Lecture 56

Authentication and Security

Module 4B

Objective and agenda

Objective

We'll 1) add DB-backed users to our Spring Boot app, 2) implement signup and login, and 3) secure key endpoints using Basic authentication and roles.

Agenda

- Warm-up quiz
- Milestone 1: Create users collection and implement signup
- Milestone 2: Implement login with passwordHash validation
- Milestone 3: Add Spring Security and configure Basic authentication using DB-backed users
- Milestone 4: Protect course endpoints using ADMIN and STUDENT roles
- Learner task: Implement role update endpoint and verify access rules

Warm-up

Warm-up quiz

Answer 6 questions.

Reply format: 1 / 2 / 3 / 4

Question 1

Spring Security is enabled in a Spring Boot REST API. A request hits POST /api/courses with no Authorization header. What should happen first, before any controller method runs?

1) Spring runs the controller method and then checks roles at the end

2) Spring Security blocks the request early in the filter chain and returns 401

3) MongoDB rejects the request because the user is missing in the users collection

4) The controller throws a validation error because credentials are missing in the body

Reply format: 1 / 2 / 3 / 4

Answer 1

Spring Security is enabled in a Spring Boot REST API. A request hits POST /api/courses with no Authorization header. What should happen first, before any controller method runs?

1) Spring runs the controller method and then checks roles at the end

2) Spring Security blocks the request early in the filter chain and returns 401

3) MongoDB rejects the request because the user is missing in the users collection

4) The controller throws a validation error because credentials are missing in the body

Correct answer: 2

Why: With Spring Security, the request is checked before controllers. Missing credentials means the request is not authenticated, so the API should return 401.

Reply format: 1 / 2 / 3 / 4

Question 2

A STUDENT user sends valid Basic authentication credentials and calls DELETE /api/courses/{id}.
Your rule is that only ADMIN can delete courses. What should the API return?

1) 200 because the user is authenticated

2) 401 because the endpoint is protected

3) 403 because the user is authenticated but not allowed

4) 404 because the course should be hidden from students

Reply format: 1 / 2 / 3 / 4

Answer 2

A STUDENT user sends valid Basic authentication credentials and calls DELETE /api/courses/{id}.
Your rule is that only ADMIN can delete courses. What should the API return?

1) 200 because the user is authenticated

2) 401 because the endpoint is protected

3) 403 because the user is authenticated but not allowed

4) 404 because the course should be hidden from students

Correct answer: 3

Why: 401 is for missing or invalid credentials. Here the credentials are valid, but the role is wrong, so it must be 403.

Reply format: 1 / 2 / 3 / 4

Question 3

Basic authentication sends credentials in every protected request.
Where do those credentials go?

1) In the request body as JSON fields: email and password

2) In the query string as ?email=...&password=...

3) In the Authorization header using the Basic scheme

4) In a MongoDB document that Spring reads automatically

Reply format: 1 / 2 / 3 / 4

Answer 3

Basic authentication sends credentials in every protected request.
Where do those credentials go?

1) In the request body as JSON fields: email and password

2) In the query string as ?email=...&password=...

3) In the Authorization header using the Basic scheme

4) In a MongoDB document that Spring reads automatically

Correct answer: 3

Why: Basic authentication is carried in the Authorization header. The server reads that header for every protected request.

Reply format: 1 / 2 / 3 / 4

Question 4

You send GET /api/courses from Postman without setting Basic Auth. The endpoint is protected.
Which result best matches what you will observe?

1) 400 because the request body is missing email and password

2) 401 because the request has no valid credentials

3) 403 because the user role is missing

4) 500 because Spring cannot connect to MongoDB

Reply format: 1 / 2 / 3 / 4

Answer 4

You send GET /api/courses from Postman without setting Basic Auth. The endpoint is protected.
Which result best matches what you will observe?

1) 400 because the request body is missing email and password

2) 401 because the request has no valid credentials

3) 403 because the user role is missing

4) 500 because Spring cannot connect to MongoDB

Correct answer: 2

Why: The endpoint requires authentication. Without credentials, Spring Security returns 401 before the controller runs.

Reply format: 1 / 2 / 3 / 4

Question 5

You sign up a user with password Tiger@123. In MongoDB you store a passwordHash. Later the user tries to log in. Which approach is correct?

