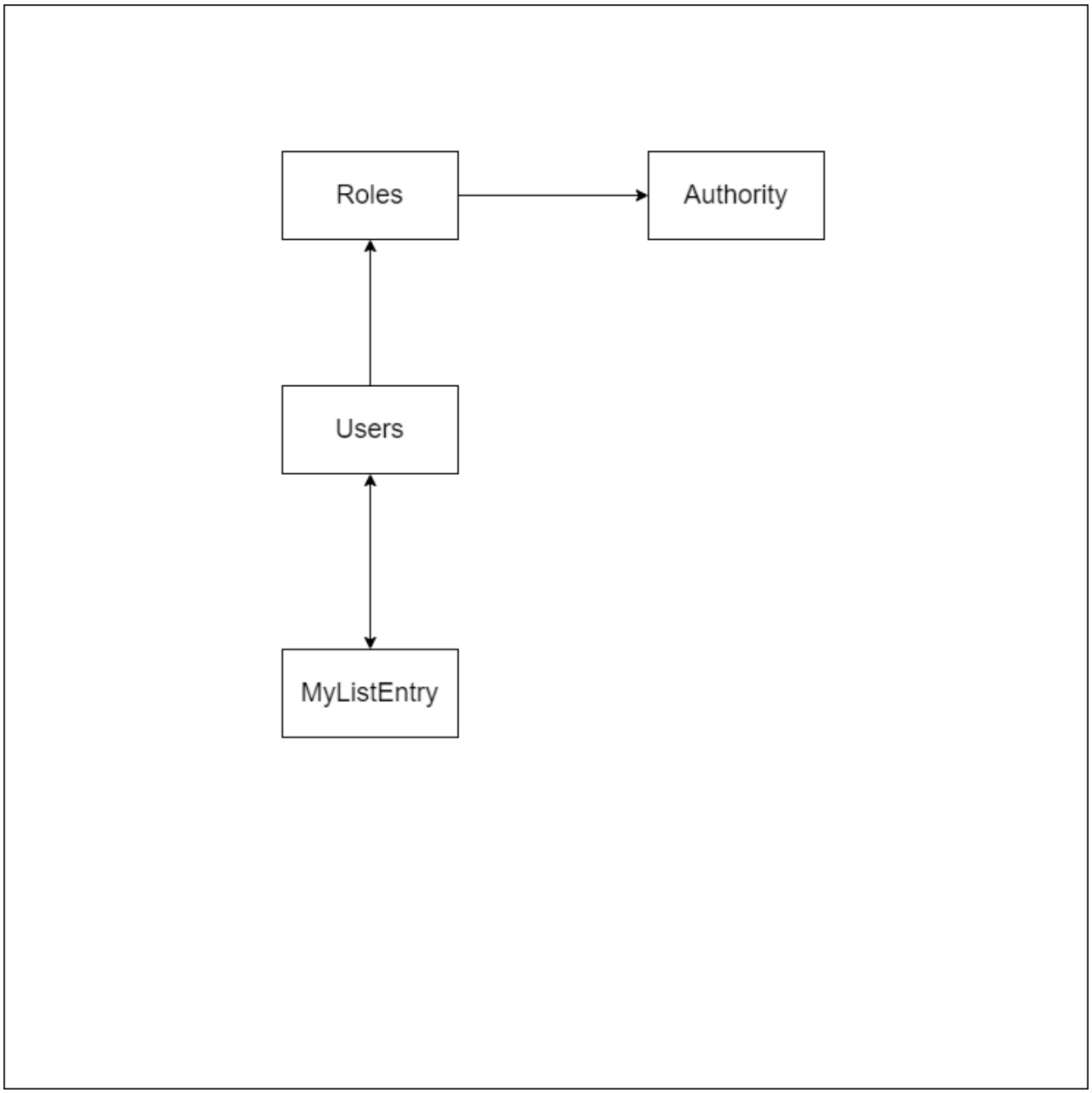
Alternative Flow:	Delete all entries: User does not exist

