WebRia T2 Notes:

Reverse Ajax

What is Reverse Ajax?

- Reverse Ajax is essentially a concept: being able to send data from the server to the client. In a standard HTTP Ajax request, data is sent to the server. Reverse Ajax can be simulated to issue an Ajax request, so the server can send events to the client as quickly as possible (low-latency communication).

Reverse Ajax Techniques

- HTTP Polling and JSONP Polling :
    - Polling involves issuing a request from the client to the server to ask for some data. (example: to see if there was an update to the webpage)
    - It is a mere Ajax HTTP request.
    - To get the server events as soon as possible, the polling interval (time between requests) must be as low as possible.
    - **Drawback:** If this interval is reduced, the client browser is going to issue many more requests, many of which won't return any useful data, and will consume bandwidth and processing resources for nothing.

Reverse Ajax with HTTP polling

- JSONP polling is essentially the same as HTTP polling. The difference is you can issue Cross-Domain Requests with JSONP Polling.
- Advantages of polling in javascript:
    - Easy to implement
    - Does not require special features on Server side
    - Works on all browsers

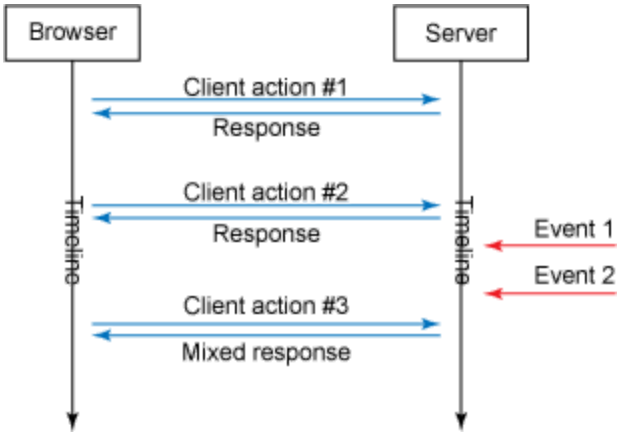
- Disadvantages of polling in javascript:

    - Method rarely implemented as it does not scale at all.

- HTTP Long Polling:
    - Client connects to server and the connection is sustained until server sends data. If server sends response, connection closes.
    - Server holds on to the response unless the server has an update. As soon as the server has an update, it sends it and then the client can send another request.

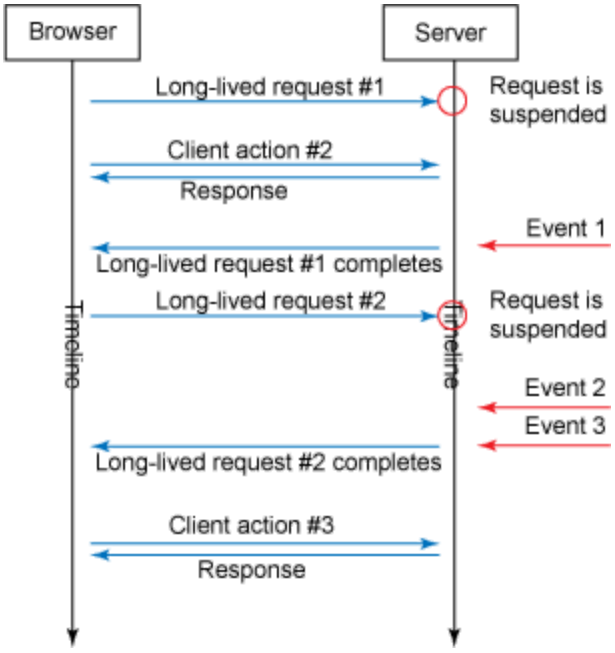
- HTTP Streaming:
    - Similar to long polling but the server responds with a header with "Transfer Encoding: chunked" and hence we do not need to initiate a new request every time the server sends some data (and hence save the additional header overhead).
    - The connection is kept open continuously and data is sent continously.

- Piggyback:
    - Piggyback polling is a much more clever method than polling since it tends to remove all non-needed requests (those returning no data). There is no interval; requests are sent when the client needs to send a request to the server. The difference lies in the response, which is split into two parts: the response for the requested data and the server events, if any occurred.



    - When implementing the piggyback technique, typically all Ajax requests targeting the server might return a mixed response.
    - **Advantages**: With no requests returning no data, since the client controls when it sends requests, you have less resource consumption. It also works in all browsers and does not require special features on the server side.
    - **Disadvantages:** You have no clue when the events accumulated on the server side will be delivered to the client because it requires a client action to request them.

