

# FIDO U2F 应用与开发(一)-原理与协议

## 1. FIDO 与 U2F

FIDO(Fast IDentity Online 联盟)是一个基于标准、可互操作的身份认证生态系统。

U2F(Universal 2nd Factor)是 FIDO 联盟提出的使用标准公钥密码学技术提供更强有力的身份认证协议。

U2F 在常用的用户名/密码的认证基础上又增加了一层第二因子(2nd Factor)的保护，这重保护是通过物理硬件来支持的。

## 2. U2F 协议原理

FIDO 的官方网站用两张图介绍了 U2F 的应用原理，它把应用过程分解为两个阶段：注册阶段，如图 1 所示：

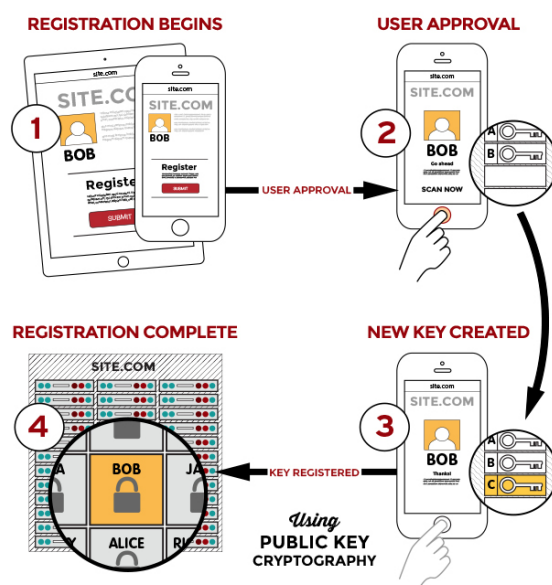


图 1

图 1 中的流程如下：

1. 客户端提示用户选择符合在线服务策略的可用 的 U2F 设备。
2. 用户使用 U2F 设备上的按钮解锁 U2F 设备。
3. U2F 设备创建一对针对本地设备、在线服务和用户账户的独有的全新公/私钥对。
4. 客户端将公钥发送给在线服务，并将其与用户的账户关联，私钥存放在 U2F 设备中。

鉴权阶段，如图 2 所示：

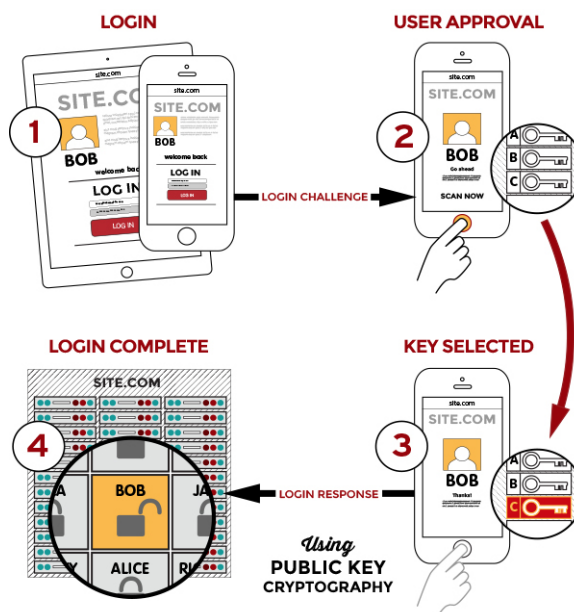


图 2

图 2 中的流程如下：

1. 在线服务要求用户使用之前注册使用的 U2F 设备登录。
2. 用户使用 U2F 设备上的按钮解锁 U2F 设备。
3. 设备使用由服务器提供的用户的账户标识来选择正确的密钥并签名服务器发出的挑战值。
4. U2F 设备将经过签名的挑战发送回服务器，由其使用存储的公钥进行验证，服务器验证成功后允许用户登录。

U2F 的原理很简单：用户需要使用用户名/口令、客户端设备(物理设备)两重安全因子完成在线服务网站的注册与登录鉴权工作；用户在向在线服务注册期间，用户的客户端设备会创建一个新的密钥对。该设备保留私钥，并向在线服务注册公钥；用户在登录鉴权期间，客户端通过对挑战值签名的方式向该服务证明私钥的拥有权，以此完成身份认证。其实这与我们日常使用的网上银行登录时需要插入 USB Key 没有什么区别，可能唯一的技术差别在于浏览器在识别银行的 USB Key 使用的是特定的插件，而对于 U2F 设备，浏览器已经内置了识别接口。

### 3. U2F 协议的消息格式

U2F 协议支持两个操作：注册(registration)与鉴权(authentication),这两个操作可分解为三个阶段，如图 3 所示。

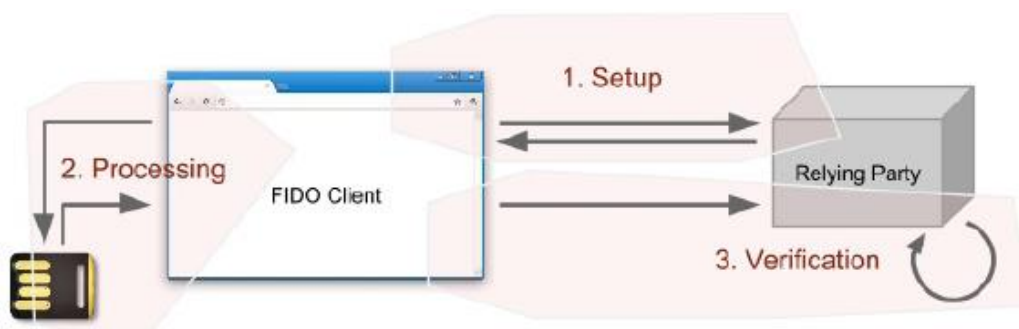


图 3

在图 3 中, Relying Party(简称 RP)就是第 2 节描述的在线服务网站。FIDO Client 为客户端, 三个阶段的流程如下:

1. **Setup**:在这个阶段, 客户端向在线服务网站请求一个挑战值, 客户端使用这个挑战值生成一个请求消息发送到 U2F 设备。
2. **Processing**:在这个阶段, U2F 设备针对请求消息做一些密码学的操作, 并创建回应消息。
3. **Verification**:在这个阶段, 客户端将 U2F 设备的回应消息交给服务端进行验证, 服务端处理回应消息并验证其正确性。对于一个正确的注册回应使得服务端为用户注册一个公钥; 对于一个正确的鉴权回应使得服务端相信用户拥有一个正确的私钥以通过身份认证。

FIDO 提供了 HID 协议实现浏览器与 U2F 设备(使用 USB 接口)之间的消息通讯, 协议细节可参见对应定义文档。

下面我们了解一下浏览器与 U2F 设备之间的数据帧格式。

### 3.1.注册请求消息

注册请求消息如图 4 所示:



图 4

消息中各字段含义如下:

- **challenge parameter [32 bytes]**: 是对由挑战值组成的 Client Data(后面会介绍, 客户端生成的一个 JSON 字符串)使用 SHA-256 算法得到 32 位摘要。
- **application parameter [32 bytes]**: 对使用 UTF-8 编码的应用 ID(application identity)使用 SHA-256 算法得到 32 位摘要。

