

# Web 2.0

## Lecture 6: Accessing and Utilizing Services

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/w20>



Modified: Fri Mar 17 2017, 12:52:11  
Humla v0.3

## Overview

- **Mashups and XHR**
- Security Mechanisms
- JSON and JSONP

## Mashups

- Web application hybrid/Web 2.0 application
  - *Uses APIs of two or more applications to provide new value-added functionality*
- Types
  - *Data mashup – integration/aggregation of data (read only)*
  - *Service mashup – more sophisticated workflows (read, write)*
  - *Visualization – involves UI, e.g., third-party data displayed on the Google map*
- Client-Server View
  - *client-side mashups (mainly in a browser)*
    - *JavaScript, Dynamic HTML, AJAX, JSON/JSONP*
  - *server-side mashup*
    - *server-side integration of services and data*
    - *third-party programming languages, very typical, nothing new*
    - *specialized environments: Google AppScript, Yahoo Pipes*
- Web Apps developments will all be about mashups!

## XMLHttpRequest (XHR)

- Interface to utilize HTTP protocol in JavaScript
  - *standardized by Web Applications WG [🔗](#) at W3C*
  - *basis for AJAX*
    - *Asynchronous JavaScript and XML*
- Typical usage
  1. *Browser loads a page that includes a script*
  2. *User clicks on a HTML element*
    - *it triggers a JavaScript function*
  3. *The function invokes a service through XHR*
    - *same origin policy, cross-origin resource sharing*
  4. *The function receives data and modifies HTML in the page*

## XHR Interface – Key Methods and Properties

- Method and properties of XHR object
  - **open**, *opens the request, parameters:*
    - method** – method to be used (e.g. GET, PUT, POST),
    - url** – url of the resource,
    - asynch** – true to make asynchronous call,
    - user, pass** – credentials for authentication.
  - **onReadyStateChange** – JavaScript function object, it is called when **readyState** changes (uninitialized, loading, loaded, interactive, completed).
  - **send, abort** – sends or aborts the request (for asynchronous calls)
  - **status, statusText** – HTTP status code and a corresponding text.
  - **responseText, responseXML** – response as text or as a DOM document (if possible).
  - **onload** – event listener to support server push.
- See XMLHttpRequest (W3C) [🔗](#), or XMLHttpRequest (Mozilla reference) [🔗](#) for a complete reference.

