用户名/密码验证 Username/Password Authentication

验证用户身份的最常见方法之一是验证用户名和密码。因此，Spring Security 为使用用户名和密码进行身份验证提供了全面的支持。

- **读取用户名和密码 Reading the Username & Password**

  Spring Security 提供了以下内置机制，用于从 HttpServletRequest 读取用户名和密码

  - 表单登录 [Form Login]
  - 基本认证 [Basic Authentication]
  - 摘要式身份验证 [Digest Authentication]

- **存储机制 Storage Mechanisms**

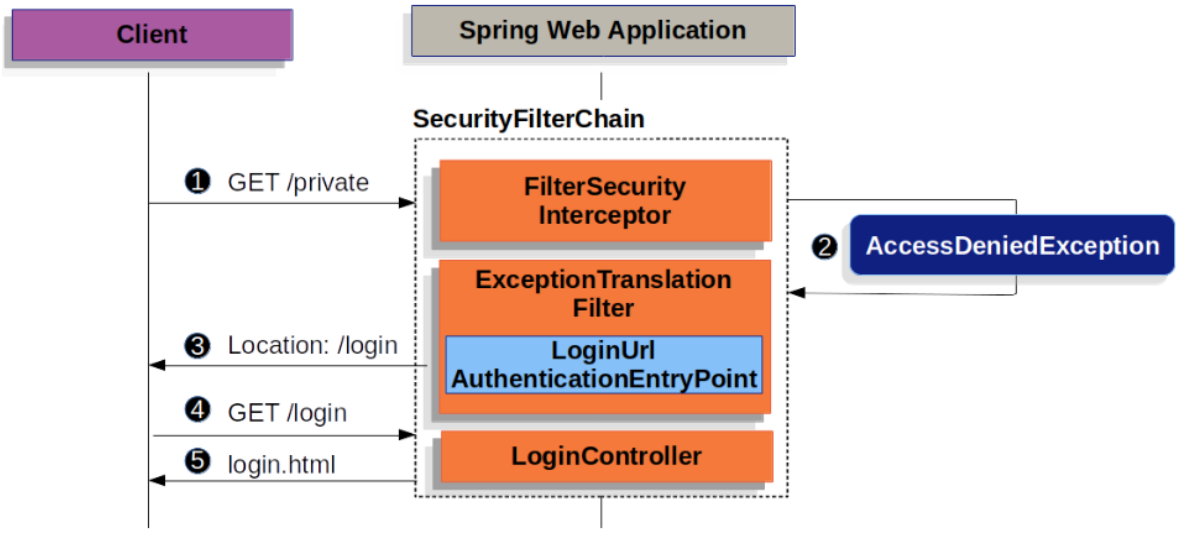
  每个支持的读取用户名和密码的机制

  - 内存中身份验证的简单存储 [In-Memory Authentication]
  - JDBC 身份验证的关系数据库 [JDBC Authentication]
  - 使用 UserDetailsService 自定义数据存储 [UserDetailsService]
  - 具有 LDAP 身份验证的 LDAP 存储 [LDAP Authentication]

