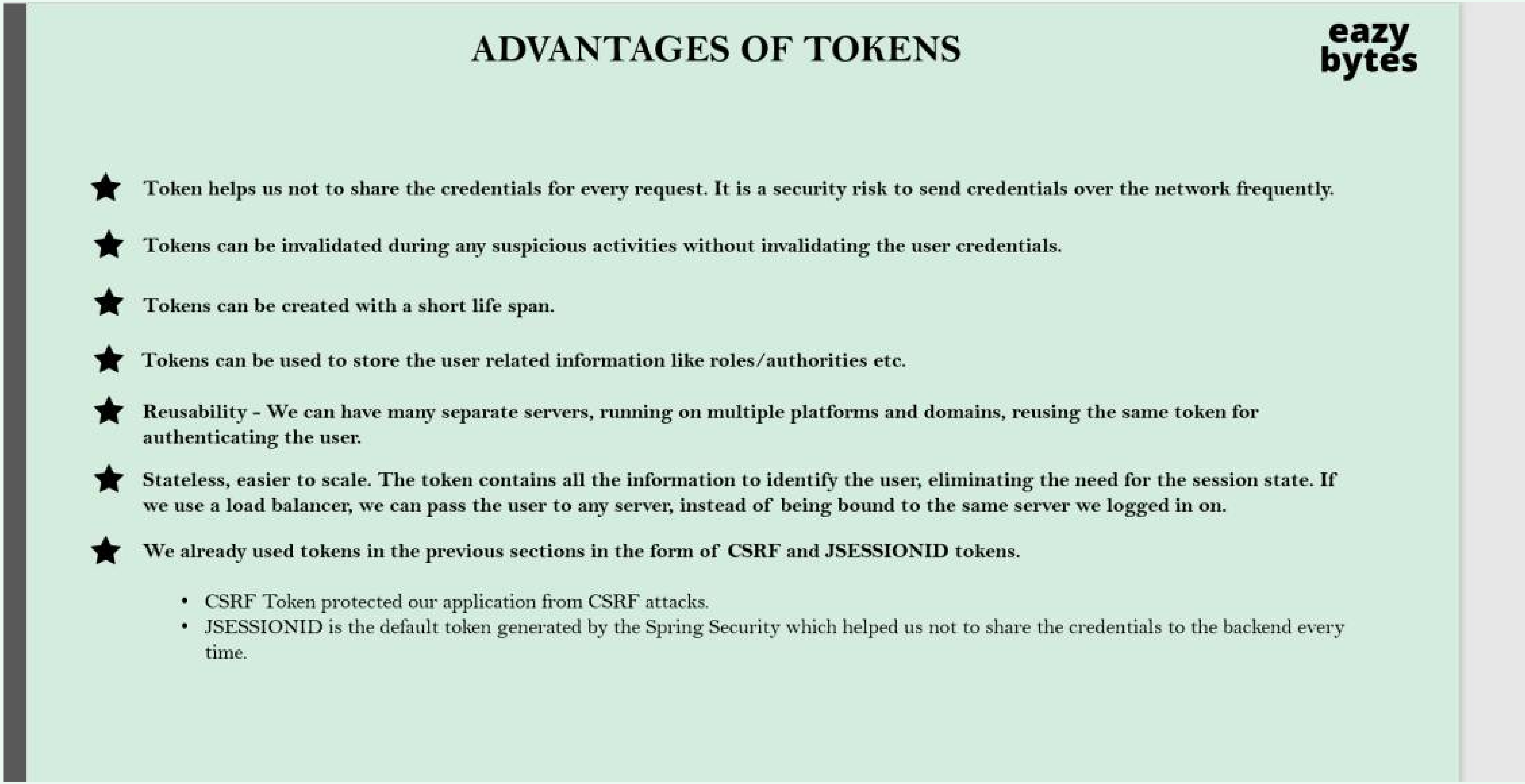
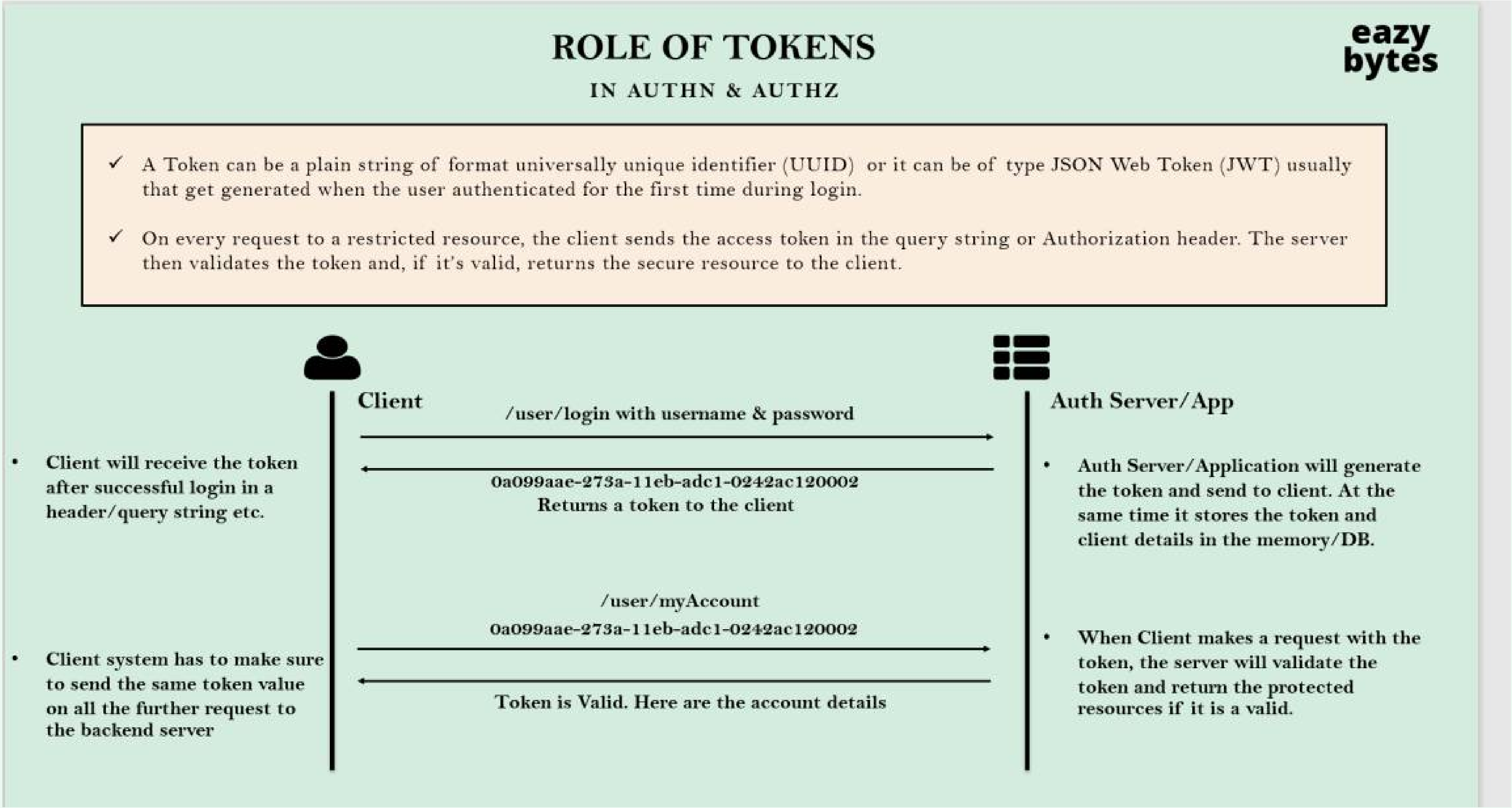
7-Tokens in AuthN AuthZ



ADVANTAGES OF TOKENS

- Token helps us not to share the credentials for every request. It is a security risk to send credentials over the network frequently.
- Tokens can be invalidated during any suspicious activities without invalidating the user credentials.
- Tokens can be created with a short life span.
- Tokens can be used to store the user related information like roles/authorities etc.
- Reusability - We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.
- Stateless, easier to scale. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.
- We already used tokens in the previous sections in the form of CSRF and JSESSIONID tokens.
  - CSRF Token protected our application from CSRF attacks.
  - JSESSIONID is the default token generated by the Spring Security which helped us not to share the credentials to the backend every time.



ROLE OF TOKENS

IN AUTHN & AUTHZ

✓ A Token can be a plain string of format universally unique identifier (UUID) or it can be of type JSON Web Token (JWT) usually that get generated when the user authenticated for the first time during login.

✓ On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it's valid, returns the secure resource to the client.

Client
Auth Server/App
/user/login with username & password
0a099aae-273a-11eb-adc1-0242ac120002
Returns a token to the client
/user/myAccount
0a099aae-273a-11eb-adc1-0242ac120002
Token is Valid. Here are the account details

- Client will receive the token after successful login in a header/query string etc.
- Client system has to make sure to send the same token value on all the further request to the backend server

- Auth Server/Application will generate the token and send to client. At the same time it stores the token and client details in the memory/DB.
- When Client makes a request with the token, the server will validate the token and return the protected resources if it is a valid.

Kindly explain the above image in details with detailed explantion

The two images provide a comprehensive explanation of tokens, their advantages, and their role in authentication (AuthN) and authorization (AuthZ). Let's break this down in detail:

Image 1: Advantages of Tokens

This image explains the benefits of using tokens for secure communication and access control in applications.

