Lecture 57

Testing and Documentation

Module 4C

Objective and agenda

Objective

- Explain how SecurityConfig controls access before a controller is reached.
- Predict when a request will return 401 versus 403 in our app.
- Complete Milestone D and verify role based access using Postman.
- Enable Swagger UI and validate that the API contract is visible.

Agenda

- Warm up quiz
- SecurityConfig recap and Milestone D plan
- Swagger and OpenAPI
- Postman baseline and repeatable testing
- Improving performance and reducing redundancy

Warm up

Warm up quiz

Answer 6 questions.

Reply format: 1 / 2 / 3 / 4

Question 1

In our SecurityConfig, what is the main reason we call authorizeHttpRequests(...)?

1) It runs our controllers in a secure mode so they can read usernames and passwords
2) It tells Spring Security which requests should be allowed or blocked before the controller is reached
3) It creates MongoDB indexes for secure fields like email and passwordHash
4) It turns on logging so we can see every request in the console

Reply format: 1 / 2 / 3 / 4

Answer 1

In our SecurityConfig, what is the main reason we call authorizeHttpRequests(...)?

1) It runs our controllers in a secure mode so they can read usernames and passwords
2) It tells Spring Security which requests should be allowed or blocked before the controller is reached
3) It creates MongoDB indexes for secure fields like email and passwordHash
4) It turns on logging so we can see every request in the console

Correct answer: 2

Why: This is where we define authorization rules so Spring Security can decide access before the request reaches any controller.

Reply format: 1 / 2 / 3 / 4

Question 2

What does requestMatchers("/auth/**") mean inside authorizeHttpRequests(...)?

1) It selects requests whose path matches /auth/** so we can apply a rule to those requests
2) It sends requests to the AuthController automatically without using @RequestMapping
3) It checks whether the user exists in MongoDB before the request continues
4) It changes the API URL so the client can call /auth/** without Basic Auth

Reply format: 1 / 2 / 3 / 4

Answer 2

What does requestMatchers("/auth/**") mean inside authorizeHttpRequests(...)?

1) It selects requests whose path matches /auth/** so we can apply a rule to those requests
2) It sends requests to the AuthController automatically without using @RequestMapping
3) It checks whether the user exists in MongoDB before the request continues
4) It changes the API URL so the client can call /auth/** without Basic Auth

Correct answer: 1

Why: A matcher selects a set of requests by pattern so the next method can define what should happen for those requests.

Reply format: 1 / 2 / 3 / 4

Question 3

When we write .requestMatchers("/auth/signup").permitAll(), what does permitAll() mean?

1) The endpoint is reachable without sending any Basic Auth credentials
2) The endpoint can be called only by ADMIN users
3) The endpoint automatically creates a user in MongoDB if they do not exist
4) The endpoint ignores request validation annotations like @NotBlank and @Email

Reply format: 1 / 2 / 3 / 4

Answer 3

When we write .requestMatchers("/auth/signup").permitAll(), what does permitAll() mean?

1) The endpoint is reachable without sending any Basic Auth credentials
2) The endpoint can be called only by ADMIN users
3) The endpoint automatically creates a user in MongoDB if they do not exist
4) The endpoint ignores request validation annotations like @NotBlank and @Email

Correct answer: 1

Why: permitAll means Spring Security will not require authentication for that request pattern.

Reply format: 1 / 2 / 3 / 4

Question 4

In our app, what is the most accurate meaning of .anyRequest().authenticated()?

1) Every request must include valid Basic Auth credentials unless an earlier rule allows it
2) Every request must be an HTTP GET request to be considered safe
3) Every request must include a CSRF token or it will be rejected
4) Every request must come from Swagger UI instead of Postman

Reply format: 1 / 2 / 3 / 4

Answer 4

In our app, what is the most accurate meaning of .anyRequest().authenticated()?

1) Every request must include valid Basic Auth credentials unless an earlier rule allows it
2) Every request must be an HTTP GET request to be considered safe
3) Every request must include a CSRF token or it will be rejected
4) Every request must come from Swagger UI instead of Postman

Correct answer: 1

Why: authenticated means the request must have a valid logged-in user, typically proven through Basic Auth in our current setup.

Reply format: 1 / 2 / 3 / 4

Question 5

Why did we store passwordHash in MongoDB instead of storing the raw password?

1) Because the raw password is too long to store in MongoDB
2) Because a password hash is a safer representation and we can still verify the password during login
3) Because Spring Boot cannot read raw passwords from request bodies
4) Because MongoDB requires passwords to be encrypted before saving

Reply format: 1 / 2 / 3 / 4

Answer 5

Why did we store passwordHash in MongoDB instead of storing the raw password?