- 折叠列表

  点击创建内容

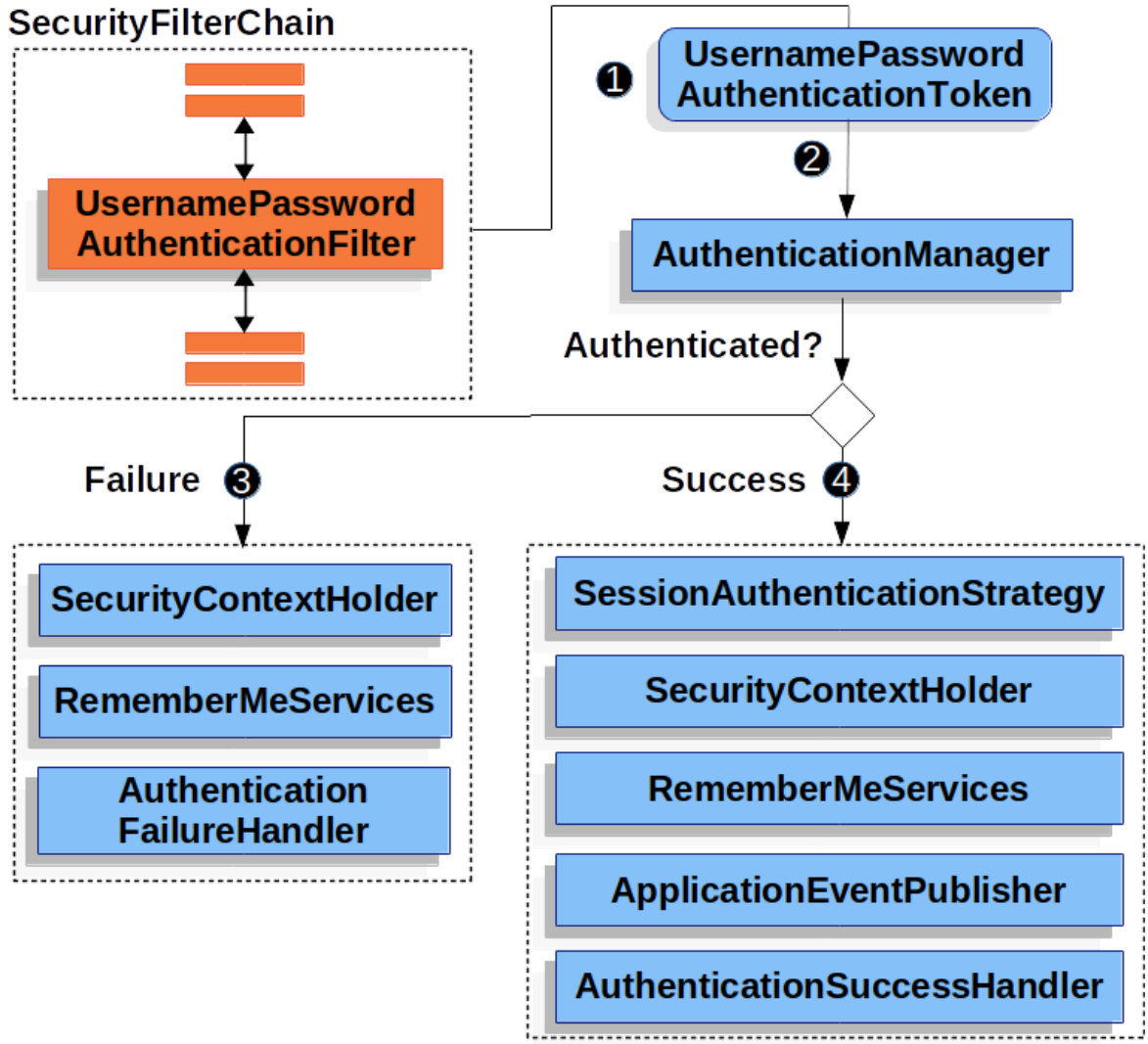
2.1.1 表单登录 [Form Login]



构建了 SecurityFilterChain

1. 首先，用户向未授权的资源 /private 发出未经身份验证的请求
2. Spring Security 的 FilterSecurityInterceptor 表示通过抛出 AccessDeniedException 拒绝未经身份验证的请求
3. 由于用户未通过身份验证，因此 ExceptionTranslationFilter 启动 Start Authentication 并使用配置的 AuthenticationEntryPoint 将重定向发送到登录页面。在大多数情况下，AuthenticationEntryPoint 是 LoginUrlAuthenticationEntryPoint 的一个实例(在前后端分离的开发中通常是返回用户未登录的信息)
4. 然后浏览器将请求它被重定向到的登录页面。
5. 应用程序中的里面必须呈现登录页面

提交用户名和密码后，UsernamePasswordAuthenticationFilter 会验证用户名和密码。 UsernamePasswordAuthenticationFilter 扩展了 AbstractAuthenticationProcessingFilter，因此该图看起来与 AbstractAuthenticationProcessingFilter 流程非常相似。



