Lecture 44

Authentication and Security

Module 4B

What are we doing today

Objective

- Understand authentication and authorization with clear examples.
- Learn how Spring Security protects REST APIs using a filter chain.
- Introduce Basic authentication and a DB-backed user model.

Agenda

- Warm-up quiz.
- Security foundations.
- Request lifecycle and where security fits.
- Introduction to Spring Security.
- Basic authentication.
- DB-backed users and roles.
- Signup and login flow.
- Authorization and protected routes.
- Activity.

Warm-up

Warm-up quiz

- Answer 6 questions.
- Reply format: 1 / 2 / 3 / 4

Question 1

You created an Atlas cluster and copied the Compass connection string.
When you try to connect from MongoDB Compass, the connection fails from your laptop.

What is the most common first fix in Atlas?

1) Add your current public IP address to the Network Access IP access list

2) Create a new database named studentdb in Atlas

3) Create an index on email before connecting

4) Import a JSON file into the students collection first

Reply format: 1 / 2 / 3 / 4

Answer 1

You created an Atlas cluster and copied the Compass connection string.
When you try to connect from MongoDB Compass, the connection fails from your laptop.

What is the most common first fix in Atlas?

1) Add your current public IP address to the Network Access IP access list

2) Create a new database named studentdb in Atlas

3) Create an index on email before connecting

4) Import a JSON file into the students collection first

Correct answer: 1

Why: Atlas blocks connections from unknown IP addresses by default. Adding your current IP in Network Access is usually the first step to allow Compass to connect.

Reply format: 1 / 2 / 3 / 4

Question 2

In MongoDB Compass, you want to search for students where name contains the text "ki" ignoring case.

Which filter best matches that requirement?

1) { "name": "ki" }

2) { "name": { "$regex": "ki", "$options": "i" } }

3) { "name": { "$contains": "ki" } }

4) { "$regex": { "name": "ki" } }

Reply format: 1 / 2 / 3 / 4

Answer 2

In MongoDB Compass, you want to search for students where name contains the text "ki" ignoring case.

Which filter best matches that requirement?

1) { "name": "ki" }

2) { "name": { "$regex": "ki", "$options": "i" } }

3) { "name": { "$contains": "ki" } }

4) { "$regex": { "name": "ki" } }

Correct answer: 2

Why: The $regex operator matches patterns inside a string and the option i makes the match case-insensitive.

Reply format: 1 / 2 / 3 / 4

Question 3

In MongoDB Compass, you run this query often.
Find active students in Bengaluru and sort by name.

Which index is the best fit to support this query pattern?

1) { "name": 1 }

2) { "city": 1 }

3) { "active": 1, "city": 1, "name": 1 }

4) { "email": 1 }

Reply format: 1 / 2 / 3 / 4

Answer 3

In MongoDB Compass, you run this query often.
Find active students in Bengaluru and sort by name.

Which index is the best fit to support this query pattern?

1) { "name": 1 }

2) { "city": 1 }

3) { "active": 1, "city": 1, "name": 1 }

4) { "email": 1 }

Correct answer: 3

Why: The query filters by active and city and then sorts by name. A compound index that starts with the filter fields and includes the sort field supports this pattern efficiently.

Reply format: 1 / 2 / 3 / 4

Question 4

Context
A food delivery app stores orders. Each order has 1 to 5 items, and each item has name, quantity, and price.
Most screens show the order together with its items.

How should you model order items in MongoDB?

1) Store items inside the Order document as an embedded array

2) Store items in a separate items collection and store itemIds in Order

3) Store items in the courses collection because arrays are not allowed in MongoDB

4) Store items in a separate database because items are large

Reply format: 1 / 2 / 3 / 4

Answer 4

Context
A food delivery app stores orders. Each order has 1 to 5 items, and each item has name, quantity, and price.
Most screens show the order together with its items.

How should you model order items in MongoDB?

