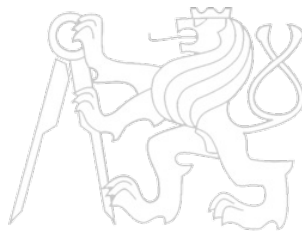


Middleware Architectures 2

Lecture 2: Browser Networking

doc. Ing. Tomáš Vitvar, Ph.D.

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



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <https://vitvar.com/lectures>



Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Modified: Sun Mar 10 2024, 21:33:02
Humla v1.0

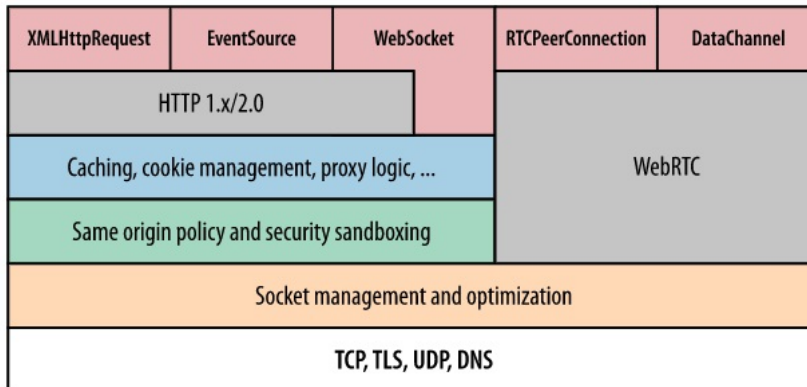
Overview

- **Browser Networking**
 - *XHR*
 - *Fetch API*
- Security Mechanisms
- JSON and JSONP

Browser Networking

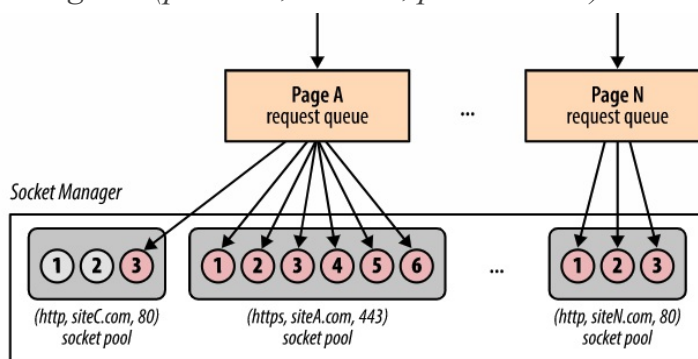
- Browser

- Platform for fast, efficient and secure delivery of Web apps
- Many components
 - parsing, layout, style calculation of HTML and CSS, JavaScript execution speed, rendering pipelines, and **networking stack**
- When network is slow, e.g. waiting for a resource to arrive
 - all other steps are blocked



Connection Management

- Network socket management and optimization
 - Socket reuse
 - Request prioritization
 - Protocol negotiation
 - Enforcing connection limits
- Socket manager
 - Sockets organized in pools (connection limits and security constraints)
 - origin = (protocol, domain, port number)



Network Security

- No raw socket access for app code
 - Prevents apps from initiating any connection to host
 - For example port scan, connect to mail server, etc.
- Network security
 - **Connection limits**
 - protect both client and server from resource exhaustion
 - **Request formatting and response processing**
 - Enforcing well-formed protocol semantics of outgoing requests
 - Response decoding to protect user from malicious servers
 - **TLS negotiation**
 - TLS handshake and verification checks on certificates
 - User is warned when verification fails, e.g. self-signed cert is used
 - **Same-origin policy**
 - Constraints on requests to be initiated and to which origin

Mashups

- Web application hybrid
 - App uses APIs of two or more applications
- Types
 - Data mashup – integration/aggregation of data (read only)
 - Service mashup – more sophisticated workflows (read, write)
 - Visualization – involves UI
 - For example, third-party data displayed on the Google map
- Client-Server View
 - client-side mashups (in a browser)
 - JavaScript, Dynamic HTML, AJAX, JSON/JSONP
 - server-side mashups
 - server-side integration of services and data
 - Any language

Overview

- Browser Networking
 - *XHR*
 - *Fetch API*
- Security Mechanisms
- JSON and JSONP

