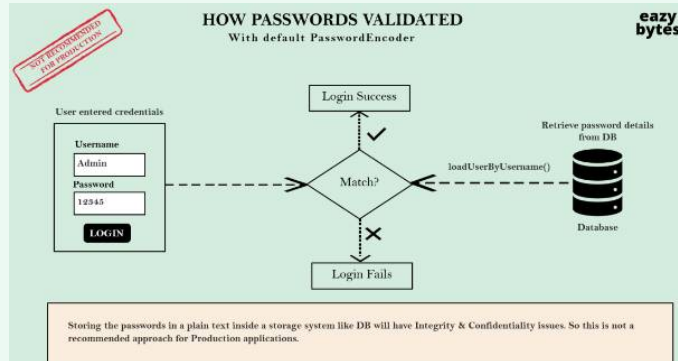


# 4-Password Validation and Security



Kindly explain the above image in details



The image explains how passwords are validated using the default `PasswordEncoder` in applications, and highlights the risks of storing passwords in plain text. Here's a detailed explanation of the process illustrated in the diagram:

## 1. User Entered Credentials:

- The user provides their **username** (e.g., `Admin`) and **password** (e.g., `12345`) on the login screen.
- These credentials are submitted via a "LOGIN" button.

## 2. Password Validation Process:

- The application retrieves the user details (including the stored password) from the **Database** using the method `loadUserByUsername(username)`.
- The **retrieved password** from the database is compared with the **password entered by the user**.

### 3. Decision Point (Match or Not):

- A decision is made at a comparison stage:
    - **If the passwords match:** Login is successful, and the user is authenticated.
    - **If the passwords do not match:** Login fails, and access is denied.
- 

### 4. Risks Highlighted:

- **Plain Text Password Storage:**

The image emphasizes that storing passwords in plain text in a database poses serious **integrity and confidentiality issues**. It is marked as "NOT RECOMMENDED FOR PRODUCTION".

- Plain text passwords are vulnerable to theft or exposure if the database is compromised.
  - This practice violates industry security standards (e.g., GDPR, PCI DSS).
- 

### Important Takeaways:

- Using plain text password storage is insecure and should be avoided in production systems.
  - Best practices include:
    - Storing passwords as **hashed values** using secure algorithms (e.g., BCrypt, Argon2).
    - Using a robust `PasswordEncoder` to hash and verify passwords securely.
- 

This diagram serves as a cautionary example, urging developers to avoid insecure practices and adopt secure password handling mechanisms in applications.



Encoding Vs Encryption Vs Hashing		
Different ways of Pwd management		
Encoding	Encryption	Hashing
<ul style="list-style-type: none"> <li>✓ Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.</li> <li>✓ It involves no secret and completely reversible.</li> <li>✓ Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding. Ex: ASCII, BASE64, UNICODE.</li> </ul>	<ul style="list-style-type: none"> <li>✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.</li> <li>✓ To achieve confidentiality, encryption requires the use of a secret which, in cryptographic terms, we call a "key".</li> <li>✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.</li> </ul>	<ul style="list-style-type: none"> <li>✓ In hashing, data is converted to the hash value using some hashing function.</li> <li>✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.</li> <li>✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data.</li> </ul>

Kindly explain the above image in details



The image compares **Encoding**, **Encryption**, and **Hashing** as different methods of password and data management, explaining their characteristics, purposes, and limitations. Here's a detailed breakdown:

## 1. Encoding

- **Definition:**  
Encoding is the process of converting data from one form to another for easier transmission or storage. It is not related to cryptography or security.
- **Key Characteristics:**
  - No secret is involved in encoding, and it is **completely reversible**.
  - Encoding **cannot be used for securing data**, as it is not meant for confidentiality.
- **Examples:**
  - Common encoding schemes include **ASCII**, **BASE64**, and **UNICODE**.

## 2. Encryption

- **Definition:**  
Encryption is the process of transforming data to ensure **confidentiality**. This is achieved by making the data readable only to those who have a secret

key.