### 3.2.注册回应消息

注册回应消息如图 5 所示:

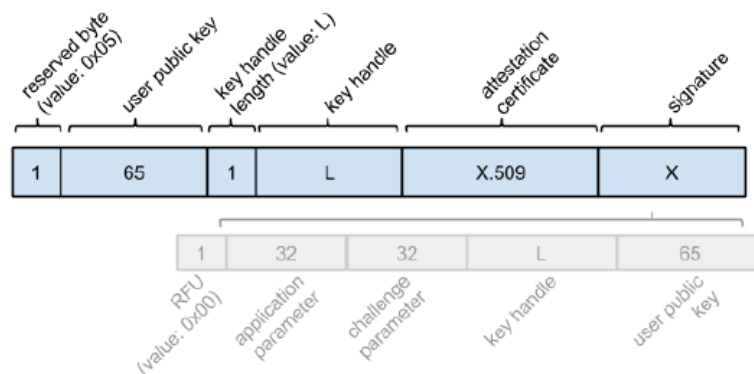


图 5

消息中字段含义如下：

- reserved byte [1 byte]: 固定值 0x05。
- user public key [65 bytes]: 65 字节的公钥。
- key handle length byte [1 byte]: key handle 用于定位私钥，这里使用 1 字节表示长度。
- key handle [length specified in previous field]: key handle 的值。
- attestation certificate [variable length]: 使用 X.509 DER 格式的证书（有点类似于厂商提供的根证书），其中的公钥用于验证后面的签名。
- signature [variable length, 71-73 bytes]: 使用 ECDSA 算法的签名值，使用 ANSI X9.62 格式编码。签名的原文为：
  - 一个为 0x00 的字节
  - 请求中的 application parameter
  - 请求中的 challenge parameter
  - 上面提及的 key handle 的值
  - 上面提及的 user public key

### 3.3. 鉴权请求消息

鉴权请求消息如图 6 所示：

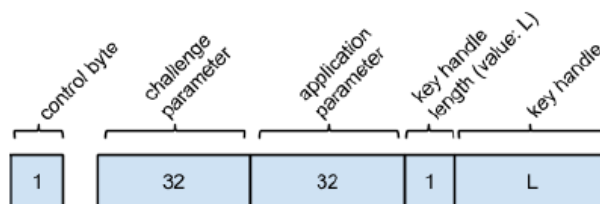


图 6

消息中字段含义如下：

- Control byte (P1)：取值可为 3 个值：0x07 ("check-only")，0x03 ("enforce-user-presence-and-sign")，0x08 ("dont-enforce-user-presence-and-sign")
- challenge parameter [32 bytes]: 是对由挑战值组成的 Client Data(后面会介绍，客户端生成的一个 JSON 字符串)使用 SHA-256 算法得到 32 位摘要。
- application parameter [32 bytes]: 对使用 UTF-8 编码的应用 ID (application identity) 使用使用 SHA-256 算法得到 32 位摘要。

- key handle length byte [1 byte]: 1 字节表示 key handle 的长度。
- key handle [length specified in previous field]: key handle 的值。

### 3.4. 鉴权响应消息

鉴权响应消息如图 7 所示:

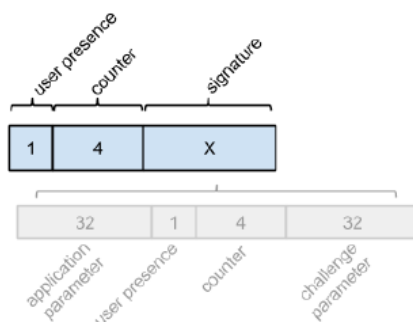


图 7

消息中各字段含义如下:

- user presence byte [1 byte]: 0 表示用户不存在, 1 表示用户存在。
- counter [4 bytes]: U2F 设备鉴权计数, 大字节序。
- signature: 使用 ECDSA 算法的签名值, 使用 ANSI X9.62 格式编码。签名的原文为:
  - application parameter [32 bytes]: 鉴权请求中的 application parameter。
  - user presence byte [1 byte]: 上面提到的 user presence byte。
  - counter: 上面提到的 counter。
  - challenge parameter [32 bytes]: 鉴权请求中的 challenge parameter 值。

### 3.5. clientdata

前面的请求/响应消息中对 challenge parameter 的构建使用到 clientdata, clientdata 为一个 JSON 对象字符串, 其对象结构 **ClientData** 定义如下:

```

dictionary ClientData {
    DOMString      typ;
    DOMString      challenge;
    DOMString      origin;
    (DOMString or JwkKey) cid_pubkey;
};
  
```

ClientData 结构中各属性含义如下:

- typ: 注册时使用值 'navigator.id.finishEnrollment', 鉴权时使用值 'navigator.id.getAssertion'。
- challenge: 使用 websafe-base64 编码(通常也叫 urlbase64 编码, 见 RFC4648 第 5 章)

的字符串，由在线服务网站提供。

- `origin`: 网站标识。
- `cid_pubkey`: 可选参数。

## 4. 参考文献

1. <https://fidoalliance.org/how-fido-works/>
2. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/FIDO-U2F-COMPLETE-v1.2-ps-20170411.pdf>
3. <http://www.ietf.org/rfc/rfc4648.txt>

# FIDO U2F 应用开发(二)-编程接口

## 1. U2F JS API

FIDO U2F 定义了 JavaScript API 供开发者开发支持 U2F 设备的在线服务网站。U2F JS API 分为两类：底层基于消息端口的 API 和上层应用 API。在 FIDO 的规格文档中介绍底层 API 用于与 U2F 设备进行消息通讯（使用 MessagePort Object），发送和接收消息。本文重点关注屏蔽了通讯细节的上层 API 接口。

## 2. 接口定义

### 2.1. u2f 接口

使用 WebIDL 定义的 u2f 接口定义如下：

```
interface u2f {  
    void register (DOMString appId, sequence<RegisterRequest>  
registerRequests, sequence<RegisteredKey> registeredKeys,  
function(RegisterResponse or Error) callback, optional unsigned long?  
opt_timeoutSeconds);  
    void sign (DOMString appId, DOMString challenge,  
sequence<RegisteredKey> registeredKeys, function(SignResponse or Error)  
callback, optional unsigned long? opt_timeoutSeconds);  
};
```

### 2.2. register 方法

#### 2.2.1. 请求参数

register 方法中各参数描述如下：

参数名称	类型	可否为空	是否可选	描述
appId	DOMString	✗	✗	请求中的应用 ID

registerRequests	sequence<RegisterRequest>	✗	✗	注册请求序列
registeredKeys	sequence<RegisteredKey>	✗	✗	已经注册到 U2F 设备的信息
callback	function(RegisterResponse or Error)	✗	✗	注册请求回调函数
opt_timeoutSeconds	unsigned long	✓	✓	客户端等待请求处理的超时时间

