○ WEB API (つづき)

- プログラム言語へのラッパー
上記のリクエストと応答をプログラムが処理するためには
  JSON / XML とプログラム言語固有のデータ構造との相互変換が必要
そのためのライブラリが Wrapper

例: Twitter4j (http://twitter4j.org/ja/index.html)
Twitter API を Java言語から利用するための wrapper
Twitter が提供するデータ構造に対応したクラスの定義
  リクエストを発行するためのメソッドなど

ブラウザ上ではない, 独立したアプリケーションからでもWEBサービスの利用が可能
  ex. さまざまなTwitterアプリケーション

最近はWEBサービスの利用頻度を制限するために, 登録キーを必須にして,
それを基準にAPI利用を制限することが多くなってきた

- javascript から WEB API を利用してみる

-- ブラウザから
http://pr.ice.uec.ac.jp/~terada/learn/javascript/gethttp/ に
以下の二つのファイルを置く
=== run-pr.html ===
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0.1//EN" "http://www.w3.org/TR/html4/strict.dtd">

<html lang="ja">
<head>
<meta http-equiv="Content-Type" Content="text/html;charset=utf8">
<meta http-equiv="Content-Script-Type" content="text/javascript">

<title>gethttp テスト</title>

<script src="gethttp-pr.js"></script>

</head>
<body>

<h1>gethttp テスト</h1>

<form>
<input type="button" value="run" onClick="run()">
</form>

<div id="result"></div>

</body>
</html>
```
=== gethttp-pr.js ===
```
function run(){

var URL = "http://pr.ice.uec.ac.jp/index.html";

    var req = new XMLHttpRequest();
    req.open("GET", URL);
    req.addEventListener("load",
         function(event){
             console.log(event);
             document.getElementById("result").innerHTML =
                event.target.responseText;
         });
    req.send();
}
```
==========

javascriptでは、XMLHttpRequestというクラスのオブジェクトが
httpによるコンテンツの取得を行う。
取得が完了すると addEventListener で登録してある関数が

呼び出される（コールバックされる）仕組みになっている

ブラウザで runボタンを押すとjavascriptで定義した関数run()が動き、
上記の方法でURLの内容を取得する。

だが、ブラウザによるこの方法は、取り出す側（上記の run-pr.html）と、
取り出される対象（上記のURL）が同じサーバに存在する必要がある。
これはブラウザによる制限で、ユーザが気づかないうちに別のサイトへの
WEB API 呼び出しを行わないように保護するためである。
（cross-site request forgeries という）

つまり、ブラウザから WEB API 呼び出しを行えるのは
対象サイトにあるページからだけということになる。

-- 独立したjavascriptプログラムから

node.js という、ブラウザからは独立して javascript プログラムを
動作させる仕組みがある。
これを利用すると、前述の制限はかからないので、たとえば Google API を
手元から呼び出して結果を利用することができる。

前回の資料にあった Google Places API を取得してみる。

```
=== gethttp_node.js ===
var https=require('https');

function run(){

var output="json";
var key="検索に必要となるキー";
var location="35.652191,139.543945";
var radius="100";
var sensor="false";

var URL = "https://maps.googleapis.com/maps/api/place/nearbysearch/" +
    output +
    "?location=" + location +
    "&radius=" + radius +
    "&key=" + key +
    "&sensor=" + sensor;   // APIの引数をGETメソッドのパラメータにエンコード

    var req = https.get(URL, function(res){
        var body = "";
        res.on('data', function(chunk){       // APIの結果が順次到着
            body += chunk;
        });
        res.on('end', function(res){          // 取得完了
            var result = JSON.parse(body);    // 結果のJSONを分解
            var results = result['results'];  // そこから店の配列をとりだし
            results.forEach(function(r){
                console.log(r['name']);       // 店名をプリント
            });
        });
    });
}

run();
=====================
```

（追記）
たとえば学内LANからのように、プロキシを経由しないと接続できない&https:の場合、
上記のコードは動かない。プロキシ経由で行うには以下のようにすると
動作するようである。

