法規掰掰！
Spring Authorization Server 的 OTP 整合快攻

Sam Wang

@Softleader 松凌科技

今日議程

- 背景: 為何需要 2FA 與 OIDC code flow 簡介
- 起點: 如何快速啟動一個 Spring-Authorization-Server with OIDC
- 預設行為: Spring Security 預設的登入流程是什麼樣子？
- 挑戰: 我們該在哪裡「插隊」加入 2FA？
- 解決方案: 一個針對 Form Login 的快速實現

Why I need 2FA ?

法條內容

法規名稱： 保險業辦理資訊安全防護自律規範
修正日期： 民國 113 年 07 月 18 日

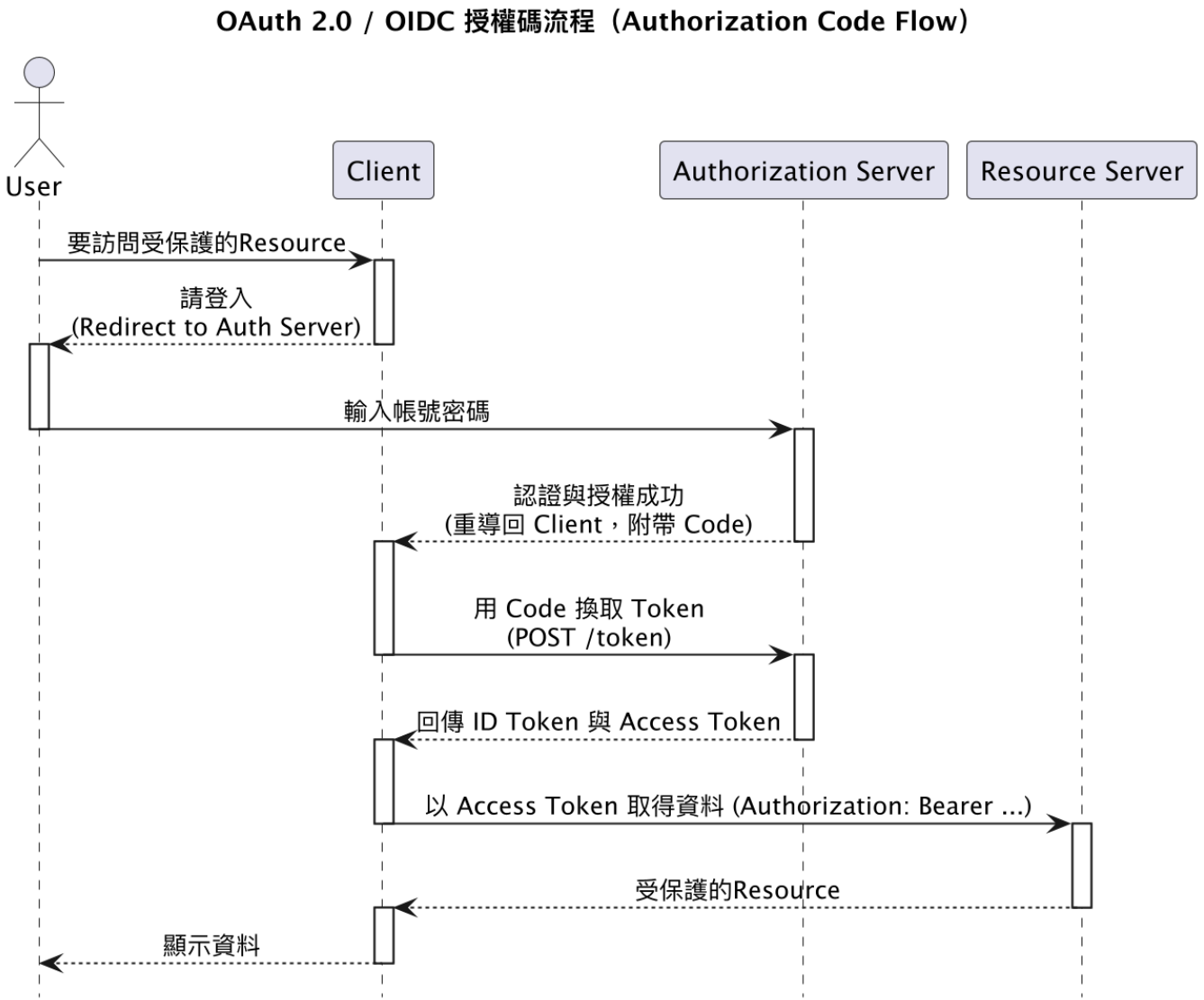
第 18 條　各會員公司應強化對跨機構合作夥伴（含保險經紀人、代理人等合作關係）之資訊安全風險評估與措施，並遵循下列事項：