- COMET:
    - Reverse Ajax with polling or piggyback is very limited: it does not scale and does not provide low-latency communication (when events arrive in the browser as soon as they arrive on the server).
    - Comet is a web application model where a request is sent to the server and kept alive for a long time, until a time-out or a server event occurs. When the request is completed, another long-lived Ajax request is sent to wait for other server events.
    - With Comet, web servers can send the data to the client without having to explicitly request it.
    - The big advantage of Comet is that each client always has a communication link open to the server. The server can push events on the clients by immediately committing (completing) the responses when they arrive, or it can even accumulate and send bursts. Because a request is kept open for a long time, special features are required on the server side to handle all of these long-lived requests.



- Comet using HTTP streaming:
    - In streaming mode, one persistent connection is opened. There will only be a long-lived request (#1 in Above Figure) since each event arriving on the server side is sent through the same connection. Thus, it requires on the client side a way to separate the different responses coming through the same connection.
    - Two common techniques used:
        - Forever Iframes: The Forever Iframes technique involves a hidden Iframe tag put in the page with its src attribute pointing to the path returning server events. Each time an event is received, the server writes

and flushes a new script tag with the JavaScript code inside. The iframe content will be appended with this script tag that will get executed.

- **Advantage:** Simple to implement, and it works in all browsers supporting iframes
- **Disadvantage**: There is no way to implement reliable error handling or to track the state of the connection, because all connection and data are handled by the browser through HTML tags. You then don't know when the connection is broken on either side.

- Multi-part XMLHttpRequest:
  - An Ajax request is sent and kept open on the server side. Each time an event comes, a multi-part response is written through the same connection.
  - On the server side, You must first set up the multi-part request, and then suspend the connection.
    - **Advantage**: Only one persistent connection is opened. This is the Comet technique that saves the most bandwidth usage.
    - **Disadvantage:** The multi-part flag is not supported by all browsers. For example, chunks of data (multi-parts) may be buffered and sent only when the connection is completed or the buffer is full, which can create higher latency than expected.

- **Need for heartbeat() function in Hidden frames technique**:

```
//The heartbeat function. This is fired by the server
// on every iteration. Each time we fire this function,
//we will postpone the next connection by 10 seconds.
// If the server closes, the heartbeat is not received
// and we will reconnect in 10 seconds.
heartbeat: function()
{
        if(timer)
        {
                clearTimeout(timer);
        }

        timer = setTimeout(obj.monitor,10000);

}
```

- **readyState == 3 in XHR technique**:

```
//We expect partial output from the server everytime
// until the server is forced to shut the script
// We will have readyState == 3 (partial output)
// and then finally readyState == 4, when we reconnect
if(this.readyState == 3 && this.status == 200)
{
        str = this.responseText;

        newdiv = document.createElement("div");
        newdiv.innerHTML = str.slice(strlen);


        //CLear the responseText and hope all's good :)
        if(this.responseText.indexOf("Fatal") == -1)
        {
                document.body.appendChild(newdiv);
        }
        strlen = str.length;

}
```

- **Using Server-Sent Events**:

```
monitor: function()
{
        //HTML5....for SSEs
        ev = new EventSource("http://localhost/sse.php");
        //ev = new EventSource("http://localhost/sse.php", {WithCredentials, True}"); for allowing CORS
        ev.addEventListener("myevent", obj.updateDiv,false);
        ev.onerror = function() { alert("OOps..server closed");};
},
```

Kai's Notes for SSE: (covers advantages, disadvantages and procedure)

```
Server sent Events:
    HTML5 way of doing. We first create an event source object and point it to URL.
    var es=new EventSource("url",{withCredentials: true});
    es.addEventListener("event-name", function(e){e.data has the shizz}, true); #Called when server sets event key:
                                                                                  event-name
    OR
    es.onmessage = function (e) { ... }         # This will be called when server does not set event key


    Server side:
        header("Content-type: text/event-stream\n\n");
        while(1)  #design choice
            echo "event: event-name\n"  #optional
            echo "data: {...}"
            echo "\n\n"

            ob_end_flush();
            flush();
            sleep(1) #design choice

    onerror function can be set
    es.close();

    + Polyfilled into browsers which do not natively support it
    + Code is simpler when websockets is overkill
    Useful when you do not have to send data back to the server
    + Uses HTTP
    + Builtin support for reconnection

    - Not full duplex
    - Cannot detect when client gets disconnected unless send is attempted
    - Only UTF-8
```

Web Services

Web Services vs Websites

- A website serves the user html typically to be interpreted and displayed in a web browser to a human user. This is the typical GET request over HTTP(S).
- A web service can respond to many different types of requests (GET, PUT, POST, DELETE etc). Interacting with a web service can result in changing data on a remote location, getting information back regarding some data etc. Furthermore, a web service can respond in many different ways, serving data in text, XML or even an empty response. Requests to web services are usually obfuscated form the user.