```
=== gethttp_node.js ===
var request=require('request');

function run(){
```

```
    var output="json";
    var key="検索に必要となるキー";
    var location="35.652191,139.543945";
    var radius="100";
    var sensor="false";

    var URL = "https://maps.googleapis.com/maps/api/place/nearbysearch/" +
        output +
        "?location=" + location +
        "&radius=" + radius +
        "&key=" + key +
        "&sensor=" + sensor;

    request({
        url: URL,
        proxy: 'http://proxy.uec.ac.jp:8080'
    }, function (error, response, body) {
        if(error){
            console.log(error);
        } else {
            var result = JSON.parse(body);
            var results = result['results'];
            results.forEach(function(r){
                console.log(r['name']);
            });
        }
    });
}

run();
=======================

------------------------------------------------------------
HTTP での認証 (authentication)

static なページでも使える認証 (HTTPで実装されている)
Basic認証 と Digest認証 がある

○ Basic認証

ディレクトリごとに, アクセスを許可するユーザ名とパスワードを設定
クライアントからアクセスがあったときは, それらを要求
正しければアクセスを認める

Apache(ウェブサーバ) での例

---- .htaccess ---- (対象ディレクトリに置くファイル)
AuthType Basic
AuthName      "Authentication required"
AuthUserFile  /home/terada/FS/public_html/learn/auth/basic/.htpasswd
AuthGroupFile /dev/null
Require       valid-user

---- .htpasswd ---- (.htaccess で指定した場所に置くファイル)
testuser:$apr1$t9dvp/..$1dfGG8ZhVnOEsZtOY15qw0

ユーザ名と, パスワードをハッシュした文字列を格納 (この例では md5 ハッシュ関数を利用)

通信の実行例

(1) client -> server (保護されているディレクトリにあるファイルを要求)
0:GET http://pr.ice.uec.ac.jp/~terada/learn/auth/basic/index.html HTTP/1.1
  後略

(2) server -> client (認証できないのでアクセス拒否 401)
1:HTTP/1.1 401 Unauthorized
1:Date: Thu, 22 Oct 2015 06:13:18 GMT
```

```
1:Server: Apache
1:WWW-Authenticate: Basic realm="Authentication required" (ここでBasic認証が必要であるとを
示す)
  中略
1:Connection: keep-alive
1:
1:<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
1:<html><head>
1:<title>401 Authorization Required</title>
1:</head><body>
1: ... (ここは人間向けの「認証が必要」というメッセージのHTML)
1:</body></html>

(3) ブラウザはユーザ名とパスワードを入力するためのポップアップを表示する
(4) 人間がそれらを入力すると, client はふたたび server に GET を送る
0:GET http://pr.ice.uec.ac.jp/~terada/learn/auth/basic/index.html HTTP/1.1
0:Host: pr.ice.uec.ac.jp
  中略
0:Authorization: Basic dGVzdHVzZXI6dGVzdHBhc3N3b3Jk (ここが人間が入力した認証情報)
0:

(5) server -> client (認証に成功したのでコンテンツを返信する)
1:HTTP/1.1 200 OK
  中略
1:Connection: keep-alive
1:
1:<html>
1:<head>
1:</head>
1:<body>
1:Content (index.html)
1:</body>
1:</html>

ステップ(4)で client は認証情報を送っているが,
この情報は base64 encoding であって, 容易に復号可能
  % echo -n dGVzdHVzZXI6dGVzdHBhc3N3b3Jk | base64 -d
  testuser:testpassword
httpの通信を傍受されると, パスワードが見えてしまう欠点

○ Digest認証

人間の入力したパスワードを, 平文ではなくハッシュ関数を使って暗号化して送る方式
利用者から見ると, ブラウザの振舞はBasic認証のときと同じ

---- .htdigest ---- (ユーザ名, realm(認証領域) と, MD5でハッシュしたパスワード)
testuser:Learn Digest:ed5e2a4618444dba2be6c5496957d2cc

通信の実行例

(1) client -> server (保護されているディレクトリにあるファイルを要求)
0:GET http://pr.ice.uec.ac.jp/~terada/learn/auth/digest/index.html HTTP/1.1
  後略

(2) server -> client (認証できないのでアクセス拒否 401)
3:HTTP/1.1 401 Unauthorized
3:Date: Thu, 22 Oct 2015 07:08:05 GMT
3:Server: Apache
3:WWW-Authenticate: Digest realm="Learn Digest", nonce="mR0SKKwiBQA=23b14f0da5e1cdc9d16ae39
6a0bb8da3730039ce", algorithm=MD5, domain="/~terada/learn/auth/digest/", qop="auth"
  後略