## How XHR works

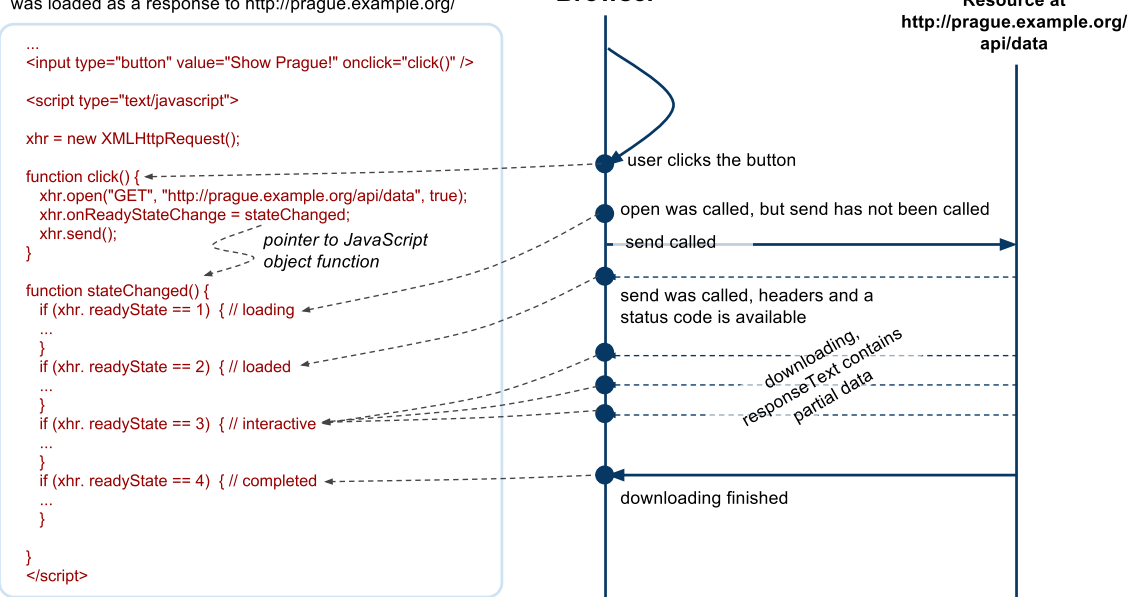
### HTML with JavaScript code

was loaded as a response to <http://prague.example.org/>

```
...  
<input type="button" value="Show Prague!" onclick="click()" />  
<script type="text/javascript">  
  xhr = new XMLHttpRequest();  
  
  function click() {  
    xhr.open("GET", "http://prague.example.org/api/data", true);  
    xhr.onreadystatechange = stateChanged;  
    xhr.send();  
  }  
  
  function stateChanged() {  
    if (xhr.readyState == 1) { // loading  
      ...  
    }  
    if (xhr.readyState == 2) { // loaded  
      ...  
    }  
    if (xhr.readyState == 3) { // interactive  
      ...  
    }  
    if (xhr.readyState == 4) { // completed  
      ...  
    }  
  }  
</script>
```

### Browser

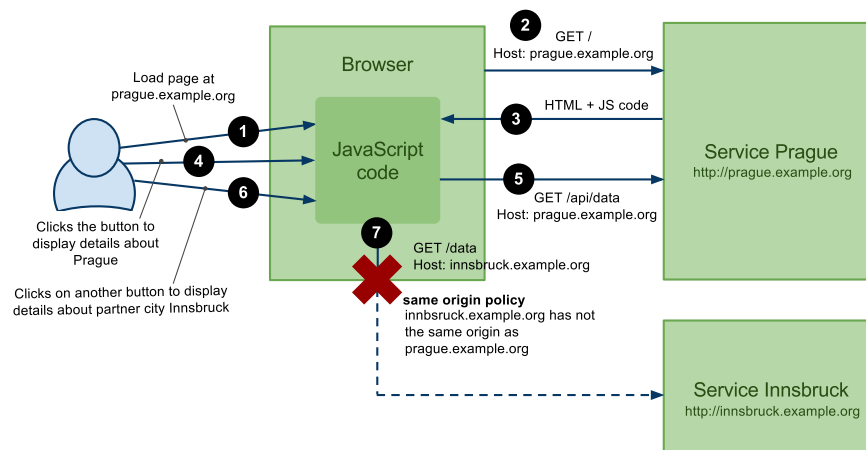
Resource at  
<http://prague.example.org/api/data>



## Overview

- Mashups and XHR
- **Security Mechanisms**
  - *Scripting Attacks*
  - *Cross-origin Resource Sharing Protocol (CORS)*
- JSON and JSONP

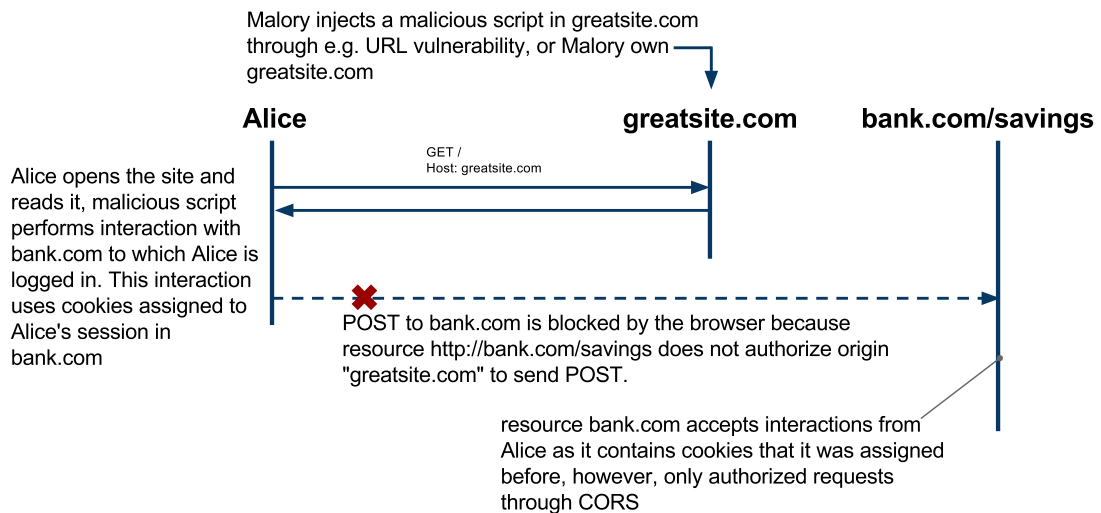
## Same Origin Policy



- JavaScript code can only access resources on the same domain
  - *XHR to GET, POST, PUT, UPDATE, DELETE*
  - *Browsers apply same origin policy*
- Solutions
  - *JSON and JSONP (GET only)*
  - *Cross-origin Resource Sharing Protocol (CORS)*