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

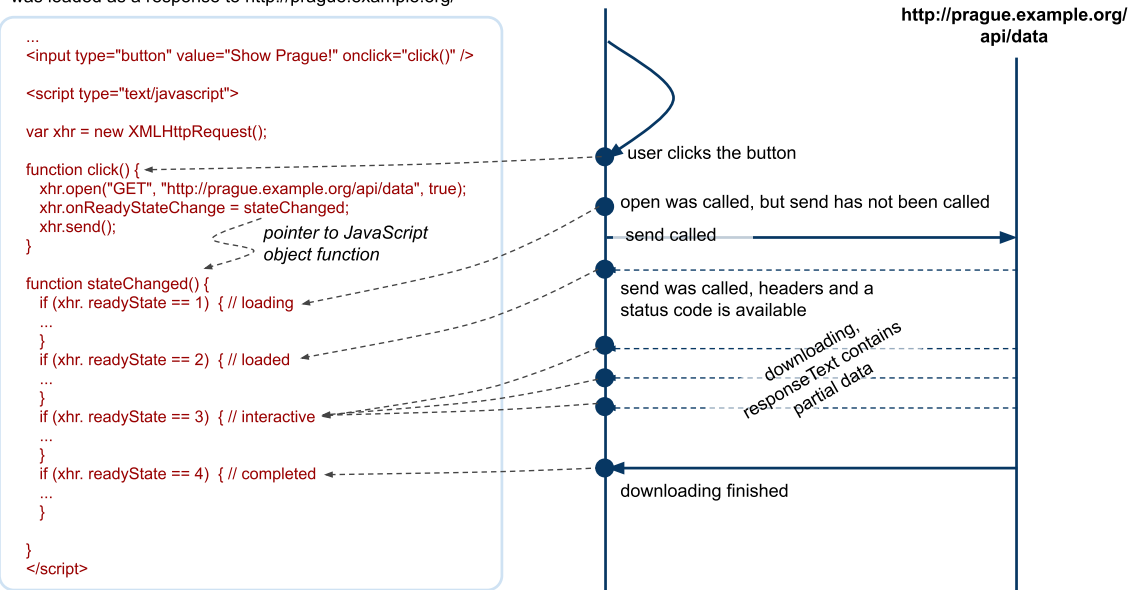
HTML with JavaScript code

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

```
...  
<input type="button" value="Show Prague!" onclick="click()" />  
<script type="text/javascript">  
var 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

- Browser Networking
 - *XHR*
 - *Fetch API*
- Security Mechanisms
- JSON and JSONP

Fetch API

- XHR is callback-based, Fetch is promise-based
- Interface to accessing requests and responses
 - Provides global **fetch** method to fetch resources asynchronously
 - Can be easily used in service workers
 - Supports CORS and other extensions to HTTP
- Interfaces
 - **Request** – represents a request to be made
 - **Response** – represents a response to a request
 - **Headers** – represents response/request headers
- Basic usage:

```
1  async function logMovies() {
2      const response = await fetch("http://example.com/movies.json");
3      const movies = await response.json();
4      console.log(movies);
5  }
```

Making request

- A **fetch** function is available in global **window**

- It takes **path** and returns **Promise**

```
1 fetch('https://api.github.com/users/tomvit')
2   .then(response => response.json())
3   .then(data => console.log(data))
4   .catch(error => console.error('Error:', error));
```

- You can make **no-cors** request

– *With Fetch, the request will be handled as with putting **src** to **img***