1. **Security Without Repeated Credentials Sharing:**
    - Tokens eliminate the need to send user credentials (like username and password) with every request.
    - This reduces the risk of credentials being intercepted over the network.
2. **Revocation of Tokens:**
    - Tokens can be invalidated if suspicious activities are detected, ensuring that user credentials remain safe and unaffected.
3. **Short Lifespan:**
    - Tokens can be configured to expire after a specific duration. This minimizes security risks if a token is compromised.
4. **Role-Based Information:**
    - Tokens can store user-specific information such as roles or authorities, allowing secure access control.
5. **Reusability Across Platforms:**
    - Tokens enable authentication across multiple servers, platforms, and domains without requiring multiple credentials.
6. **Stateless and Scalable:**
    - Since tokens contain all necessary user information, they eliminate the need for server-side session storage.
    - This makes systems stateless and easier to scale, especially when using load balancers. Any server can process the token without maintaining session data.
7. **Previous Usage Examples:**
    - **CSRF Tokens:** Protect applications from Cross-Site Request Forgery attacks.
    - **JSESSIONID Tokens:** Generated by Spring Security to maintain sessions without repeatedly sharing credentials with the backend.

Image 2: Role of Tokens in AuthN & AuthZ

This image illustrates how tokens are used in the authentication (AuthN) and authorization (AuthZ) process with a diagram and additional details.

Key Concepts:

1. **Token Formats:**
    - A token can be a plain string (e.g., UUID) or a structured format like a JSON Web Token (JWT).

  - Tokens are typically generated when the user logs in for the first time.
2. **Token Usage:**
   - For every request to a restricted resource, the client includes the token in the query string or `` `Authorization` `` header.
   - The server validates the token and grants access to secure resources if the token is valid.

Flow of Events (Illustrated in the Diagram):

1. **Login Process:**
   - The client sends credentials (e.g., username and password) to the authentication server (Auth Server).
   - The server authenticates the user and generates a token.
   - The token is sent back to the client, usually in a header or query string.
2. **Accessing Protected Resources:**
   - The client includes the token in every subsequent request to the backend server.
   - The server validates the token against its memory or database and ensures it’s valid.
   - If valid, the server grants access to the requested resource.

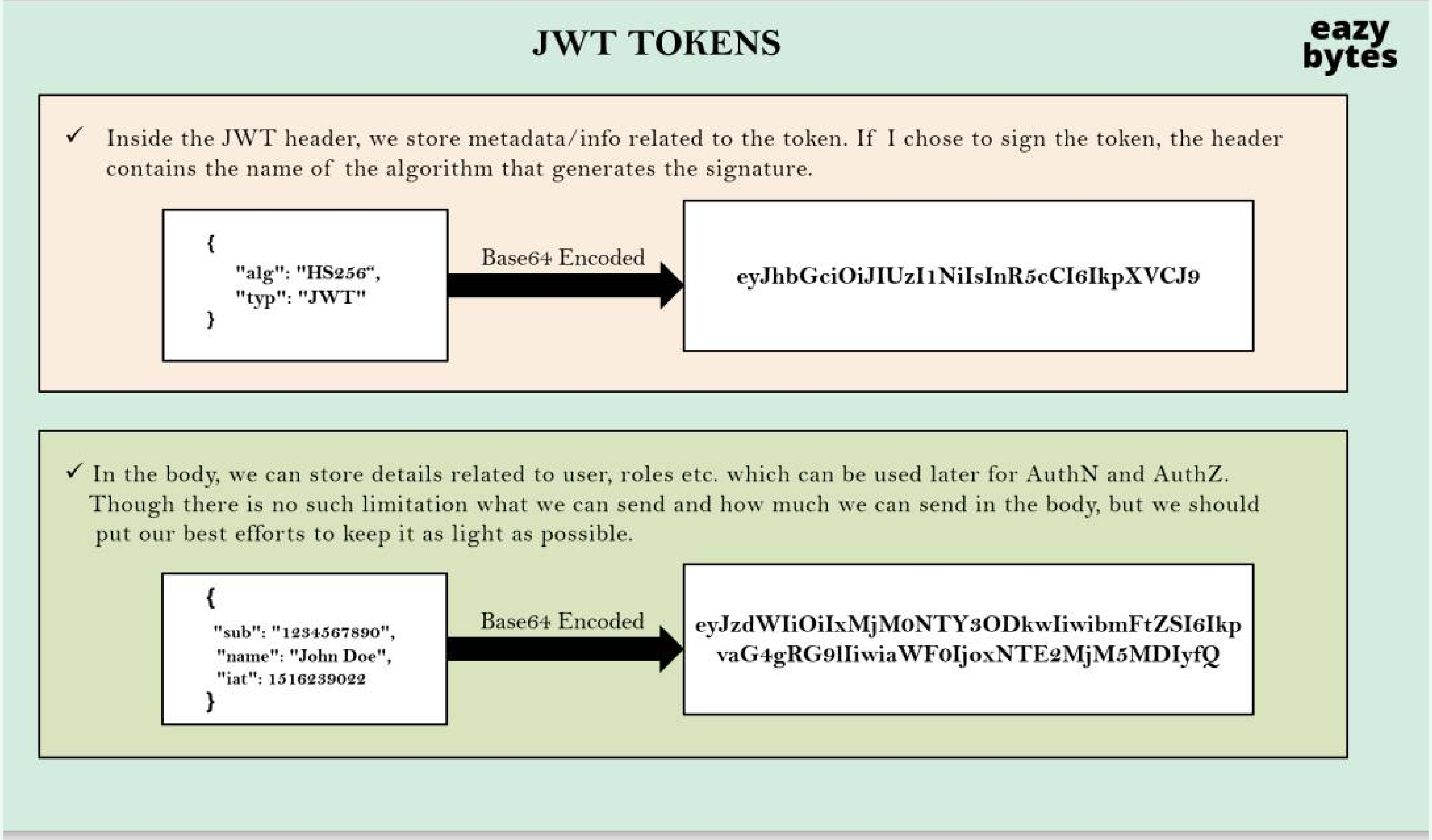
Responsibilities:

- **Client Side:**
  - Store and reuse the token for subsequent requests.
- **Auth Server/Backend:**
  - Generate and validate tokens.
  - Store token and client details in memory or a database.

Key Takeaways:

- **Efficiency:** Tokens streamline authentication by eliminating the need for repeated credential verification.
- **Security:** Short-lived, revocable, and stateless tokens provide enhanced security.
- **Scalability:** Stateless systems using tokens can efficiently scale across distributed systems and load balancers.
- **Flexibility:** Tokens work across multiple domains and platforms, ensuring consistent authentication.

Would you like further details on token implementation or specific use cases like JWT?



JWT TOKENS

eazy bytes

✓ Inside the JWT header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

Base64 Encoded

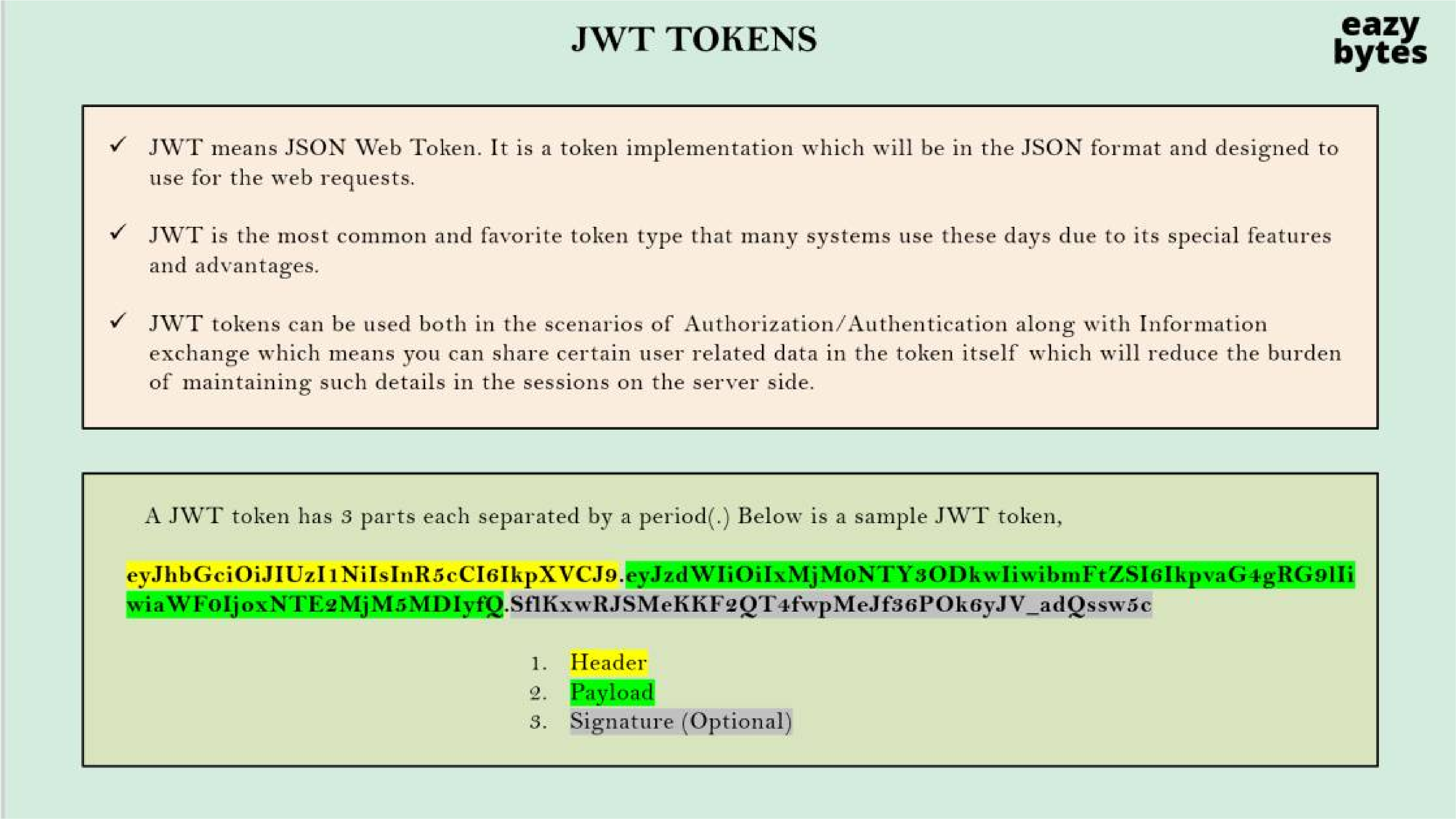
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