- **Key Characteristics:**

- Requires a **secret key** to encrypt and decrypt the data.
- Encryption is **reversible**, meaning encrypted data can be transformed back to its original form if the key is available.
- As long as the key remains confidential, encryption provides strong data protection.

### 3. Hashing

- **Definition:**

Hashing converts data into a fixed-length hash value using a **hashing function**. It is primarily used for data integrity and secure storage of passwords.

- **Key Characteristics:**

- Hashing is **non-reversible**, meaning the original data **cannot be retrieved** from the hash value.
- Hashing allows verification: Given a piece of data, you can check if its hash matches a stored hash without needing the original data.
- This property makes it ideal for password storage and data integrity verification.

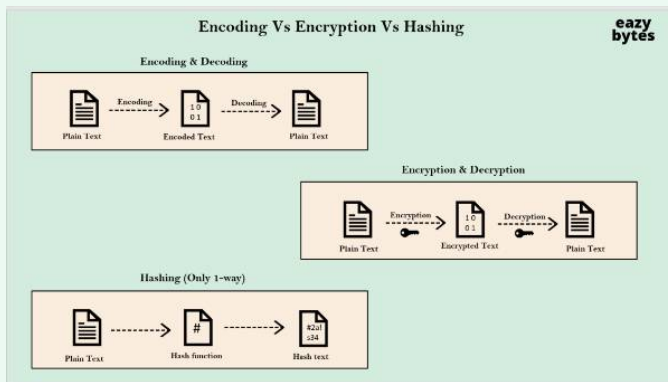
### Comparison Highlights:

Aspect	Encoding	Encryption	Hashing
Purpose	Format conversion	Confidentiality	Integrity and verification
Reversible	Yes	Yes (with a key)	No
Use of Secret	No	Yes (requires a key)	No
Example Algorithms	ASCII, BASE64, UNICODE	AES, RSA, DES	MD5, SHA-256, bcrypt

### Key Takeaways:

1. **Encoding** is not for security; it's for data transmission or storage.
2. **Encryption** ensures confidentiality and requires key management.
3. **Hashing** secures data like passwords by storing them in a way that cannot be reversed but can be verified.

This comparison helps in understanding the appropriate usage of each method based on the context and requirements.



Kindly explain the above image in details



The image provides a visual representation of the differences between **Encoding**, **Encryption**, and **Hashing**, focusing on their processes and characteristics. Here's a detailed explanation of each section:

## 1. Encoding & Decoding

- **Process:**
  - Data in **plain text** is converted into another format (e.g., binary, base64) using an encoding scheme. This process is **Encoding**.
  - The encoded data can be converted back to its original form using **Decoding**.
- **Key Points:**
  - Encoding is **reversible**; anyone with knowledge of the encoding scheme can decode the data.
  - It is not intended for security but for ensuring compatibility (e.g., transmitting data over a network).
- **Example:**
  - Plain text → Encoded text (e.g., Base64) → Decoded back to Plain text.

## 2. Encryption & Decryption

- **Process:**

- **Encryption** converts plain text into an unreadable format (encrypted text) using a **key**.
- **Decryption** reverses this process to retrieve the original plain text using the same key (symmetric encryption) or a different key (asymmetric encryption).

- **Key Points:**

- Encryption is **reversible** with the correct key.
- It ensures **confidentiality** by making data unreadable to unauthorized users.
- Commonly used for protecting sensitive information (e.g., emails, bank transactions).

- **Example:**

- Plain text → Encrypted text (using a key) → Decrypted back to Plain text (using the same or another key).

### 3. Hashing (Only One-Way)

- **Process:**

- Plain text is transformed into a fixed-length string (hash) using a **hashing function**.
- The result is called a **hash value**, which cannot be converted back to the original text.

- **Key Points:**

- Hashing is **non-reversible** (one-way process).
- It ensures data integrity and is widely used for storing passwords and verifying data authenticity.
- Hashing functions (e.g., SHA-256, bcrypt) are designed to produce unique hash values for unique inputs.
- Even a slight change in the input results in a completely different hash (a property called the "avalanche effect").

- **Example:**

- Plain text → Hash function → Hash text (e.g., ``#2a!s34``).