```
1 fetch('https://google.com', {
2   mode: 'no-cors',
3 }).then(function (response) {
4   console.log(response.type);
5 });
```

- You can access low-level body stream

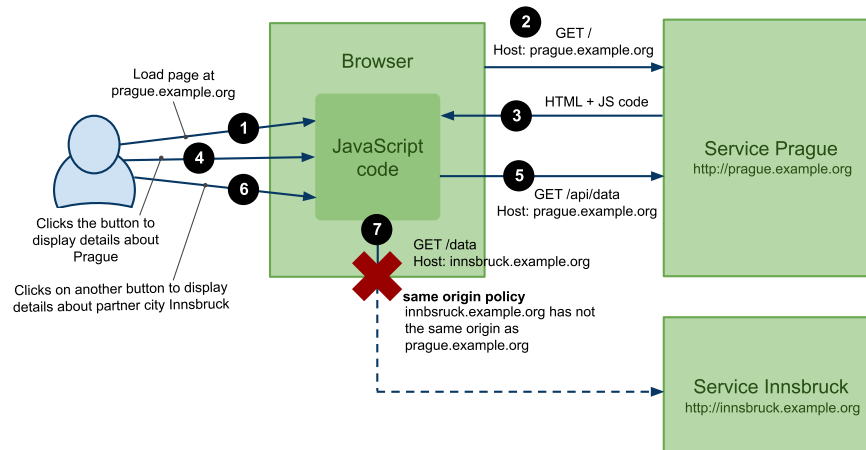
– *With XHR, the whole **responseText** would be loaded into memory.*

– *With Fetch, you can read chunks of response and cancel the stream when needed.*

Overview

- Browser Networking
- **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)*

Overview

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

Overview of Web Threats

- Scripting/Web attacks
 - *CSRF*: attacker causes a user's browser to send an authenticated request
 - *XSS*: attacker gets script to run in a trusted origin (steal data / act as user)
 - *Clickjacking*: victim is tricked into clicking UI elements
- Modern context
 - *Single Page Apps (SPAs), APIs, OAuth logins, third-party scripts*
 - *Browsers added new defenses: SameSite cookies, CSP, CORS, Trusted Types*
- Roles in security scenarios
 - *Alice, Bob*: regular users / sites
 - *Eve*: passive attacker, man in the middle (MITM) (observes, phishes)
 - *Mallory*: active attacker (injects content, modifies requests, drives victim browser)

Recall: State, Sessions, and Cookies

- HTTP is stateless; sites add state via tokens
 - *Cookies (session cookies, persistent cookies)*
 - *Bearer tokens (often in Authorization: Bearer ...)*
- Session management patterns
 - *Stateful*: server stores session data
 - *Stateless*: signed tokens (e.g., JWT) stored client-side; server validates each request
- Cookie flags
 - **Secure**: only over HTTPS
 - **HttpOnly**: not readable by JavaScript
 - Mitigates some XSS cookie theft, but not all attack scenarios
 - **SameSite**: limits cross-site cookie sending
 - Major CSRF mitigation

Cross-site Request Forgery (CSRF)

- How it works
 - Victim's browser sends a request to **bank.com** with user's credentials (cookies)
 - Attacker cannot read the response (SOP), but the **side effect** may still happen
- CSRF today
 - Many browsers now default to **SameSite=Lax** (reduces CSRF via cross-site navigation)
 - But CSRF is still relevant
 - misconfigured **SameSite=None**, legacy apps, subdomain issues, OAuth flows
- Typical attack shapes
 - Cross-site `<form method="POST">` submit
 - State-changing GET is still a bug, but many frameworks avoid it now
- Modern best practice
 - Use **CSRF tokens** (synchronizer token pattern) or **double-submit cookie**
 - Set cookies: **SameSite=Lax** (default) or **Strict** where possible
 - only use **None; Secure** when required
 - Validate **Origin** header for state-changing requests (often better than **Referer**)
 - Re-auth / step-up auth for high-risk actions

CSRF Example

- Attacker page causes a victim browser to submit a form to a target origin

```
1 <form action="https://bank.com/transfer" method="POST">
2   <input type="hidden" name="to" value="mallory" />
3   <input type="hidden" name="amount" value="50000" />
4 </form>
5 <script>document.forms[0].submit();</script>
```
- What makes this work?
 - If the browser includes cookies for **bank.com** on this cross-site POST
 - If server does not require or validate a CSRF token (or **Origin**)
- Why attackers like it
 - No need to read responses; only need the action to succeed

CSRF Tokens

- Goal
 - Prevent **cross-site request forgery** even when cookies are attached by the browser
 - Server accepts a state-changing request only if it contains a **secret token**
- Synchronizer Token Pattern (most common)
 - Server creates a random token bound to the user session
 - Token is embedded into HTML (form field) or sent as a header by JS
 - On POST/PUT/DELETE: server verifies token matches the session
- Why it works
 - Attacker can trigger a cross-site request (e.g., `<form>`)
 - but cannot read the response page to obtain the token (prevented by SOP)
 - Guessing a random token is not possible
- Implementation notes
 - Regenerate tokens periodically (or per form) and invalidate on logout
 - Protect token delivery with HTTPS; avoid exposing it to third-party origins
 - Use together with **SameSite** cookies and **Origin** checks

