LI Databases & Web Programming

Week 9 - lab sheet - SpringBoot Constraints Masterclass

Search

Search this course

Week 9 - lab sheet - SpringBoot Constraints Masterclass

In this masterclass: We will modify the generated JHipster code and become familiar with the template. We will customise the backend to implement API versioning, custom queries, database management, and security constraints.

Prerequisites:

A terminal with Java 21+, Maven, and JHipster installed.

An IDE e.g. IntelliJ (or VS Code)

Part 1: Generate the Base Application

To ensure everyone starts from the same place, we will generate a fresh "Task Manager" application.

Create a file named taskapp.jdl with the following content:

```
entity Task {

title String required,

description TextBlob,

dueDate Instant,

isComplete Boolean,

isPublic Boolean

}

// Relationship: A Task belongs to a User

relationship ManyToOne {

Task{user(login)} to User with builtInEntity

}
```

hint: to create a jdl file in the terminal, use the nano editor:

nano taskapp.jdl //opens or uses the file named taskapp.jdl in the current folder

<paste or type the code>ctrl + O //to Output the file to disk

ctrl + X //to eXit from nano

Generate the application:

jhipster

and using the following settings:

app name: taskApp

application type: monolith

base java package name: com.bham.taskapp

authentication: jwt

productin database type: postgresql

development database type: h2 with disk persistence

frontend client framework: angular

Generate the REST API boilerplate JDL:

jhipster jdl taskapp.jdl

Run the project, then open in a web browser:

./mvnw

you can stop the application using

Ctrl+C

Follow-up 1: Try to use the generated application's CRUD functionality to Create, Update, Delete using the front end UI, login as a different user - which tasks can you read and edit for both users?

Follow-up 2: Extract the admin user authentication token using the authentication endpoint. Use the token with curl to read all tasks (GET) and to create a task (POST). The incompete curl command is:

curl -X POST http://localhost:8080/api/authenticate \

-H "Content-Type: application/json" \

-d '{"username": }'

Follow-up 3: Do the same for the non-admin "user" user - first extract the user authentication token. Use the token with curl to read all tasks (GET) and to create a task (POST).

Part 2: API Versioning

JHipster generates a REST endpoints as /api/tasks.

Let's simulate working for a professional development organisation, where we version our API and follow the REST conventions from the lecture last week.

Step 1: Modify API end point in code

File: src/main/java/com/bham/taskapp/web/rest/TaskResource.java

Find the class declaration. You will see a @RequestMapping annotation. Change it to include a version number.

Change:

@RestController

@RequestMapping("/api/tasks")

public class TaskResource {

To:

@RestController

@RequestMapping("/api/v1/tasks") // Versioning added

public class TaskResource {

Step 2: Verify the update

Restart your backend (stop ./mvnw and run it again).

Try to access the old endpoint:

curl -X 'GET' \

'http://localhost:8080/api/tasks' \

-H 'accept: /' \

-H 'Authorization: Bearer YOUR_TOKEN_HERE'

-> 404 Not Found

(add the API token to the command, otherwise you will get HTTP/1.1 401 Unauthorized)

3. Try the new endpoint:

curl -X 'GET' \

'http://localhost:8080/api/v1/tasks' \

-H 'accept: /' \

-H 'Authorization: Bearer YOUR_TOKEN_HERE'

-> 200 OK

Follow up 1: Open the app in the web browser. Why does this change break the frontend? Find and update task.service.ts. Identify the line that needs to be changed to match the updated API. n.b. for this lab, we are focusing on the Backend.

Follow up 2: What would we need to do to version the API when we have more than one Entity in our JDL?

Part 3: Implementing Constraints for Ownership Security

Currently, any logged-in user can see everyone's tasks. This is a security flaw. We want to enforce a rule or constraint: users can only manage their own tasks.

Step 1: Modify the Create / HTTP POST request Logic

When a task is saved, we shouldn't trust the frontend to tell us who the user is. We should grab the currently logged-in user from the security context.

File: src/main/java/com/bham/taskapp/web/rest/TaskResource.java

Find the createTask method.

Add the imports:

import com.bham.taskapp.service.UserService;

import com.bham.taskapp.domain.User;

import java.util.Optional;

Inject the UserService:

Add this field to the class and add it to the constructor (or use @Autowired):

private final UserService userService; // Add this line to fields

// Update Constructor

public TaskResource(TaskRepository taskRepository, UserService userService) {

this.taskRepository = taskRepository;

this.userService = userService;

}

Modify createTask:

Before the task is saved, force the user to be the currently logged-in user.

@PostMapping("")

public ResponseEntity<Task> createTask(@Valid @RequestBody Task task) throws URISyntaxException {

// ... existing checks ...