1) Store items inside the Order document as an embedded array

2) Store items in a separate items collection and store itemIds in Order

3) Store items in the courses collection because arrays are not allowed in MongoDB

4) Store items in a separate database because items are large

Correct answer: 1

Why: Items are owned by the order and are usually read together with the order. Embedding keeps reads simple and avoids extra lookups.

Reply format: 1 / 2 / 3 / 4

Question 5

You updated your code locally and want those exact changes to appear on GitHub in your repository.

Which sequence is correct?

1) Stage the files (git add), commit the snapshot (git commit), push the commit (git push)

2) Commit the snapshot (git commit), stage the files (git add), push the commit (git push)

3) Push the commit (git push), stage the files (git add), commit the snapshot (git commit)

4) Pull from GitHub (git pull), push to GitHub (git push), then commit locally (git commit)

Reply format: 1 / 2 / 3 / 4

Answer 5

You updated your code locally and want those exact changes to appear on GitHub in your repository.

Which sequence is correct?

1) Stage the files (git add), commit the snapshot (git commit), push the commit (git push)

2) Commit the snapshot (git commit), stage the files (git add), push the commit (git push)

3) Push the commit (git push), stage the files (git add), commit the snapshot (git commit)

4) Pull from GitHub (git pull), push to GitHub (git push), then commit locally (git commit)

Correct answer: 1

Why: You stage changes with git add, save a snapshot with git commit, and then send commits to GitHub with git push.

Reply format: 1 / 2 / 3 / 4

Question 6

You are building a REST API that has endpoints to create and delete courses.
You do not want everyone on the internet to call these endpoints.

What is the first security goal you should solve before talking about roles like admin?

1) Ensure the request body is valid JSON

2) Ensure only known users can call the endpoint at all

3) Ensure the database has indexes for faster reads

4) Ensure the endpoint returns paginated responses

Reply format: 1 / 2 / 3 / 4

Answer 6

You are building a REST API that has endpoints to create and delete courses.
You do not want everyone on the internet to call these endpoints.

What is the first security goal you should solve before talking about roles like admin?

1) Ensure the request body is valid JSON

2) Ensure only known users can call the endpoint at all

3) Ensure the database has indexes for faster reads

4) Ensure the endpoint returns paginated responses

Correct answer: 2

Why: Before deciding what different users can do, you must first ensure the caller is a known user. That is the starting point for protecting any API.

Reply format: 1 / 2 / 3 / 4

Foundations of security for APIs

Why security exists in an API

An API exposes actions over the network, which means the caller is not automatically trusted. If an endpoint is reachable, it can be called by a real user, a script, or an attacker.

Common risks when an API has no security

- Data exposure can happen when private information is returned to the wrong caller.
    - Example: someone fetches student profiles, emails, or enrollment details that were never meant to be public.
- Unauthorized changes can happen when actions are not restricted.
    - Example: someone deletes a course or changes an enrollment status.
- Abuse at scale can happen when requests are automated.
    - Example: someone scrapes data or attempts credential guessing.

Core definition: Authentication

Authentication answers one simple question. It tells the server who the caller is.

How authentication works in plain terms

- The caller presents an identity, such as an email address or a username.
- The caller presents proof, such as a password.
- The server verifies the proof and decides whether the identity is valid.

Example from our system

If a caller (aka the client) says they are an admin and sends a password, the server must verify those credentials before it treats the caller as an admin user.

Core definition: Authorization

Authorization answers a different question. It tells the server what the authenticated caller is allowed to do.

How authorization works

- A request comes in from a caller who is already authenticated.
- The server checks rules that decide which actions that caller is allowed to perform.
- The server either allows the action or blocks it.

Examples from our system

A student can read their own profile and enroll into a course.
A student should not be allowed to delete a course or view all students.
An admin can create and delete courses and can view all students.

Identity, credentials, roles, and permissions

These terms show up throughout security. We will keep the meanings consistent.

Identity

- Identity is how we represent a person or system inside our application.
- Example: email, userId.

Credentials

- Credentials are the proof that the identity belongs to the caller.
- Example: password, API key.