- 该图是构建了我们的 SecurityFilterChain 图解

1. 当用户提交他们的用户名和密码时，UsernamePasswordAuthenticationFilter 通过从 HttpServletRequest 中提取用户名和密码来创建一个 UsernamePasswordAuthenticationToken 这是一种身份验证
2. 接下来，将 UsernamePasswordAuthenticationToken 传递给 AuthenticationManager 进行身份验证。 AuthenticationManager 的详细信息取决于用户信息的存储方式
3. 如果认证失败
    a. SecurityContextHolder 被清除
    b. 调用 RememberMeServices.loginFail()。如果记住我没有配置，这是一个空操作
    c. AuthenticationFailureHandler 被调用
4. 如果认证成功
    a. SessionAuthenticationStrategy 收到新登录通知
    b. Authentication 在 SecurityContextHolder 上设置
    c. RememberMeServices.loginSuccess。如果记住我没有配置，这是一个空操作
    d. ApplicationEventPublisher 发布一个 InteractiveAuthenticationSuccessEvent
    e. AuthenticationSuccessHandler 被调用。通常这是一个 SimpleUrlAuthenticationSuccessHandler，当我们重定向到登录页面时，它将重定向到 ExceptionTranslationFilter 保存的请求

Java

```java
/**
 * 默认情况下启用 Spring Security 表单登录。
 * 但是，一旦提供了任何基于 servlet 的配置，就必须明确提供基于表单的登录。
 * 可以在下面找到最小的显式 Java 配置
 */
protected void configure(HttpSecurity http) {
    http
        // ...
        .formLogin(withDefaults());
}
```

在此配置中，Spring Security 将呈现默认登录页面。大多数生产应用程序都需要自定义登录表单。

下面的配置演示了如何提供自定义登录表单

Java

```java
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .formLogin(form -> form
            .loginPage("/login")
            .permitAll()
        );
}
```