## 2.2.2. 返回值

`register` 方法成功返回的数据（`callback` 的参数）使用 `RegisterResponse` 结构。

```
dictionary RegisterResponse {
    DOMString version;
    DOMString registrationData;
    DOMString clientData;
};
```

其中各属性含义如下：

- **version:** U2F 协议版本，如 “U2F\_V2”
- **registrationData:** 使用 `websafe-base64` 编码后的注册数据，数据格式参看《FIDO U2F 设备应用与开发(一)-原理与协议》3.2 节。
- **clientData:** 使用 `websafe-base64` 编码后的 `clientData`，数据格式参看《FIDO U2F 设备应用与开发(一)-原理与协议》3.5 节。

## 2.3. sign 方法

### 2.3.1. 请求参数

`sign` 方法中各参数描述如下：

参数名称	类型	可否为空	是否可选	描述
appId	DOMString	✗	✗	请求中的应用 ID
challenge	DOMString	✗	✗	使用 WEBSAFE-BASE64 编码的挑战值



registeredKeys	sequence<RegisteredKey>	✗	✗	待签名用户的注册信息
callback	function(SignResponse or Error)	✗	✗	签名请求回调函数
opt_timeoutSeconds	unsigned long	✓	✓	客户端等待请求处理的超时时间

### 2.3.2. 返回值

sign 方法成功返回的数据(callback 的参数)使用 SignResponse 结构。

```
dictionary SignResponse {
    DOMString keyHandle;
    DOMString signatureData;
    DOMString clientData;
};
```

其中各属性含义如下：

- keyHandle: 请求中提供的 key handle
- signatureData: 使用 websafe-base64 编码后的签名数据，数据格式参看《FIDO U2F 设备应用与开发(一)-原理与协议》3.4 节。
- clientData: 使用 websafe-base64 编码后的 clientData，数据格式参看《FIDO U2F 设备应用与开发(一)-原理与协议》3.5 节。

## 2.4. 错误码

register 和 sign 方法失败时返回的错误码定义如下：

```
interface ErrorCode {
    const short OK = 0;
    const short OTHER_ERROR = 1;
    const short BAD_REQUEST = 2;
    const short CONFIGURATION_UNSUPPORTED = 3;
    const short DEVICE_INELIGIBLE = 4;
    const short TIMEOUT = 5;
};
```

## 2.5. 接口中的数据结构

### 2.5.1. RegisterRequest

使用 WebIDL 定义的 RegisterRequest 结构如下：

---

```
dictionary RegisterRequest {  
    DOMString version;  
    DOMString challenge;  
};
```

---

属性含义如下：

- version: U2F 协议版本，如 “U2F\_V2”
- challenge: 使用 websafe-base64 编码的挑战值

### 2.5.2. RegisteredKey

使用 WebIDL 定义的 RegisteredKey 结构如下：

---

```
dictionary RegisteredKey {  
    DOMString version;  
    DOMString keyHandle;  
    Transports? transports;  
    DOMString? appId;  
};
```

---

各属性含义如下：

- version: U2F 协议版本，如 “U2F\_V2”
- keyHandle: 用于签名用户的 key handle
- transports: 传输方式，可选参数
- appId: 在线服务网站应用 Id

## 3. 编程接口实例探究

### 3.1. 注册过程

让我们来到 yubico 的 U2F 设备测试网站 (<https://demo.yubico.com/u2f>)，使用 yubico 的安全 key，看看 register 方法和 sign 方法如何使用。

从淘宝上购买的两个 U2F Key,如图 1 所示，其中一个飞天品牌的带蓝牙功能。



图 1

首先测试注册过程，如图 2 所示。

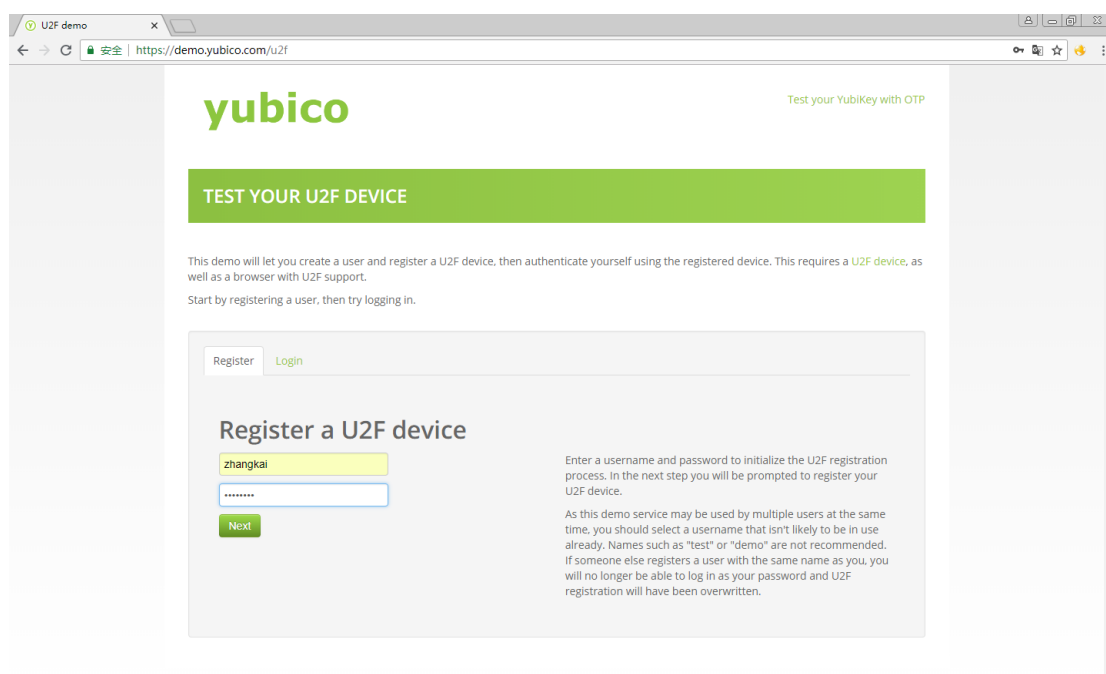


图 2

通过跟踪网站客户端与服务端的交互消息，我们发现开始注册过程后，客户端向服务端的提交了两请求，对应上篇文章《FIDO U2F 应用开发(一)-原理与协议》第 3 节描述的 3 个阶段中的第 1 和第 3 阶段。