ID:	1.1
Short description:	The backend let's the logged in user know that the user they were looking for does not exist.
Preconditions:	The user entered the username of a non-existing user
Actor(Primary):	User
Actor(Secondary):	None
Alternative Flow: 1. The Alternative Flow starts after 1.3 in the main process. 2. The current user is informed that the user they were looking for does not exist.	
Postconditions:	None

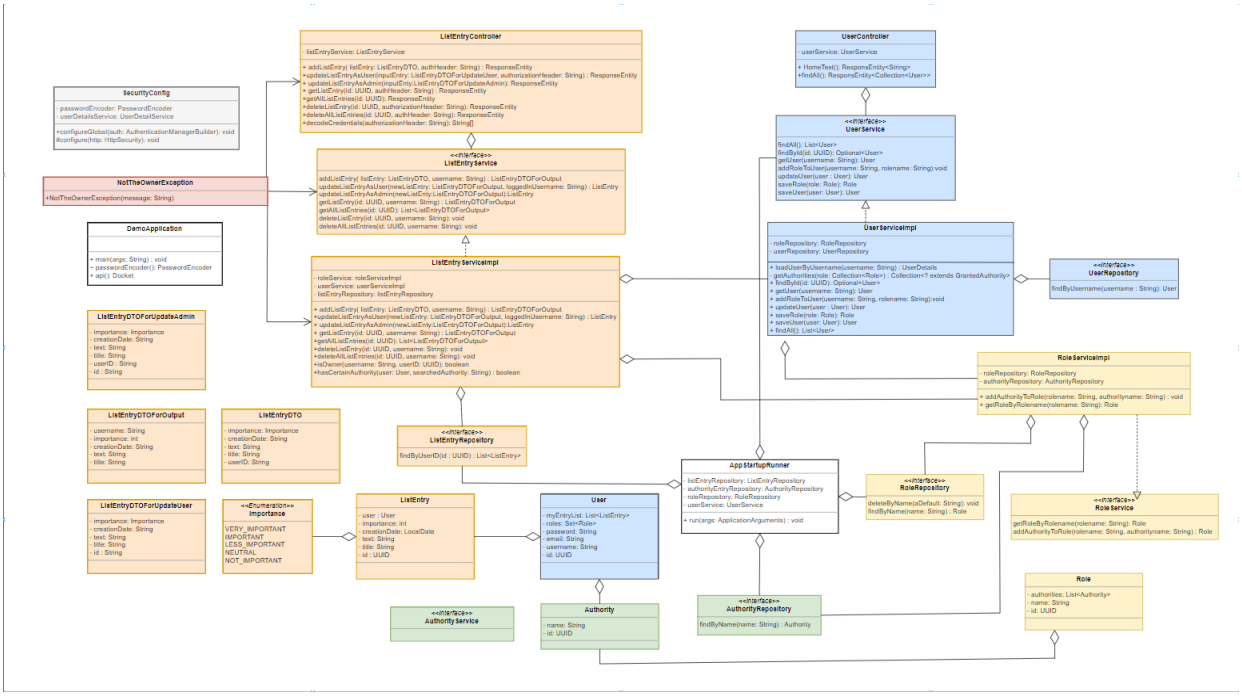
Unsuccessful Use Case

Use Case:	get a specific listEntry
Use Case ID:	2
Short Description:	The user tries to get a specific listEntry from the database.
Preconditions:	An existing user is logged in.
Actor (Primary):	User
Actor(Secondary):	None
Main process: 1. The Use Case starts when the User tries to access the Get-Endpoint via "http://localhost:8080/list/get/ListEntryID". 2. Calls method in the Service layer. 3. Checks if the listEntry exists. 4. Throws InstanceNotFoundException. 5. Returns error message.	
Postcondition:	None
Alternative Flows: None	