一、就保險業與跨機構合作夥伴共同使用之網際網路應用系統（如網路投保、網路要保等直接提供客戶自動化服務之系統），其系統管控機制應包括資料傳輸之保密方式、系統使用權限之區隔及系統帳號權限控管等相關資訊安全機制。

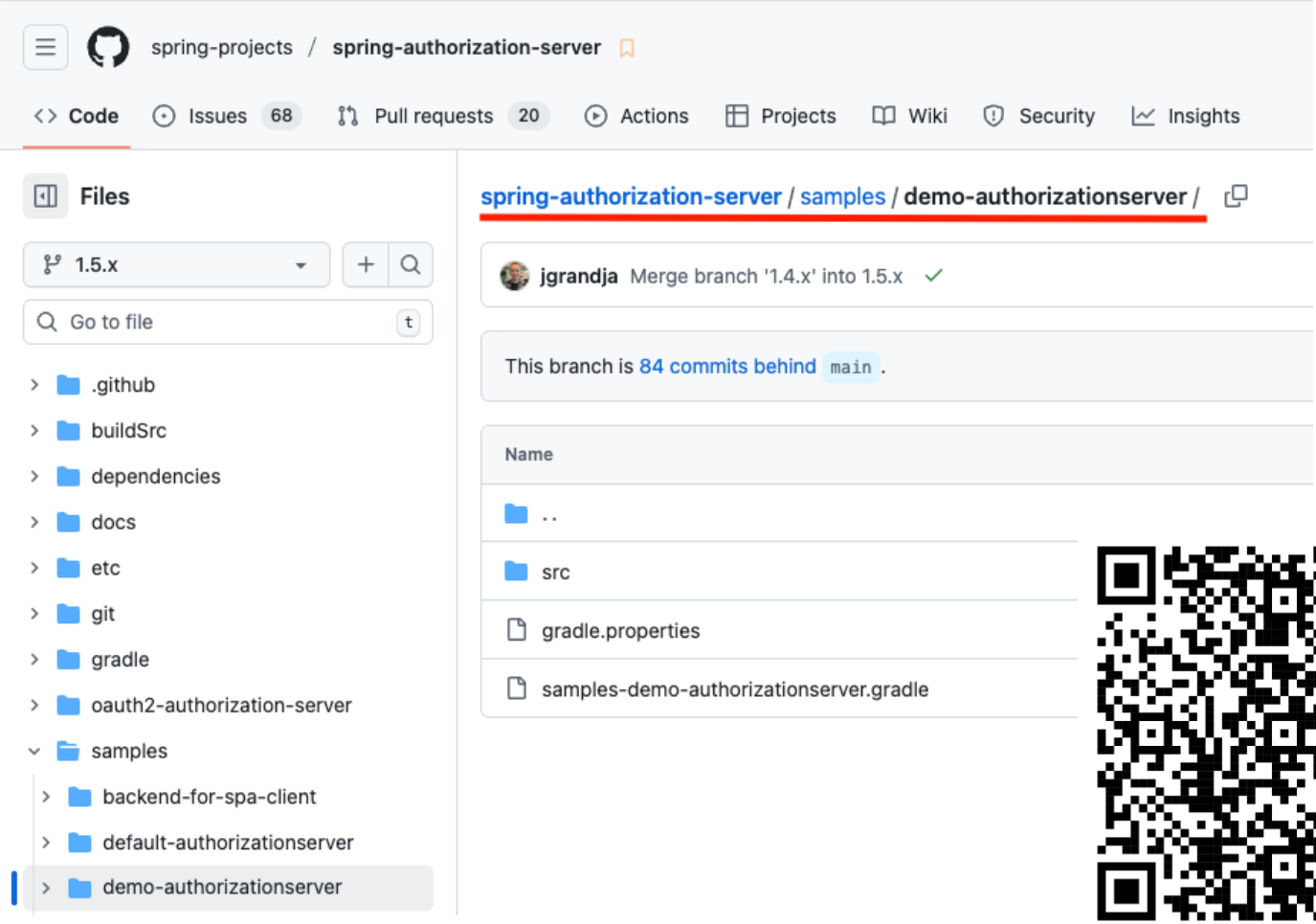
二、與跨機構合作夥伴合約簽訂時，應進行風險評估並規劃風險處置措施，並於雙方簽訂備忘錄或契約中載明相關要求，其內容需包含資訊安全及保戶個人資料保護相關條款、禁止多人共用同一帳號，以及相關業務往來之查核機制或控管措施，以確保資訊安全維護能力與水準。