✓ In the body, we can store details related to user, roles etc. which can be used later for AuthN and AuthZ. Though there is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Base64 Encoded

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ



JWT TOKENS

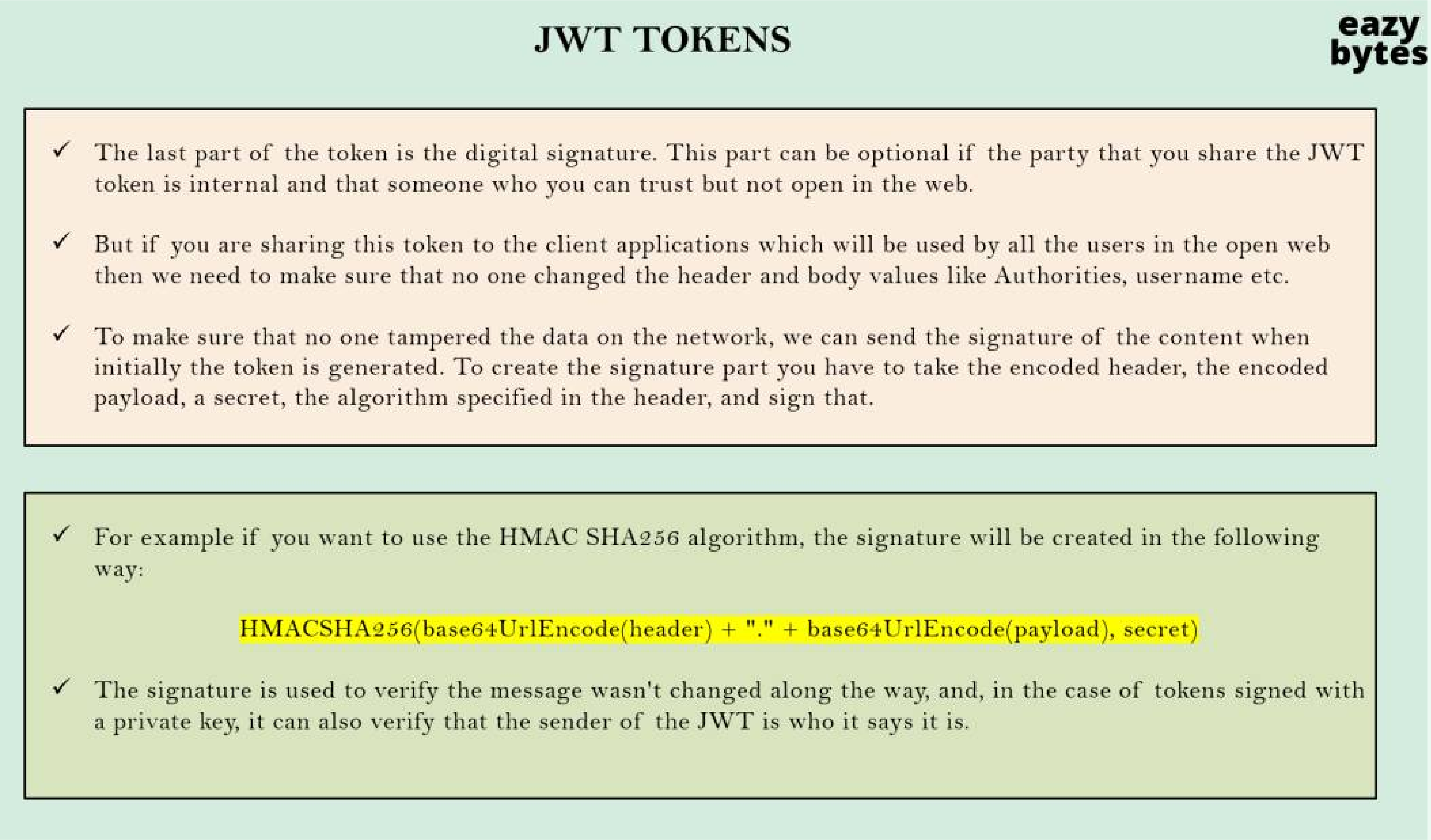
eazy bytes

- ✓ JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
- ✓ JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.
- ✓ JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.

A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1. Header
2. Payload
3. Signature (Optional)



JWT TOKENS

- ✓ The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.
- ✓ But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.
- ✓ To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

- ✓ For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

- ✓ The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Kindly explain the above image in details with detailed explantion

The images explain **JWT (JSON Web Token)** in detail, focusing on its structure, components, and significance. Let’s go through the details systematically:

Image 1: JWT Token Structure

1. **Header:**
    - The **header** contains metadata about the token.
    - Typically includes:
        - **Algorithm (alg):** The hashing algorithm used for signing (e.g., HS256 for HMAC with SHA-256).
        - **Type (typ):** Specifies that it’s a JWT.
    - Example:

```json
{
  "alg": "HS256",
  "typ": "JWT"
}
```

    - This header is **Base64-encoded** into a string.

2. **Payload:**
    - The **payload** holds user-specific information or claims.
    - Can include:
        - **Registered claims:** Predefined fields like `sub` (subject), `iat` (issued at), `exp` (expiration).
        - **Public claims:** Custom data, like user roles or permissions.
        - **Private claims:** Application-specific data.
    - Example:

```json
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

   - Like the header, the payload is also **Base64-encoded**.
3. **Signature:**
   - The signature ensures that the token is not tampered with.
   - Generated by:
     - Taking the encoded header and payload.
     - Using the specified algorithm and a secret key.
     - Producing a hash that gets appended to the token.
   - Formula:

```
plaintext


 HMACSHA256(
   Base64UrlEncode(header) + "." + Base64UrlEncode(payload),
   secret
 )
```

4. **Complete Token Structure:**
   - The final JWT consists of three parts:
     - Header
     - Payload
     - Signature
   - Separated by periods (`` ` `` `.` `` ` ``), e.g.:

```
plaintext


 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwp
 MeJf36POk6yJV_adQssw5c
```

Image 2: Advantages and Use Cases of JWT

1. **What is JWT?**

   - JWT is a **token implementation** in JSON format, commonly used for web requests.
   - It's widely adopted due to its simplicity and versatility.
2. **Uses of JWT:**
   - **Authentication & Authorization:**
     - A client can authenticate using a JWT token, which carries all necessary claims (e.g., roles, permissions).
   - **Information Exchange:**
     - Tokens include user-specific data, reducing the need for server-side session storage.
3. **Stateless Nature:**
   - Since all required data is included in the token itself, the server doesn't need to maintain session state.

Image 3: Security of JWT

1. **Digital Signature:**
   - The signature is crucial for verifying the token's integrity.
   - It ensures that:
     - The data has not been altered.
     - The token originates from a trusted source.
2. **Usage of Signature:**
   - If shared over open networks, the signature protects the data from tampering.
   - The process involves encoding the header and payload, then signing it with a secret key or private key (in asymmetric algorithms).
3. **Example of Signature Algorithm:**
   - **HMACSHA256:**

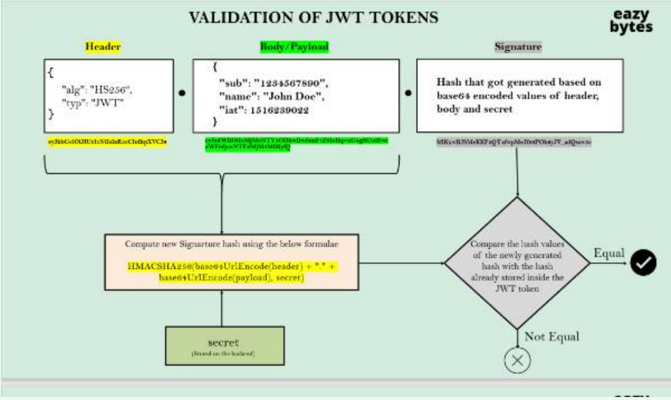
     ```
     plaintext

     HMACSHA256(
       Base64UrlEncode(header) + "." + Base64UrlEncode(payload),
       secret
     )
     ```
4. **Tamper Resistance:**
   - The signature ensures no unauthorized modifications to the payload or header.

5. **Verification:**
    - On receiving the token, the server:
        - Decodes the token.
        - Validates the signature using the shared secret or public key.
        - Checks claims like expiration (`` `exp` ``).

Key Benefits of JWT:

1. **Compact and Portable:**
    - Encoded as a single string, making it ideal for URLs, HTTP headers, etc.
2. **Secure:**
    - Prevents tampering with the payload using signatures.
3. **Decentralized:**
    - Removes the need for centralized session storage on the server.
4. **Versatile:**
    - Can be used across different platforms and systems.

Would you like further details on implementing JWT or its security considerations?


VALIDATION OF JWT TOKENS
eazy bytes
Header
Body/Payload
Signature
{ "alg": "HS256", "typ": "JWT" }
{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }
Hash that got generated based on base64 encoded values of header, body and secret
Compute new Signarture hash using the below formulae
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
secret
Compare the hash values of the newly generated hash with the hash already stored inside the JWT token
Equal
Not Equal

Kindly explain the above image in details with detailed explanation

The image explains the process of **JWT (JSON Web Token) validation**, breaking down the steps involved in verifying a token’s integrity and authenticity. Here's a detailed explanation:

Components of JWT Validation

1. **Header:**
    - Contains metadata about the token, such as:
        - **`alg`**: Algorithm used to sign the token (e.g., HS256).
        - **`typ`**: Specifies the token type (e.g., JWT).
    - This is Base64-encoded.

    Example Header:

    ```json
    {
        "alg": "HS256",
        "typ": "JWT"
    }
    ```

2. **Payload/Body:**
    - Contains claims (user-specific information) such as:
        - **`sub`**: Subject or user ID.
        - **`name`**: Name of the user.
        - **`iat`**: Issued at timestamp.
    - This section is also Base64-encoded.

    Example Payload:

```
json


  {
      "sub": "1234567890",
      "name": "John Doe",
      "iat": 1516239022
  }
```

3. **Signature:**
    - A hashed value generated using:
        - The encoded header.
        - The encoded payload.
        - A secret key stored on the server.
    - The formula used for the signature:

      ```
      plaintext


        HMACSHA256(
          Base64UrlEncode(header) + "." + Base64UrlEncode(payload),
          secret
        )
      ```

   Example Signature:

   ```
   plaintext


     SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
   ```

   The signature ensures the token has not been tampered with.

Validation Process

1. **Step 1: Decode the JWT**

  - The JWT is composed of three parts separated by dots (`.`):
    - Header (Base64-encoded).
    - Payload (Base64-encoded).
    - Signature.

  Example JWT:

  ```plaintext
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf3
  6POk6yJV_adQssw5c
  ```

2. **Step 2: Compute a New Signature**
   - The server uses the same formula to compute a new signature:

     ```plaintext
     HMACSHA256(
       Base64UrlEncode(header) + "." + Base64UrlEncode(payload),
       secret
     )
     ```

   - The **secret key** (stored securely on the backend) ensures the integrity of the signature.
3. **Step 3: Compare Signatures**
   - The server compares the newly computed signature with the signature included in the received JWT.
   - If the two signatures match:
     - The token is valid.
     - The payload (claims) has not been tampered with.
   - If the signatures do not match:
     - The token is invalid.
     - It may have been tampered with or altered.
4. **Decision Point:**
   - **Equal:** If the signatures match, the server considers the token valid and grants access.
   - **Not Equal:** If the signatures do not match, the server rejects the token, and access is denied.

Key Points of JWT Validation

1. **Integrity Check:**
    - The signature ensures that the token's data (header and payload) has not been altered.
2. **Secret Key Dependency:**
    - The secret key used to generate and validate the signature must be securely stored on the server.
    - Any compromise of the secret key can lead to token forgery.
3. **Stateless Validation:**
    - No database lookup is required during token validation, making it a lightweight and efficient process.
4. **Expiration and Claims Validation:**
    - While this diagram focuses on signature validation, the server should also check:
        - The `exp` (expiration) claim to ensure the token hasn't expired.
        - Other claims like `aud` (audience) or `iss` (issuer) to validate the token's context.

Summary

- JWT validation ensures the authenticity and integrity of the token.
- It uses cryptographic methods (e.g., HMACSHA256) to validate the token without storing session data on the server.
- The process relies heavily on a securely stored secret key to prevent tampering.

Would you like to dive deeper into implementing JWT validation in a specific language or framework?

Write the logic how to implement Jwt token in SpringBoot.Kindly write the whole logic for it and explain the code as well'

Below is the implementation logic for JWT in a Spring Boot application. The implementation includes generating a JWT, validating it, and securing APIs using JWT authentication.

Step 1: Maven Dependency

Add the following dependencies to your `pom.xml`:

```
xml



<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Step 2: JWT Utility Class

Create a utility class for generating and validating JWT tokens.

```
java



import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.function.Function;

@Component
public class JwtUtil {

    private String SECRET_KEY = "mySecretKey"; // Use a strong secret in production

    // Generate a token
    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10)) // 10 hours expiration
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
                .compact();
    }
```

```java
    // Extract username from token
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    // Extract expiration date
    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    // Extract specific claim
    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
    }

    // Check if the token is expired
    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    // Validate token
    public Boolean validateToken(String token, String username) {
        final String extractedUsername = extractUsername(token);
        return (extractedUsername.equals(username) && !isTokenExpired(token));
    }
}
```

Step 3: Custom UserDetailsService

Implement a `` `UserDetailsService` `` to fetch user information.

java

```java
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
```

```
@Service
public class MyUserDetailsService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // In real-world applications, fetch user details from DB
        return new User("user", "password", new ArrayList<>());
    }
}
```

Step 4: JWT Request and Response Classes

Create classes to handle authentication requests and responses.

java

```
public class AuthenticationRequest {
    private String username;
    private String password;

    // Getters and Setters
}

public class AuthenticationResponse {
    private String jwt;

    public AuthenticationResponse(String jwt) {
        this.jwt = jwt;
    }

    // Getter
}
```

Step 5: Authentication Controller

Create a REST controller for handling authentication.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.AuthenticationException;
import org.springframework.web.bind.annotation.*;

@RestController
public class AuthController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private MyUserDetailsService userDetailsService;

    @PostMapping("/authenticate")
    public AuthenticationResponse createAuthenticationToken(@RequestBody AuthenticationRequest request) throws Exception {
        try {
            authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(request.getUsername(), request.getPassword())
            );
        } catch (AuthenticationException e) {
            throw new Exception("Incorrect username or password", e);
        }

        final String jwt = jwtUtil.generateToken(request.getUsername());
        return new AuthenticationResponse(jwt);
    }
}
```

Step 6: JWT Filter

Create a filter to validate JWT in incoming requests.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private MyUserDetailsService userDetailsService;

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
            throws ServletException, IOException {

        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);
            if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {
                UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        chain.doFilter(request, response);
    }
}
```

Step 7: Security Configuration

Configure Spring Security to use the JWT filter.

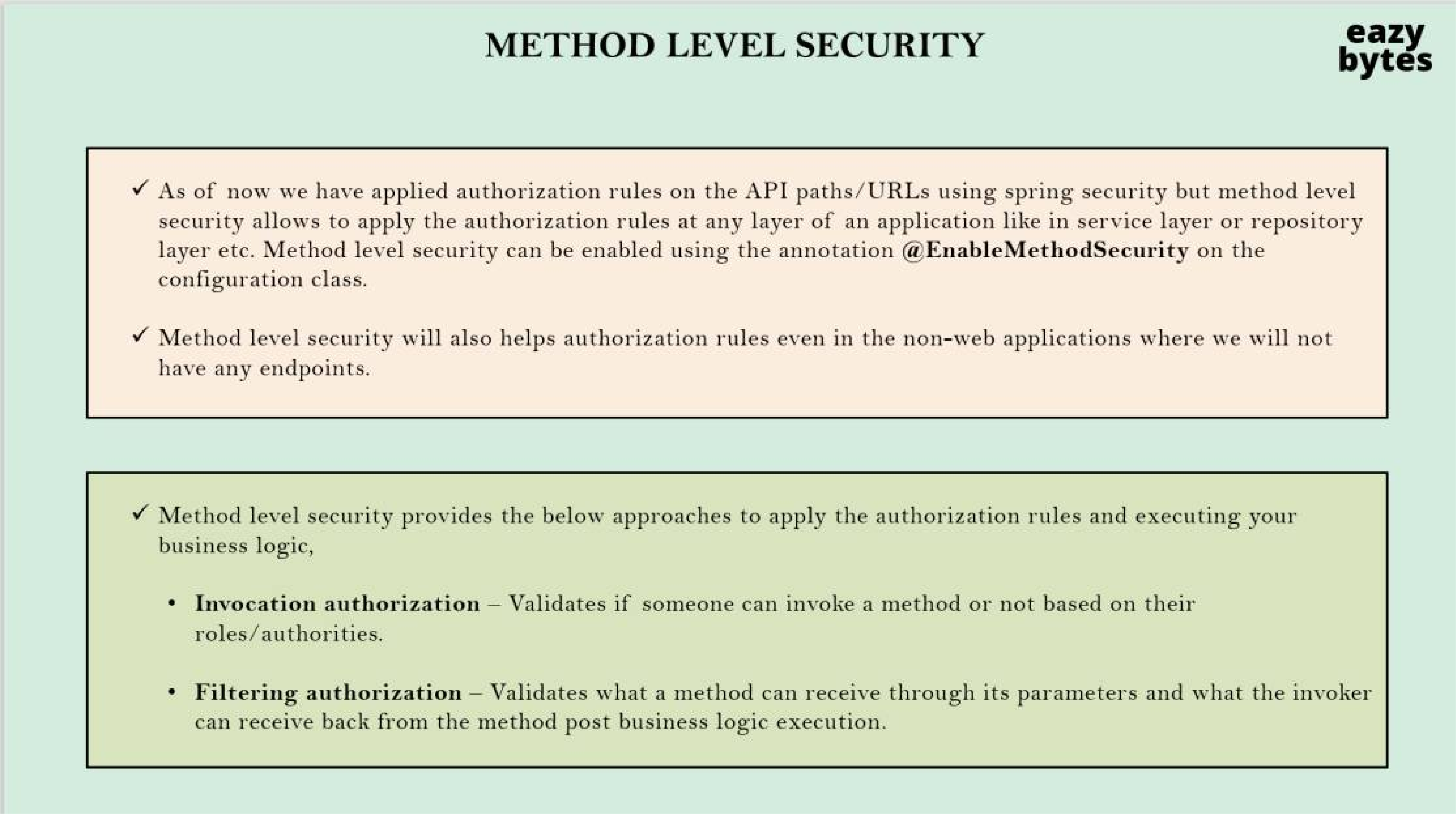
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private MyUserDetailsService userDetailsService;