1) Because the raw password is too long to store in MongoDB
2) Because a password hash is a safer representation and we can still verify the password during login
3) Because Spring Boot cannot read raw passwords from request bodies
4) Because MongoDB requires passwords to be encrypted before saving

Correct answer: 2

Why: Hashing protects the real password value while still allowing the system to verify login by comparing the input password against the stored hash.

Reply format: 1 / 2 / 3 / 4

Question 6

CSRF stands for Cross-Site Request Forgery. In our current Basic Auth + Postman flow, why do we disable CSRF protection?

1) Because CSRF protection blocks many POST, PUT, and DELETE requests unless a CSRF token is present
2) Because CSRF protection prevents Basic Authentication from working
3) Because CSRF protection is only needed when using MongoDB Atlas
4) Because CSRF protection is required only for GET requests

Reply format: 1 / 2 / 3 / 4

Answer 6

CSRF stands for Cross-Site Request Forgery. In our current Basic Auth + Postman flow, why do we disable CSRF protection?

1) Because CSRF protection blocks many POST, PUT, and DELETE requests unless a CSRF token is present
2) Because CSRF protection prevents Basic Authentication from working
3) Because CSRF protection is only needed when using MongoDB Atlas
4) Because CSRF protection is required only for GET requests

Correct answer: 1

Why: Without disabling CSRF, state-changing requests can return 403 because Postman is not sending the CSRF token Spring expects by default.

Reply format: 1 / 2 / 3 / 4

Moving to implementation

Recall: What is SecurityConfig?

SecurityConfig is a Spring configuration class.

Its job is to define security setup for the entire application when the app starts.

This class does not handle requests like a controller.
Instead, it prepares the security system that will run before any controller is reached.

Recall: SecurityFilterChain

In our SecurityConfig, we create a bean of type SecurityFilterChain.

That bean represents the security pipeline that will run for incoming requests.

A SecurityFilterChain contains two important parts.

- A way to decide to which requests this chain applies
- A list of security filters that will run in order for those requests

We build this bean using HttpSecurity.
HttpSecurity is the builder that collects our settings and rules.

How requests actually flow

When a request reaches our backend, it does not go directly to the controller.

It first passes through the security layer.

A simplified view of the flow looks like this.

- Request enters the server
- Spring Security is triggered through a filter integration
- Spring Security selects the matching security chain
- Filters run in order and may allow or block the request
- Only then does the request reach the controller method

If security blocks the request, the controller code never runs.

Why chaining works: HttpSecurity is a builder

The SecurityConfig code looks like many methods are running one after another.

What is actually happening is that we are configuring a *builder* object.

HttpSecurity is a builder.
Each method call updates the builder with one piece of security configuration.
At the end, http.build() creates the final SecurityFilterChain bean.

This is similar to patterns you already know in Java. Example:

- StringBuilder collects changes and produces a final string when you call toString().

How do we read the authorization rules block?

Inside SecurityConfig, we have a block where we define access rules.

The easiest way to read each rule is in two steps.

- Step 1: select which requests we are talking about
- Step 2: attach the access decision for those requests

requestMatchers(...) is the selection step.
It selects requests based on URL patterns like /auth/** or /api/**.

Then we attach a decision method such as permitAll() or authenticated().

Example:

- requestMatchers("/auth/**") selects authentication endpoints
- permitAll() means those requests are allowed without login

What is authorizeHttpRequests building?

authorizeHttpRequests is where Spring Security collects the authorization rules.

Under the hood, it builds a structure that looks like this.

- Request pattern matcher
- Access decision for requests that match that pattern

So you can think of it as a table of rules.

- If request matches pattern A, apply decision A
- If request matches pattern B, apply decision B
- If request matches nothing above, apply the default rule at the end

Spring Security uses the idea of an AuthorizationManager to represent the access decision logic.
You do not call it directly in your code.
Spring Security uses it internally while handling requests.

Where does 401 and 403 come from?

Spring Security handles two steps before a request reaches the controller.

Step 1: Authentication.

- Spring tries to identify who the user is.
- If Basic Authentication is enabled, it reads the Authorization header and verifies the credentials.

Step 2: Authorization.

- Spring checks the rules we defined and decides whether this user is allowed to access this endpoint.

This leads to two common outcomes.

401 happens when authentication did not happen or failed.

- Missing credentials
- Wrong credentials

403 happens when authentication succeeded but authorization failed.

- The user is logged in
- The user does not have the required role for the endpoint

This is why a STUDENT can log in successfully but still be blocked from an ADMIN-only route.

Rule order matters: first match wins

Spring Security checks the rules in the order we write them.

The first rule that matches the request decides what happens.

This is why specific rules must be written before broad rules.

Example.

- If we want /api/courses/** to be ADMIN-only, that rule must appear before /api/** authenticated.
- Otherwise, the broad /api/** rule may match first and the specific rule may never apply.