## Why Same Origin Policy?

- Without the same origin policy, the following POST would be possible



## Overview

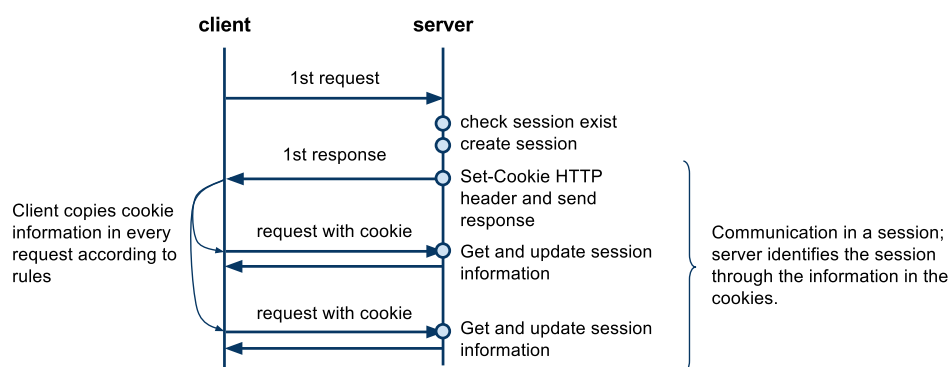
- Mashups and XHR
- Security Mechanisms
  - Scripting Attacks*
  - Cross-origin Resource Sharing Protocol (CORS)*
- JSON and JSONP

## Overview

- Scripting Attacks
  - Intruders make users perform action that has side effects on their resources
  - Intruders inject malicious code to Web pages
- Roles in Security Scenarios
  - Alice, Bob
    - Normal users, usually Alices wants to send a message to Bob or Alice accesses a Bob's site.
  - Eve
    - A user with bad intentions, usually a passive attacker.
  - Mallory
    - An active attacker, usually sends a link to a page with malicious code.

## Recall: State management in HTTP

- Request-response interaction with cookies
  - Session is a logical channel maintained by the server



- Stateful Server
  - Server remembers the session information in a server memory
  - Server memory is a non-persistent storage, when server restarts the memory content is lost!

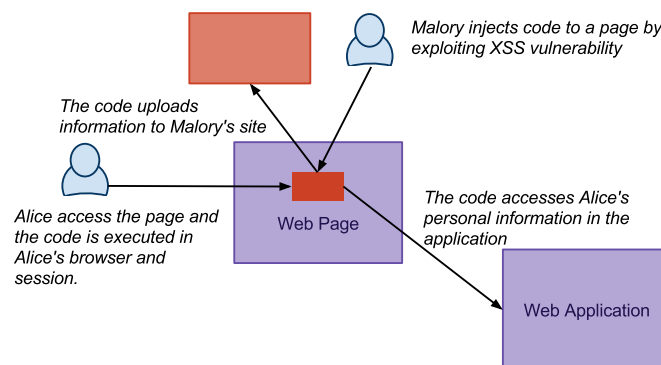
## Cross-site Request Forgery (CSRF)

- Exploits a trust of a website in a user's browser
- Scenario
  1. Mallory sends a link to Alice (in an email, in a chat, etc.)
    - The link points to a page that has HTML code with hrefs to Alice's private resources
    - For example, to perform an action on Alice's account, it is possible to use `img` like this:

```
1 | 
```
  2. Alice loads the page in her browser
    - Alice is authenticated to the bank's website, the browser sends Alice's authentication cookies with the request.
- Issues and Prevention
  - The bank site violates REST, i.e. overloading of GET for making actions
  - The bank should check HTTP `referer` header
  - It is a "blind" attack, Mallory does not see the result
  - To perform POST, current browsers today use *CORS protocol*

## Cross-site Scripting Attack (XSS)

- Exploits a trust of a user in a website



- Example Scenario
  1. An attacker injects a code to a page
  2. A users executes the code in his/her browser's session
  3. The code provides information (cookies) to the attacker
  4. The attacker uses the cookies to access the user's data

## XSS Examples

- Twitter in Sep 2010

- *Injection of JavaScript code to a page using a tweet*
- *You posted following tweet to Twitter*

```
1 | There is a great event happening at
2 | http://someurl.com/@"onmouseover="alert('test xss')"/
```

