ChatGPT

TokenController – Authentication Endpoints

The **TokenController** is a Spring MVC controller in the authentication service. Its role is to handle user login and token-refresh requests, generating JWT access tokens and managing refresh tokens. In this application, it defines two POST endpoints under `/auth/v1/`: one for **login** and one for **refreshToken**. When a user logs in, this controller authenticates the credentials and returns a JSON payload containing a new JWT (`accessToken`) and a refresh token. When a refresh request is made, it validates the supplied refresh token and issues a new access token.

Overview of annotations: The class is annotated with `@Controller`, and the methods use `@PostMapping` and `@RequestBody`. In a REST-style API that returns JSON, it is common to use `@RestController` (a shorthand for `@Controller` + `@ResponseBody`) [1]. Here, `@Controller` alone marks it as a Spring MVC controller, and each method's return value (a DTO or `ResponseEntity`) will be serialized to JSON (as if `@ResponseBody` were present). The `@PostMapping("auth/v1/login")` and `@PostMapping("auth/v1/refreshToken")` annotations map HTTP POST requests on those paths to the respective methods. The `@RequestBody` annotation on method parameters tells Spring to deserialize the JSON request body into the given DTO object. Dependencies are injected with `@Autowired`, e.g. the `AuthenticationManager`, `RefreshTokenService`, and `JwtService` fields.

Injected Dependencies:

- `AuthenticationManager` – a Spring Security component (typically a `ProviderManager`) that performs authentication checks. Under the hood it will invoke configured `AuthenticationProvider`s (e.g. a `DaoAuthenticationProvider` tied to a `UserDetailsService` and `PasswordEncoder`) to validate credentials [2]. In this controller it's used to authenticate the username/password.
- `RefreshTokenService` – a custom service (bean) responsible for managing refresh tokens. It provides methods like `createRefreshToken(username)` and `findByToken(token)` / `verifyExpiration(...)`. It likely stores tokens in a database and checks their expiry.
- `JwtService` – a custom service for handling JWT generation. The method `GenerateToken(username)` creates a signed JWT string for the given user.

Each dependency is injected into the controller via `@Autowired` so the controller can call their methods.

POST /auth/v1/login

```
@PostMapping("auth/v1/login")
public ResponseEntity AuthenticateAndGetToken(@RequestBody AuthRequestDTO
authRequestDTO) {
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(authRequestDTO.getUsername(),
authRequestDTO.getPassword()));
```

```
    if (authentication.isAuthenticated()) {
        RefreshToken refreshToken =
refreshTokenService.createRefreshToken(authRequestDTO.getUsername());
        return new ResponseEntity<>(JwtResponseDTO.builder()
            .accessToken(jwtService.GenerateToken(authRequestDTO.getUsername()))
            .token(refreshToken.getToken())
            .build(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>("Exception in User Service",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

- **Method signature:** `public ResponseEntity AuthenticateAndGetToken(AuthRequestDTO authRequestDTO)` returns a Spring `ResponseEntity`. The generic type is not specified, but in practice a `JwtResponseDTO` is returned on success.
- **Input parameter:** `AuthRequestDTO authRequestDTO` – this object is populated from the JSON request body (due to `@RequestBody`). It must contain at least a `username` and `password`. For example, the client should send:

```
{ "username": "user1", "password": "secret" }
```

- **Behavior:** The method calls `authenticationManager.authenticate(...)` with a `UsernamePasswordAuthenticationToken` built from the supplied username and password. This triggers Spring Security's authentication process [2]. If the credentials are valid, `authenticate()` returns an `Authentication` whose `isAuthenticated()` is true. (If invalid, it will throw an `AuthenticationException`.)
- If authenticated, the code then:
- Calls `refreshTokenService.createRefreshToken(username)` to generate a new refresh token for the user.
- Calls `jwtService.GenerateToken(username)` to create a new JWT access token (a string).
- Builds a `JwtResponseDTO` (via its builder) containing the `accessToken` (the JWT string) and `token` (the new refresh token).
- Returns a `ResponseEntity` with that DTO as the body and **HTTP 200 OK** status. (`ResponseEntity` is a Spring type representing the full HTTP response, including status and body [3].)
- **Outputs:** On success, the response body is a JSON object like:

```
{ "accessToken": "<JWT string>", "token": "<refresh_token>" }
```

and the HTTP status is 200 OK.
- **Failure:** If `authenticate()` fails (wrong credentials), an exception is thrown and not caught here, which will result in an error (typically HTTP 401 or 500, depending on global exception handling). The code's `else` block (`isAuthenticated() == false`) is unlikely to be reached because bad

credentials cause an exception; but if it did run, it returns a 500 Internal Server Error with a plain message.

Return type: The method returns `ResponseEntity`, allowing it to specify both the response body (the `JwtResponseDTO`) and the HTTP status code explicitly. This matches Spring's normal practice of returning data with a status [3].

POST /auth/v1/refreshToken