    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests().antMatchers("/authenticate").permitAll()
                .anyRequest().authenticated()
                .and().sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
```

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

Explanation of Code

1. **JwtUtil:**
    - Handles token generation and validation.
    - Uses a secret key to sign the token.
2. **UserDetailsService:**
    - Fetches user details (can be extended to fetch from a database).
3. **Authentication Controller:**
    - Authenticates user credentials.
    - Generates a JWT for valid users.
4. **JwtRequestFilter:**
    - Intercepts requests to validate the JWT.
    - Adds authentication to the security context if the token is valid.
5. **SecurityConfig:**
    - Configures Spring Security to use stateless sessions with JWT.
    - Adds the custom JWT filter to the security filter chain.

This implementation is a basic template. In a production-grade application, use environment variables or configuration files for the secret key and strengthen validation mechanisms. Would you like help deploying or testing this code?



METHOD LEVEL SECURITY

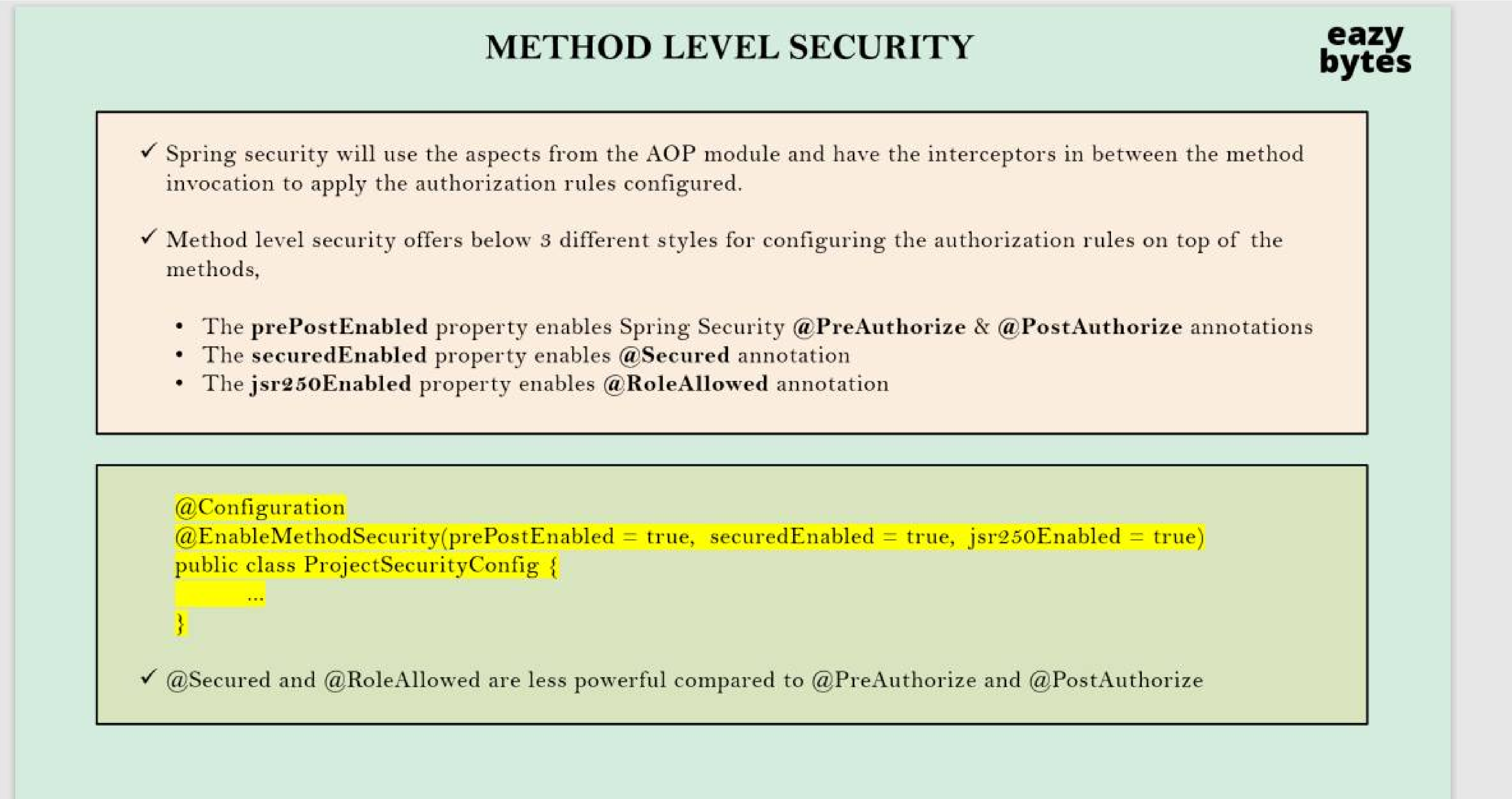
- ✓ As of now we have applied authorization rules on the API paths/URLs using spring security but method level security allows to apply the authorization rules at any layer of an application like in service layer or repository layer etc. Method level security can be enabled using the annotation **@EnableMethodSecurity** on the configuration class.

- ✓ Method level security will also helps authorization rules even in the non-web applications where we will not have any endpoints.

- ✓ Method level security provides the below approaches to apply the authorization rules and executing your business logic,

  - **Invocation authorization** – Validates if someone can invoke a method or not based on their roles/authorities.

  - **Filtering authorization** – Validates what a method can receive through its parameters and what the invoker can receive back from the method post business logic execution.



METHOD LEVEL SECURITY

- ✓ Spring security will use the aspects from the AOP module and have the interceptors in between the method invocation to apply the authorization rules configured.
- ✓ Method level security offers below 3 different styles for configuring the authorization rules on top of the methods,
  - The **prePostEnabled** property enables Spring Security **@PreAuthorize** & **@PostAuthorize** annotations
  - The **securedEnabled** property enables **@Secured** annotation
  - The **jsr250Enabled** property enables **@RoleAllowed** annotation

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true,  securedEnabled = true,  jsr250Enabled = true)
public class ProjectSecurityConfig {
        ...
}
```

- ✓ @Secured and @RoleAllowed are less powerful compared to @PreAuthorize and @PostAuthorize

Kindly explain the above diagram with detailed explanation

The diagrams explain **Method Level Security** in Spring Security, which provides fine-grained access control at the method level. Here's a detailed breakdown:

What is Method Level Security?

- Traditionally, authorization rules are applied at API path or URL levels. However, **Method Level Security** allows you to apply these rules at the service or repository layer or directly to specific methods.
- This can be enabled using the `` `@EnableMethodSecurity` `` annotation in the Spring Security configuration class.

Key Benefits of Method Level Security:

1. It applies **authorization rules** even to non-web applications or layers where endpoints are not exposed.
2. It ensures that critical methods are protected regardless of how they are accessed.

Approaches in Method Level Security

1. **Invocation Authorization:**
   - Validates whether a specific user or role is allowed to invoke a method.
   - For example, ensuring only users with the `` `ADMIN` `` role can call a method.
2. **Filtering Authorization:**
   - Controls:
     - What input parameters a method can receive.
     - What data can be returned to the invoker after the method's execution.

Enabling Method Level Security

Spring Security leverages the **Aspect-Oriented Programming (AOP)** module and interceptors to enforce authorization at the method level.

Steps to Enable Method Level Security:

1. **Add `` `@EnableMethodSecurity` `` to the Security Configuration:**

```java
@Configuration
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
```

```
public class ProjectSecurityConfig {
    // Other configurations
}
```

- `` `prePostEnabled` ``: Enables `` `@PreAuthorize` `` and `` `@PostAuthorize` ``.
- `` `securedEnabled` ``: Enables `` `@Secured` ``.
- `` `jsr250Enabled` ``: Enables `` `@RolesAllowed` ``.