Milestone D

Milestone D is where we prove authorization using roles.

We will add an endpoint that can update a user’s roles in the users collection.

Then we will use the same credentials again and observe that access changes based on role.
First the user has role STUDENT and gets blocked
Then the user gets role ADMIN and the same action is allowed

This will help learners connect roles in MongoDB to authorization decisions in SecurityConfig.

Postman verification

We will verify role-based authorization using a simple proof loop.

Step 1.
Create a user using signup.
Start with role STUDENT.

Step 2.
Call a protected course endpoint with STUDENT credentials.
Try POST /api/courses and observe 403.

Step 3.
Update the user’s roles to ADMIN using the new endpoint we build.

Step 4.
Call the same course endpoint again with the same credentials.
Observe success.

OpenAPI and Swagger

What is OpenAPI?

OpenAPI is a standard format for describing a REST API.

It captures the API contract in a structured way so tools can read it.

An OpenAPI description usually includes:

- The list of endpoints and their HTTP methods
- Path parameters and query parameters
- Request body shape for POST and PUT
- Response shapes for success and error cases
- Response codes such as 200, 201, 400, 401, 403, 404
- Security requirements for protected endpoints

When the contract is accurate, someone can understand how to call the API without reading controller code.

What is Swagger UI?

Swagger UI is a web interface that renders the OpenAPI contract as interactive documentation.

It helps with three practical tasks:

- Discovering what endpoints exist in the backend
- Understanding what payload an endpoint expects and what it returns
- Trying a request quickly from the browser and observing the response

Swagger UI is good for discovery and quick trials
Tools like Postman is better for repeatable testing and collections.

Why API documentation matters?

Our project already has multiple resources and multiple endpoints.

- Students
- Courses
- Enrollments
- Authentication and users

As we add security, the same endpoint can behave differently based on credentials and roles.

Documentation reduces confusion by making these details visible:

- What endpoints exist
- What request body is required
- What the response looks like
- What error responses look like
- Whether authentication is required
- Whether an admin role is required

It also supports smoother collaboration in larger teams because the API contract becomes a shared reference.

How to validate Swagger UI output?

When Swagger UI is running, validate the output in this order.

1. Endpoints
Confirm key endpoints appear and the method and path look correct

2. Request schema
Confirm the request model matches the DTO you expect
Confirm required fields are clearly indicated

3. Response schema
Confirm the success response looks correct
Confirm error responses are visible as well

4. Status codes
Confirm expected codes like 200 or 201 are visible
Confirm common failures like 400, 401, 403, 404 are included

5. Security clarity
Confirm it is clear which endpoints are public and which are protected
Confirm protected endpoints indicate that credentials are required

High level integration steps in Spring Boot

We will enable Swagger UI by adding OpenAPI support to our Spring Boot app.

Steps:

- Add the OpenAPI and Swagger UI dependency
- Start the Spring Boot application
- Open Swagger UI in the browser
- Confirm endpoints are discovered and listed

If endpoints do not appear, the most common causes are:

- Dependency or version mismatch
- The application did not restart cleanly
- Controllers are not being picked up by component scanning

We will keep the first integration minimal. Once the UI is visible, we can improve documentation details later.

Swagger UI walkthrough

We will use a short walkthrough to confirm Swagger UI is working correctly.

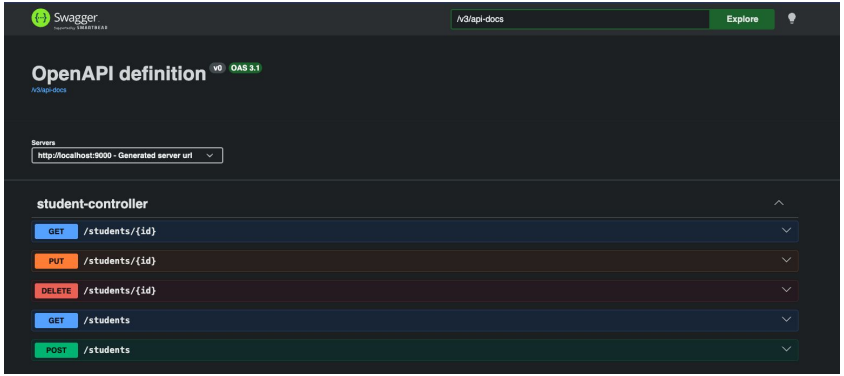
Step 1
Open Swagger UI and confirm the endpoint list loads

Step 2
Expand one endpoint and inspect request and response details

Step 3
Try one request from Swagger UI and observe the response code and body

Step 4
Try one protected endpoint without credentials and observe the response

The purpose is to confirm documentation is generated and usable.



Activity

Goal: Enable Swagger UI for this project and ensure the API contract is visible.