HTML

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.
    <head>
        <title>Please Log In</title>
    </head>
    <body>
        <h1>Please Log In</h1>
        <div th:if="${param.error}">
            Invalid username and password.</div>
        <div th:if="${param.logout}">
            You have been logged out.</div>
        <form th:action="@{/login}" method="post">
            <div>
            <input type="text" name="username" placeholder="Username"/>
            </div>
            <div>
            <input type="password" name="password" placeholder="Password"/>
            </div>
            <input type="submit" value="Log in" />
        </form>
    </body>
</html>
```

- 关于默认 HTML 表单有几个关键点

> 表单应该发送一个 Post 请求到 /login
>
> 该表格需要包含一个由 Thymeleaf 生产的 自动包含的 CSRF 令牌。
>
> 表单应在名为 username 的参数中指定用户名
>
> 表单应在名为 password 的参数中指定密码
>
> 如果发现 HTTP 参数错误，说明用户未能提供有效的用户名/密码
>
> 如果找到 http 参数 logout，说明用户已经注销成功

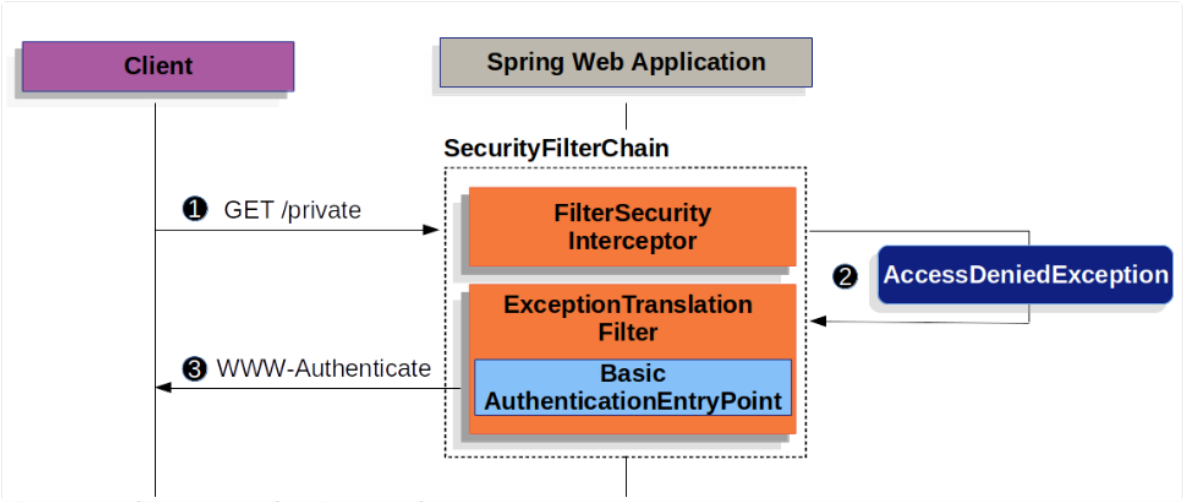
许多用户只需要自定义登录页面即可。但是，如果需要，上述所有内容都可以通过其他配置进行自定义

如果您使用的是 Spring MVC，您将需要一个将 GET /login 映射到我们创建的登录模板的控制器。

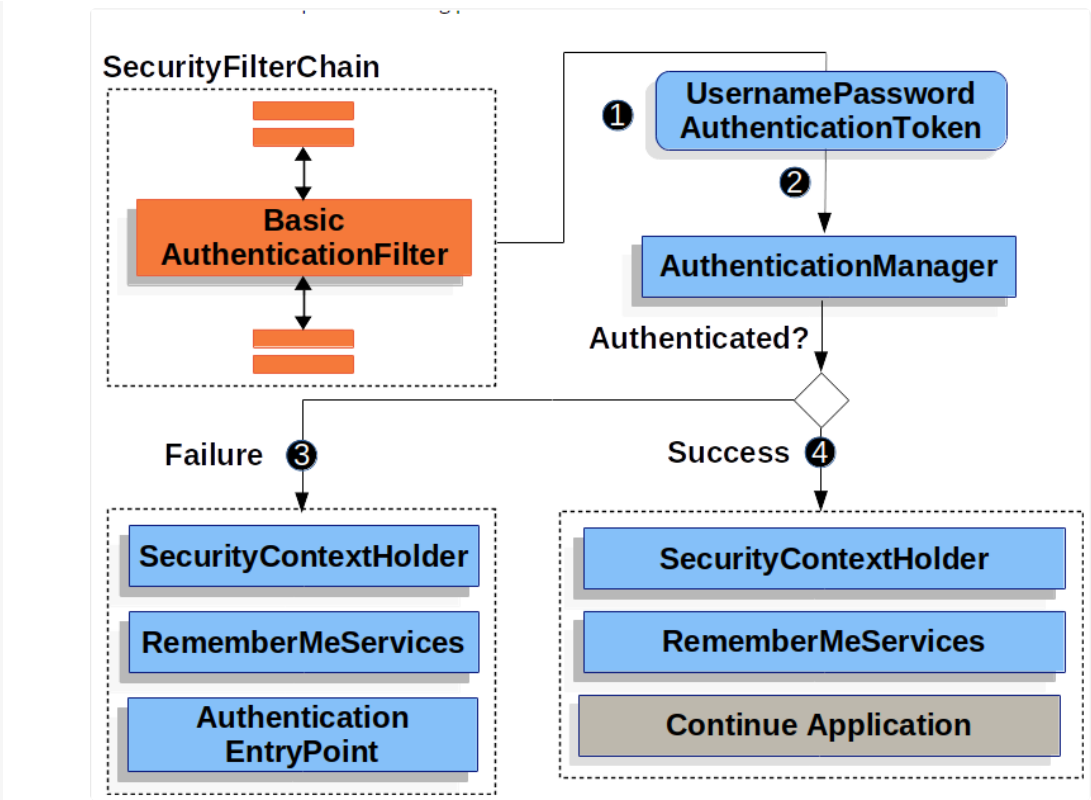
最小示例 LoginController:

```java
@Controller
class LoginController {
    @GetMapping("/login")
    String login() {
        return "login";
    }
}
```

2.1.2 基本认证 [Basic Authentication]



1. 首先，用户向未授权的资源/private 发出未经身份验证的请求。
2. Spring Security 的 FilterSecurityInterceptor 指示通过抛出 AccessDeniedException 拒绝未经身份验证的请求
3. 由于用户未通过身份验证，因此 ExceptionTranslationFilter 会启动 Start Authentication。配置的 AuthenticationEntryPoint 是 BasicAuthenticationEntryPoint 的一个实例，它发送一个 WWW-Authenticate 标头。 RequestCache 通常是一个不保存请求的 NullRequestCache，因为客户端能够重放它最初请求的请求
4. 当客户端收到 WWW-Authenticate 标头时，它知道应该使用用户名和密码重试。以下是处理用户名和密码的流程。



Spring Security 的 HTTP 基本身份验证支持默认启用。但是，一旦提供了任何基于 servlet 的配置，就必须显式提供 HTTP Basic, 如下给出最简配置