Properties of Web Services

- Self-Contained: On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, you only require an HTTP server and a SOAP server.

- Self-Describing: A Web Service Description Language (WSDL) file provides all of the information you need to implement a Web service as a provider or to invoke a Web service as a requester.
- Discoverable: Web services can be published, located, and invoked across the Web
- Modular: Simple Web services can be aggregated to form more complex ones by calling lower-layer Web services from a Web service implementation.

Types of Web Services

- Big Web Services (SOAP-Based)
    - Based on SOAP standard (Simple Object Access Protocol standard, an XML language defining a message architecture and message formats.)
    - They often contain a WSDL to describe the interface that the web service offers.
    - The details of the contract may include messages, operations, bindings, and the location of the web service.
    - Big web services includes architecture to address complex non-functional requirements like *transactions, security, addressing, trust, coordination,* and also handles *asynchronous processing and invocation.*
    - This is a great solution when you need :
        - **Asynchronous Processing**
        - **Reliability**
        - **Stateful Operations**

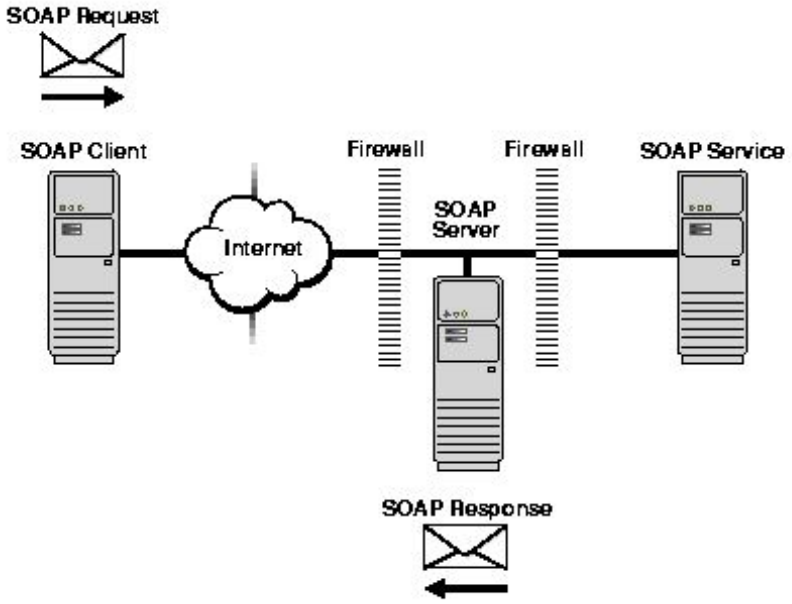
- RESTful Web Services
    - RESTful web services are based on the way how the World Wide Web works.
    - The WWW is based on an Architectural style called REST (Representational State Transfer)
    - Web services following this architectural style are said to be RESTful Web services.
    - In the web, everything is identified by resources. When we type a URL in the browser we are actually requesting a resource present on the server. A representation of the resource (normally a page) is returned to the user which depicts the state of the application. On clicking any other link, the application transfers state with the new representation of the resource. Hence the name Representational State Transfer.
    - REST-style architecture follows this concept and consists of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources which are identified by URI (Uniform Resource Identifier).
    - RESTful web services are based on HTTP protocol and its methods are mainly PUT, GET, POST, and DELETE. These web services are better integrated with

HTTP than SOAP-based services are, and as such do not require XML SOAP messages or WSDL service definitions.

How SOAP-based Services work?

- The SOAP specification describes a standard, XML-based way to encode requests and responses, including:
  - Requests to invoke a method on a service, including *in parameters*
  - Responses from a service method, including return value and *out parameters*
  - Errors from a service

- SOAP describes the structure and data types of message payloads by using the emerging W3C XML Schema standard issued by the World Wide Web Consortium (W3C). SOAP requests and responses travel using HTTP, HTTPS, or some other transport mechanism.