Expected outcome:

- Swagger UI loads in the browser
- The main endpoints are visible in Swagger UI
- At least one endpoint can be tried from Swagger UI and returns a valid response

What to share:

- A screenshot showing Swagger UI with the endpoint list visible
- A screenshot of one endpoint expanded showing request and response details

Postman

What Postman is used for

Postman is a tool used to send HTTP requests to an API and inspect the response.

It is useful during backend development because it makes API testing fast and repeatable.
You can test an endpoint without building a frontend
You can verify request bodies, headers, and status codes
You can re-run the same request multiple times while you are changing code

Postman also helps when debugging security.
It makes it easy to compare what happens with no credentials versus correct credentials.

Three Postman concepts to know

To use Postman well, you only need three concepts.

Request
A single API call with a method, URL, headers, and an optional body.

Collection
A saved group of related requests, organized like a folder.
A collection becomes a reusable test suite for your API.

Environment
A saved set of variables such as base URL, usernames, and passwords.
Environments prevent copy pasting values across requests and reduce mistakes.

Testing secured endpoints in Postman

When security is added, testing should follow a consistent sequence.

Step 1: Confirm a public endpoint works
Call an endpoint that is meant to be open
Verify you get the expected status code and response

Step 2: Confirm a protected endpoint blocks without credentials
Call a protected endpoint with no Authorization header
Verify you get 401, meaning identity was not proven

Step 3: Confirm the same endpoint works with valid credentials
Add Basic Authentication credentials to the request
Verify you get 200 or 201 if the user has permission

Step 4: Confirm role based behavior
Use the same credentials and call an admin protected endpoint
Verify you get 403 when the role is not sufficient
Update the user role, then retry and verify success

This sequence helps you separate authentication issues from authorization issues.

What to save after testing

After testing, save the work so it can be repeated without retyping anything.

Save a collection that includes:
Signup request
Login verification request if applicable
A public endpoint request
A protected endpoint request that requires authentication
A protected endpoint request that requires an admin role

Save an environment that includes:
Base URL
One student account email and password
One admin account email and password

When the collection and environment are saved, you can re-run the same checks quickly after every code change.

Performance and redundancy

Where redundancy shows up in backend code

Redundancy means the same work is repeated in multiple places.

In Spring Boot projects, redundancy commonly appears in these ways:
Controllers repeating the same validation checks instead of relying on DTO validation
Multiple controllers formatting errors differently instead of using one global exception handler
Services repeating the same mapping logic from entities to response DTOs
Repeating the same database lookup logic in several methods instead of centralizing it

Redundancy makes code harder to maintain.
When the behavior changes, you have to update many places and mistakes become more likely.

Patterns to reduce redundancy

There are a few simple patterns that reduce repetition without adding complexity.

Thin controllers
Controllers should focus on request and response handling.
They should delegate business rules to services.

Service owns business rules
Business rules should live in the service layer so logic is not copied across endpoints.

DTO and mapping discipline
Define clear request and response DTOs.
Convert between DTOs and database models in a consistent way.
This avoids repeating conversion logic in every endpoint.

Centralized exception handling
Use a global exception handler to convert exceptions into consistent error responses.
This removes repeated try catch logic from controllers.

Validation through annotations
Use validation annotations on DTOs and @Valid in controllers so required checks are enforced consistently.

Where performance issues show up with MongoDB

Performance issues often come from avoidable patterns, not from MongoDB itself.

Common causes in API projects:
Returning unbounded lists instead of using pagination
Missing indexes on fields that are frequently searched or filtered
Fetching full documents when only a small subset of fields is needed
Repeating database calls in the same request, such as fetching the same record multiple times
Doing work in loops that can be done once outside the loop

Most of these issues can be solved with a few good defaults and habits.

Practical performance hygiene for this project

These are simple rules we can apply immediately in this codebase.

Pagination by default
List endpoints should use paging parameters so responses stay bounded.

Indexes for lookup fields
Add indexes on fields that are frequently used in queries such as email and codes.
This helps reads stay fast as data grows.

Projection for list responses
For list screens, return only the fields needed for the UI.
Do not return entire documents if the UI does not need everything.

Avoid duplicate queries
If you already have a record, reuse it in the same request.
Do not fetch the same record repeatedly in different parts of the same service method.

Basic measurement
When something feels slow, measure before guessing.
Even simple request timing logs can reveal where time is spent.

Optional next tools and practices

Once the basics are solid, there are tools that can help observe and improve performance further.

Metrics and health checks
Spring Boot Actuator can expose health and basic metrics for the application.

Structured monitoring
Metrics libraries help track request latency and error rates over time.

Caching
Caching can help when reads are frequent and data does not change often.
Caching should be introduced only after measuring and identifying a real bottleneck.

Profiling and query review
When performance becomes a priority, profiling and query inspection help locate expensive paths in code and database access.

That is a wrap