客户端第一次向服务端提交请求后，参数包含用户名和口令，如图 3 所示。

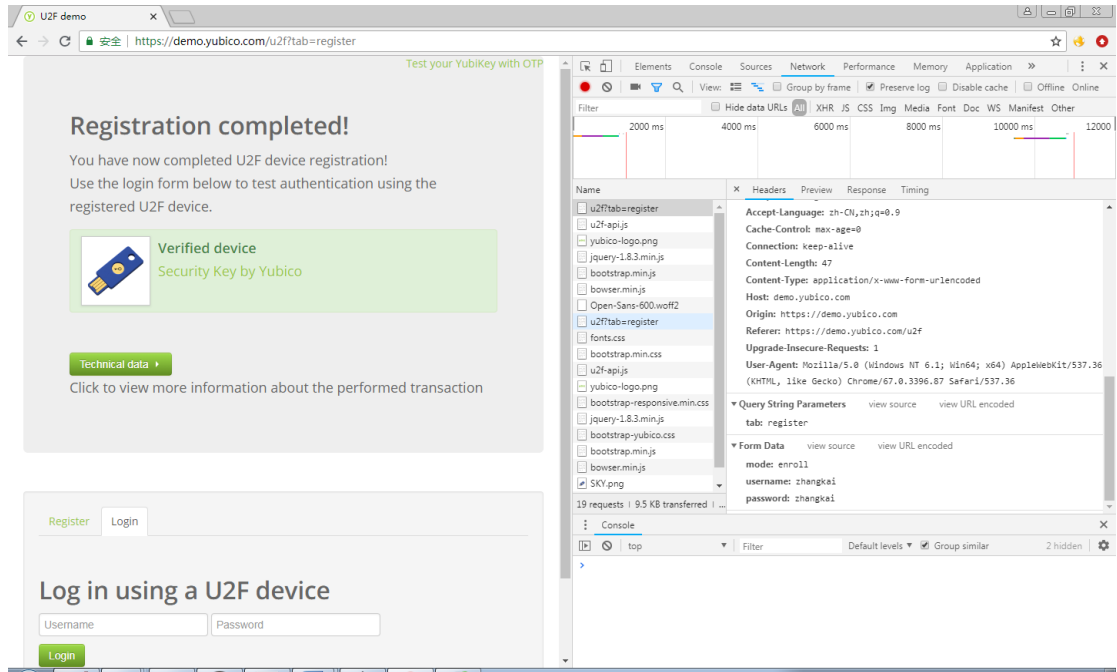


图 3

在客户端 JS 脚本对服务端返回的注册请求进行 `register` 函数调用后，将 `register` 的注册数据提交给服务端，提交的表单数据如图 4 所示。

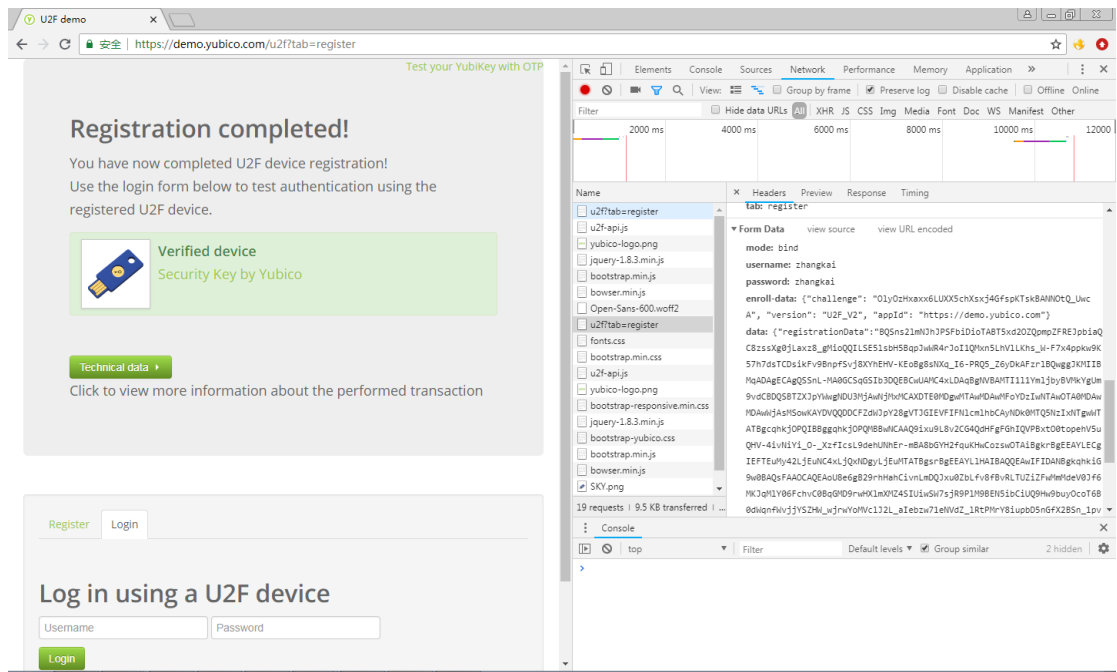


图 4

由图 3 提交的数据可以看到 `register` 返回成功，提交的表单数据为：

名称	值
<b>mode</b>	bind
<b>username</b>	zhangkai

<b>password</b>	zhangkai
<b>enroll-data</b>	{ "challenge": "0ly0zHxaxx6LUXX5chXsxj4GfspKTskBANN0tQ_UwcA", "version": "U2F_V2", "appId": "https://demo.yubico.com" }
<b>data(u2f 设备接口调用返回数据)</b>	{ "registrationData": "BQSns2lmNJhJPSFbiDioTABT5xd20ZQpmpZFREJpbiaQC8zsXg0jLaxz8_gMioQQILSE5lsbH5BqpJwNR4rJoI1QMxn5LhV1LKhs_W-F7x4ppkw9K57h7dsTCDsikFv9BnpfSvj8XYhEHV-KEoBg8sNXq_I6-PRQ5_Z6yDkAFzr1BQwggJKMIIBMqADAgECAGQSSnL-MA0GCSqGSIb3DQEBChUAMC4xLDAqBgNVBAMTI111Ym1jbyBVMkYgUm9vdCBDQSBTZXJpYWwNDU3MjAwNjMxMCAXDTE0MDgwMTAwMDAwMfoYDzIwNTAwOTA0MDAwMDAwWjAsMSowKAYDVQQDDCFZdWJpY28gVTJGIEVFIENlcm1hbCAyNDk0MTQ5NzIxNTgwWTATBgqhkJOPQIBBggqhkJOPQMBBwNCAAQ9ixu9L8v2CG4QdHfGfGhIQVPBxt00topehV5uQHV-4ivNiYi_0-_XzfIcsL9dehUNhEr-mBA8bGYH2fquKHwCozswOTAiBgkrBgEEAYLEcGIEFTEuMy42LjEuNC4xLjQxNDgyLjEuMTATBgSrBgEEAYL1HAIBAQQEAWIFIDANBgkqhkiG9w0BAQsFAA0CAQEAAoU8e6gB29rhHahCivnLmDQJxu0ZbLf8fBvRLTUZiZfWmMdeV0Jf6MKJqM1Y06FchvC0BqGMD9rwhX1mXMZ4SIuiwSW7sjR9P1M9BEN5ibCiUQ9Hw9buyOcoT6B0dWqnfWvjYSHW_wjrwYoMvclJ2L_aIebzw71eNVdZ_1rtPMrY8iupbD5nGfX2BSn_1pvUt-D6JSjpdnIuC5_i8ja9MgBdf-Jcv2nkzPsR12AbqzJSPG6siBFqVvYpIwgIm2sAD1B-8ngXqKka7XhCkneBgoKT2omdqNNaMSr6MYdDVbkCfoKMqeBksALWLo2M8HRJI XU9NePIfF1XeUU-dzBFAiAtXTkSxA8NFX8RU-qNtKdzBkuVSk-rIFjhkJRALTIbWiHAKjY3XT8vJggy0yGhEyXGF8zQonpWvdOwFoTe77c0v-", "version": "U2F_V2", "challenge": "0ly0zHxaxx6LUXX5chXsxj4GfspKTskBANN0tQ_UwcA", "attestation": "direct", "clientData": "eyJ0eXAiOiJuYXZpZ2F0b3IuawQuZmluaXNoRW5yb2xsbWVudCIsImNoYWxsZW5nZSI6Ik9seU96SHheHg2TFVYWDVjaFhzeGo0R2ZzcEtUc2tCQU50T3RRX1V3Y0EiLCJvcmlnaW4iOiJodHRwciovL2R1bW8ueXViaWNvLmNvbSIsImNpZF9wdWJrZXkiOiJ1bnVzZWQifQ" }