// NEW CODE START

// Get the current user from the Security Context

Optional<User> currentUser = userService.getUserWithAuthorities();

if (currentUser.isPresent()) {

    task.setUser(currentUser.get()); // Force the owner to be the logged-in user

}

// NEW CODE END

Task result = taskRepository.save(task);

// ... return statement ...

}

Follow up 1: Restart the app using mvnw. Login to the app what changes do you notice for the created Task entites?

Follow up 2: Which field no longer has an effect in the form to create a Task? How might we fix this?

Step 2: Modify the Read / HTTP GET Logic

Now we need to ensure that when we ask for "All Tasks", we only get our tasks.

File: src/main/java/com/bham/taskapp/repository/TaskRepository.java

JHipster generates a method for this automatically, but check and make sure it is there with a JPA Derived Query.

public interface TaskRepository extends JpaRepository<Task, Long> {

// JPA magic: selects where user.login matches the current security principal

@Query("select task from Task task where task.user.login = ?#{authentication.name}")

List<Task> findByUserIsCurrentUser();

}

also add a new method:

// New method for eager loading with user filter

@Query("select task from Task task left join fetch task.user where task.user.login = ?#{authentication.name}")

List<Task> findByUserIsCurrentUserWithEagerRelationships();

File: src/main/java/com/bham/taskapp/web/rest/TaskResource.java

Update the getAllTasks method to use this specific query instead of the generic findAll().

@GetMapping("")

```
public List<Task> getAllTasks() {

log.debug("REST request to get all tasks for current user");

// OLD:

    if (eagerload) {

        return taskRepository.findAllWithEagerRelationships();

    } else {

        return taskRepository.findAll();

    }

// NEW:

    if (eagerload) {

        return taskRepository.findByUserIsCurrentUserWithEagerRelationships();

    } else {

        return taskRepository.findByUserIsCurrentUser();

    }

}
```

Follow up 1: Restart the app. Log in as 'admin', create a task. Log in as 'user', create a task. The 'user' should NOT see the 'admin' task in the list.

Follow up 2: How would we express the above constriant that we have implemented in English? What similar constraints can you think of for the petstore.jdl from last week?

Part 4: Implementing Relations on Endpoints

According to the REST design conventions for a uniform interface, we should be able to get all tasks for a specific user from the user endpoint. e.g. /users/{login}/tasks where {login} is the username of the user who's tasks we should get back.

File: src/main/java/com/bham/taskapp/repository/TaskRepository.java

Add a new query signature. JPA will convert this Method Name into SQL automatically.

```
// Example of a Derived Query Method
```

List<Task> findAllByUserLogin(String login);

File: src/main/java/com/bham/taskapp/web/rest/PublicUserResource.java

Create a completely new endpoint method.

```
// In PublicUserResource.java

@GetMapping("/users/{login}/tasks")

public List<Task> getUserTasks(@PathVariable String login) {

LOG.debug("REST request to get tasks for user: {}", login);

return taskRepository.findAllByUserLogin(login);

}
```

Test the new endpoint.

```
curl -v http://localhost:8080/api/users/admin/tasks
```

n.b. the above will require an API token in the header e.g.

```
curl -X 'GET' \

'http://localhost:8080/api/users/admin/tasks' \

-H 'accept: /' \

-H 'Authorization: Bearer <YOUR-TOKEN>'
```

Follow up 1: Test the command curl -v http://localhost:8080/api/users/user/tasks what do we have to do to make sure this curl command returns results?

Follow up 2: Also change the version of this endpoint so the following command works: curl -v http://localhost:8080/api/v1/users/admin/tasks

Follow up 3: Whats similar relational endpoints would be useful for the petstore JDL from last week?

Part 5: Liquibase & FakerJS

JHipster uses FakerJSLinks to an external site. to pre-populate our database with random data when we start our app. This can helpful when developing our apps, so we do not need to manually insert data but we should know how to swithcit off.

Disable FakerJS

This is in the dev profile only, disable in:

src/main/resources/config/application-dev.yml

on line 48 remove fake:

spring:

liquibase:

contexts: dev

This can be re enabled later on.

To make the change take effect, delete the local/development H2 database in the folder target/h2db. n.b. if we delete the whole target folder, the compilation will take longer next time.

Liquibase to insert specific data

We can now create our own Liquibase changelogs with handcrafted data.