### Comparison

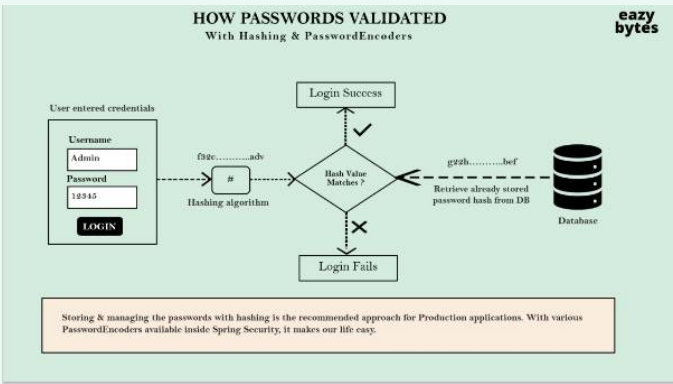
Aspect	Encoding	Encryption	Hashing
Reversible	Yes	Yes (with key)	No
Security Purpose	No	Yes	Yes (for verification)
Key/Secret Required	No	Yes	No

Aspect	Encoding	Encryption	Hashing
Use Case	Data compatibility	Data confidentiality	Data integrity, password storage

Conclusion

This diagram clarifies that:

- **Encoding** is for representation, not security.
- **Encryption** ensures confidentiality but requires secure key management.
- **Hashing** is ideal for data verification and is irreversible, making it suitable for password storage and checksums.



Kindly explain the above image in details



The image demonstrates how passwords are validated securely using **Hashing** and **PasswordEncoders**, emphasizing its use in production environments. Below is a detailed explanation of the process outlined in the diagram:

1. User Enters Credentials

- The user inputs their **username** (e.g., `Admin`) and **password** (e.g., `12345`) on the login form.

- These credentials are sent to the application for validation after clicking the **LOGIN** button.



## 2. Password Hashing

- Once the user enters their password:
  - The password is passed through a **Hashing Algorithm** to generate a hash value.
  - For example, the plain-text password ``12345`` is hashed into a string like ``f32c.....adv``.

## 3. Database Interaction

- The application retrieves the **already stored password hash** from the database for the corresponding username (e.g., ``g22h.....bef``).

## 4. Hash Value Comparison

- The system compares:
  - The **hash of the user-entered password** (calculated in step 2).
  - The **hash stored in the database** (retrieved in step 3).
- **Possible Outcomes:**
  - **If the hash values match:** The password is valid, and the user is authenticated. (Login Success )
  - **If the hash values do not match:** The password is incorrect, and the authentication fails. (Login Fails )

## Key Features of This Approach

### 1. Secure Password Storage:

- Only the hashed password is stored in the database, not the plain text.
- Even if the database is compromised, the original passwords cannot be derived from the hash.

### 2. One-Way Hashing:

- Hashing is irreversible, meaning the original password cannot be reconstructed from the hash.
- Popular hashing algorithms include **bcrypt**, **PBKDF2**, and **SHA-256** (bcrypt is highly recommended).

### 3. Salting (Not Explicitly Shown):

- Best practices involve adding a random salt to the password before hashing, ensuring unique hash values even for identical passwords.



## Advantages Highlighted

- The diagram emphasizes that **storing and managing passwords with hashing** is the **recommended approach** for production applications.
- Using tools like **PasswordEncoders** available in frameworks like **Spring Security** simplifies the implementation of secure password hashing.

## Conclusion

This diagram illustrates a secure password validation flow where:

1. Plain-text passwords are hashed before comparison.
2. Hashing ensures that passwords are never stored in their original form, reducing security risks.
3. PasswordEncoders and hashing algorithms make this process secure and production-ready.