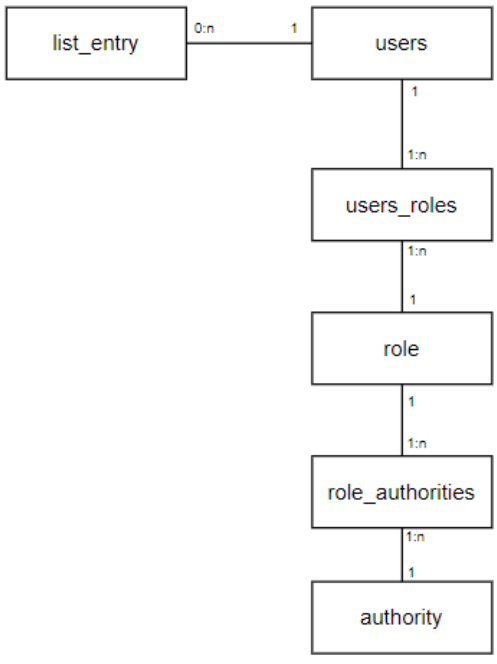
Domain Model



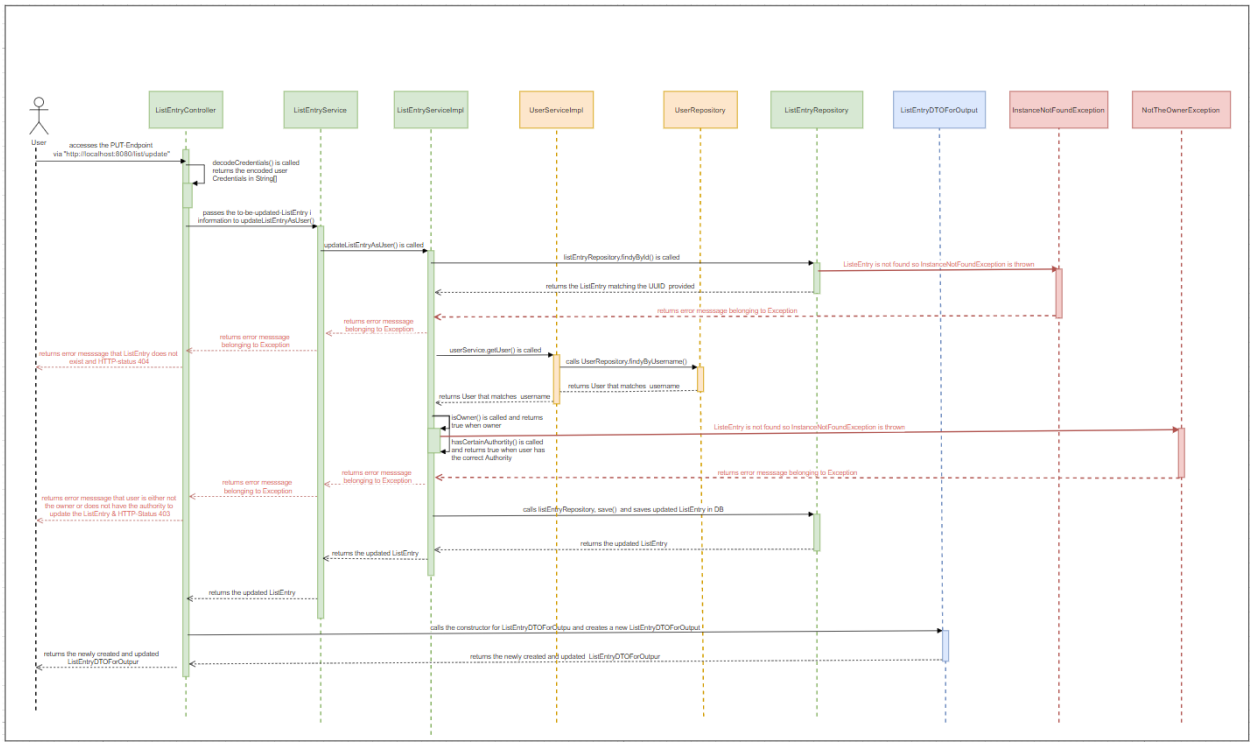
Class Diagram



ERM



Sequence Diagram



Endpoints

I. AddListEntry()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for creating a new ListEntry looks as follows:

```
@PostMapping("add")
public ResponseEntity addListEntry(@RequestBody ListEntryDTO listEntry, @RequestHeader("Authorization") String authHeader) {
    ListEntryDTOForOutput returnedListEntry = null;
    try {
        returnedListEntry = listEntryService.addListEntry(listEntry, decodeCredentials(authHeader)[0]);
    } catch (InstanceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    }
    return ResponseEntity.ok().body(returnedListEntry);
}
```

With the addition “add” to the pathing when using the post-annotation a new ListEntry will be created by calling the addListEntry method in the ListEntryServiceImpl.