```
1 <form action="/transfer" method="POST">
2   <input type="hidden" name="csrf_token" value="RANDOM_SESSION_BOUND_TOKEN"> ...
3 </form>
```

Double-Submit Cookie

- Goal
 - Block CSRF by requiring a **token to be sent twice**:
 - as a **cookie** (automatically attached by the browser)
 - and as a **request value** (hidden form field or custom header)
 - Server accepts the request only if the two values **match**
- Typical flow
 - Server generates random token **T**
 - Browser stores **T** in cookie **csrf=T**
 - POST/PUT/DELETE: **T** in **csrf_token** (form field) or **X-CSRF-Token** (header)
 - Server verifies: `Cookie[csrf] == token_in_body_or_header`
- Why it works
 - Attacker can trigger a cross-site request that includes cookies, but cannot read victim cookies/pages due to **SOP**
 - Attacker cannot reliably include the matching token in the request body/header
- Implementation notes
 - Used for **stateless** setups (server does not store CSRF token in session)
 - Use with **Secure** and **HTTPS**; combine with **SameSite** and **Origin** checks

Cross-site Scripting (XSS)

- How XSS works
 - Attacker-controlled script runs in a trusted origin (e.g., <https://app.com>)
 - Can read/modify DOM, make same-origin requests, extract data, act as the user
- Types
 - **Stored**: payload stored on server (posts, profiles)
 - **Reflected**: payload in request (query param) reflected into HTML
 - **DOM-based**: client-side JS inserts untrusted data into DOM unsafely
- Today's prevention
 - Many apps use frameworks that reduce HTML injection
 - But DOM-XSS via unsafe sinks still happens
 - Third-party scripts and supply-chain risks can lead to "XSS-like" outcomes

XSS: What Attackers Do Today

- Common impact
 - Session hijack is harder if cookies are **HttpOnly**, but attackers can still:
 - Perform actions as the user (same-origin **fetch**)
 - Steal sensitive DOM data (CSRF tokens in DOM, PII rendered in page)
 - Keylog / credential phishing overlays
 - Token theft when apps store tokens in **localStorage**
- Mitigations
 - Output encoding + safe templating
 - Sanitize HTML if you must render it (allowlist-based sanitizers)
 - **Content Security Policy (CSP)** (reduce exploitability)
 - **Trusted Types** (Chrome/Chromium): block DOM-XSS sinks unless explicitly allowed

Content Security Policy (CSP)

- Goal
 - Mitigate XSS attacks by restricting what the page can load and execute
 - Browser enforces a policy sent by the server (HTTP header or `<meta>`)
- Core idea
 - Define allowlists for resource types (scripts, styles, images, connections, frames, ...)
 - Block unexpected code execution (e.g., injected `<script>`, inline handlers)
- Common directives
 - `script-src`: where scripts can come from; can require `'nonce-...'` / hashes
 - `connect-src`: where JS can send requests (`fetch`/XHR/WebSocket)
 - `img-src`, `style-src`, `font-src`: resource allowlists
 - `object-src 'none'`: disable plugin content
 - `base-uri`: restrict `<base>` tag abuse
 - `frame-ancestors`: who may embed the page (clickjacking defense)

Nonce-based CSP Example

- Policy (sent as HTTP response header)
 - Only load scripts from `self`
 - only execute inline scripts that carry the correct `nonce`
 - Allow XHR/WebSocket only to own origin and a trusted API
 - Prevent plugins and clickjacking

```
1 Content-Security-Policy:  
2   default-src 'self';  
3   script-src 'self' 'nonce-ABC123';  
4   connect-src 'self' https://api.example.com;  
5   object-src 'none';  
6   base-uri 'self';  
7   frame-ancestors 'none'
```

- Scripts in a page
 - server must inject the same nonce into allowed inline scripts

```
1 <script nonce="ABC123">  
2   // allowed inline script  
3   window.appBoot();  
4 </script>  
5  
6 <script>  
7   // blocked inline script (missing nonce)  
8   alert('XSS');  
9 </script>
```

XSS Example: DOM-based (Common in SPAs)

- Bug pattern: untrusted data flows into a dangerous DOM sink