1. Create a new changelog file in: src/main/resources/config/liquibase/changelog/

Name it something like 20251124000000-student-data.xml

2. Add our handcrafted data using Liquibase insert statements:

<?xml version="1.0" encoding="utf-8"?><databaseChangeLog

xmlns="http://www.liquibase.org/xml/ns/dbchangelog"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-latest.xsd">

<changeSet id="20251124000000-1" author="student">

    <comment>Add handcrafted task data</comment>

```
    <insert tableName="task">

        <column name="id" value="101"/>

        <column name="title" value="Complete Liquibase Assignment"/>

        <column name="description" value="Create custom data changelog with meaningful
tasks"/>

        <column name="due_date" value="2025-12-01"/>

        <column name="is_complete" value="false"/>

        <column name="is_public" value="true"/>

        <column name="user_id" valueComputed="(SELECT id FROM jhi_user WHERE login
= 'admin')"/>

    </insert>



    <insert tableName="task">

        <column name="id" value="201"/>

        <column name="title" value="Learn Database Migrations"/>

        <column name="description" value="Understand how Liquibase manages schema
changes"/>

        <column name="due_date" value="2025-12-05"/>

        <column name="is_complete" value="true"/>

        <column name="is_public" value="true"/>

        <column name="user_id" valueComputed="(SELECT id FROM jhi_user WHERE login
= 'user')"/>

    </insert>

</changeSet>

</databaseChangeLog>3 .Include the new changelog in the main changelog master file
(src/main/resources/config/liquibase/master.xml):
```

<include file="config/liquibase/changelog/20251124000000-student-data.xml" relativeToChangelogFile="false"/>To make the change take effect, restart the app.

Follow up 1: Create one more data migration that has another task item in it.

Follow up 2: For the petstore.jdl - suggest two possible uses for liquibase, to pre-populate the database.

Part 6: Public APIs

Currently, all API endpoints require authentication. Let's create some public endpoints that can be accessed without authentication.

Step 1: Create a Public Controller

Create a new controller for public endpoints:

File: src/main/java/com/bham/taskapp/web/rest/PublicTaskResource.java

```
package com.bham.taskapp.web.rest;

import com.bham.taskapp.domain.Task;

import com.bham.taskapp.repository.TaskRepository;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;

import java.util.List;

/**

REST controller for public Task access.


*/

@RestController

@RequestMapping("/api/v1/public/tasks")

public class PublicTaskResource {
```

```
private final Logger log = LoggerFactory.getLogger(PublicTaskResource.class);

private final TaskRepository taskRepository;

public PublicTaskResource(TaskRepository taskRepository) {
    this.taskRepository = taskRepository;
}

/**
 * GET /api/v1/public/tasks : get all public tasks.
 * This endpoint is accessible without authentication.
 *
 * @return the ResponseEntity with status 200 (OK) and the list of tasks in body
 */
@GetMapping("")
public List<Task> getAllPublicTasks() {
    log.debug("REST request to get all public tasks");
    // For now, return all tasks (we'll filter this later)
    return taskRepository.findAll();
}

/**
 * GET /api/v1/public/tasks/{id} : get a specific public task by ID.
 * This endpoint is accessible without authentication.
```

```
 *
 * @param id the id of the task to retrieve
 * @return the ResponseEntity with status 200 (OK) and with body the task, or with status 404 (Not Found)
 */
@GetMapping("/{id}")
public ResponseEntity<Task> getPublicTask(@PathVariable Long id) {
    log.debug("REST request to get public Task : {}", id);
    // For now, return any task (we'll add public/private logic later)
    return taskRepository.findById(id)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}
}
```

Step 2: Configure Security to Allow Public Access

Modify the security configuration to permit public access to these endpoints:

File: src/main/java/com/bham/taskapp/config/SecurityConfiguration.java

Find the filterChain method and the authorizeHttpRequests section and add the public API pattern:

```
.requestMatchers(mvc.pattern("/api/register")).permitAll()
    .requestMatchers(mvc.pattern("/api/activate")).permitAll()
    .requestMatchers(mvc.pattern("/api/account/reset-password/init")).permitAll()
    .requestMatchers(mvc.pattern("/api/account/reset-password/finish")).permitAll()
    // ADD THIS LINE FOR PUBLIC TASKS API
    .requestMatchers(mvc.pattern("/api/public/**")).permitAll()
```

```
    .requestMatchers(mvc.pattern("/api/admin/**")).hasAuthority(AuthoritiesConstants.ADMIN)
    .requestMatchers(mvc.pattern("/api/**")).authenticated()

.requestMatchers(mvc.pattern("/v3/api-docs/**")).hasAuthority(AuthoritiesConstants.ADMIN)
    .requestMatchers(mvc.pattern("/management/health")).permitAll()
```

Place the changed line after the other public API patterns and before the admin and authenticated API rules, as shown above. Explaination:

Requests to /api/public/** are permitted without authentication

The rule comes before the general /api/** rule that requires authentication

It follows the same pattern as the other public endpoints using mvc.pattern()

Important: Make sure the new line is added before the .requestMatchers(mvc.pattern("/api/**")).authenticated() line, as Spring Security processes rules in order and the first match wins.