1.1.1 Parameters

The new ListEntry is written in the JSON-format and handed over to the method by adding it to the RequestBody in Postman. To make sure that the current user has the needed authorities and/or is the owner of the listEntry the RequestHeader is used to get the encoded user information under the “Authorization” tag. This ListEntryDTO and String are parameters of this method.

1.1.2 Method Body

When calling the matching method the authHeader(from the RequestHeader) has to be decoded so that the username can be determined. To do so the decodeCredentials()-method is used to decode and split the String into the username and password.

1.1.3 Error-Handling

This method tries to call the matching method in the ListEntryService-Layer. If the userID that was given in the RequestBody does not belong to an existing user the error message belonging to the InstanceNoFoundException is displayed.

1.1.4 Return

This method either returns the newly created ListEntry in the ListEntryDTOForOutput form or the error message from the error handling.

1.2 ListEntryServiceImpl

In the service layer the addListEntry method is implemented as follows:

```
public ListEntryDTOForOutput addListEntry(ListEntryDTO listEntry, String username) throws InstanceNotFoundException {
    Optional<User> optionalUser;
    ListEntry newListEntry;
    if (userService.getUser(username).getRoles().contains(roleService.getRoleByRolename("ADMIN"))) {
        if ((optionalUser = userService.findById(UUID.fromString(listEntry.getUserID()))).isEmpty()) {
            throw new InstanceNotFoundException("User does not exist");
        }
        User user = optionalUser.get();
        newListEntry = listEntryRepository.save(new ListEntry( id: null, listEntry.getTitle(), listEntry.getText(), LocalDate.parse(listEntry.getCreationDate()),
                listEntry.getImportance().getNumVal(), user));
        return new ListEntryDTOForOutput(newListEntry);
    }
    newListEntry = listEntryRepository.save(new ListEntry( id: null, listEntry.getTitle(), listEntry.getText(), LocalDate.parse(listEntry.getCreationDate()),
            listEntry.getImportance().getNumVal(), userService.getUser(username)));
    return new ListEntryDTOForOutput(newListEntry);
}
```

1.2.1 Parameters

The new ListEntryDTO and the username of the currently logged in user is handed over by the endpoint in the web layer.

1.2.2 Method Body

Before the method in the ListEntryRepository is called the role of the user is checked. If the user is an admin the save method from the JPA-Repository is called to add the new ListEntry to the database. The username that was given in the Request body is used to get the respective user. If the current user is a normal user, the method will save the new ListEntry by setting the current user as the owner of the ListEntry.

1.2.3 Error-Handling

If the user that was provided does not exist an InstanceNotFoundException is thrown.

1.2.4 Return

This method either returns the newly created ListEntry in the ListEntryDTOForOutput form by turning the returned ListEntry into the ListEntryDTOForOutput.

II. GetListEntry()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for getting a specific ListEntry looks as follows:

```
@GetMapping("get/{listEntryID}")
@PreAuthorize("hasAuthority('READ_LIST_ENTRY')")
public ResponseEntity getListEntry(@PathVariable("listEntryID") UUID id) {

    Optional<ListEntryDTOForOutput> returnedListEntry = null;
    try {
        returnedListEntry = Optional.ofNullable(listEntryService.getListEntry(id));
    } catch (InstanceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    }
    return ResponseEntity.ok().body(returnedListEntry);
}
```

With the addition “get/{listEntryID}” to the pathing when using the get-annotation the path variable is used to specify which ListEntry is being searched. With @PreAuthorize we check if the current user has the authority to read ListEntries before the method is executed.

1.1.1 Parameters

As mentioned above the id of the wanted ListEntry is given in the URL and extracted by using the @PathVariable annotation.

1.1.2 Method Body

When calling the matching method from the service layer use Optional.ofNullable, because there is a chance that the ListEntry we were looking for is null.

1.1.3 Error-Handling

If the ListEntry does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed.

1.1.4 Return

This method either returns the ListEntry that belongs to the given ID in the ListEntryDTOForOutput form or the error message from the error handling.