Role

- A role is a label that groups a set of permissions into something easy to manage.
- Example: STUDENT, ADMIN.

Permission

- A permission is a specific allowed action in the system.
- Example: course:create, course:delete, student:read:all.

A concrete mapping using our student system

We will use a few actions from our project so these concepts feel real.

Actions we expose through APIs

- Read a student profile.
- Create a course.
- Delete a course.
- Create an enrollment.
- Update an enrollment status.

A simple rule set we can start with

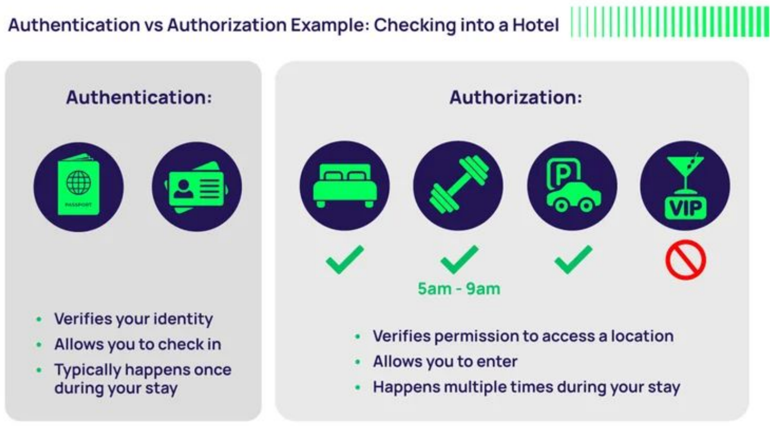
- A student can read their own student profile and create their own enrollments.
- A student cannot delete courses and cannot read all students.
- An admin can create and delete courses and can view all students.

Remembering the difference

Authentication is the step where you prove who you are. Authorization is the step where the system decides what you are allowed to do.

A helpful mental picture

- Authentication is the entry check, where you show an ID and the system recognizes you.
- Authorization is the permission check, where the system decides which actions you can perform after you enter.


Authentication vs Authorization Example: Checking into a Hotel
Authentication:
PASSPORT
Verifies your identity
Allows you to check in
Typically happens once during your stay
Authorization:
VIP
5am - 9am
Verifies permission to access a location
Allows you to enter
Happens multiple times during your stay

REST request lifecycle & where security fits

Recap: A REST request lifecycle

When a client calls an API endpoint, the request moves through a sequence of steps in the backend.

Typical steps in our Spring Boot app

- The client sends an HTTP request to the server.
- Spring matches the URL and HTTP method to a controller method.
- The controller reads inputs such as path variables, query parameters, and request body.
- The controller calls the service layer to run business logic.
- The service calls the repository layer to read or write data in MongoDB.
- The server builds an HTTP response and sends it back to the client.

Why this matters for security

Security must happen before business logic, because we should not execute protected actions for unknown callers.

- Authentication should happen before the request reaches controllers and services.
- Authorization should block requests that are not allowed before any data changes occur.
- A consistent security layer prevents accidental gaps across endpoints.

Where security fits in the same lifecycle

Spring Security adds security checks early in the request flow, before your controller code runs.

What changes when Spring Security is enabled

- The request first passes through a chain of filters.
- The filter chain tries to authenticate the caller using the request.
- If authentication succeeds, Spring stores the authenticated user for the rest of the request.
- Spring then checks whether the authenticated user is allowed to access the endpoint.
- Only after these checks pass does the request reach your controller and service code.

What this prevents

If the request is not authenticated, Spring can stop the request immediately.
If the request is authenticated but not allowed, Spring can stop the request before it reaches your business logic.

Three common outcomes of validating a request

Once security is applied, every protected endpoint usually ends in one of these outcomes.

Not authenticated

The request does not provide valid credentials. The server responds with 401.

Authenticated but not allowed

The request has valid credentials, but the user does not have permission. The server responds with 403.

Authenticated and allowed

The request has valid credentials and the correct permission. The request reaches the controller and the normal business flow runs.