三、提供跨機構合作夥伴資訊服務者，應採用雙因子驗證或相關身分驗證方式，並應定期辦理帳號密碼變更及帳號清查。

OIDC Authorization Code Flow 概覽



spring-authorization-server sample



快速啟用 OIDC Server

Title

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-authorization-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

authorizationServerSecurityFilterChain

Title

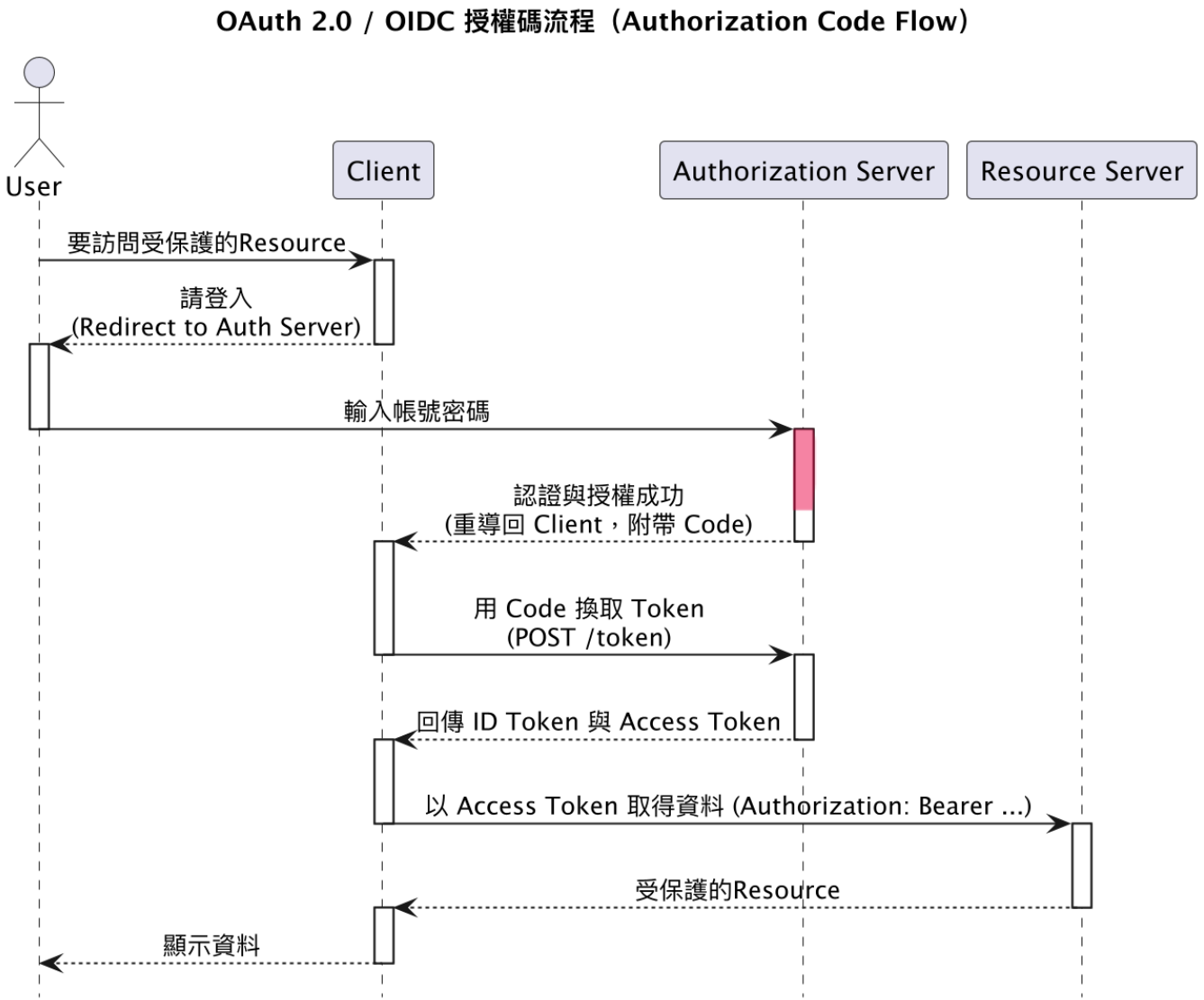
```java

@Configuration
public class AuthorizationServerConfig {

  @Bean
  @Order(Ordered.HIGHEST_PRECEDENCE)
  public SecurityFilterChain authorizationServerSecurityFilterChain(HttpSecurity http) throws Exception {

   final OAuth2AuthorizationServerConfigurer configurer = OAuth2AuthorizationServerConfigurer.authorizationServer();
   http.securityMatcher(configurer.getEndpointsMatcher())
        .with(
            configurer,
            serverConfigurer -> serverConfigurer.oidc(Customizer.withDefaults())) // Enable OpenID Connect 1.0
        .authorizeHttpRequests((authorize) -> authorize.anyRequest().authenticated())
        .exceptionHandling(
            (exceptions) ->
                exceptions.defaultAuthenticationEntryPointFor(
                    new LoginUrlAuthenticationEntryPoint("/login"),
                    new MediaTypeRequestMatcher(MediaType.TEXT_HTML)));
    return http.build();
  }

  // exists code ...
}
```