このとき, サーバは要求する認証情報に関するデータを送ってくる
nonce : ランダムな文字列を生成したもの

(3) ブラウザはユーザ名とパスワードを入力するためのポップアップを表示する
(4) 人間がそれらを入力すると, client はふたたび server に GET を送る
2:GET http://pr.ice.uec.ac.jp/~terada/learn/auth/digest/index.html HTTP/1.1
```

```
2:Host: pr.ice.uec.ac.jp
  中略
2:Authorization: Digest username="testuser", realm="Learn Digest", nonce="mR0SKKwiBQA=23b14
f0da5e1cdc9d16ae396a0bb8da3730039ce", uri="/~terada/learn/auth/digest/index.html", algorith
m=MD5, response="9783d6d1cc4493ddbd5375ef7c6dcccc", qop=auth, nc=00000001, cnonce="9daa8c29
15cbeaeb"
2:
```

client は,
  人間が入力したユーザ名
  人間が入力したパスワード
  サーバからの nonce
  クライアントが作成した cnonce (ランダム文字列)
を暗号化して, response としてサーバに送る

サーバは 保存されている .htdigest から正しいresponseを求め,
response と比較する. 一致したら認証成功.

(5) server -> client (認証に成功したのでコンテンツを返信する)
```
3:HTTP/1.1 200 OK
  後略
```

クライアントによる response の計算を実際に計算してみる (RFC 2617)
A1 = testuser:Learn Digest:testpassword (人間の入力内容)
A2 = GET:/~terada/learn/auth/digest/index.html

```
MD5(A1) ed5e2a4618444dba2be6c5496957d2cc
nonce   mR0SKKwiBQA=23b14f0da5e1cdc9d16ae396a0bb8da3730039ce (サーバから入手してある)
nc      00000001
cnonce  9daa8c2915cbeaeb (クライアントが生成)
qop     auth
MD5(A2) 69cb0936b11dbc72ef80069036a89fdf
```

```
response = MD5(MD5(A1) ":" nonce ":" nc ":" cnonce ":"  qop ":" MD5(A2))
MD5(ed5e2a4618444dba2be6c5496957d2cc:mR0SKKwiBQA=23b14f0da5e1cdc9d16ae396a0bb8da3730039ce:0
0000001:9daa8c2915cbeaeb:auth:69cb0936b11dbc72ef80069036a89fdf)

9783d6d1cc4493ddbd5375ef7c6dcccc
```

これをサーバは同じ方法で正解をもとに計算して比較する

- ネットワーク上を平文のパスワードが流れない
- 平文のパスワードは人間の頭の中だけにあればよい

cnonceの必要性
attackerが通信(server->client)を横取りして、常に同じnonceを送るようにすると、
attackerは辞書攻撃によって複数のパスワードを少ない労力で破れることになる。
（その唯一nonceで辞書をハッシュしておき、返信のハッシュと比較する）

OAuth

ウェブサービスで, データへのアクセス権限の一部を移譲するための仕組み (認可)

3人の登場人物
- Service Provider
  データを保持しているサービス (例: Twitter)
- User
  Service Provider の利用者. 自分の認証情報を持っている.
- Consumer
  ユーザの代わりにService Providerの一部の機能を実行するサービス.
  そのためにユーザのアクセス権限の一部が必要.
    ウェブサービスの場合 (例: Twitterに連携した各種サービス)
    独立したアプリケーションの場合 (例: Twitterクライアント)

認証情報(パスワードなど)をConsumerに渡すのは望ましくない
  「全権委任」になる
    機能の一部だけを移譲することができない (例: タイムラインの読み出し)
    最悪の場合, パスワード変更でアカウント乗っ取り

認可の取消しが難しい

手順
0. Consumer は事前に Service Provider に登録しておく
    Consumer Key, Consumer Secret
1. User が Consumer にアクセス
2. Consumer は User を Service Provider にリダイレクト
3. User は Service Provider にログインし, Consumer の要求するアクセスを認可
4. 認可情報が Consumer へ渡る
5. Consumer は認可情報と引き換えにアクセストークンを入手する
    Access Token, Access Token Secret
6. Consumer は Service Provider を利用
    Consumer Key, Consumer Secret, Access Token, Access Token Secret

---