```java
protected void configure(HttpSecurity http) {
    http
        // ...
        .httpBasic(withDefaults());
}
```

2.1.3 摘要式身份验证 [Digest Authentication]

您不应该在现代应用程序中使用摘要式身份验证，因为它不被认为是安全的。最明显的问题是您必须以明文、加密或 MD5 格式存储密码。所有这些存储格式都被认为是不安全的。相反，您应该使用 Digest Authentication 不支持的单向自适应密码哈希（即 bCrypt、PBKDF2、SCrypt 等）存储凭据

摘要式身份验证尝试解决基本身份验证的许多弱点，特别是通过确保绝不会通过网络以明文形式发送凭据。许多浏览器支持摘要式身份验证。

2.1.4 内存中身份验证的简单存储 [In-Memory Authentication]

> Spring Security 的 InMemoryUserDetailsManager 实现 UserDetailsService 以支持在内存中检索的基于用户名/密码的身份验证。 InMemoryUserDetailsManager 通过实现 UserDetailsManager 接口提供对 UserDetails 的管理。当 Spring Security 配置为接受用户名/密码进行身份验证时，使用基于 UserDetails 的身份验证。

Java

```
/**
 * 在本示例中，我们使用 Spring Boot CLI 对 password 的密码进行编码，
 * 并得到 {bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76k
 */
@Bean
public UserDetailsService users() {
    UserDetails user = User.builder()
        .username("user")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

在下面的示例中，我们利用 User.withDefaultPasswordEncoder() 来确保存储在内存中的密码受到保护。但是，它可以反编译源代码获得密码。出于这个原因，User.withDefaultPasswordEncoder() 应该只用于"入门"，而不是用于生产

Java

```
@Bean
public UserDetailsService users() {
    // The builder will ensure the passwords are encoded before saving in m
    UserBuilder users = User.withDefaultPasswordEncoder();
    UserDetails user = users
```

```
            .username("user")
            .password("password")
            .roles("USER")
            .build();
        UserDetails admin = users
            .username("admin")
            .password("password")
            .roles("USER", "ADMIN")
            .build();
        return new InMemoryUserDetailsManager(user, admin);
}
```

2.1.5 DBC 身份验证的关系数据库 [JDBC Authentication]

> Spring Security 的 JdbcDaoImpl 实现 UserDetailsService 以支持使用 JDBC 检索的基于用户名/密码的身份验证。 JdbcUserDetailsManager 扩展了 JdbcDaoImpl 以通过 UserDetailsManager 接口提供对 UserDetails 的管理。当 Spring Security 配置为接受用户名/密码进行身份验证时，使用基于 UserDetails 的身份验证。

JDBC 身份验证默认表格

用户表

SQL

```sql
-- org/springframework/security/core/userdetails/jdbc/users.ddl
-- MySQL schema
create table users(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(500) not null,
    enabled boolean not null
);

create table authorities (
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username) references users(
);
create unique index ix_auth_username on authorities (username,authority);
```

```
-- Oracle schema
CREATE TABLE USERS (
    USERNAME NVARCHAR2(128) PRIMARY KEY,
    PASSWORD NVARCHAR2(128) NOT NULL,
    ENABLED CHAR(1) CHECK (ENABLED IN ('Y','N') ) NOT NULL
);

CREATE TABLE AUTHORITIES (
    USERNAME NVARCHAR2(128) NOT NULL,
    AUTHORITY NVARCHAR2(128) NOT NULL
);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_UNIQUE UNIQUE (USERNAME,
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_FK1 FOREIGN KEY (USERNAM
```

组

SQL

```
create table groups (
    id bigint generated by default as identity(start with 0) primary key,
    group_name varchar_ignorecase(50) not null
);

create table group_authorities (
    group_id bigint not null,
    authority varchar(50) not null,
    constraint fk_group_authorities_group foreign key(group_id) references
);

create table group_members (
    id bigint generated by default as identity(start with 0) primary key,
    username varchar(50) not null,
    group_id bigint not null,
    constraint fk_group_members_group foreign key(group_id) references grou
);
```