```
1 // URL: https://app.com/welcome?name=<img src=x onerror=alert(1)>
2 const params = new URLSearchParams(location.search);
3 const name = params.get("name");
4
5 // Dangerous sink:
6 document.querySelector("#welcome").innerHTML = "Hi " + name;
```

- Fix

- Use `textContent` instead of `innerHTML`
- If HTML is required, sanitize with a proven library and strict allowlist

Historical 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

Quick Comparison: CSRF vs XSS

- CSRF
 - Attacker **cannot** run code in the target origin
 - Relies on browser automatically attaching credentials (cookies)
 - Defenses: **SameSite**, CSRF tokens, **Origin** checks
- XSS
 - Attacker **can** run code in the target origin
 - Bypasses CSRF defenses (because requests are same-origin)
 - Defenses: encoding/sanitization, CSP, Trusted Types, reduce secrets in DOM

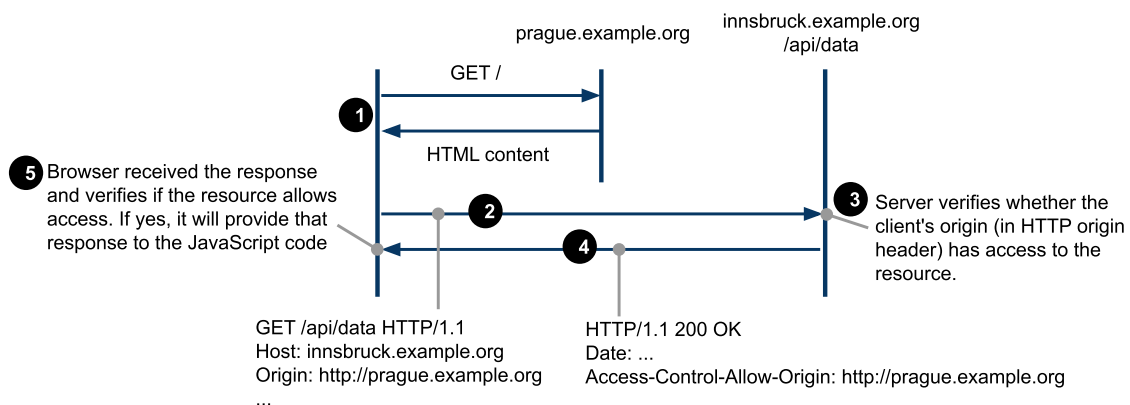
Overview

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

Overview

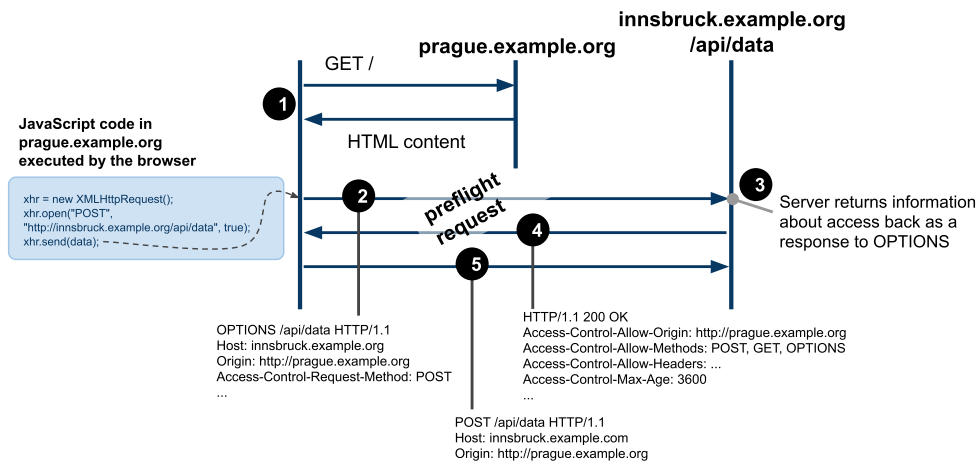
- Increasing number of mashup applications
 - *client-side mashups involving multiple sites*
 - *mechanism to control an access to sites from within JavaScript*
- Allow for **cross-site HTTP requests**
 - *HTTP requests for resources from a different domain than the domain of the resource making the request.*
- W3C Recommendation
 - *see Cross-origin Resource Sharing [↗](#)*
 - *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

- Browser Networking
- 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 (var i = 0; i < data.people.length; i++) {
6     var 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   {"title":"Web 2.0 announcements",
6     "description":"Pipes Output",
7     "link":"http://pipes.yahoo.com/pipes/pipe.info...",
8     "pubDate":"Mon, 07 Mar 2011 18:27:20 +0000",
9     "generator":"..."
10  }
11 }
```

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 returns

```
{ "people" : ["tomas", "peter", "alice", "jana"] }
```
 - then the resource at 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 (var 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 }
```