The diagram, titled "DETAILS OF PASSWORDENCODER" and attributed to "eazy bytes", provides a comprehensive overview of password encoding in Spring Security. It is divided into two main sections:

- Methods inside PasswordEncoder Interface:** This section displays the Java code for the `PasswordEncoder` interface. The methods include `encode(CharSequence rawPassword)` for hashing, `matches(CharSequence rawPassword, String encodedPassword)` for validation, and `defaultPasswordEncoderEncoding(CharSequence rawPassword)` for returning the encoded password.
- Different implementations of PasswordEncoder inside Spring Security:** This section lists six implementations, each preceded by a star icon:
  - `NoOpPasswordEncoder` (Not recommended for Prod apps)
  - `StandardPasswordEncoder` (Not recommended for Prod apps)
  - `PbkdfPasswordEncoder`
  - `BCryptPasswordEncoder`
  - `SCryptPasswordEncoder`
  - `Argon2PasswordEncoder`

Kindly explain the above image in details



The image provides an overview of the **PasswordEncoder** interface in Spring Security, its methods, and various implementations. Below is a detailed explanation:

## 1. Methods in the `PasswordEncoder` Interface

The **`PasswordEncoder`** interface is designed to handle password encoding and verification in a secure manner. It defines the following methods:

a. `String encode(CharSequence rawPassword);`

- **Purpose:** Encodes the raw password (plain text) into a hashed format.
- **Usage:** Used when storing passwords in the database after securely encoding them.
- **Example:**

```
java

String encodedPassword = passwordEncoder.encode("mypassword");
```

b. `boolean matches(CharSequence rawPassword, String encodedPassword);`

- **Purpose:** Checks if the raw password matches the previously encoded password.
- **Usage:** Compares a user's entered password (raw) with the hashed password stored in the database.
- **Example:**

```
java

boolean isPasswordMatch = passwordEncoder.matches("mypassword", encodedPassword);
```

c. `default boolean upgradeEncoding(String encodedPassword);`

- **Purpose:** Determines whether the current encoding of a password should be upgraded.
- **Default Behavior:** Returns `false`, meaning no upgrade is required unless overridden.
- **Usage:** Helps in scenarios where you want to switch to a stronger hashing algorithm over time.

---

## 2. Implementations of `PasswordEncoder` in Spring Security

Spring Security provides several implementations of the `PasswordEncoder` interface. Here's an overview:

### Not Recommended for Production

#### 1. `NoOpPasswordEncoder`:

- Encodes passwords as plain text without hashing.
- **Why Not Recommended:** Storing passwords as plain text is a serious security risk.

#### 2. `StandardPasswordEncoder`:

- Uses SHA-256 hashing.
- **Why Not Recommended:** Lacks additional security features like salting or adaptive hashing.

### Recommended for Production

#### 1. `BCryptPasswordEncoder`:

- Uses the BCrypt hashing algorithm.
- **Features:**
  - Strong adaptive hashing.
  - Built-in salt to ensure unique hashes for the same password.
- **Recommended Use:** Default choice for secure password encoding.

#### 2. `SCryptPasswordEncoder`:

- Uses the SCrypt hashing algorithm.
- **Features:**
  - Memory-intensive computation, making brute-force attacks slower.
- **Recommended Use:** For applications requiring high levels of computational security.

#### 3. `Pbkdf2PasswordEncoder`:

- Uses the PBKDF2 (Password-Based Key Derivation Function 2) algorithm.
- **Features:**
  - Adjustable iteration count for computational difficulty.
- **Recommended Use:** For password hashing with customizable security levels.

#### 4. `Argon2PasswordEncoder`:

- Uses the Argon2 hashing algorithm.
- **Features:**

- Memory, time, and parallelism-hard hashing.
  - Winner of the Password Hashing Competition (PHC).
  - **Recommended Use:** Modern and highly secure hashing solution.
- 

## Key Takeaways

- The **PasswordEncoder** interface is essential for securely encoding and validating passwords in Spring Security.
- **Recommended Implementations:** Use `BCryptPasswordEncoder`, `SCryptPasswordEncoder`, `Pbkdf2PasswordEncoder`, or `Argon2PasswordEncoder` for production environments.
- **Avoid:** `NoOpPasswordEncoder` and `StandardPasswordEncoder` in production, as they do not provide adequate security.

By leveraging these implementations, developers can ensure secure password storage and validation in their applications.