Mapping outcomes to our student system

We will use examples from our endpoints so the status codes feel intuitive.

Example 1

A user calls DELETE /courses/{id} without credentials. This should return 401 because the caller is not authenticated.

Example 2

A student calls DELETE /courses/{id} with valid credentials. This should return 403 because the student is authenticated but not allowed to delete courses.

Example 3

An admin calls DELETE /courses/{id} with valid credentials. This should succeed because the admin is authenticated and allowed.

Introduction to Spring Security

Why Spring Security exists

In a real backend, you do not want every API method to handle security checks on its own.
If you do that, you end up repeating the same work across many endpoints. Recall: this is the same reason GlobalExceptionHandler exists.

What goes wrong without a security layer

- Every endpoint has to check who the caller is before doing anything important.
- Every endpoint has to decide whether the caller is allowed to do that action.
- Some endpoints will forget to check, and those become security holes.
- The code becomes messy because security logic mixes with business logic.

What Spring Security gives you

- One central place to define which endpoints are protected.
- A consistent way to verify the caller for each request.
- A consistent way to allow or block actions based on roles.
- Standard responses when access is denied.

The core idea: a *filter chain* in front of controllers

Spring Security runs requests through a chain of security filters before they reach your controllers.
Every request is checked before your controller method runs.

What a filter can do

- Read the request, especially headers that carry credentials.
- Verify whether the caller is a known user.
- Stop the request early and return an error response.
- Attach the authenticated user to the request so the rest of the app can use it.

Why this is useful

Security checks happen before business logic runs, so protected operations are blocked early.

SecurityFilterChain and what it controls

In Spring Security, we define the rules for our API in a SecurityFilterChain configuration. This configuration is the security policy for the entire application.

What we define in the policy

- Which endpoints are public.
- Which endpoints require a logged-in user.
- Which endpoints require a specific role, such as ADMIN.
- Which authentication method we want to use, such as Basic authentication.

Why this matters

We write the rules once, and Spring Security applies them consistently to all requests.

What happens after a request is authenticated?

When Spring Security verifies the credentials in a request, it marks the request as authenticated. It also stores information about the user for the rest of the request.

What gets stored

- The user identity, such as email or username.
- The roles for that user.
- An Authentication object that represents the logged-in user.

Where it is stored

Spring Security keeps it in the SecurityContext for the current request.

Why this matters

Your controller and service code can access the current user, instead of re-checking credentials again.

Principal and Authentication

These two terms will appear in logs and in code, so we will keep them clear.

Principal

Principal represents who the user is, usually the username or email.

Authentication

Authentication represents the full logged-in state, including roles.

What this enables

Once the request has a user and roles, Spring Security can allow or block protected endpoints based on rules.

Basic authentication

What is Basic Authentication?

Basic Authentication is a simple way to secure an API.
For every request, the client sends a username and password so the server can verify who is calling.

How the client sends credentials

The client includes an Authorization header in the HTTP request. The value starts with Basic, followed by a Base64 encoded string.

Example format

```
Authorization: Basic <base64 of username:password>
```

Important note

Base64 is not encryption. It only turns text into a different readable form. Anyone who sees it can decode it back, so Basic Auth must be used with HTTPS

Why Basic Authentication is used & when it makes sense

Basic Authentication is popular because it is easy to set up and easy to test. It works especially well when you control the environment and you are not building a public consumer app.

Good fits

- Internal admin tools.
- Private APIs behind a VPN.
- Learning and demo projects.
- Services protected by an API gateway.

Bad fits

- Public apps where sending a password repeatedly is risky.
- Systems that need logout, token expiry, or session tracking.
- Clients that cannot safely store long-term credentials.

One rule you cannot ignore

If you use Basic authentication, you must use HTTPS. Without HTTPS, credentials can be captured on the network.

What happens on every protected request

Basic Authentication does not create a server-side login session by default. The server verifies credentials again for each protected request.