## 3.2. 鉴权过程

注册成功后，可执行鉴权(login)过程，如图 5 所示。

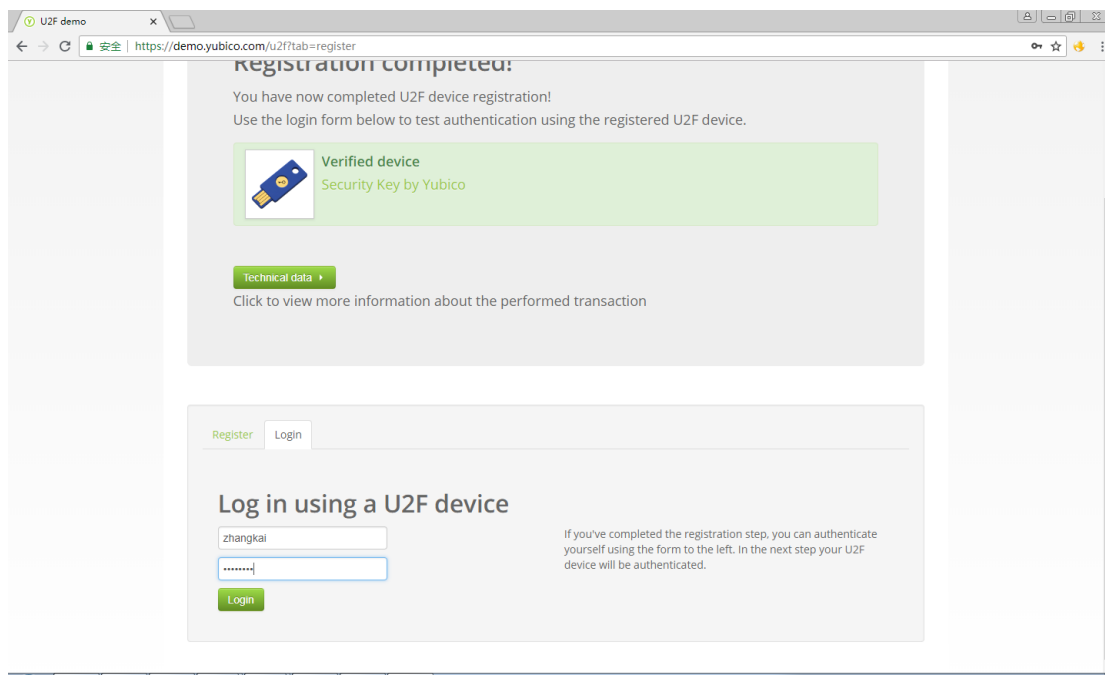


图 5

通过跟踪网站客户端与服务端的交互消息，第一次客户端请求时携带了用户名和密码，如图 6 所示。

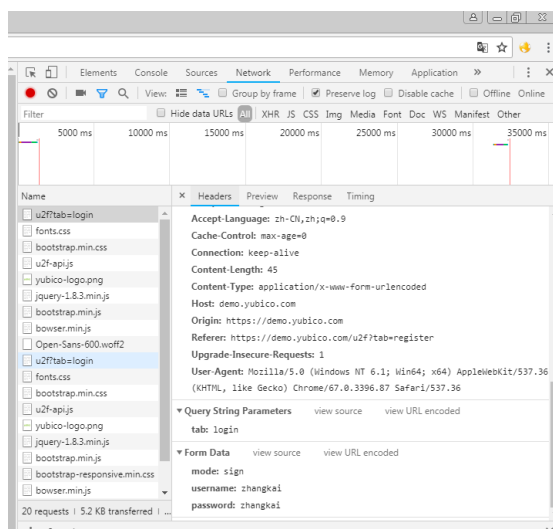


图 6

在客户端 JS 脚本对服务端返回的签名请求进行 sign 函数调用后，U2F 设备产生签名后，客户端将签名数据提交到服务端，如图 7 所示。

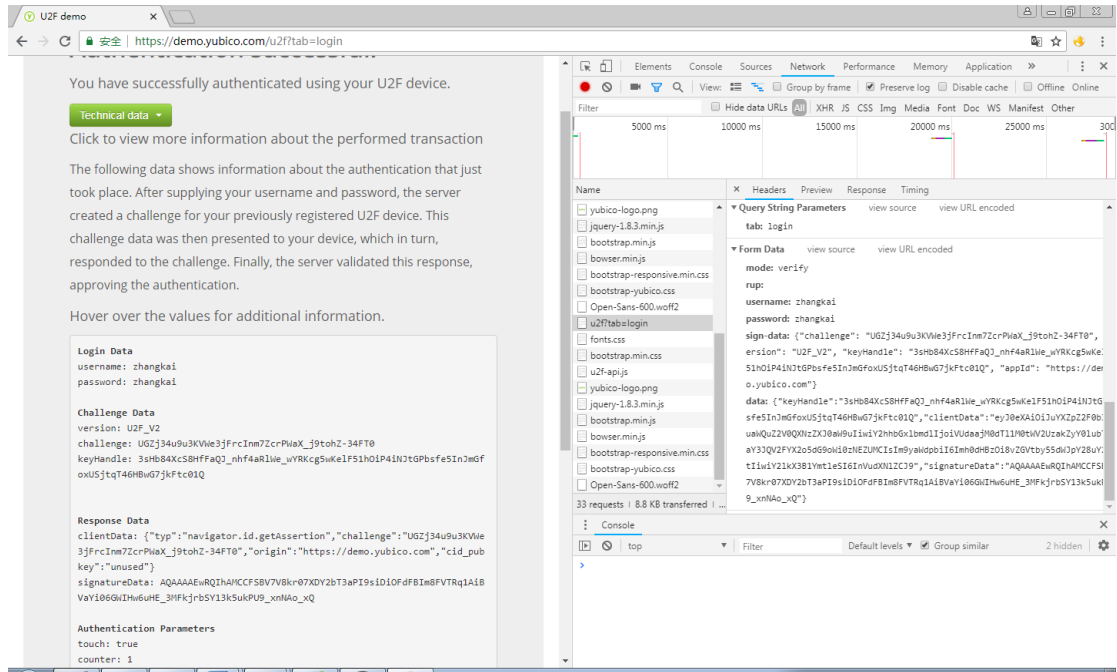


图 7

提交的表单数据为:

名称	值
<b>mode</b>	verify
<b>rup</b>	
<b>username</b>	zhangkai
<b>password</b>	zhangkai
<b>sign-data</b>	{ "challenge": "UGZj34u9u3KVWe3jFrcInm7ZcrPwaX_j9tohz-34FT0", "version": "U2F_V2", "keyHandle": "3sHb84XcS8HfFaQJ_nhf4aRlWe_wYRKcg5wKelF51h0iP4iNjTGpBsfe5InJmGfoxUSjtqT46HBwG7jkFtc01Q", "appId": "https://demo.yubico.com" }
<b>data(u2f 设备接口返回数据)</b>	{ "keyHandle": "3sHb84XcS8HfFaQJ_nhf4aRlWe_wYRKcg5wKelF51h0iP4iNjTGpBsfe5InJmGfoxUSjtqT46HBwG7jkFtc01Q", "clientId": "eyJ0eXAiOiJuYXZpZ2Z0b3IuawQuZ2V0QXNzZXJ0aW9uIiwiaWY2hhdGxlbmdlIjoiaVUdaajM0dTl1M0tWV2UzakZyY01ubTdaY3JQV2FYX2o5dG9oWi0zNEZUMCIIm9yaWdpbiI6Imh0dHBzOi8vZGVtbW55dWJpY28uY29tIiwiaWY2lkX3B1YmtleSI6InVudXN1ZCJ9", "signatureData": "AQAAAAEwRQIhAMCCFSBV7V8kr07XDY2bT3aPI9siDiOFdFBIm8FVTRq1AiBVaYi06GWIHw6uHE_3MFkjrbSY13k5ukPU9_xnNAo_xQ" }

服务端 验证签名后，返回验证成功信息。

### 3.3. 异常处理

实验过程中，如果在交互时不按 U2F 设备的按钮和不插入设备，register 和 sign 函数都会返回错误码。错误码的定义可参看 2.4 节。

### 3.4. u2f-api.js

u2f-api.js 是 yubico 提供的 U2F js api, 封装了第 2 节接口规范中描述的接口。可从地址: <https://demo.yubico.com/js/u2f-api.js> 处获取。u2f-api.js 中的主要定义如下:

```
var u2f = u2f || {};  
u2f.register = function(appId, registerRequests, registeredKeys, callback, opt_timeoutSeconds)  
u2f.sign = function(appId, challenge, registeredKeys, callback, opt_timeoutSeconds)
```

请注意在这个脚本中将 register 和 sign 操作的超时时间定义为 30 秒:

```
u2f.EXTENSION_TIMEOUT_SEC = 30;
```

在执行 3.1 节的注册过程时, 通过跟踪浏览器消息, 可以看到第一次向服务器请求后返回的页面中包含如下 JS 代码:

```
setTimeout(function() {  
    var request = {"challenge": "OlyOzHxaxx6LUXX5chXsxj4GfspKTskBANNOrQ_UwcA", "version":  
"U2F_V2", "appId": "https://demo.yubico.com"};  
    console.log("Register: ", request);  
    var appId = request.appId;  
    var registerRequests = [{version: request.version, challenge: request.challenge, attestation: 'direct'}];  
    $('#promptModal').modal('show');  
    console.log(appId, registerRequests);  
    u2f.register(appId, registerRequests, [], function(data) {  
        console.log("Register callback", data);  
        $('#promptModal').modal('hide');  
        $('#bind-data').val(JSON.stringify(data));  
        $('#bind-form').submit();  
    });  
}, 1000);
```

这段代码中, 使用 U2F 的上层函数 register 进行了注册, 读者可以与 2.2 节的函数参数做一下比对, 在这段代码中 registeredKeys 参数使用是空数组 “[]”。

仔细阅读 u2f-api.js, 会发现脚本使用了 EXTENSION\_ID 为 “krendfapggjehodndflmmgagdbamhnfd” 的 chrome 内置扩展完成与 U2F 设备的通讯。

在执行 3.2 节的鉴权过程时, 通过跟踪浏览器消息, 可以看到第一次向服务器请求后返回的页面中包含如下 JS 代码:

```
setTimeout(function() {  
    var request = {"challenge": "UGZj34u9u3KVWe3jFrcInm7ZcrPWaX_j9tohZ-34FT0", "version":  
"U2F_V2", "keyHandle":  
"3sHb84XcS8HfFaQJ_nhf4aRIWe_wYRKcg5wKelF51hOiP4iNJtGPbsfe5InJmGfoxUSjtqT46HBwG7jkFtc01Q",  
"appId": "https://demo.yubico.com"};
```



```

console.log("sign: ", request);
var appId = request.appId;
var challenge = request.challenge;
var registeredKeys = [{version: request.version, keyHandle: request.keyHandle}];
$('#promptModal').modal('show');
u2f.sign(appId, challenge, registeredKeys, function(data) {
    $('#promptModal').modal('hide');
    $('#verify-data').val(JSON.stringify(data));
    $('#verify-form').submit();
});
}, 1000);

```

这段代码中，使用 U2F 的上层函数 sign 进行了注册，

## 4. 浏览器兼容测试

使用购买的 U2F 设备在 PC 上对 chrome、firefox、IE 浏览器进行了测试。其中 **chrome** 版本为 69，**firefox** 版本为 **firefox quantum 62**，这两种浏览器目前都支持 U2F 设备。

在 firefox 中使用 U2F 需要打开一个开关（默认没有打开），如图 8 所示。

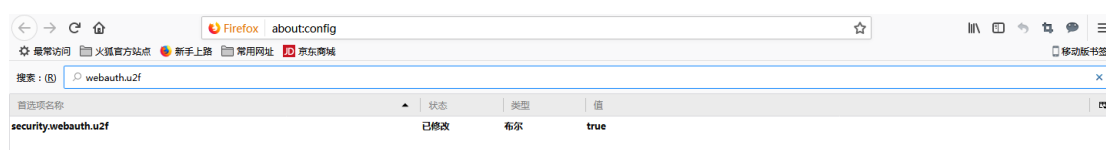


图 8

这里有个有趣的问题，测试时使用的网站仍然是 <https://demo.yubico.com/u2f>，JS 脚本仍然使用的是 u2f-api.js，前面提到这个脚本是针对 chrome 内置扩展应用与 USB 口通信的，在 firefox 中怎么也能正常使用呢？

原因就是 firefox 实现了自己的 u2f 对象，而且这个 u2f 对象的所有属性都是只读的，u2f-api.js 没有改写这个对象，从控制台如下输出可以看到：

```
TypeError: setting getter-only property "u2f"
```

**IE 浏览器使用的版本为 11，不支持 U2F 设备。**

在安卓手机上安装了 chrome app，使用 U2F 设备的 BLE 模式(蓝牙功能)测试了 U2F 的支持，没有成功，在 chrome 调用 register 函数时，U2F 设备没有闪烁。