- *Twitter parses the link and wraps it with `<a>` element*

```
1 | There is a great event happening at
2 | <a href="http://someurl.com/@"onmouseover="alert('test xss')"  
3 |   target="_blank">http://someurl.com/@"onmouseover=  
4 |   "alert('test xss')"/</a>
```

- *See details at Twitter mouseover exploit [↗](#)*

- Other example: Google Contacts

## Overview

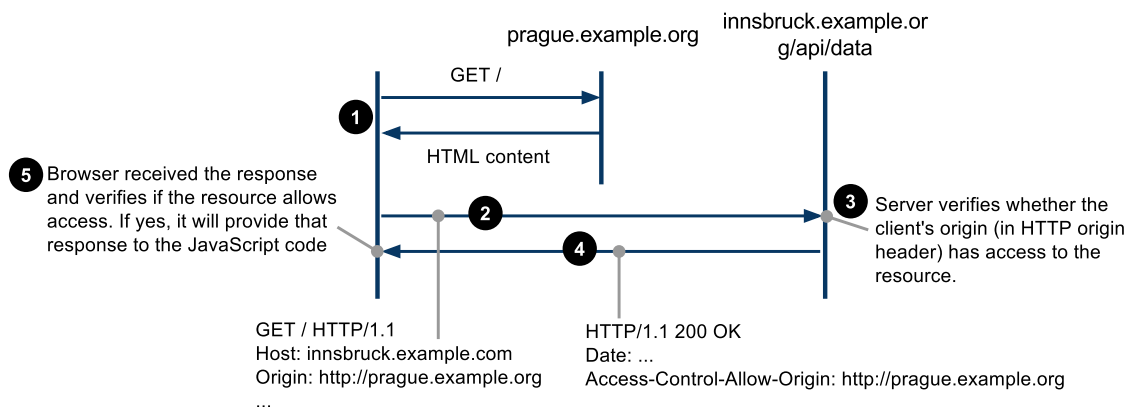
- Mashups and XHR
- Security Mechanisms
  - *Scripting Attacks*
  - *Cross-origin Resource Sharing Protocol (CORS)*
- JSON and JSONP



## Overview

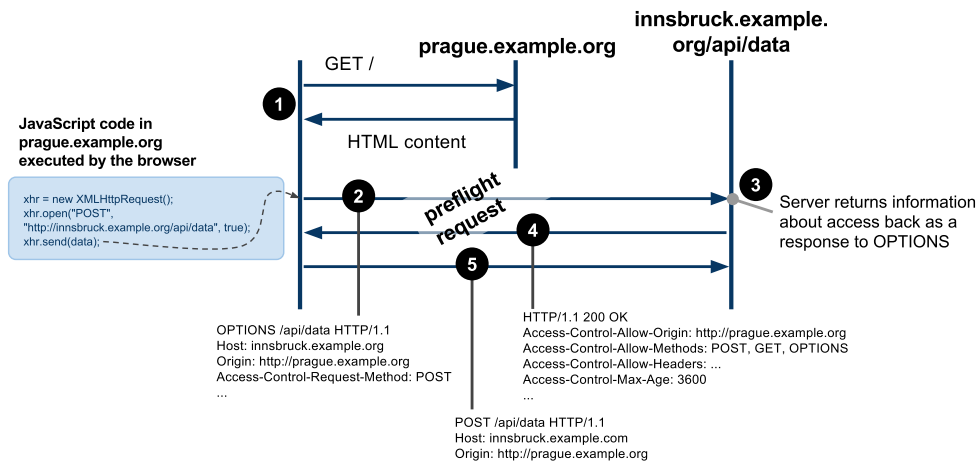
- Increasing number of mashup applications
  - *client-side mashups involving multiple sites*
  - *mechanism to control an access to sites from within JavaScript*
- Allow for **cross-site HTTP requests**
  - *HTTP requests for resources from a different domain than the domain of the resource making the request.*
- W3C specification, working draft
  - *see Cross-origin Resource Sharing [↗](#)*
  - *already browsers support it*
    - *see HTTP Access Control [↗](#) at Mozilla*

## CORS Protocol – GET



- Read-only resource access via HTTP GET
- Headers:
  - **Origin** – identifies the origin of the request
  - **Access-Control-Allow-Origin** – defines who can access the resource
  - either the full domain name or the wildcard (\*) is allowed.