What the server does each time

- Reads the Authorization header.
- Decodes username and password.
- Verifies the credentials against the user store.
- Loads roles for the user.
- Applies authorization rules for the endpoint.

Why this matters

This repeated verification is one reason many systems later move to token-based approaches. Tokens reduce how often the server needs to re-check passwords.

How will we use Basic Authentication?

We will start with Basic authentication in our project to secure our REST APIs.

Our starting policy

- Signup is public so a new user can be created.
- All other endpoints require authentication.
- Destructive actions require the ADMIN role, such as deleting a course.

How we will verify it using Postman

- Call a protected endpoint without credentials and confirm 401.
- Call the same endpoint with valid credentials and confirm success.
- Call an admin-only endpoint as a student and confirm 403.

DB-backed users and roles in our project

Why we need a User collection in this project

Right now, our Student document represents a person in the domain, like name, email, and active status.
For security, we also need a separate record that represents identity and access.

What a User record is responsible for

- Storing login identity, such as email.
- Storing proof of identity as a password hash.
- Storing roles, such as STUDENT or ADMIN.
- Enabling or disabling access without deleting the student data.

Why we keep it separate from Student

- Student stays clean as domain data.
- User stays focused on authentication and authorization data.

What do we store in the users collection?

We will create a collection named users, and each user will be one document.

Fields we will store

- Email, used as the login identity.
- PasswordHash, used to verify the password securely.
- Roles, a list of role strings such as STUDENT or ADMIN.
- Enabled, so we can disable access quickly.
- CreatedAt and UpdatedAt exist so we can audit basic changes.

Example user document

```
{
  "_id": "65c0a13aa9c7e24e0f4e9012",
  "email": "kiran.p@gmail.com",
  "passwordHash": "5f4dcc3b5aa765d61d8327deb882cf99",
  "roles": ["STUDENT"],
  "enabled": true
}
```

Uniqueness rule and why it matters

In a login system, the email must uniquely identify one user. That means we must not allow two user documents with the same email.

What we enforce

- Create a unique index on email in the users collection.
- If someone tries to create a second user with the same email, MongoDB rejects the write.

Why we enforce this in the database

Even if the API makes a mistake, the database rule still protects data integrity. This keeps login behavior consistent and predictable.

How User links to Student in our system

We will keep Student and User separate and connect them using an id reference.

Recommended link

- Student stores userId.
- User stays independent as the login identity record.

Impact of this design

- One user can represent access even if the student profile changes.
- You can later add other user types without changing Student heavily.
- Roles are owned by User, not by Student.

Example student with a user link

```
{
  "_id": "65b7f3b2a9c7e24e0f4e1a91",
  "name": "Kiran Pillai",
  "email": "kiran.p@gmail.com",
  "active": true,
  "userId": "65c0a13aa9c7e24e0f4e9012"
}
```

How Spring Security will use our DB-backed users

When a request comes in with Basic authentication, Spring Security needs to load the user from MongoDB.
That means our backend must provide a way to find a user by email and read roles.

What happens at a high level

- Spring Security reads email and password from the request.
- Our code loads the user document using the email.
- Our code compares the incoming password with the stored passwordHash.
- Our code returns roles so authorization rules can be applied.

What this enables

Once Spring Security can load users from MongoDB, we can protect endpoints and apply role rules consistently.

Signup and login flow

What does signup means in our system?

Signup is when we create a new user record in our database. After signup, the user can prove their identity and access protected endpoints.

What we store for a new user

- Email as the identity.
- PasswordHash instead of the raw password.
- Roles such as STUDENT or ADMIN.
- Enabled flag so an account can be disabled later.

What we validate at signup

- Email is present and looks like an email.
- Password is present and meets basic length rules.
- Email is unique in the users collection.

Password hash and why we use it

A password hash is a transformed version of the password that we store instead of the real password. The goal is simple. Even if someone reads the database, they should not be able to see the real passwords.

A simple way to think about it

If the password is "Tiger123", we never store "Tiger123"
Instead, we store something that looks like random text, which came from running a hashing function.