## 5. 参考文献

1. <https://fidoalliance.org/how-fido-works/>
2. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/FIDO-U2F-COMplete-v1.2-ps-20>

170411.pdf

3. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-javascript-api-v1.2-ps-20170411.html>
4. <https://demo.yubico.com/js/u2f-api.js>
5. <https://www.yubico.com/2017/11/how-to-navigate-fido-u2f-in-firefox-quantum/>

## FIDO U2F 应用与开发(三)-开发支持 U2F 的站点

### 1. 设计中的约束

U2F 协议只能在支持 HTTPS 的网站上才可被支持，对于商业站点这不是问题，但对于局域网内或内部开发环境，我们使用自签名证书来实现 HTTPS。

U2F 设备在网页上进行签名和注册时，会对请求中的“appId”进行检查，“appId”中必须为域名或机器名，不可为 IP 地址，否则会出现错误。

### 2. 一个支持 U2F 的站点 DEMO

#### 2.1. 如何获取代码

[https://github.com/solarkai/FIDO\\_U2F\\_KEDACOM](https://github.com/solarkai/FIDO_U2F_KEDACOM) 提供了一个支持 U2F 的站点项目。虽然只是一个站点 DEMO，但提供了对用户和 U2F 设备管理的完整功能。

#### 2.2. 如何构建运行

该项目客户端使用 jquery 和 jquery-ui 编写，服务端使用 spring-boot 框架编写。可使用 maven 进行构建，使用如下命令：

```
./mvnw clean package
```

对于生成的 jar 包,可使用如下命令(建议 JDK1.8)直接运行(需要注意当前工作目录下要有 tomcat.keystore 文件，不然会报 spring 的注入错误)：

```
java -jar Kedacom-U2F-DEMO-0.0.1-SNAPSHOT.jar
```

上面的程序运行后，启动了一个 tomcat 服务器，支持 http 和 https 两种模式。用户可在浏览器中使用“http://localhost:8080”和“https://localhost:8443”两种模式访问，在 http 模式下不支持 U2F 设备。

#### 2.3. 如何支持 HTTPS

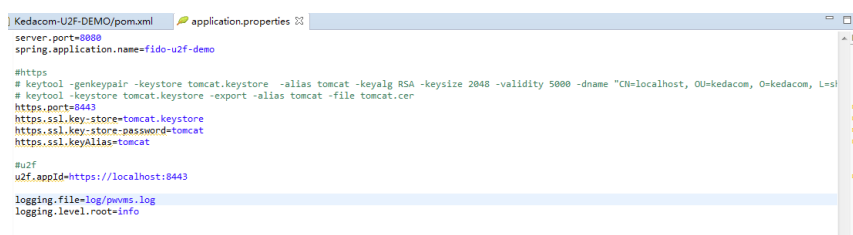
为使用 HTTPS，需要使用自签名证书，我们使用 JDK 自带的 keytool 生成自签名证书 tomcat.keystore，项目运行时与运行 jar 包放在同一目录下。生成命令如下：

```
keytool -genkeypair -keystore tomcat.keystore -alias tomcat -keyalg RSA -keysize 2048  
-validity 5000 -dn "CN=localhost, OU=kedacom, O=kedacom, L=shanghai, ST=shanghai,  
C=cn" -ext "SAN=DNS:localhost,IP:172.16.64.59" -ext "BC=ca:true"
```

为使得浏览器对自签名证书不产生告警，需要从 tomcat.keystore 中导出公钥证书(cer 文件)以导入浏览器的“受信任的根证书颁发机构”，导出公钥证书的命令如下：

```
keytool -keystore tomcat.keystore -export -alias tomcat -file tomcat.cer
```

在项目的 application.properties 文件中，定义了相关 HTTPS 参数，这些参数在项目启动时，被 spring 注入到变量中，application.properties 定义如图 1 所示：



```
server.port=8080  
spring.application.name=fido-u2f-demo  
  
#https  
# keytool -genkeypair -keystore tomcat.keystore -alias tomcat -keyalg RSA -keysize 2048 -validity 5000 -dn "CN=localhost, OU=kedacom, O=kedacom, L=shanghai, ST=shanghai, C=cn" -ext "SAN=DNS:localhost,IP:172.16.64.59" -ext "BC=ca:true"  
# keytool -keystore tomcat.keystore -export -alias tomcat -file tomcat.cer  
https.port=8443  
https.ssl.key.store=tomcat.keystore  
https.ssl.key.store.password=tomcat  
https.ssl.key.alias=tomcat  
  
#u2f  
u2f.appId=https://localhost:8443  
  
logging.file=log/pwms.log  
logging.level.root=info
```

图 1

如果读者使用该项目的代码构建自己的站点时，一定要注意保证 application.properties 文件中“u2f.appId”，tomcat.keystore 中 CN,SAN 对域名(机器名)的一致性。

## 2.4. 如何实现用户数据的持久化

该项目中未实现用户数据的磁盘持久化，这意味着服务器一重启，之前保存的用户数据都将丢失。但要实现持久化对于有兴趣的读者而言也是非常简单的事情，项目中对于用户数据的操作是使用 com.kedacom.u2f.users.IUserStore 实现的，系统启动时注入该接口的实现对象，目前项目代码中使用的是 com.kedacom.u2f.users.UsersStoreInmemory 对象注入的。读者只需将实现 IUserStore 的自定义持久化对象替代 UsersStoreInmemory 注入即可。