```
@PostMapping("auth/v1/refreshToken")
public JwtResponseDTO refreshToken(@RequestBody RefreshTokenRequestDTO
refreshTokenRequestDTO) {
    return refreshTokenService.findByToken(refreshTokenRequestDTO.getToken())
        .map(refreshTokenService::verifyExpiration)
        .map(RefreshToken::getUserInfo)
        .map(userInfo -> {
            String accessToken =
jwtService.GenerateToken(userInfo.getUsername());
            return JwtResponseDTO.builder()
                .accessToken(accessToken)
                .token(refreshTokenRequestDTO.getToken())
                .build();
        })
        .orElseThrow(() -> new RuntimeException("Refresh Token is not in
DB..!!"));
}
```

- **Method signature:** `public JwtResponseDTO refreshToken(RefreshTokenRequestDTO refreshTokenRequestDTO)`. It returns a `JwtResponseDTO` object directly (no `ResponseEntity`). As a result, Spring will serialize this object to JSON with a 200 OK status by default.
- **Input parameter:** `RefreshTokenRequestDTO refreshTokenRequestDTO` – populated from JSON request body. It should contain a `token` field, which is the refresh token previously issued. For example:

```
{ "token": "<refresh_token>" }
```

- **Behavior:**
- `refreshTokenService.findByToken(token)` looks up the `RefreshToken` entity/object corresponding to the token string. This returns an `Optional<RefreshToken>`.
- `.map(refreshTokenService::verifyExpiration)` likely checks if the token is expired; if expired, it may throw an exception or return null, causing the Optional to be empty.
- `.map(RefreshToken::getUserInfo)` retrieves user information associated with the refresh token (presumably a user entity or DTO).

- If present, `.map(userInfo -> { ... })` is invoked: it generates a new access token via `jwtService.GenerateToken(userInfo.getUsername())`. It then builds and returns a `JwtResponseDTO` containing the **new access token** and the **same refresh token** string (`refreshTokenRequestDTO.getToken()`).
- If at any point the token is not found or expired, the `Optional` chain is empty and `orElseThrow` triggers: throwing a `RuntimeException("Refresh Token is not in DB..!!")`. This will result in a 500 Internal Server Error unless a controller advice handles it.
- **Outputs:** On success, the method returns a `JwtResponseDTO` (which Spring serializes to JSON). The JSON will look like:

  ```
  { "accessToken": "<new_JWT>", "token": "<same_refresh_token>" }
  ```

  with HTTP 200 OK. On failure (invalid or missing token), an exception is thrown, leading to an error response (typically 500).
- **Return type:** `JwtResponseDTO` – a simple DTO (likely with fields for `accessToken` and `token`).

Input/Output Summary

- `AuthRequestDTO` (login input): expected JSON with `username` and `password` fields.
- `RefreshTokenRequestDTO` (refresh input): expected JSON with `token` field (a string).
- **Successful login output:** JSON with `accessToken` (JWT string) and `token` (refresh token), HTTP 200.
- **Failed login:** The code as written would throw an exception or return 500; it does not explicitly return a 401.
- **Successful refresh output:** JSON with a new `accessToken` and the same `token`, HTTP 200.
- **Failed refresh:** If token is invalid/expired, a `RuntimeException` is thrown (resulting in a 500 error response by default).

Security and Validation

- **Authentication:** The `/login` endpoint uses `AuthenticationManager.authenticate(...)`. Internally, this checks the supplied username/password against the user data (via a `UserDetailsService` and `PasswordEncoder`) [2]. If the credentials are incorrect, Spring Security will throw a `BadCredentialsException` (or similar), resulting in an authentication failure.
- **Refresh token check:** The `/refreshToken` endpoint validates the refresh token by looking it up and calling `verifyExpiration`. If the token is expired or not found, the code throws an exception. In a robust implementation one might catch these and return a 401 or 400, but here they propagate as a 500 error.
- **Status codes:** The controller explicitly returns `HttpStatus.OK` on success for the login. It returns `HttpStatus.INTERNAL_SERVER_ERROR` on the (unlikely) `else` branch. In practice, invalid credentials or tokens would typically lead to 401 Unauthorized, but that behavior would depend on global exception handling outside this controller.
- **Validation:** There is no explicit input validation (e.g. no `@Valid` or null checks). The assumption is that Spring's data binding will populate the DTOs. If required fields are missing or JSON is malformed, Spring would return a 400 Bad Request automatically.

- **Token generation:** The `JwtService.GenerateToken(username)` presumably signs the JWT with a secret/key. That detail is handled in `JwtService`, not shown here.

Overall, **TokenController** ties together Spring Security and custom token services to implement a stateless JWT-based login flow. It relies on Spring's `AuthenticationManager` for security and uses DTOs for request/response payloads. The use of `@Controller` (instead of `@RestController`) means each method's return value is serialized as the response body [1]. Successful responses use HTTP 200 and include a `JwtResponseDTO` body (built via its `builder()` in code), while failures result in exceptions (currently leading to 500 errors by default).

Sources: The behavior of `@Controller` vs `@RestController` and usage of `ResponseEntity` are standard in Spring [1] [3]. The authentication call uses Spring Security's `AuthenticationManager` (typically a `ProviderManager` with `DaoAuthenticationProvider`) to validate credentials [2]. These patterns are consistent with common Spring Boot JWT authentication examples.

---

[1] java - Difference between spring @Controller and @RestController annotation - Stack Overflow
https://stackoverflow.com/questions/25242321/difference-between-spring-controller-and-restcontroller-annotation

[2] java - How the Spring Security AuthenticationManager authenticate() method is able to check if the username and password sent are correct? - Stack Overflow
https://stackoverflow.com/questions/70101522/how-the-spring-security-authenticationmanager-authenticate-method-is-able-to-c

[3] How to Use Spring ResponseEntity to Manipulate the HTTP Response? | GeeksforGeeks
https://www.geeksforgeeks.org/how-to-use-spring-responseentity-to-manipulate-the-http-response/