DefaultSecurityFilterChain for login page

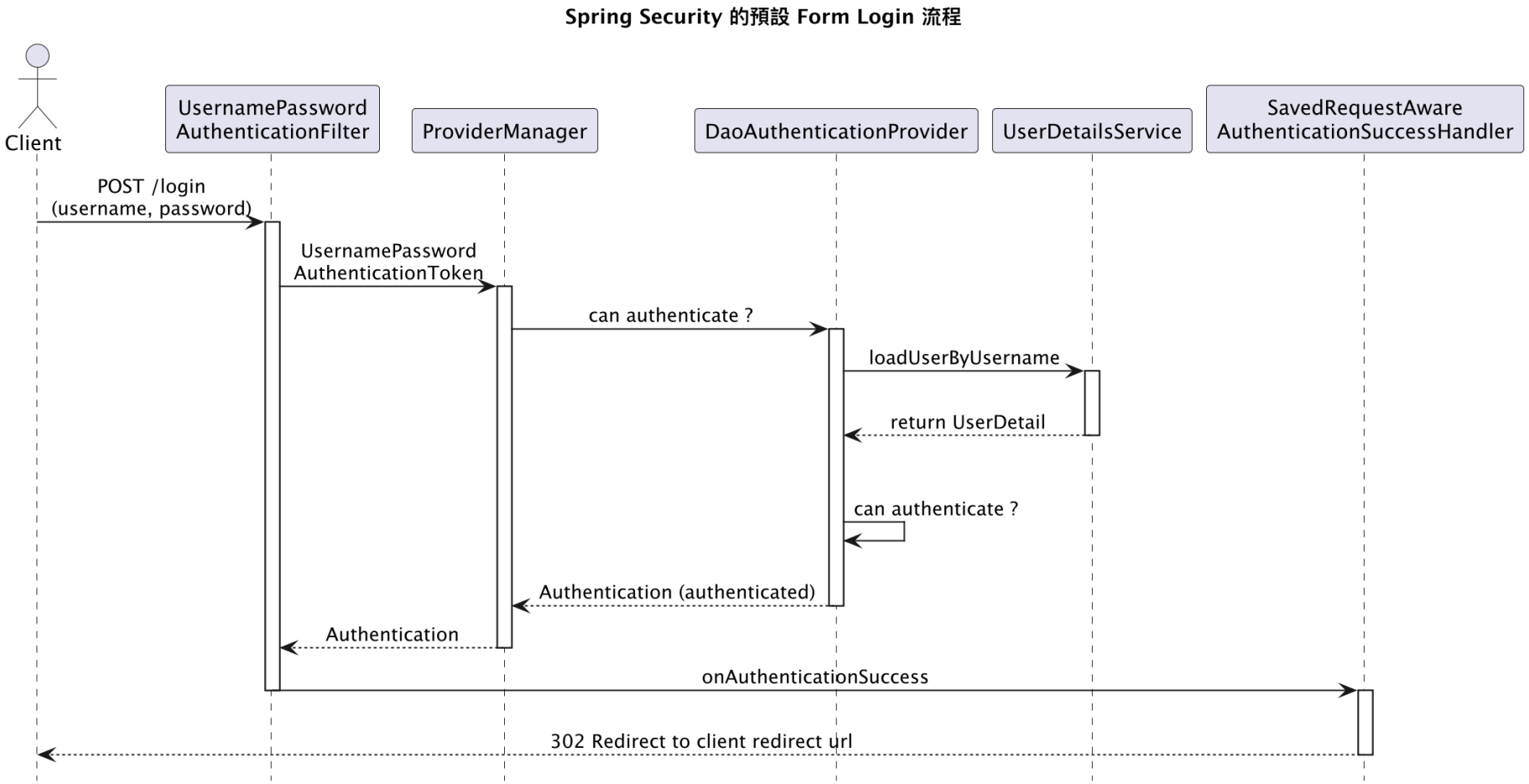
Title

```
1
2 @EnableWebSecurity
3 @Configuration
4 public class DefaultSecurityConfig {
5
6   @Bean
7   @Order(Ordered.HIGHEST_PRECEDENCE + 1)
8   public SecurityFilterChain defaultSecurityFilterChain(
9       HttpSecurity http, CustomAuthenticationProvider authenticationProvider) throws Exception {
10
11    http
12        .securityMatcher("/assets/**", "/login")
13        .authorizeHttpRequests((authorize) -> authorize.anyRequest().permitAll())
14        .formLogin(formLogin -> formLogin.loginPage("/login"))  // --> enable form login
15        .exceptionHandling(
16            exceptionHandling ->
17                exceptionHandling.defaultAuthenticationEntryPointFor(
18                    new LoginUrlAuthenticationEntryPoint("/login"),
19                    new MediaTypeRequestMatcher(MediaType.TEXT_HTML)));
20    return http.build();
21  }
22
23
24  // other required beans...
25 }
```