1.2 ListEntryServiceImpl

In the service layer the getListEntry method is implemented as follows:

```
@Override
public ListEntryDTOForOutput getListEntry(UUID id) throws InstanceNotFoundException {
    Optional<ListEntry> optionalListEntry;

    if ((optionalListEntry = listEntryRepository.findById(id)).isEmpty()) {
        throw new InstanceNotFoundException("Element does not exist");
    }
    ListEntry listEntry = optionalListEntry.get();

    return new ListEntryDTOForOutput(listEntry);
}
```

1.2.1 Parameters

The UUID of ListEntry we are looking for is handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to find the ListEntry by the given id by using the findById method from the ListEntryRepository. We check if this method returns a ListEntry or not.

1.2.3 Error-Handling

If the return of the findyById method is empty, an InstanceNotFoundException is thrown because the ListEntry does not exist.

1.2.4 Return

This method should return the ListEntry adjusted to the ListEntryDTOForOutput form.

III. GetAllListEntries()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for getting all the ListEntries of a specific user looks like this:

```
@GetMapping("{userID}")
@PreAuthorize("hasAuthority('READ_LIST_ENTRY')")
public ResponseEntity getAllListEntries(@PathVariable("userID") UUID id) {
    List<ListEntryDTOForOutput> returnedList = null;
    try {
        returnedList = listEntryService.getAllListEntries(id);
    } catch (InstanceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    }
    return ResponseEntity.ok().body(returnedList);
}
```

With the addition “{userID}” to the pathing when using the get-annotation the path variable is used to specify which user’s list we are looking for. With @PreAuthorize we check if the current user has the authority to read ListEntries before the method is executed.

1.1.1 Parameters

As mentioned above the id of the wanted ListEntry is given in the URL and extracted by using the @PathVariable annotation.

1.1.2 Method Body

When calling the matching method from the service layer use Optional.ofNullable, because there is a chance that the ListEntry we were looking for is null.

1.1.3 Error-Handling

If the ListEntry does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed.

1.1.4 Return

This method either returns the ListEntry that belongs to the given ID in the ListEntryDTOForOutput form or the error message from the error handling.

1.2 ListEntryServiceImpl

In the service layer the getAllListEntries method is implemented as follows:

```
@Override
public List<ListEntryDTOForOutput> getAllListEntries(UUID id) throws InstanceNotFoundException {
    List<ListEntry> listEntries;
    if (!(listEntries = listEntryRepository.findByUserID(id)).isEmpty()) {
        List<ListEntryDTOForOutput> listEntryDTOForOutputs = new ArrayList<>();
        for (int i = 0; i < listEntries.size(); i++) {
            ListEntry listEntry = listEntries.get(i);
            ListEntryDTOForOutput listEntryDTOForOutput = new ListEntryDTOForOutput(listEntry);
            listEntryDTOForOutputs.add(listEntryDTOForOutput);
        }
        return listEntryDTOForOutputs;
    } else {
        throw new InstanceNotFoundException("User does not exist");
    }
}
```

1.2.1 Parameters

The UUID of the user who's list we are looking for is handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to find all the ListEntries that belong to the user by using the findByUserID method from the ListEntryRepository. All the returned ListEntries are saved in a List, through which we iterate through with a for-loop that transforms the ListEntries into ListEntryDTOForOutput's and adds them to another List.

1.2.3 Error-Handling

If the return of the findyByUserID method is empty, an InstanceNotFoundException is thrown because the user does not exist.

1.2.4 Return

This method should return a List of ListEntryDTOForOutput's of all the ListEntries that belong to the given user.