Annotations for Method Level Security

1. `` `@PreAuthorize` ``:
    - Used to check access before the method is invoked.
    - Example:

```java
@PreAuthorize("hasRole('ADMIN')")
public void adminMethod() {
    // Admin-specific logic
}
```

2. `` `@PostAuthorize` ``:
    - Used to validate the return value of a method after its execution.
    - Example:

```java
@PostAuthorize("returnObject.owner == authentication.name")
public Resource getResource(Long id) {
    // Return a resource
}
```

3. `` `@Secured` ``:

- Specifies roles allowed to access a method.
- Example:

  ```java
  @Secured("ROLE_USER")
  public void userMethod() {
      // User-specific logic
  }
  ```

4. **`@RolesAllowed`**:
   - Similar to **`@Secured`**, but part of the JSR-250 specification.
   - Example:

     ```java
     @RolesAllowed({"ROLE_ADMIN", "ROLE_USER"})
     public void multipleRolesMethod() {
         // Logic for multiple roles
     }
     ```

Configuration Example

- **Security Configuration:**

  ```java
  @Configuration
  @EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
  public class SecurityConfig {
      // Spring Security configuration
  }
  ```

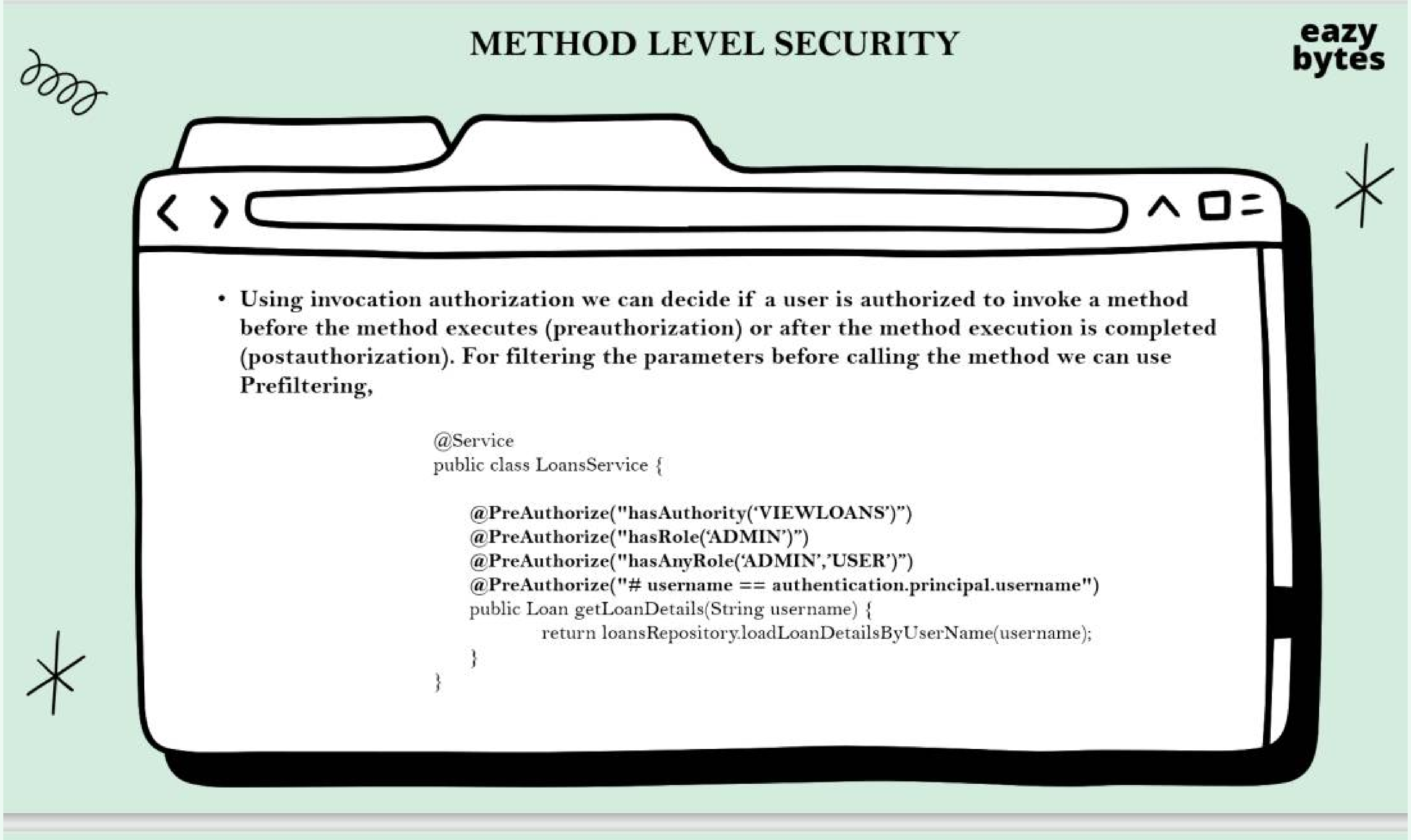
- **Service Layer with Method Security:**

```java
@Service
public class MyService {

    @PreAuthorize("hasRole('ADMIN')")
    public String adminOnlyMethod() {
        return "Admin Access Granted!";
    }

    @PostAuthorize("returnObject.owner == authentication.name")
    public Resource getResource(Long id) {
        // Business logic to fetch resource
        return new Resource("John", "Sample Data");
    }

    @Secured("ROLE_USER")
    public String userMethod() {
        return "User Access Granted!";
    }

    @RolesAllowed({"ROLE_ADMIN", "ROLE_USER"})
    public String commonAccessMethod() {
        return "Access Granted for Admin or User!";
    }
}
```

Summary

- **Purpose:** Method Level Security enables fine-grained control over who can access specific methods, beyond URL-based authorization.
- **Flexibility:** Works for both web and non-web applications.
- **Annotations:**
    - `@PreAuthorize` and `@PostAuthorize` are the most powerful.
    - `@Secured` and `@RolesAllowed` are simpler alternatives.
- **Use Case Example:**
    - Protect critical service or repository layer methods.
    - Ensure only specific roles or users can execute sensitive business logic.

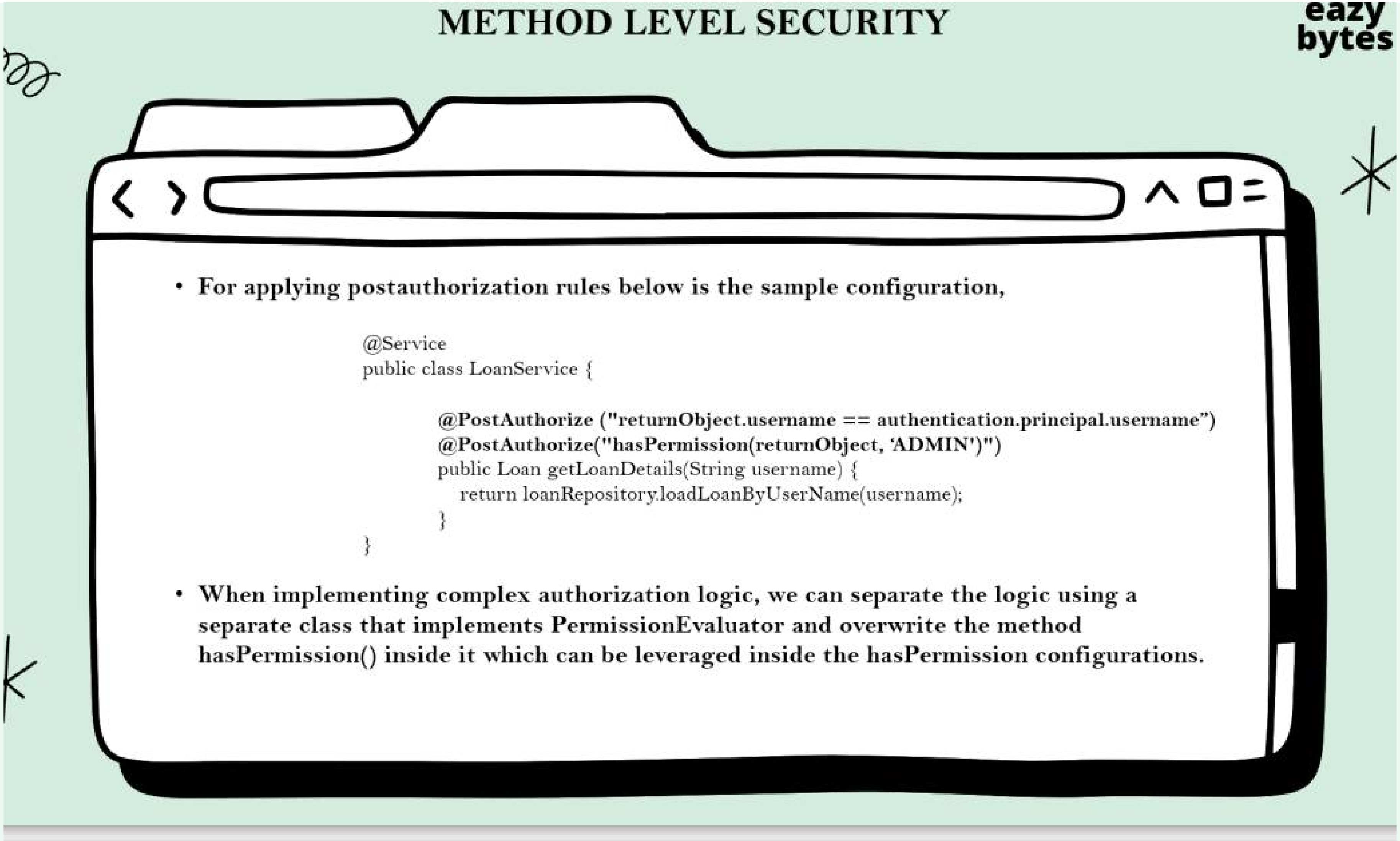
Would you like help implementing this in your project or customizing it further?