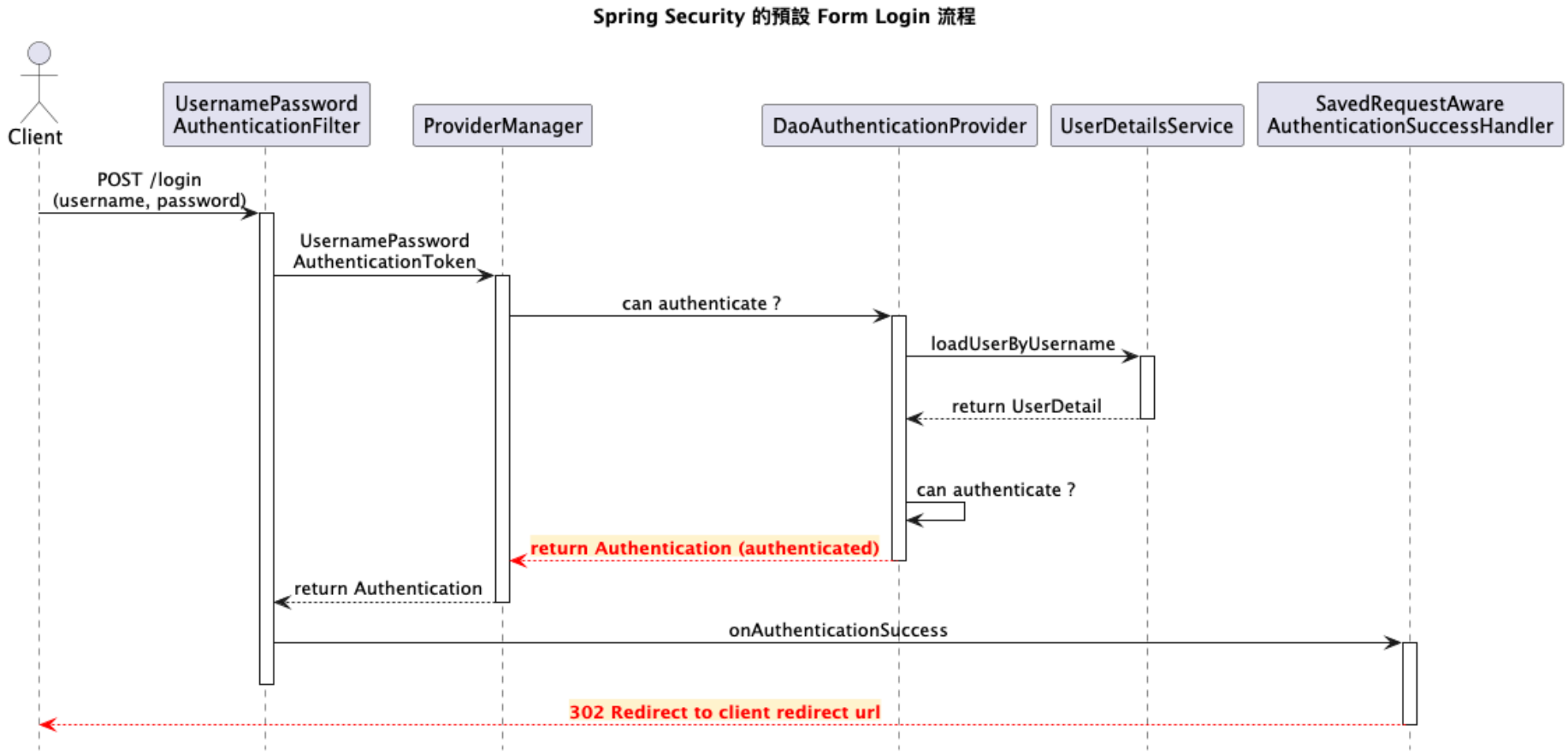
OIDC Authorization Code Flow 概覽



Spring Security 的預設 Form Login 流程



Spring Security 的預設 Form Login 流程

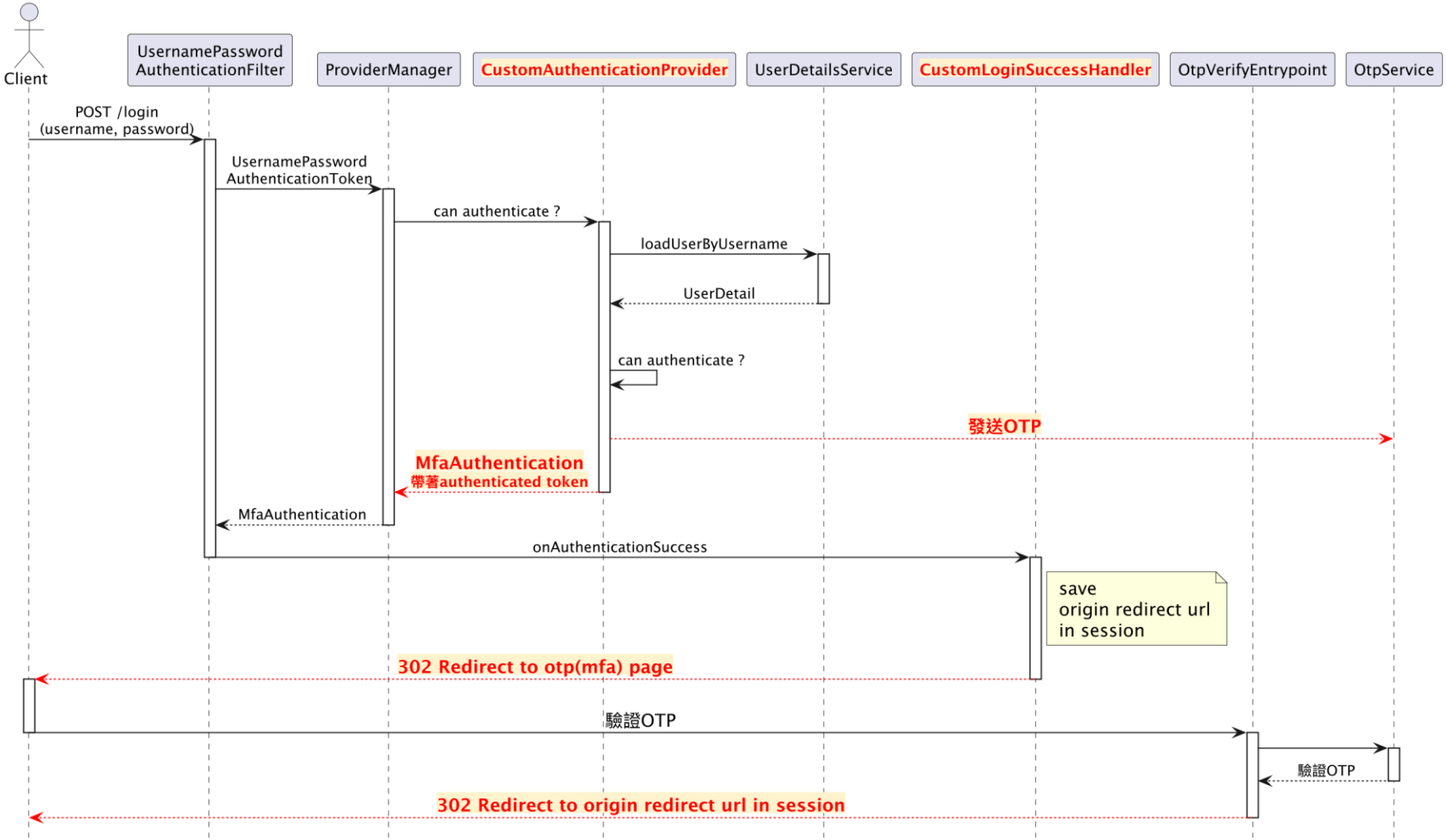


解決方案：客製化核心元件

- **CustomAuthenticationProvider:**
  - 取代預設的 DaoAuthenticationProvider。
  - 職責：只驗證帳號密碼。成功後，不回傳「完全認證」的 Authentication，而是回傳一個自訂的、**「半認證」**的 MfaAuthentication 物件。