Components of the SOAP Architecture

- In general, a SOAP service remote procedure call (RPC) request/response sequence includes the following steps:
  - A SOAP client formulates a request for a service. This involves creating a conforming XML document.
  - A SOAP client sends the XML document to a SOAP server. This SOAP request is posted using HTTP or HTTPS to a SOAP Request Handler running as a servlet on a Web server.
  - The Web server receives the SOAP message, an XML document, using the SOAP Request Handler Servlet. The server then dispatches the message as a

service invocation to an appropriate server-side application providing the requested service.
    - A response from the service is returned to the SOAP Request Handler Servlet and then to the caller using the standard SOAP XML payload format.

- The SOAP specification does not describe how the SOAP server should handle the content of the SOAP message body. The content of the body may be handed to a SOAP service, depending on the SOAP server implementation.

Components of Web Services

- XML-RPC :
    - This is the simplest XML-based protocol for exchanging information between computers.
    - Requests are encoded in XML and sent via HTTP POST.
    - XML responses are embedded in the body of the HTTP response.

- SOAP :
    - SOAP is an XML-based protocol for exchanging information between computers.
    - SOAP is a format for sending messages.
    - SOAP is designed to communicate via Internet.
    - SOAP is simple and extensible.
    - SOAP allows you to get around firewalls.

- WSDL :
    - WSDL definition describes how to access a web service and what operations it will perform.
    - WSDL is a language for describing how to interface with XML-based services.
    - WSDL is an integral part of UDDI, an XML-based worldwide business registry.
    - WSDL is the language that UDDI uses.
    - WSDL is pronounced as 'wiz-dull' (Do not know why I included this)

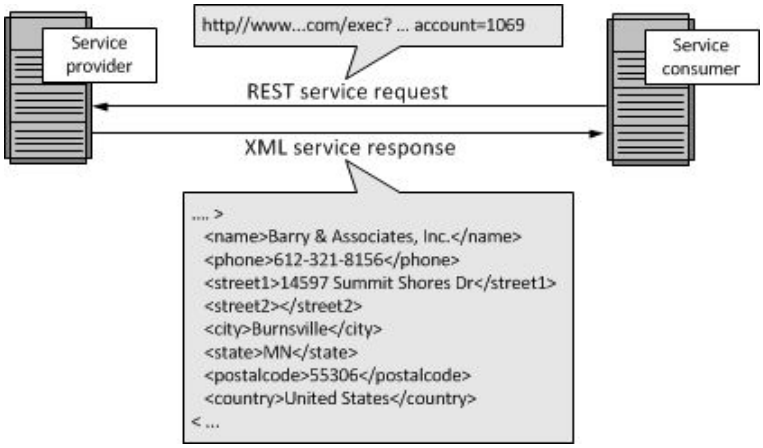
- UDDI :
    - UDDI is an XML-based standard for describing, publishing, and finding web services.
    - UDDI stands for Universal Description, Discovery, and Integration.
    - UDDI is a specification for a distributed registry of web services.
    - UDDI uses WSDL to describe interfaces to web services.
    - UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.

REST Architecture

An application or architecture considered RESTful or REST-style is characterized by:

- State and functionality are divided into distributed resources

- Every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet)
- The protocol is client/server, stateless, layered, and supports caching



REST Constraints

- REST constraints are design rules that are applied to establish the distinct characteristics of the REST architectural style.
- The formal REST constraints are:
    - Client-Server: enforces the separation of concerns in the form of a client-server architecture.
    - Stateless: The communication between service consumer (client) and service (server) must be stateless between requests. This means that each request from a service consumer should contain all the necessary information for the service to understand the meaning of the request, and all session state data should then be returned to the service consumer at the end of each request.
    - Cache: Response messages from the service to its consumers are explicitly labeled as cacheable or non-cacheable. This way, the service, the consumer, or one of the intermediary middleware components can cache the response for reuse in later requests.
    - Uniform Interface: Defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently.
    - Layered System: A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.

RESTful Web Services

- The World Wide Web complies with all these constraints and hence the WWW architecture is RESTful.
- In RESTful Web Services, the WWW is used to implement the services.

- The URI acts as the object and it usually supports the basic CRUD operations. (Create, Read, Update and Delete).
- When implemented with a PHP, a single script performs Create (POST), Read (GET), Update (PUT) and Delete (DELETE) operations.
- But 4 URIs are provided for the four operations. The URLs and rewritten (mapped) to the single PHP which then does the job. The return value is XML or JSON.
- The mapping is performed in the Web Server's config file.