设置 DataSource

Java

```
@Bean
DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
```

```
        // 使用H2作为数据库
        .setType(H2)
        // 将默认用户表写入到数据库中
        .addScript("classpath:org/springframework/security/core/userdetails
        .build();
}
```

JDBC 用户管理实例

Java

```
@Bean
UserDetailsManager users(DataSource dataSource) {
    UserDetails user = User.builder()
        .username("user")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")
        .password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0
        .roles("USER", "ADMIN")
        .build();
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.createUser(user);
    users.createUser(admin);
}
```

2.1.6 用户信息 UserDetails

UserDetails 由 UserDetailsService 返回。 DaoAuthenticationProvider 验证 UserDetails，然后返回一个 Authentication，该 Authentication 的主体是配置的 UserDetailsService 返回的 UserDetails

2.1.7 用户详情服务 UserDetailsService

DaoAuthenticationProvider 使用 UserDetailsService 来检索用户名、密码和其他属性，以使用用户名和密码进行身份验证。 Spring Security 提供 UserDetailsService 的内存和 JDBC 实现

可以通过将自定义 UserDetailsService 公开为 bean 来定义自定义身份验证。例如，以下将自定义身份验证假设 CustomUserDetailsService 实现 UserDetailsServic

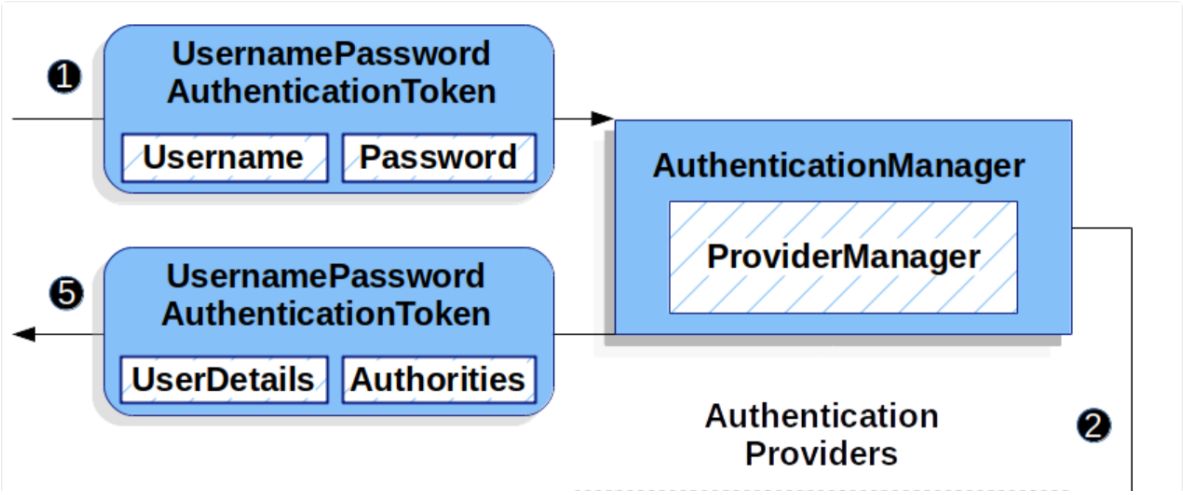
```java
// 这仅在尚未填充 AuthenticationManagerBuilder 且未定义 AuthenticationProvider
@Bean
CustomUserDetailsService customUserDetailsService() {
    return new CustomUserDetailsService();
}
```

2.1.8 密码编码器 PasswordEncoder

Spring Security 的 servlet 支持通过与 PasswordEncoder 集成来安全地存储密码。可以通过公开 PasswordEncoder Bean 来自定义 Spring Security 使用的 PasswordEncoder 实现

2.1.9 Dao 认证提供者 DaoAuthenticationProvider

DaoAuthenticationProvider 是一个 AuthenticationProvider 实现，它利用 UserDetailsService 和 PasswordEncoder 来验证用户名和密码。


1
UsernamePassword
AuthenticationToken
Username
Password
AuthenticationManager
ProviderManager
5
UsernamePassword
AuthenticationToken
UserDetails
Authorities
Authentication
Providers
2