- **CustomLoginSuccessHandler**:
  - 取代預設的SavedRequestAwareAuthenticationSuccessHandler。
  - 職責：檢查傳入的 Authentication 物件。
    - 如果拿到「半認證」的 MfaAuthentication，就**導向到 MFA 輸入頁面**。

調整後的 Form Login 流程



自訂的MfaAuthentication

Title

```java
@Getter
public class MfaAuthentication extends AbstractAuthenticationToken {

    private final Authentication firstFactorAuthentication;

    public MfaAuthentication(Authentication firstFactorAuthentication) {
      super(null); // 沒有 authorities
      this.firstFactorAuthentication = firstFactorAuthentication;
      setAuthenticated(false); // 明確設定為未驗證
    }

    @Override
    public Object getCredentials() {
      return firstFactorAuthentication.getCredentials();
    }

    @Override
    public Object getPrincipal() {
      return firstFactorAuthentication.getPrincipal();
    }

}
```

CustomAuthenticationProvider

Title

```java
public class CustomAuthenticationProvider extends DaoAuthenticationProvider {

  // .....

  @Override
  public Authentication authenticate(Authentication authentication) throws AuthenticationException {

    // 首先，使用父類別的邏輯驗證使用者名稱和密碼
    Authentication firstFactorAuthentication = super.authenticate(authentication);

    // 如果使用者名稱和密碼驗證成功
    UserDetails user = (UserDetails) firstFactorAuthentication.getPrincipal();

    // 再產生並發送 OTP
    otpService.generateAndSendOtp(user.getUsername());

    // 返回一個代表「需要 MFA」的中間狀態
    return new MfaAuthentication(firstFactorAuthentication);
  }

  // .....
}
```

CustomLoginSuccessHandler

Title

```java
public class CustomLoginSuccessHandler implements AuthenticationSuccessHandler {

  private final RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();

  @Override
  public void onAuthenticationSuccess(
      HttpServletRequest request, HttpServletResponse response, Authentication authentication)
      throws IOException {

    // 使用 HttpSessionRequestCache 來存取/取得先前因未授權而被保存於 Session 的原始請求資訊（例如登入前想前往的 URL）
    RequestCache requestCache = new HttpSessionRequestCache();

    // 從快取中取回前次被攔截並保存的請求物件，用於後續決定成功登入後要重導至哪裡
    SavedRequest savedRequest = requestCache.getRequest(request, response);

    // 自 SavedRequest 取得原始欲重導的目標 URL
    String originalRequestUrl = savedRequest.getRedirectUrl();

    // 先暫存此 URL 以便 Mfa 完成後再導回
    request.getSession().setAttribute("MFA_AUTH", authentication);
    request.getSession().setAttribute("MFA_ORIGINAL_REQUEST_URL", originalRequestUrl);
    redirectStrategy.sendRedirect(request, response, "/mfa");
  }

}
```

