

Spring Boot JWT Authentication (Without Bcrypt / With PasswordEncoder)

1 . Flow Overview

- User registers → password hashed → stored in DB.
 - User logs in → username & password sent to login endpoint.
 - `AuthenticationManager` validates credentials using `PasswordEncoder`.
 - If valid → JWT token generated and returned.
 - Client uses JWT in `Authorization` header to access protected routes.
 - JWT filter validates token for each request.
-

2 . Registration

- 1 . User sends JSON `{ username, password }`.
- 2 . Password is hashed via configured `PasswordEncoder`.
- 3 . Save `username` + hashed password in database.

Example: ``PasswordEncoder Bean``

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // or NoOpPasswordEncoder for plain
text
}
```

3 . Login (AuthService)

```
Authentication auth = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(username, password)
);
```

Behind the Scenes:

- 1 . `AuthenticationManager` calls `UserDetailsService.loadUserByUsername(username)` → fetch user from DB.
- 2 . Retrieves stored password.
- 3 . Calls `passwordEncoder.matches(rawPassword, storedPasswordFromDB)` internally.
- 4 . If matches → authentication succeeds, else throws `BadCredentialsException`.

Key Note: In Spring, you do **not manually call bcrypt**; `PasswordEncoder` handles it automatically.

4 . JWT Token Generation

- Once authentication succeeds, generate JWT with `username` and `exp` claims.
- Return token in JSON:

```
{ "access_token": "<JWT_TOKEN>" }
```

5 . Accessing Protected Routes

- Client sends request with header:

```
Authorization: Bearer <JWT_TOKEN>
```

- `JwtAuthFilter` intercepts request:
- Extracts token from header.
- Validates token using `JwtUtil`.
- Sets authentication context if valid.
- Controller executes business logic for authenticated user.

6 . Key Points

- Password comparison occurs **inside AuthenticationManager** using the configured `PasswordEncoder`.
- No explicit bcrypt comparison is needed in your service.
- Can use `NoOpPasswordEncoder` for plaintext (unsafe) or `BCryptPasswordEncoder` for hashed passwords.
- Spring Security handles **authentication, validation, and error handling** automatically.
- JWT makes session stateless and scalable.

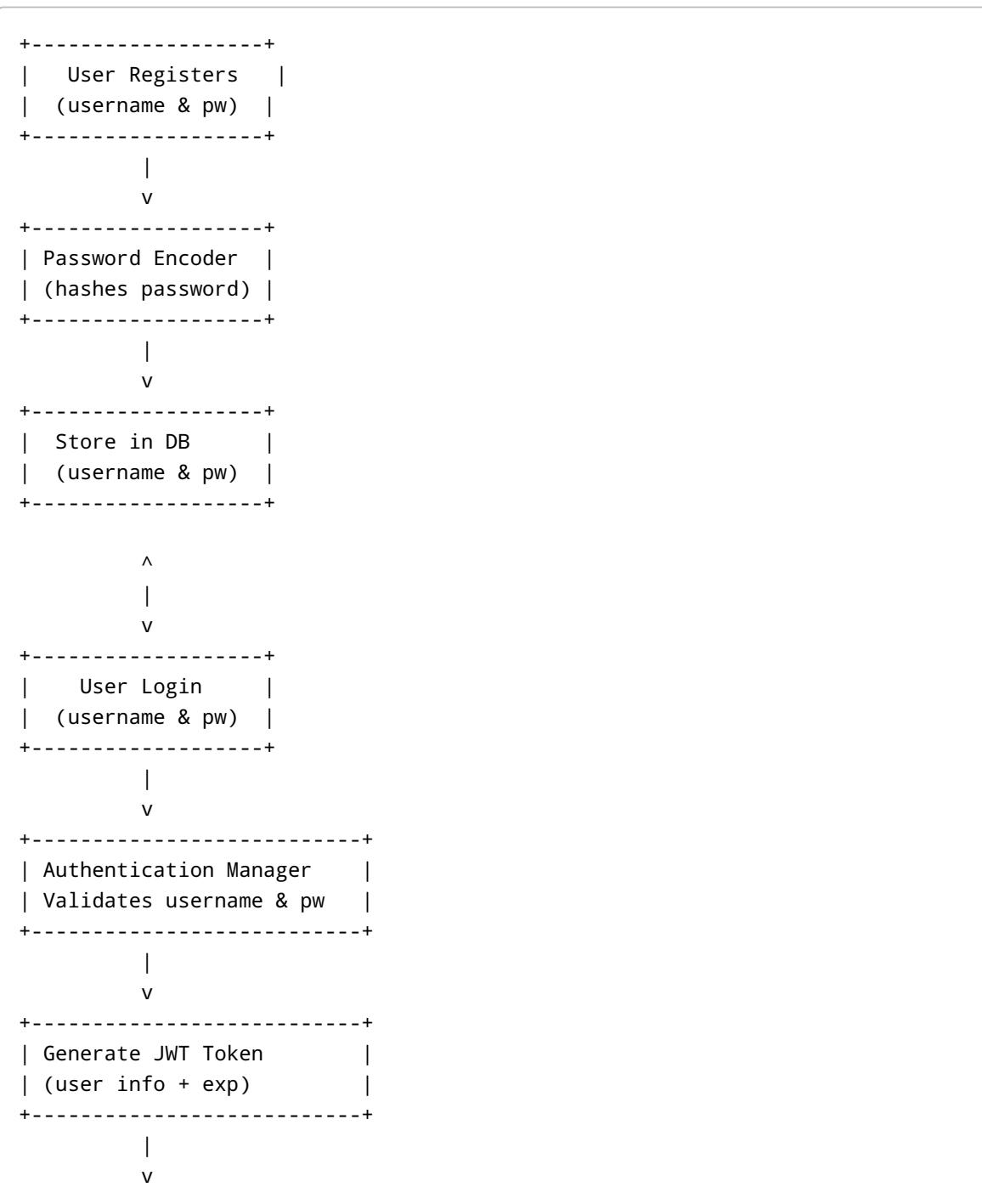
7 . Summary Table: Node.js/Python vs Spring Boot

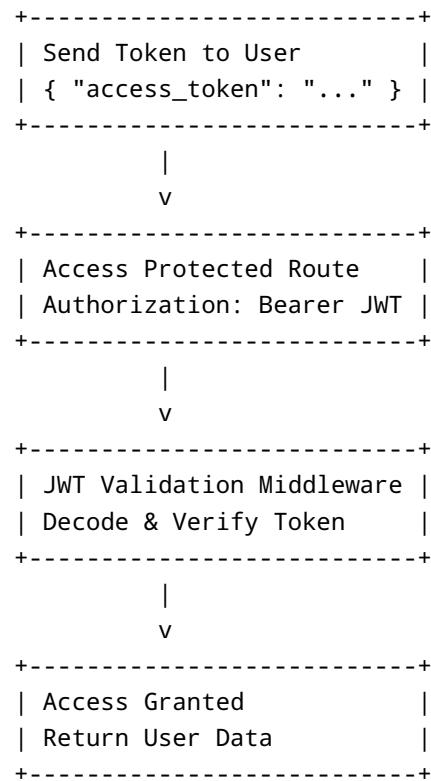
Step	Node.js / Python	Spring Boot
Password hash compare	<code>bcrypt.compare(raw, hash)</code> manually	<code>PasswordEncoder.matches(raw, hash)</code> internally in AuthenticationManager
Who validates	Your code	Spring Security AuthenticationManager
Exception on failure	Manual 4 0 1	<code>BadCredentialsException</code>

Step	Node.js / Python	Spring Boot
Token generation	After manual check	After authentication succeeds

This note summarizes **Spring Boot JWT Authentication**, highlighting where password validation happens internally and how JWT token flow works.

ASCII Flow Diagram: JWT Authentication in Spring Boot





Step-by-Step Explanation

1 . User Registration

- The user sends a **username and password** to the registration endpoint.
- The **password is hashed** using a **PasswordEncoder** (Spring's default encoder or any hash function).
- User credentials (username + hashed password) are **stored in the database**.
- Hashing ensures passwords are not stored in plaintext.

2 . User Login

- The user submits **username and password** to the login endpoint.
- **AuthenticationManager** is used to validate credentials:
- Compares submitted password (after encoding) with the stored hashed password in the DB.
- If credentials are valid → proceed, else return **401 Unauthorized**.

3 . JWT Token Generation

- Once authenticated, Spring Boot generates a **JWT token**.
- Token contains:
 - **sub** claim → username
 - **exp** claim → token expiration

- Token is **signed with a secret key** to prevent tampering.
- Token is returned to the client in JSON:

```
{ "access_token": "<JWT_TOKEN>" }
```

4 . Accessing Protected Routes

- The client sends requests to protected endpoints with:

```
Authorization: Bearer <JWT_TOKEN>
```

- Middleware (or a filter) **extracts and verifies the token**:
- Checks signature
- Checks expiration
- Extracts user info

5 . Access Granted

- If token is valid:
- Request proceeds
- User data or protected resource is returned
- If token is invalid or expired:
- `401 Unauthorized` is returned.

1 . Spring Security Authentication Flow

When you use **Spring Security** and `AuthenticationManager`, here's what happens during login:

- 1 . User sends JSON (or form) with **username & password**.
- 2 . Spring Security creates an `UsernamePasswordAuthenticationToken` with the submitted username and password.
- 3 . `AuthenticationManager` delegates to a `UserDetailsService.loadUserByUsername(username)` to load user details from the database.
- 4 . `UserDetailsService.loadUserByUsername(username)` returns a `UserDetails` object:
 - `username`
 - `password` (already hashed in DB)
 - authorities/roles
- 5 . `AuthenticationManager` checks credentials using a `PasswordEncoder` (e.g., `BCryptPasswordEncoder`, `NoOpPasswordEncoder`, or any custom encoder).

2 . Password Comparison in Spring

- If you used **bcrypt**,
`BCryptPasswordEncoder.matches(rawPassword, encodedPassword)` is called internally.
- If you **don't use bcrypt**, you can use `==`, which basically does:

```
boolean matches = rawPassword.equals(storedPassword);
```

So the comparison still happens just that Spring handles AuthenticationManager → you don't manually call `bcrypt.compare()` like in Node.js or Python.

3 . Example Without bcrypt

```
@Bean  
public PasswordEncoder passwordEncoder() {  
    // No hashing, plain text (not recommended for prod)  
    return NoOpPasswordEncoder.getInstance();  
}
```

- During login:

```
Authentication auth = authenticationManager.authenticate(  
    new UsernamePasswordAuthenticationToken(username, password)  
);
```

- Internally:
 - Loads `UserDetails` from DB (username + storedPassword)
 - Calls `passwordEncoder.matches(password, storedPassword)`
 - If true → authentication succeeds, else exception (`BadCredentialsException`)

4 . Summary

StepNode.js /
PythonSpring Boot

Password hash comparison	<code>bcrypt.compare()</code>	<code>PasswordEncoder.matches()</code>
Who handles it	Your code manually	Spring Security internally
Exception on failure	Return 4 0 1 manually	Throws <code>BadCredentialsException</code> handled by Spring
Token generation	After manual check	After authentication succeeds, you generate JWT manually

✓ Key point:

Even if you don't use bcrypt, Spring still compares the raw password with the stored one whatever `PasswordEncoder` you configured. You don't see the call explicitly because Spring handles it in `AuthenticationManager`.

key piece in your `AuthService`:

```
Authentication auth = authenticationManager.authenticate(  
    new UsernamePasswordAuthenticationToken(username, password)  
);
```

1 . What `` does

- When you call `authenticate()`, Spring Security internally:
- Calls your `` to load the user from DB by username.
- Retrieves the stored password from the database (usually already hashed).
- Calls the configured `` to compare the entered password with the stored password.
- If it matches → returns a fully authenticated `Authentication` object.
- If it doesn't → throws a `BadCredentialsException`.

2 . Where the comparison happens

In your `SecurityConfig` you defined:

```
@Bean  
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

- This bean is automatically picked up by Spring Security.
- During authentication:

```
passwordEncoder.matches(enteredPassword, storedPasswordFromDB)
```

is called internally.

- **You don't see this call** because Spring Security handles it inside `AuthenticationManager`.

So even though your `AuthService` looks like it's just passing the raw password:

```
new UsernamePasswordAuthenticationToken(username, password)
```

...behind the scenes, Spring Security **applies the password encoder to check it against the DB value**.

3 . How it would work without bcrypt

If you don't want to use bcrypt at all, you can configure:

```
@Bean  
public PasswordEncoder passwordEncoder() {  
    return NoOpPasswordEncoder.getInstance(); // plain text comparison  
}
```

Then `matches()` simply does:

```
rawPassword.equals(storedPassword)
```

- Authentication still works.
- But passwords are stored in plaintext — **not secure**.

4 . Flow inside Spring Security

- 1 . User calls `/login` with username & password.
- 2 . `AuthService.login()` creates a `UsernamePasswordAuthenticationToken`.
- 3 . `AuthenticationManager.authenticate()`:
- 4 . Loads user from DB via `UserDetailsService`.
- 5 . Gets hashed password from DB.
- 6 . Calls `passwordEncoder.matches(rawPassword, storedPassword)`.
- 7 . If valid → returns `Authentication` object.
- 8 . If invalid → throws `BadCredentialsException`.
- 9 . `AuthService` can now generate JWT for the authenticated user.



Key Points

- Unlike Node.js/Python where you manually call `bcrypt.compare()`, Spring Security does it for you **inside AuthenticationManager**.
- The `PasswordEncoder` bean you configure determines **how passwords are compared**.
- If you don't configure bcrypt, any other encoder (including `NoOpPasswordEncoder`) is used.