1.3 ListEntryRepository

In the ListEntryRepository we have a custom query that looks for all the ListEntries that have the user_id that we are looking for and sorts them by the importance starting with the highest.

```
@Query(value = "select * from list_entry  where list_entry.user_id = :id order by list_entry.importance desc", nativeQuery = true)
List<ListEntry> findByUserID(@Param("id") UUID id);
```

IV. DeleteAllListEntries()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for deleting all the ListEntries of a specific user looks as follows:

```
@DeleteMapping("{id}")
public ResponseEntity deleteAllListEntries(@PathVariable UUID id, @RequestHeader("Authorization") String authHeader) {
    try {
        listEntryService.deleteAllListEntries(id, decodeCredentials(authHeader)[0]);
        return ResponseEntity.ok().body("All list entries were deleted");
    } catch (InstanceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    } catch (NotTheOwnerException e) {
        return ResponseEntity.status(403).body(e.getMessage());
    }
}
```

With the addition "{id}" to the pathing when using the delete-annotation the path variable is used to specify which user's ListEntries need to be deleted.

1.1.1 Parameters

The id of the wanted user is given in the URL and extracted by using the @PathVariable annotation. To check the current user we need the request header as well to get the username of the user.

1.1.2 Method Body

When calling the matching method from the service we give it the user id and the decoded and split in half authHeader that equals the username.

1.1.3 Error-Handling

If the user does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed. Is the user not the owner of the list of ListEntries and does not have the authority to delete ListEntries, the NotTheOwnerExcept will be caught.

1.1.4 Return

This method either returns a String that confirms that all the ListEntries were deleted or the error messages with responding HTTP-statuses that belong to the error handling.

1.2 ListEntryServiceImpl

In the service layer the deleteAllListEntries method is implemented as follows:

```
@Override
public void deleteAllListEntries(UUID id, String username) throws InstanceNotFoundException, NotTheOwnerException {
    List<ListEntry> listEntries = listEntryRepository.findByUserID(id);
    if (listEntries.isEmpty()) {
        throw new InstanceNotFoundException("User does not exist");
    }
    if (!userService.getUser(username).getRoles().contains(roleService.getRoleByRolename("ADMIN"))) {
        if (!isOwner(username, listEntries.get(0).getUser().getId()) && !hasCertainAuthority(userService.getUser(username), searchedAuthority: "DELETE_LIST_ENTRY")) {
            throw new NotTheOwnerException("You do not own this list of entries and do not have the authority to delete those entries");
        }
    }
    for (int i = 0; i < listEntries.size(); i++) {
        listEntryRepository.delete(listEntries.get(i));
    }
}
```

1.2.1 Parameters

The UUID of the user and the username of the logged-in user are handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to find all the ListEntries that belong to the user by using the findByUserID method and save them in a list.We check if the current user is the owner of the listEntries and/or has the authority to delete those entries. If everything is working correctly we iterate through the list and delete an element with each loop.

1.2.3 Error-Handling

If the return of the findyByUserID method is empty, an InstanceNotFoundException is thrown because the user does not exist. The NotTheOwnerException is thrown if the user is not the owner, is not an admin and does not have the correct authorities to delete a ListEntry.

V. DeleteListEntry()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for deleting a single List Entry of a specific user looks as follows:

```
@DeleteMapping("delete/{id}")
public ResponseEntity deleteListEntry(@PathVariable UUID id,
                                      @RequestHeader("Authorization") String authorizationHeader) {
    log.trace("Accessed the deleteListEntry Endpoint");
    try {
        listEntryService.deleteListEntry(id, decodeCredentials(authorizationHeader)[0]);
        return ResponseEntity.ok("deleted");
    } catch (InstanceNotFoundException e) {
        log.error("ListEntry was not found");
        return ResponseEntity.status(404).body(e.getMessage());
    } catch (NotTheOwnerException e) {
        log.error("User was not the owner or did not have the authority to execute transaction");
        return ResponseEntity.status(403).body(e.getMessage());
    } catch (Exception e) {
        log.trace("Unexpected error: " + e.getMessage());
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

With the addition “{id}” to the pathing when using the delete-annotation the path variable is used to specify which user’s ListEntries need to be deleted.

1.1.1 Parameters

The id of the wanted list entry is given in the URL and extracted by using the @PathVariable annotation. To check the current user we need the request header as well to get the username of the user.

1.1.2 Method Body

When calling the matching method from the service we give it the list entry id and the decoded and split in half authHeader that equals the username.

1.1.3 Error-Handling

If the list entry does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed. If the user is not the

owner of the ListEntry and does not have the authority to delete ListEntries, the NotTheOwnerExcept will be caught.

1.1.4 Return

This method either returns a String that confirms that the ListEntry was deleted or the error messages with responding HTTP-statuses that belong to the error handling.

1.2 ListEntryServiceImpl

In the service layer the deleteAllListEntries method is implemented as follows:

```
@Override
public void deleteListEntry(UUID id, String username) throws InstanceNotFoundException, NotTheOwnerException {
    Optional<ListEntry> optionalListEntry = listEntryRepository.findById(id);
    if (optionalListEntry.isEmpty())
        throw new InstanceNotFoundException("List Entry doesn't exist");
    if (!isOwner(username, optionalListEntry.get().getUser().getId()) && !hasCertainAuthority(userService.getUser(username),
            searchedAuthority: "DELETE_LIST_ENTRY"))
        throw new NotTheOwnerException("You're not the owner of this entry and you do not have the authority to delete it");
    listEntryRepository.delete(optionalListEntry.get());

}
```

1.2.1 Parameters

The UUID of the list entry and the username of the logged-in user are handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to find all the ListEntries that belong to the user by using the findByUserID method and save them in a list.We check if the current user is the owner of the listEntries and/or has the authority to delete that entry. If everything is working correctly we iterate through the list and delete an element with each loop.

1.2.3 Error-Handling

If the return of the findyByUserID method is empty, an InstanceNotFoundException is thrown because the user does not exist. The NotTheOwnerException is thrown if the user is not the owner, is not an admin and does not have the correct authorities to delete a ListEntry.

VI. updateListEntryAsUser()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for updating a single List Entry looks as follows:

```
@PutMapping("update")
public ResponseEntity updateListEntryAsUser(@RequestBody ListEntryDTOForUpdateUser inputEntry,
                                            @RequestHeader("Authorization") String authorizationHeader) {
    log.trace("Accessed the updateListEntryAsUser Endpoint");
    try {
        return ResponseEntity.ok(new ListEntryDTOForOutput(listEntryService.updateListEntryAsUser(inputEntry,
                decodeCredentials(authorizationHeader)[0])));
    } catch (InstanceNotFoundException e) {
        log.error("ListEntry was not found");
        return ResponseEntity.status(404).body(e.getMessage());
    } catch (NotTheOwnerException e) {
        log.error("User was not the owner or did not have the authority to execute transaction");
        return ResponseEntity.status(403).body(e.getMessage());
    } catch (Exception e) {
        log.trace("Unexpected error: " + e.getMessage());
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

The endpoint requires a JSON Object in the format of ListEntryDTOForUpdateUser.

1.1.1 Parameters

The JSON object required is given in the request body in the same format as the constructor of ListEntryDTOForUpdateUser. To check the current user we need the request header as well to get the username of the user.

1.1.2 Method Body

When calling the matching method from the service we give it the DTO and the decoded and split in half authHeader that equals the username.

1.1.3 Error-Handling

If the list entry does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed. If the user is not the owner of the ListEntry and does not have the authority to delete ListEntries, the NotTheOwnerExcept will be caught. Any other error is assumed to be a Bad Request error.

1.1.4 Return

This method either returns the update entry that confirms that the ListEntry was updated or the error messages with responding HTTP-statuses that belong to the error handling.

1.2 ListEntryServiceImpl

In the service layer the updateListEntryAsUser method is implemented as follows:

```
@Override
public ListEntry updateListEntryAsUser(ListEntryDTOForUpdateUser newListEntry,
                                       String loggedInUsername) throws InstanceNotFoundException, NotTheOwnerException {
    Optional<ListEntry> oldListEntryOptional;
    if ((oldListEntryOptional = listEntryRepository.findById(UUID.fromString(newListEntry.getId()))).isPresent()) {
        ListEntry oldListEntry = oldListEntryOptional.get();
        oldListEntry.setTitle(newListEntry.getTitle());
        oldListEntry.setText(newListEntry.getText());
        oldListEntry.setCreationDate(LocalDate.parse(newListEntry.getCreationDate()));
        oldListEntry.setImportance(newListEntry.getImportance().getNumVal());
        if (!isOwner(loggedInUsername, oldListEntry.getUser().getId()) &&
                !hasCertainAuthority(userService.getUser(loggedInUsername), searchedAuthority: "UPDATE_LIST_ENTRY"))
            throw new NotTheOwnerException("You're not the owner of this entry and you do not have the authority to edit it");
        return listEntryRepository.save(oldListEntry);
    } else {
        throw new InstanceNotFoundException("List Entry doesn't exist");
    }
}
```