# CORS Protocol – other methods and "preflight"



- Preflight request queries the resource using **OPTIONS** method
  - requests other than `GET` (except `POST` w/o payload) or with custom headers
  - A browser should run preflight automatically for any XHR request meeting preflight conditions
  - The browser caches responses according to **Access-Control-Max-Age**

## Overview

- Mashups and XHR
- Security Mechanisms
- **JSON and JSONP**

## Recall: JSON

- JSON = JavaScript Object Notation
  - *Serialization format for data representation*
  - *Very easy to use in JavaScript*
    - *no need to use a parser explicitly*
  - *Also great support in many programming environments*
- Key constructs
  - **object** is a collection of comma-separated key/value pairs:  
`{"name" : "tomas", "age" : 18, "student" : false, "car" : null}`
  - **array** is an order list of values:  
`[ "prague", "innsbruck", 45 ]`
  - can be nested: objects as values in an **array**:  
`[ { "name" : "tomas", "age" : 18 },  
 { "name" : "peter", "age" : 19 } ]`
  - and the other way around: array as values in an **object**:  
`{ "cities" : [ "prague", "innsbruck",  
 "states" : [ "CZ", "AT" ] }`
  - A complete grammar see *JavaScript Object Notation* [↗](#)

## JSON in JavaScript

- Native data format

```
1 // data needs to be assigned
2 var data = { "people" : ["tomas", "peter", "alice", "jana"] };
3
4 // go through the list of people
5 for (i = 0; i < data.people.length; i++) {
6   man = data.people[i];
7   // ... do something with this man
8 }
```
- Responses of service calls in JSON
  - *Many support JSON, how can we load that data?*
- Example Request-Response

```
1 GET http://pipes.yahoo.com/pipes/pipe.run?_id=638c670c40c97b62&_render=json
2
3 {
4   "count":1,"value":
5   {
6     "title":"Web 2.0 announcements",
7     "description":"Pipes Output",
8     "link":"http://pipes.yahoo.com/pipes/pipe.info...",
9     "pubDate":"Mon, 07 Mar 2011 18:27:20 +0000",
10    "generator":"..."
11  }
12 }
```

# JSONP

- Service that supports JSONP
  - allows to specify a query string parameter for a wrapper function to load the data in JavaScript code
  - otherwise the data cannot be used in JavaScript
    - they're loaded into the memory but assigned to nothing
- Example
  - if a resource at [http://someurl.org/json\\_data](http://someurl.org/json_data) returns

```
{ "people" : [ "tomas", "peter", "alice", "jana" ] }
```
  - then the resource at [http://someurl.org/json\\_data?\\_callback=loadData](http://someurl.org/json_data?_callback=loadData) returns

```
loadData({ "people" : [ "tomas", "peter", "alice", "jana" ] });
```
- A kind of workaround for the same origin policy
  - only **GET**, nothing else works obviously
  - no XHR, need to load the data through the dynamic **<script>** element

## JSONP in JavaScript

- JSONP example
  - loads JSON data using JSONP by dynamically inserting **<script>** into the current document. This will download JSON data and triggers the script.

```
1  var TWITTER_URL = "http://api.twitter.com/1/statuses/user_timeline.json?" +
2    "&screen_name=web2e&count=100&callback=loadData";
3
4  // this needs to be loaded in window.onload
5  // after all document has finished loading...
6  function insertData() {
7    var se = document.createElement('script');
8    se.setAttribute("type", "text/javascript");
9    se.setAttribute("src", TWITTER_URL);
10   document.getElementsByTagName("head")[0].appendChild(se);
11   // And data will be loaded when loadDta callback fires...
12 }
13
14 // loads the data when they arrive
15 function loadData(data) {
16   // we need to know the the structure of JSON data that is returned
17   // and code it here accordingly
18   for (i = 0; i < data.length; i++) {
19     data[i].created_at // contains date the tweet was created
20     data[i].text // contains the tweet
21   }
22 }
```

# JSON Vulnerability

- What it is
  - JSON array data accessible via GET (normal access is via XHR)
  - Attacker may load the data in a **script**, redefine **Array** object, and assign the data to a variable.
  - Attacker's page with a script that you access:
    - your browser uses your cookies to load the resource

```
1 | <script type="text/javascript">
2 |   var yourData;
3 |   Array = function() {
4 |     yourData = this;
5 |   }
6 | </script>
7 |
8 | <script src="yourdomain.cz/yoursecretdata.json" type="text/javascript"></script>
```

- Prevention
  - Using prefix in the data – the prefix makes the JSON data invalid; the client must strip the prefix before parsing the data as JSON

```
1 | [ "a": "account", 433, 5543 ]
```
  - Use only POST for sensitive data