1) Decrypt the stored hash to get the original password and compare

2) Hash the incoming password and compare the two hash values

3) Store both password and passwordHash so you can debug login issues

4) Compare only the email, because Basic Auth already validates passwords

Reply format: 1 / 2 / 3 / 4

Answer 5

You sign up a user with password Tiger@123. In MongoDB you store a passwordHash. Later the user tries to log in. Which approach is correct?

1) Decrypt the stored hash to get the original password and compare

2) Hash the incoming password and compare the two hash values

3) Store both password and passwordHash so you can debug login issues

4) Compare only the email, because Basic Auth already validates passwords

Correct answer: 2

Why: A hash is not decrypted. We hash the incoming password the same way and compare the results.

Reply format: 1 / 2 / 3 / 4

Question 6

In UserRepository, you add a method like existsByEmailIgnoreCase(email) to check duplicates before saving. You also add a unique index on users.email. Why do we still need the unique index?

1) Because without it, Spring Security will not load users from MongoDB

2) Because the database is the final guard even if two signup requests happen close together

3) Because unique indexes automatically convert emails to lowercase

4) Because it removes the need to check duplicates in the service layer

Reply format: 1 / 2 / 3 / 4

Answer 6

In UserRepository, you add a method like existsByEmailIgnoreCase(email) to check duplicates before saving. You also add a unique index on users.email. Why do we still need the unique index?

1) Because without it, Spring Security will not load users from MongoDB

2) Because the database is the final guard even if two signup requests happen close together

3) Because unique indexes automatically convert emails to lowercase

4) Because it removes the need to check duplicates in the service layer

Correct answer: 2

Why: The service check is helpful, but the database rule is the final guard. It guarantees uniqueness even if the API layer misses it.

Reply format: 1 / 2 / 3 / 4

Implementation

Milestone 1 signup

What we will build

- A users collection that stores login identity and access data.
- A signup endpoint that creates a new user in MongoDB.

How we will build it

- Create a User model and UserRepository.
- Add POST /auth/signup in an AuthController.
- Validate the request using a DTO.
- Prevent duplicate emails using a repository check and a database rule.

What we will verify

- We get 201 and a new document appears in users.
- Signing up with an existing email returns 409.

Milestone 2 login

What we will build

- A login endpoint that validates credentials using MongoDB users.

How we will build it

- Read the user by email using UserRepository.
- Hash the incoming password and compare with passwordHash.
- Return a success response or a clear 401 response.

What we will verify

- Correct email and password returns 200.
- Wrong email or password returns 401.

Spring Security

What changes when we add Spring Security?

- When we add Spring Security, requests no longer go directly to controllers.
- Spring Security checks the request first and decides whether the request is allowed.
- If a request is not allowed, the controller method does not run.

How you will notice this

Endpoints that used to work may start returning 401 until we configure security rules.

Why this happens after adding the dependency?

- When we add spring-boot-starter-security in pom.xml, Spring Boot auto-configures security at startup.
- Auto-configuration means Spring Boot creates security components for us, even if we did not write any security code yet.
- By default, Spring Security protects most endpoints, so requests without authentication get blocked.

What we will do next?

We will write our own security configuration so the rules match our project.

Configuration classes vs controllers and services

Controllers and services

- Controllers and services run when a request is handled.
- A controller maps a URL to a method.
- A service contains business logic called by controllers.

Configuration class

- A configuration class runs when the application starts.
- Its job is to define global setup and global rules.
- It creates and wires beans that the whole application uses.

Why this matters?

- Security rules are global.
- We define the policy once, instead of writing checks in every controller.

Key annotations for Spring Security

@Configuration

This means the class defines beans that Spring should build at startup.

@EnableWebSecurity

This enables Spring Security for web requests in this application.

@Bean

This tells Spring to create an object and store it in the Spring container.

Example we will write

We will create a bean method that returns a SecurityFilterChain.

What is a filter chain?

- A filter is a step that runs before the request reaches the controller.
- A chain means multiple steps run in a fixed order.

What this means in practice

- Requests are checked early.
- If checks fail, Spring returns a response and the controller does not run.

Why this matters

- Security is enforced consistently for every request.
- We do not need to manually check credentials in each controller.

What is a SecurityFilterChain?

- A SecurityFilterChain is the security rule book for our application.
- We define it once, and Spring Security uses it for every incoming request.

What we will write?

- A bean method that returns a SecurityFilterChain.
- Rules for public endpoints and protected endpoints.
- Rules for role-based access in Milestone 4.