METHOD LEVEL SECURITY

- Using invocation authorization we can decide if a user is authorized to invoke a method before the method executes (preauthorization) or after the method execution is completed (postauthorization). For filtering the parameters before calling the method we can use Prefiltering,

```
@Service
public class LoansService {

    @PreAuthorize("hasAuthority('VIEWLOANS')")
    @PreAuthorize("hasRole('ADMIN')")
    @PreAuthorize("hasAnyRole('ADMIN','USER')")
    @PreAuthorize("# username == authentication.principal.username")
    public Loan getLoanDetails(String username) {
            return loansRepository.loadLoanDetailsByUserName(username);
    }
}
```



METHOD LEVEL SECURITY

- For applying postauthorization rules below is the sample configuration,

```
@Service
public class LoanService {

        @PostAuthorize ("returnObject.username == authentication.principal.username”)
        @PostAuthorize("hasPermission(returnObject, ‘ADMIN')")
        public Loan getLoanDetails(String username) {
           return loanRepository.loadLoanByUserName(username);
        }
}
```

- When implementing complex authorization logic, we can separate the logic using a separate class that implements PermissionEvaluator and overwrite the method hasPermission() inside it which can be leveraged inside the hasPermission configurations.

Kindly explain the above image with detailed explanation

The images elaborate on **Method Level Security** in Spring Security, highlighting **pre-authorization** (`@PreAuthorize`) and **post-authorization** (`@PostAuthorize`) rules. Here's a detailed breakdown:

First Image: Pre-Authorization

Concept of Pre-Authorization

- **Pre-authorization** checks if the user is allowed to execute a method **before** the method is invoked.
- These checks are specified using the `` `@PreAuthorize` `` annotation and are based on roles, authorities, or other conditions.

Code Explanation

```java
@Service
public class LoansService {

    @PreAuthorize("hasAuthority('VIEWLOANS')")
    @PreAuthorize("hasRole('ADMIN')")
    @PreAuthorize("hasAnyRole('ADMIN', 'USER')")
    @PreAuthorize("#username == authentication.principal.username")
    public Loan getLoanDetails(String username) {
        return loansRepository.loadLoanDetailsByUserName(username);
    }
}
```

1. **Annotations Used:**
    - `` `@PreAuthorize("hasAuthority('VIEWLOANS')")` ``: Ensures the user has the specific authority `` `VIEWLOANS` ``.
    - `` `@PreAuthorize("hasRole('ADMIN')")` ``: Ensures the user has the role `` `ADMIN` ``.
    - `` `@PreAuthorize("hasAnyRole('ADMIN', 'USER')")` ``: Ensures the user has either `` `ADMIN` `` or `` `USER` `` roles.
    - `` `@PreAuthorize("#username == authentication.principal.username")` ``: Ensures the username passed as a parameter matches the authenticated user's username.
2. **How It Works:**
    - When the `` `getLoanDetails` `` method is called:
        - The framework checks the specified pre-authorization rules.
        - If the user does not meet the conditions, the request is denied, and the method is not executed.

Second Image: Post-Authorization

Concept of Post-Authorization

- **Post-authorization** evaluates conditions **after** a method has been executed.
- These checks are performed on the returned object or result of the method.

Code Explanation

```java
@Service
public class LoanService {

    @PostAuthorize("returnObject.username == authentication.principal.username")
    @PostAuthorize("hasPermission(returnObject, 'ADMIN')")
    public Loan getLoanDetails(String username) {
        return loanRepository.loadLoanByUserName(username);
    }
}
```

1. **Annotations Used:**
    - `@PostAuthorize("returnObject.username == authentication.principal.username")`: Ensures the `username` field of the returned object matches the authenticated user's username.
    - `@PostAuthorize("hasPermission(returnObject, 'ADMIN')")`: Checks if the authenticated user has `ADMIN` permission for the returned object.
2. **How It Works:**
    - When the `getLoanDetails` method is executed:
        - The framework evaluates the returned object (`returnObject`) against the specified post-authorization rules.
        - If the user does not meet the conditions, the response is denied or filtered.

Key Differences Between Pre and Post-Authorization

1. **Timing:**
    - **Pre-authorization (`@PreAuthorize`):** Rules are applied before method execution.

    - **Post-authorization (`@PostAuthorize`):** Rules are applied after method execution.
2. **Focus:**
    - **Pre-authorization:** Focuses on input parameters and context (e.g., roles, authorities).
    - **Post-authorization:** Focuses on the output or result of the method.

Complex Authorization Logic

- For advanced use cases, a **custom `PermissionEvaluator`** can be implemented.
- **PermissionEvaluator** provides a mechanism to evaluate complex permission logic.

Example:

```java
@Service
public class CustomPermissionEvaluator implements PermissionEvaluator {

    @Override
    public boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission) {
        // Custom logic to validate permissions
        return true; // Allow or deny based on logic
    }

    @Override
    public boolean hasPermission(Authentication authentication, Serializable targetId, String targetType, Object permission) {
        // Additional logic if needed
        return true;
    }
}
```

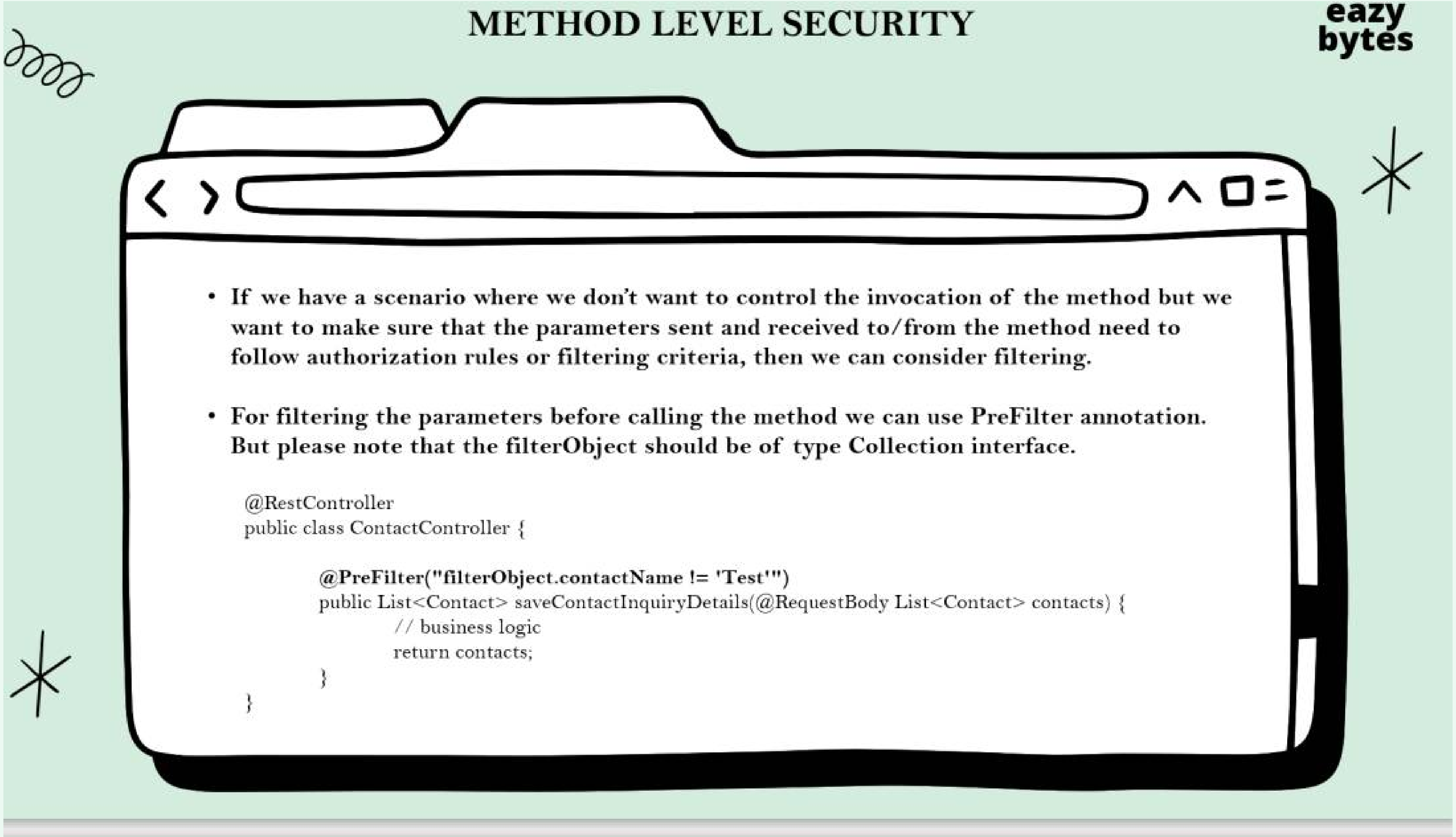
- You can integrate `hasPermission()` in `@PostAuthorize` annotations for fine-grained control.