Step 3: Test the Public Endpoints

Restart the application and test that the public endpoints work without authentication:

bash

Test public endpoint without authentication token

```
curl -v http://localhost:8080/api/public/tasks
```

Test specific task endpoint

```
curl -v http://localhost:8080/api/public/tasks/101
```

Follow-up 1: Why might we want to limit what data is exposed through public endpoints? Modify the getAllPublicTasks() method to only return tasks that are marked as complete and public (isPublic = true).

Hint: Update the TaskRepository with a query method like findByIsPublicTrue()and use this in the above new public endpoint.

Follow-up 2: What changes should we make to ensure the public API is versioned?

Part 7: Automating UserExtra and Linking to User Entities

A common JHipster task is adding extra fields to the built-in User. The best practice is creating a UserExtra entity and linking it 1-to-1. In this part we will see how to we ensure every User automatically has a UserExtra.

This part invloved regenerating from JDL. To complete this part, it is strongly recommended to create a new application template in a new folder and start from scratch in a new folder. Call this app UserExtraApp.

Add UserExtra to the previous taskapp.jdl JDL and save as userextra.jdl :

```
entity Task {

title String required,

description TextBlob,

dueDate Instant,

isComplete Boolean

}

// Relationship: A Task belongs to a User

relationship ManyToOne {

Task{user(login)} to User with builtInEntity

}

entity UserExtra {

bio String

}

relationship OneToOne {

UserExtra{internalUser(login)} to User with builtInEntity

}
```

Generate the application

```
jhipster jdl userextra.jdl
```

Update registration:

We need to hook into the Registration process.

File: src/main/java/com/bham/taskapp/service/UserService.java

Find the registerUser method. We will inject code to automatically create the extra data.

public User registerUser(AdminUserDTO userDTO, String password) {

// ... existing code that saves the User ...

User newUser = new User();

// ... (lines where password/email are set) ...

userRepository.save(newUser);

// --- NEW CODE STARTS HERE ---

// Create the extra profile automatically

UserExtra userExtra = new UserExtra();

userExtra.setUser(newUser); // Link to the new user

userExtra.setBio("Empty user bio"); // Set defaults

userExtraRepository.save(userExtra);

// --- NEW CODE ENDS HERE ---

// ...

return newUser;

}

For this code to work, add the UserExtraRepository as a constructor injected dependencyLinks to an external site. the class UserService, just like we previously did in TaskResource.

//add the new field to the clas UserService

```
private final UserExtraRepository userExtraRepository;

//update the constructor so userExtraRepository gets "injected" into our class

//dependency injection is one the major benefits of Springboot

public UserService(

    UserRepository userRepository,

    PasswordEncoder passwordEncoder,

    AuthorityRepository authorityRepository, UserExtraRepository userExtraRepository,

    CacheManager cacheManager

) {

    this.userRepository = userRepository;

    this.passwordEncoder = passwordEncoder;

    this.authorityRepository = authorityRepository;

    this.userExtraRepository = userExtraRepository;

    this.cacheManager = cacheManager;

}
```

Follow up 1: Why is the above method and field an example of depency injection and how is it different to the alternatives for depenency injection in SpringBoot? What is the anotation-based alternative to the constructor based depency injection approach?

Follow up 2: In this challenging followup task, we will try to make the same changes to the original TaskApp, that has all the other changes we made. Start making a backup of the original, customised TaskApp folder as a .zip file. Take the changes from the new userextra.jdl and copy these into the JDL in the original TaskApp taskapp.jdl. When running jhipster jdl taskapp.jdl we will be asked to overwrite files or keep the versions we modified. Use the "d" (diff) option to see the changes. Take care to not overwrite any files we have modified in the previous parts.

Hint: if you get a Liquibase checksum error like:

```
liquibase.exception.UnexpectedLiquibaseException:
liquibase.exception.CommandExecutionException:
liquibase.exception.ValidationFailedException: Validation Failed:
```

1 changesets check sum

config/liquibase/changelog/20251123200925_added_entity_Task.xml::20251123200925-1::jhipster was: 9:12c493e3d93efaab9d57dd83eb227b99 but is now: 9:5d7da969dd387396dfb2ed5cdc86d4c7

either delete the database or run

./mvnw liquibase:clearCheckSums

Follow up 3: consider the alternatives to the above approach - what alternative steps we might take in order to get the manually made changes back if we regenrated the whole app or regenerated from a modified JDL? Suggest what commands we could exploit in a version control system like git, e.g. git checkout or git cherrypick.

Summary

We have now modified the "Black Box" of the generated code. We have learned:

Controllers: How to change API paths and add new endpoints.

Repositories: How to write SQL without writing SQL for JPA Derived queries.

Security: How to inject the current user into logic to enforce ownership.

Config: How to stop the database from loading fake data and customising the date we load.

Application Users: How to have extra User-centred fiedls and linking Users to their own Entities