啟用我們的客製元件

Title

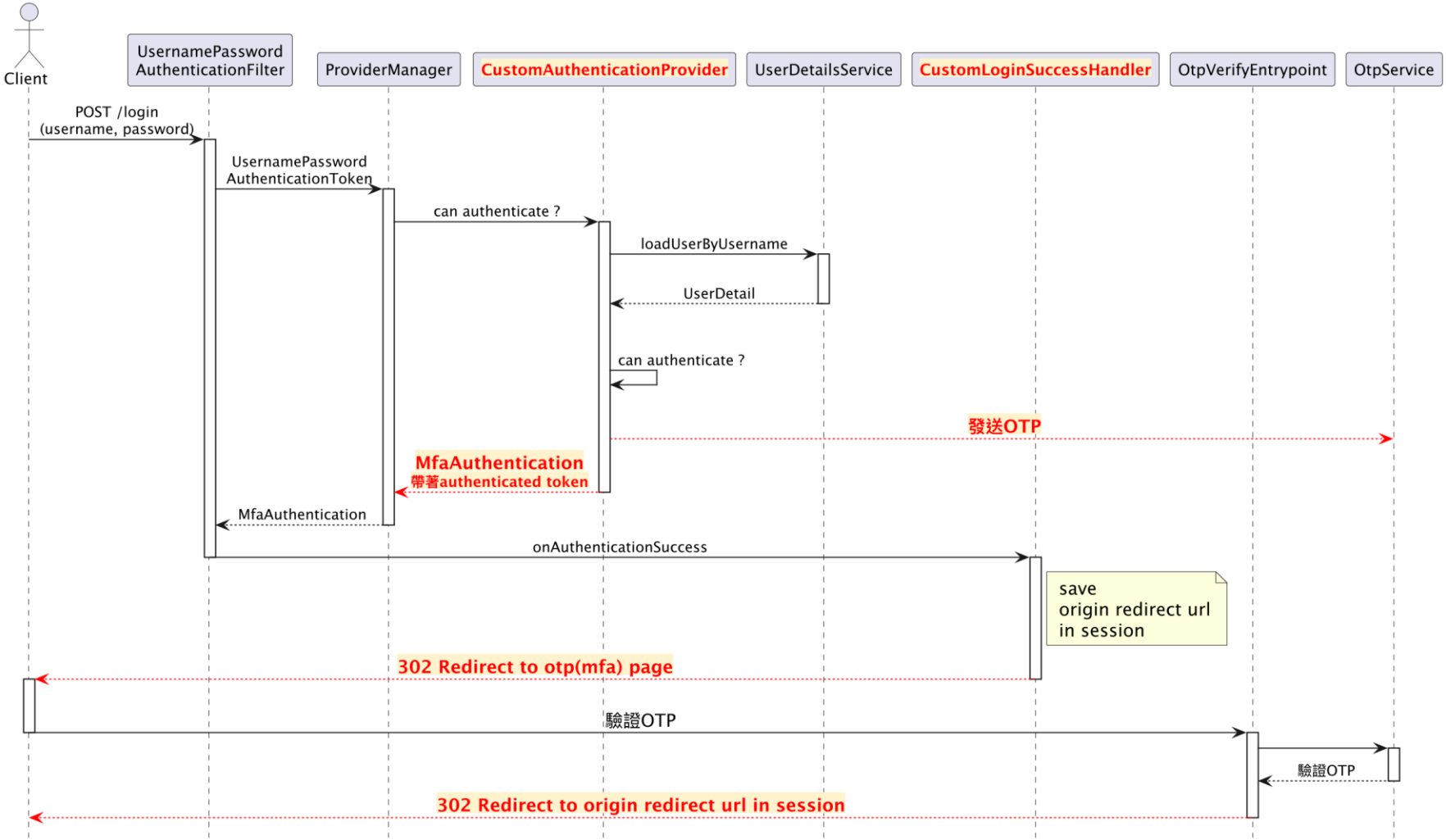
```
1
2 @EnableWebSecurity
3 @Configuration
4 public class DefaultSecurityConfig {
5
6   @Bean
7   @Order(Ordered.HIGHEST_PRECEDENCE + 1)
8   public SecurityFilterChain defaultSecurityFilterChain(
9       HttpSecurity http, CustomAuthenticationProvider authenticationProvider) throws Exception {
10
11    http
12        // ...
13        .authenticationProvider(authenticationProvider) // <--- 在這裡註冊我們的 Provider
14        .formLogin(
15            formLogin ->
16                formLogin
17                   .loginPage("/login")
18                   .successHandler(new CustomLoginSuccessHandler()) // <--- 在這裡註冊我們的 Handler
19        )
20        // ...
21
22    return http.build();
23  }
24
25
26  // other required beans...
27 }
```

OtpVerifyEntrypoint

Title

```java
@PostMapping("/verify-mfa")
public String verifyMFa(
    @RequestParam String otp, HttpSession session, HttpServletRequest request, HttpServletResponse response) {

  // 從session取出先前的MfaAuthentication
  MfaAuthentication mfaAuth = (MfaAuthentication) session.getAttribute("MFA_AUTH");

  // 驗證OTP
  UserDetails user = (UserDetails) mfaAuth.getPrincipal();
  otpService.validateOtp(user.getUsername(), otp);

  // 驗證成功後，取出被我們放在MfaAuthentication中，原本的UsernamePasswordAuthenticationToken
  Authentication finalAuth = mfaAuth.getFirstFactorAuthentication();

  // 當前的SecurityContext裡的Authentication會是我們放的MfaAuthentication
  // 要把原本的UsernamePasswordAuthenticationToken放回去
  SecurityContextHolder.getContext().setAuthentication(finalAuth);
  SecurityContextRepository contextRepository = new HttpSessionSecurityContextRepository();
  contextRepository.saveContext(SecurityContextHolder.getContext(), request, response);

  // 從session 拿出original request url ，然後導過去
  String targetUrl = (String) session.getAttribute("MFA_ORIGINAL_REQUEST_URL");

  // remove attribute from session ...
  return "redirect:" + targetUrl;

}
```

調整後的 Form Login 流程



Thank You