## 2.5. 站点功能

该项目站点 DEMO 提供完整的用户和 U2F 设备管理功能，提供用户的增加删除修改，U2F 设备的注册绑定和鉴权等功能。

### 2.5.1. 用户管理功能

站点启动时已经缺省生成了 admin 用户，可使用“admin/admin”的初始用户名和密码登录。图 2 展示了用户的增加、删除和修改密码功能。

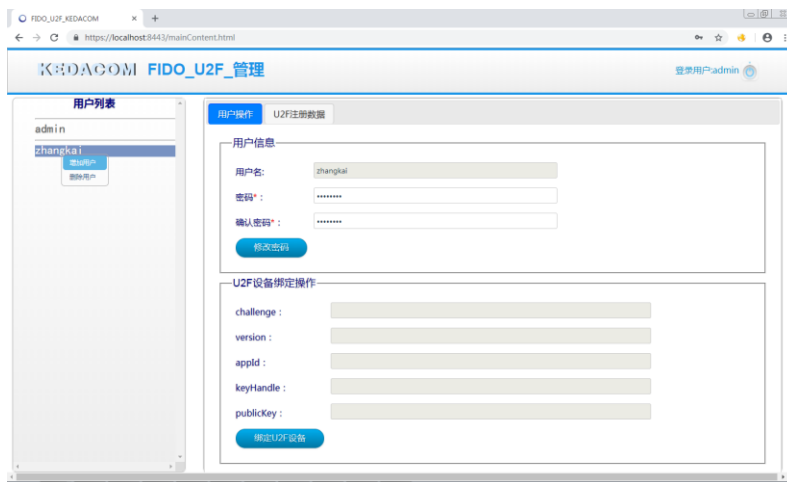


图 2

## 2.5.2. 绑定、解绑 U2F 设备

该站点中一个用户可绑定（注册）多个 U2F 设备，对同一个 U2F 设备不可绑定两次。而同一个 U2F 设备可被多个用户绑定。

图 3 显示了一个用户的设备绑定过程，站点在绑定时会提示用户触摸设备。

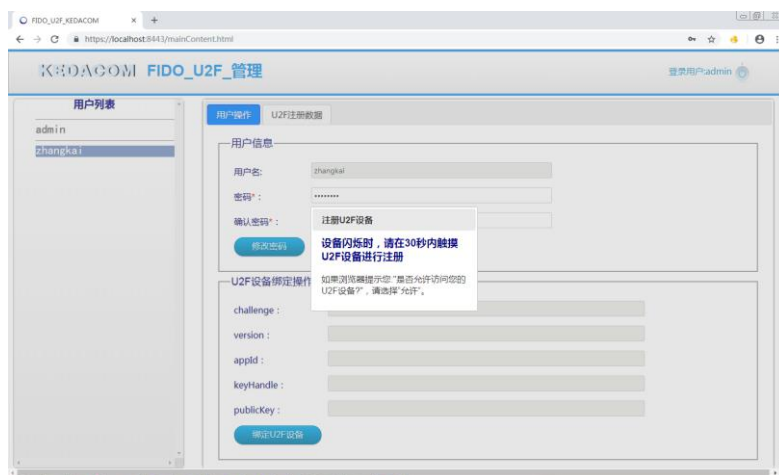


图 3

图 4 显示了设备绑定后的数据。



图 4

图 5 显示了一个用户绑定多个设备的注册数据，每个设备的绑定数据以 keyHandle 作为标识。

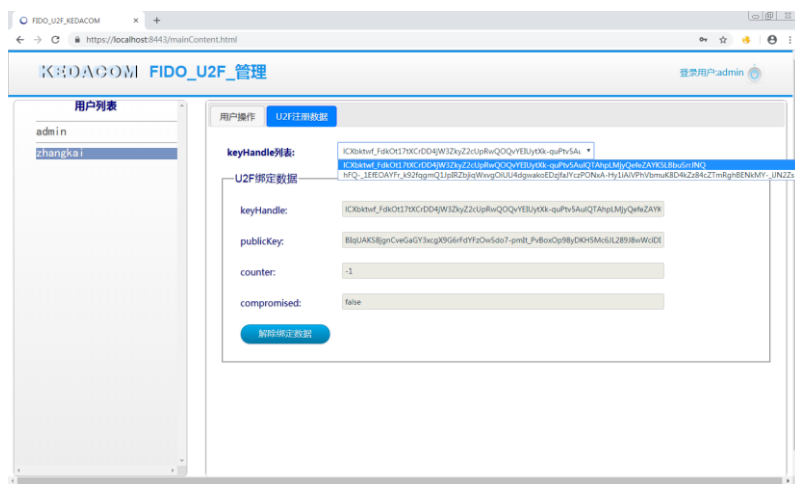


图 5

可选择其中的一个 keyHandle 解除绑定，该 keyHandle 对应的 U2F 设备在登录鉴权时将不再起作用，如图 6 所示。

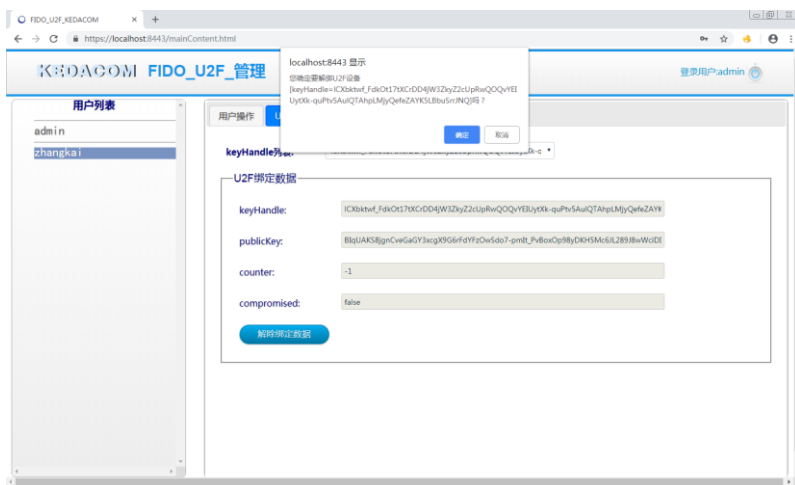


图 6

### 2.5.3. 用户登录鉴权

对于绑定了 U2F 设备的用户，在登录时不仅要校验用户名和密码，还需要验证 U2F 设备，如图 7 所示。

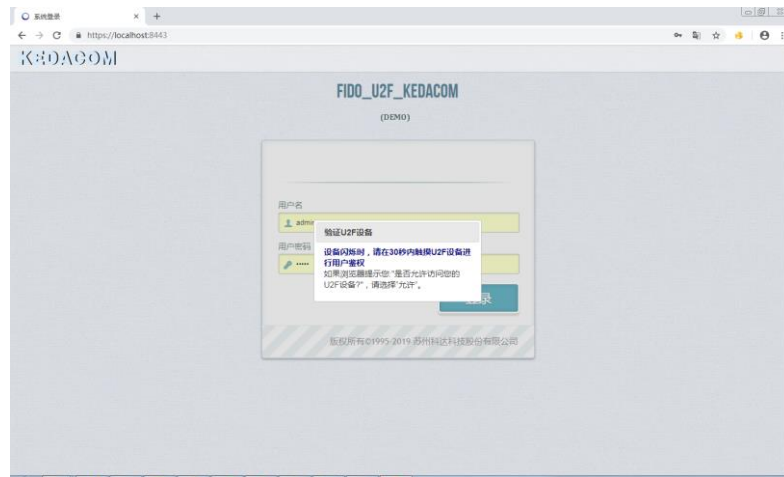


图 7

## 2.6. 使用的第三方库

该项目在客户端使用的 U2F 签名和注册接口脚本均来自 <https://demo.yubico.com/js/u2f-api.js>，这个在前面的文档中介绍后，这里不再赘述。

服务端使用了 yubico 提供的 u2flib-server-core 和 u2flib-server-attestation 这两个库，可在 pom 文件中增加如下依赖：

```
<dependency>
    <groupId>com.yubico</groupId>
    <artifactId>u2flib-server-core</artifactId>
    <version>0.19.0</version>
</dependency>

<dependency>
    <groupId>com.yubico</groupId>
    <artifactId>u2flib-server-attestation</artifactId>
    <version>0.19.0</version>
</dependency>
```

这两个库完成 U2F 设备注册信息中证书的验证、公钥的提取、签名的验证等功能，其核心类为 com.yubico.u2f.U2F 类，引用了 java.security 相关的包和类，代码值得一读。

### 3. 参考文献

1. <https://fidoalliance.org/how-fido-works/>
2. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/FIDO-U2F-COMPLETE-v1.2-ps-20170411.pdf>
3. <http://www.ietf.org/rfc/rfc4648.txt>