Example of what a stored user record looks like

```
{
  "email": "kiran.p@gmail.com",
  "passwordHash": "5f4dcc3b5aa765d61d8327deb882cf99",
  "roles": ["STUDENT"],
  "enabled": true
}
```

What happens during login

- The user types a password.
- The server runs the same hashing step on the typed password.
- The server compares the result with the stored passwordHash.
- If they match, the password is correct.

Login meaning in Basic authentication

With Basic authentication, the client sends credentials on every protected request.
The server will not store any information to allow a user to bypass authentication in subsequent requests.

What the server does on a protected request

- Loads the user by email from the database.
- Hashes the incoming password.
- Compares it with the stored passwordHash.
- Loads roles for the user and applies endpoint rules.

What the client sees

- Correct credentials lead to success.
- Wrong credentials lead to 401.

A simple endpoint to confirm the logged-in user

We can add one endpoint that returns the current user identity and roles.

Example

```
GET /api/me
```

What it should return

- The current user email.
- Roles for the current user.

Why this is useful

It helps us confirm that authentication works and that roles are being loaded correctly.

Authorization and protected routes

What a protected route means

A protected route is an endpoint that should not be accessible to everyone. The server blocks the request unless the caller is authenticated.

Examples

- POST /api/courses should not be public.
- DELETE /api/courses/{id} should not be public.
- GET /api/students should not be public if it exposes all students.

Role-based access in our project

After a user is authenticated, we decide what they are allowed to do using roles.

A simple starting rule set

- STUDENT can read their own student profile.
- STUDENT can create an enrollment for themselves.
- ADMIN can create and delete courses.
- ADMIN can view all students.

Why roles help

Roles let us express security rules clearly without writing custom checks inside every controller method.

Concrete examples using our endpoints

We will map rules to endpoints so it is easy to apply in code later.

Course endpoints

- POST /api/courses requires ADMIN.
- DELETE /api/courses/{id} requires ADMIN.
- GET /api/courses can be allowed for both STUDENT and ADMIN.

Student endpoints

- GET /api/students requires ADMIN because it lists all students.
- GET /api/students/{id} requires authentication.
- PUT /api/students/{id} should allow only the owner or ADMIN.

How the rule check fits into the request flow

Authorization happens after authentication.

The flow

- Request arrives.
- Spring Security identifies the user.
- Spring Security reads roles for that user.
- Spring Security checks the endpoint rule.
- The request is allowed or blocked before reaching controller logic.

What this protects

Even if a controller method exists, it cannot be reached unless security rules allow it.

Activity

Activity

Add authentication and basic authorization to our Spring Boot API using a DB-backed user collection. By the end, protected endpoints should return 401 or 403 correctly, based on who is calling.

You will implement three capabilities in the project.

Capability 1: User signup

- Create a new user in the users collection.
- Store email, passwordHash, roles, enabled.
- Enforce unique email.

Capability 2: Secure endpoints with Basic authentication

- Require authentication for protected endpoints.
- Verify credentials against the users collection.

Capability 3: Role-based protection

- Restrict course creation and deletion to ADMIN.
- Allow course listing for both STUDENT and ADMIN.

Testing using Postman

We’ll run these checks and note the outcomes.

Check 1

Call POST /courses without credentials. Expected result is 401.

Check 2

Signup a STUDENT user and call POST /courses with those credentials. Expected result is 403.

Check 3

Signup an ADMIN user and call POST /courses with those credentials. Expected result is 200 or 201.

Check 4

Call GET /api/courses with a STUDENT user. Expected result is 200.

Success criteria

The submission will be valid when these are true.

- Users are stored in MongoDB and email is unique.
- Basic authentication reads users from MongoDB, not in memory.
- The API returns 401 for missing or wrong credentials.
- The API returns 403 for valid users without required role.
- ADMIN can create and delete courses.
- STUDENT can view courses.

Optional extension if time remains

Protect GET /students so only ADMIN can access it.

That’s a wrap!