The SecurityConfig class shape

- We will create a SecurityConfig class.
- It will use @Configuration and @EnableWebSecurity.
- It will contain a @Bean method that returns a SecurityFilterChain.

Why it looks different

- This is startup configuration.
- This is not request handling code like controllers.

What is HttpSecurity?

- HttpSecurity is a configuration object provided by Spring Security.
- We use it to describe security rules at startup.

Why chained calls are used

- Each call adds one part of the security policy.
- build creates the final SecurityFilterChain.

Important point

- We are not handling one request here.
- We are defining rules for all requests.

Public and protected endpoints for today

Public endpoints

- POST /auth/signup
- POST /auth/login

Protected endpoints

Everything under /api/** will require authentication.

Why we do this first

- It gives us a clear baseline.
- Then we add role-based rules for course endpoints.

What are requestMatchers?

- requestMatchers is how we select which endpoints a rule applies to.
- We can match a single path or a path pattern.

Examples

Match one endpoint

```
/auth/signup
```

Match a group of endpoints

```
/api/**
```

Match course endpoints

```
/api/courses/**
```

Core rule keywords we will use

permitAll

Anyone can access the endpoint without authentication.

authenticated

A valid logged-in user is required.

hasRole

The user must be logged in and must have a specific role such as ADMIN.

anyRequest().authenticated

Any endpoint not listed above should require authentication.

Important note

- Rule order matters.
- The first matching rule is applied.

What 401 and 403 mean

401 Unauthorized

- The request does not have valid credentials.

Examples

- Missing credentials.
- Wrong credentials.

403 Forbidden

- The user is authenticated, but not allowed to do this action.

Example

- STUDENT tries to delete a course restricted to ADMIN.

DB-backed authentication

- Spring Security needs a way to load user credentials and roles.
- We will connect Spring Security to MongoDB using UserRepository.

What we will connect

- Spring Security calls our user loading code.
- Our user loading code reads the user document from users.
- Spring Security validates the credentials using that data.

How roles connect to endpoint rules

- A role is a label that describes what actions a user can perform.
- In our project, we will use roles such as STUDENT and ADMIN.

Rules for today

- Any authenticated user can list courses.
- Only ADMIN can create and delete courses.

What we will verify

- STUDENT gets 403 on course create and delete.
- ADMIN can create and delete successfully.

CSRF and why we will disable it today

- CSRF stands for Cross Site Request Forgery.
- CSRF protection is mainly relevant for browser-based apps that use cookies.
- Our goal is different, since we are building a REST API and testing it through Postman.

Why we disable it today

- With CSRF enabled, POST, PUT, and DELETE requests can get blocked with 403 unless CSRF tokens are included in the request.
- Disabling CSRF allows us to the focus on authentication and authorization.

Milestone 3 verification checklist

- GET /api/courses without credentials returns 401.
- GET /api/courses with valid credentials returns 200.
- After role rules are added, STUDENT gets 403 on course create and delete.
- ADMIN can create and delete courses successfully.

Milestone 4 preview

- After authentication is working, we will add role-based rules for course endpoints.
- This is where we define ADMIN-only endpoints using hasRole.

Endpoints we will protect

- Only ADMIN can create courses.
- Only ADMIN can delete courses.

Your turn

Task

By the end of this task, you will update a user's roles in MongoDB and verify that access rules change immediately.

What you will build

- An endpoint that updates a user's roles in the users collection.

Endpoint

```
PUT /api/users/{email}/roles
```

Request body example

```
roles: ["ADMIN"]
```

Implementation steps

1) Load the user using the email from the path.

- Use UserRepository to find the user by email.
- If the user does not exist, return 404.

2) Validate the request body.

- Roles list must not be empty.
- Roles must contain only values we support today such as STUDENT and ADMIN.
- If validation fails, return 400.

3) Update roles and save.

- Replace the roles field on the user document.
- Save using UserRepository.

4) Return a clean response.

- Return 200 with updated user details.
- Do not return passwordHash in the response.

Testing and submission

How to test in Postman

1) Create a user using signup.

    Start with role STUDENT.

2) Call a protected course endpoint with STUDENT credentials.

    Try POST /api/courses and observe 403.

3) Update the user's roles to ADMIN using your new endpoint.

4) Call the same course endpoint again with the same credentials.

Submission

Share your code in GitHub Discussions for Lecture 56.

That's a wrap