Summary

- **Pre-authorization (`@PreAuthorize`):** Validates user access before the method runs.
- **Post-authorization (`@PostAuthorize`):** Validates user access based on the method's returned object.

- **Key Use Cases:**
    - Protect sensitive business logic (``@PreAuthorize``).
    - Restrict the visibility or access to returned data (``@PostAuthorize``).
- **Advanced Customization:** Use ``PermissionEvaluator`` for complex authorization rules.

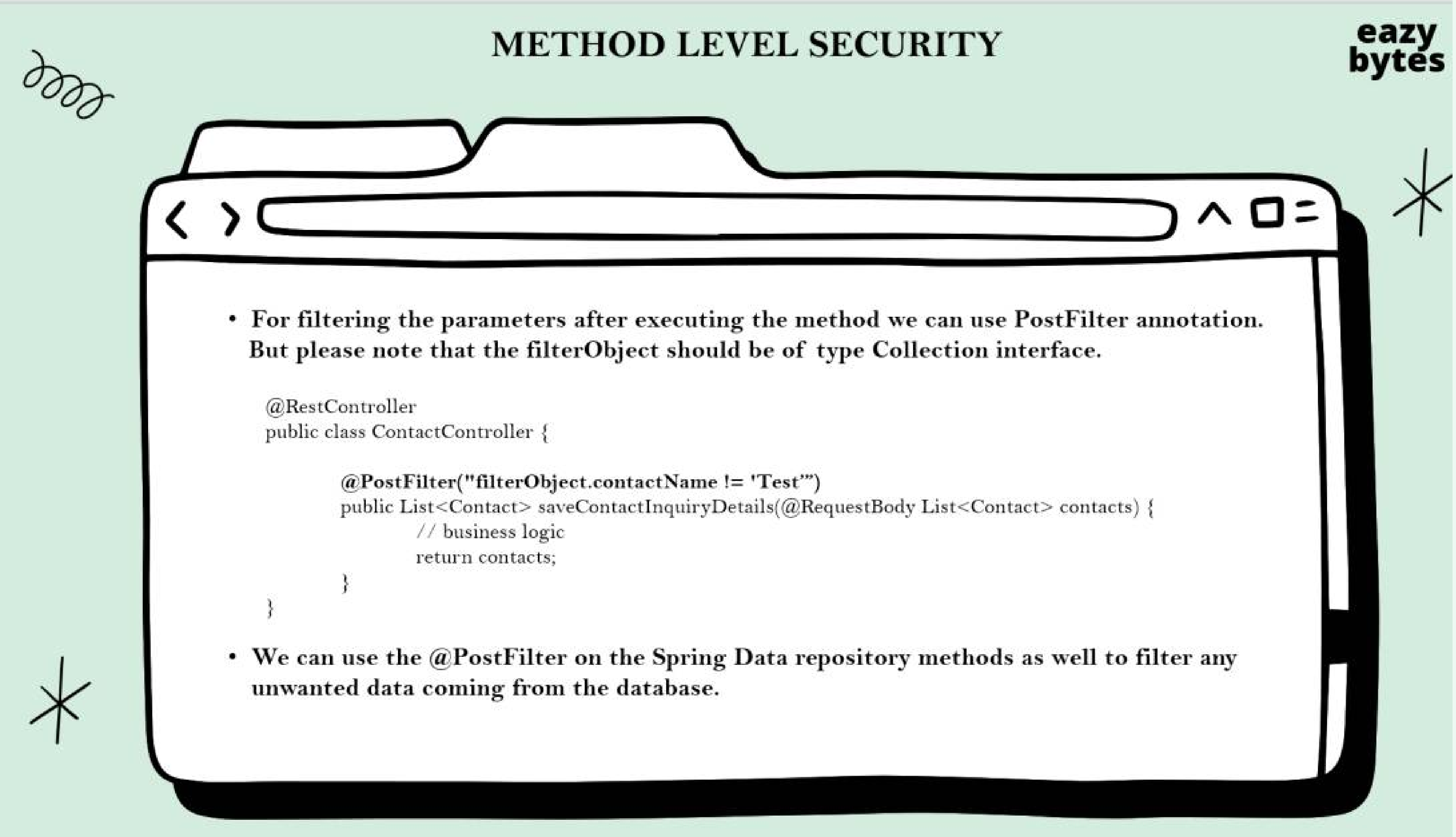
Would you like help implementing a specific aspect of this in your project?



METHOD LEVEL SECURITY

- If we have a scenario where we don't want to control the invocation of the method but we want to make sure that the parameters sent and received to/from the method need to follow authorization rules or filtering criteria, then we can consider filtering.
- For filtering the parameters before calling the method we can use PreFilter annotation. But please note that the filterObject should be of type Collection interface.

```
@RestController
public class ContactController {

        @PreFilter("filterObject.contactName != 'Test'")
        public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact> contacts) {
                // business logic
                return contacts;
        }
}
```



METHOD LEVEL SECURITY

- For filtering the parameters after executing the method we can use PostFilter annotation. But please note that the filterObject should be of type Collection interface.

```
@RestController
public class ContactController {

        @PostFilter("filterObject.contactName != 'Test'")
        public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact> contacts) {
                // business logic
                return contacts;
        }
}
```

- We can use the @PostFilter on the Spring Data repository methods as well to filter any unwanted data coming from the database.

Kindly explain the above image with detailed explanation with example

The images explain **filtering annotations in Method Level Security** in Spring Security, specifically focusing on `@PreFilter` and `@PostFilter`. These annotations are used to filter input parameters and returned results, respectively.

1. Pre-Filtering with `@PreFilter`

What is Pre-Filtering?

- **Pre-filtering** allows us to filter or validate the input parameters **before** the method is executed.
- It ensures that only valid or authorized data is passed to the method for processing.

Code Example

```java
@RestController
public class ContactController {

    @PreFilter("filterObject.contactName != 'Test'")
    public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact> contacts) {
        // Business logic to save contact details
        return contacts;
    }
}
```

1. **Explanation:**
    - The method accepts a list of `Contact` objects as input.
    - The `@PreFilter` annotation filters out any `Contact` object where the `contactName` equals `"Test"`.
    - Only the filtered list is processed by the method.
2. **Key Points:**
    - `filterObject` refers to each individual element in the collection (in this case, each `Contact` object).
    - The method parameter being filtered must implement the `Collection` **interface** (e.g., `List`, `Set`).
3. **Use Case:**
    - Ensure that unauthorized or invalid data is excluded before processing (e.g., blocking users or invalid entries).

---

2. Post-Filtering with `@PostFilter`

What is Post-Filtering?

- **Post-filtering** allows us to filter or validate the data **after** the method has executed and returned a result.
- It ensures that only authorized or relevant data is sent back to the client.

Code Example

```java
@RestController
public class ContactController {

    @PostFilter("filterObject.contactName != 'Test'")
    public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact> contacts) {
        // Business logic to save contact details
        return contacts;
    }
}
```

1. **Explanation:**
    - The method processes and returns a list of `` `Contact` `` objects.
    - The `` `@PostFilter` `` annotation filters out any `` `Contact` `` object where the `` `contactName` `` equals `` `"Test"` ``.
    - Only the filtered list is returned to the client.
2. **Key Points:**
    - `` `filterObject` `` refers to each individual element in the returned collection (in this case, each `` `Contact` `` object).
    - The returned value must implement the `` `Collection` `` **interface** (e.g., `` `List` ``, `` `Set` ``).
3. **Use Case:**
    - Ensure sensitive or unauthorized data is excluded from the response (e.g., hiding sensitive details).

Differences Between `` `@PreFilter` `` and `` `@PostFilter` ``

Feature	`` `@PreFilter` ``	`` `@PostFilter` ``
When Applied	Before the method is executed	After the method is executed

Feature	`@PreFilter`	`@PostFilter`
Scope	Filters input parameters	Filters the returned result
FilterObject	Refers to each element in the input collection	Refers to each element in the output collection
Primary Use Case	Validate or sanitize input data	Remove unauthorized data from the response

---

Use Cases for Pre and Post Filtering

1. **Pre-Filtering (`@PreFilter`):**
    - Filtering user input to prevent processing invalid data.
    - Example: Removing items from an order that a user is not allowed to purchase.
2. **Post-Filtering (`@PostFilter`):**
    - Restricting returned data to match user permissions or roles.
    - Example: Hiding confidential information from non-admin users.

Considerations

1. **Collection Type:**
    - Both `@PreFilter` and `@PostFilter` work only with parameters and return types that implement the `Collection` interface (e.g., `List`, `Set`).
2. **Performance:**
    - Excessive filtering logic in annotations can impact performance. Use them judiciously and delegate complex logic to services.
3. **Security and Logic Separation:**
    - Avoid embedding business logic in filtering annotations. Keep security and business logic separate for maintainability.

Summary

- `@PreFilter` ensures input data adheres to rules before processing.
- `@PostFilter` ensures that the output data meets authorization criteria before returning it to the client.
- Both are powerful tools for maintaining **fine-grained control** over method parameters and results.

Would you like a practical implementation or example involving these filters in a specific application?