1.2.1 Parameters

The DTO of the list entry and the username of the logged-in user are handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to edit the entry that belongs to the transferred id in the DTO using the findById method.We check if the current user is the owner of the listEntries and/or has the authority to edit that entry. If everything is working correctly we update the entry.

1.2.3 Error-Handling

If the return of the findyById method is empty, an InstanceNotFoundException is thrown because the list entry does not exist. The NotTheOwnerException is thrown if the user is not the owner, is not an admin and does not have the correct authorities to delete a ListEntry.

VII. updateListEntryAsAdmin()

1.1 ListEntryController

In the ListEntryController the Endpoint responsible for updating a single List Entry and its owner looks as follows:

```
@PutMapping("admin/update")
@PreAuthorize("hasAuthority('UPDATE_LIST_ENTRY')")
public ResponseEntity updateListEntryAsAdmin(@RequestBody ListEntryDTOForUpdateAdmin inputEntry) {
    log.trace("Accessed the updateListEntryAsUser");
    try {
        return ResponseEntity.ok(new ListEntryDTOForOutput(listEntryService.updateListEntryAsAdmin(inputEntry)));
    } catch (InstanceNotFoundException e) {
        log.error("ListEntry was not found");
        return ResponseEntity.status(404).body(e.getMessage());
    } catch (Exception e) {
        log.trace("Unexpected error: " + e.getMessage());
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }

}
```

The endpoint requires a JSON Object in the format of ListEntryDTOForUpdateAdmin.

1.1.1 Parameters

The JSON object required is given in the request body in the same format as the constructor of ListEntryDTOForUpdateAdmin. To check the current user we need the request header as well to get the username of the user.

1.1.2 Method Body

When calling the matching method from the service we give it the DTO and the decoded and split in half authHeader that equals the username.

1.1.3 Error-Handling

If the list entry does not exist the InstanceNotFoundException will be caught and the HTTP-Status 404 as well as the error message will be displayed. Any other error is assumed to be a Bad Request error.

1.1.4 Return

This method either returns the updated entry that confirms that the ListEntry was updated or the error messages with responding HTTP-statuses that belong to the error handling.

1.2 ListEntryServiceImpl

In the service layer the updateListEntryAsAdmin method is implemented as follows:

```
@Override
@Transactional(isolation = Isolation.READ_COMMITTED)
public ListEntry updateListEntryAsAdmin(ListEntryDTOForUpdateAdmin newListEntry) throws InstanceNotFoundException {
    Optional<ListEntry> oldListEntryOptional;
    User newUser;
    if ((newUser = userService.getUser(newListEntry.getUsername())) == null)
        throw new InstanceNotFoundException("User to assign entry to doesn't exist");
    if ((oldListEntryOptional = listEntryRepository.findById(UUID.fromString(newListEntry.getId()))).isPresent()) {
        ListEntry oldListEntry = oldListEntryOptional.get();
        oldListEntry.setTitle(newListEntry.getTitle());
        oldListEntry.setText(newListEntry.getText());
        oldListEntry.setCreationDate(LocalDate.parse(newListEntry.getCreationDate()));
        oldListEntry.setImportance(newListEntry.getImportance().getNumVal());
        oldListEntry.setUser(newUser);
        return listEntryRepository.save(oldListEntry);
    } else {
        throw new InstanceNotFoundException("List Entry doesn't exist");
    }
}
```

1.2.1 Parameters

The DTO of the list entry and the username of the logged-in user are handed over by the endpoint in the web layer.

1.2.2 Method Body

In the method body we try to edit the entry that belongs to the transferred id in the DTO using the findById method.We check that the new owner exists. If everything is working correctly we update the entry.

1.2.3 Error-Handling

If the return of the findyById method is empty, an InstanceNotFoundException is thrown because the list entry does not exist.