JSON vs XML:

- JSON:
  - Pros:
    - Simple syntax, which results in less "markup" overhead compared to XML.
    - Easy to use with JavaScript as the markup is a subset of JS object literal notation and has the same basic data types as JavaScript.
    - JSON Schema for description and datatype and structure validation
    - JsonPath for extracting information in deeply nested structures
  - Cons:
    - Simple syntax, only a handful of different data types are supported.
- XML:
  - Pros:
    - Generalized markup; it is possible to create "dialects" for any kind of purpose
    - XML Schema for datatype, structure validation. Makes it also possible to create new data- `types
    - built in support for namespaces
  - Cons:
    - Relatively wordy compared to JSON (results in more data for the same amount of information).

RESTful vs SOAP based services

No.	SOAP	REST
1)	SOAP is a **protocol**.	REST is an **architectural style**.
2)	SOAP stands for **Simple Object Access Protocol**.	REST stands for **REpresentational State Transfer**.
3)	SOAP **can't use REST** because it is a protocol.	REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP.
4)	SOAP **uses services interfaces to expose the business logic**.	REST **uses URI to expose business logic**.
5)	**JAX-WS** is the java API for SOAP web services.	**JAX-RS** is the java API for RESTful web services.
6)	SOAP **defines standards** to be strictly followed.	REST does not define too much standards like SOAP.
7)	SOAP **requires more bandwidth** and resource than REST.	REST **requires less bandwidth** and resource than SOAP.
8)	SOAP **defines its own security**.	RESTful web services **inherits security measures** from the underlying transport.
9)	SOAP **permits XML** data format only.	REST **permits different** data format such as Plain text, HTML, XML, JSON etc.
10)	SOAP is **less preferred** than REST.	REST **more preferred** than SOAP.

Security

Restricting data access using HTTP Authentication

- If you want to setup some sort of protection for your files on the server, then HTTP Authentication can be a very good choice.
- When files are protected by user/password combination, the server (apache) has the following lines added to httpd.conf (in windows) -or- apache2.conf(in ubuntu: path is /etc/apache2/apache2.conf)

```
<Directory "C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\restricted">
        AuthType Basic
        AuthName "THIS IS CONFIDENTIAL"
        AuthUserFile "C:\Program Files\Apache Software Foundation\Apache2.2\bin\mypwd"
        Require valid-user
</Directory>
```

- You can generate the password file(mypwd) using htpasswd.exe.

- Now if a client asks for a file in the protected directory, the server returns 401 as a status. This means that authentication is required. In response, the browser pops up the user/password fields for the user to authenticate himself. If the user/passwd is correct access is allowed, otherwise server returns another 401. Once correct,Browsers cache the user/password combination. The user/passwd dialog is not shown until the server returns another 401.
- xhr.open("GET","http://localhost/restricted/priv.txt", true,"guest","guest");
    - This will prevent the default password dialog. Also, once the response is received, the browser will cache the user/passwd combination. The dialog is not shown again.

- To ask for password each and every time a request is made, the “authorization” header is used. Code snippet attached below

```
//Ask for password each and everytime I make a
// a request. For this, use the "Authorization" header.
user = document.getElementById("usr").value;
pwd = document.getElementById("pwd").value;

//Need to encode base64 ourselves
strtoserver = user + ":" + pwd;
encStr = btoa(strtoserver);
alert(encStr);

//Assuming "restricted" was a protected directory on the server.
xhr.open("GET","http://localhost/CONF/priv.txt", true);

// Set the Authorization header. This is crucial.
// We use BASIC authentication. If you use DIGEST,
// the process gets very complicated.
// For BASIC, the username/password is sent as a base64 encoded
// string. See above(lines 49-50). The format is "usr:passwd".
// The combined string is base64 encoded. However this is not
// safe because it can easily be read using a base64 decoder.
// Hence,normal best practice in the industry is BASIC + SSL.
// By setting the "Authorization" header, the user/passwd
// combination is required for every AJAX request.
// The designer should decide whether he wants the user
// to authenticate himself on every call or not.

// So the header field has the following format
// "Authorization:BASIC e4ffabdd445="
xhr.setRequestHeader("Authorization","BASIC " + encStr);

xhr.send();
```

The configuration directives in Apache

- The configuration directives is specified in either httpd.conf or apache.conf.
    - ServerRoot: The ServerRoot directive specifies the top-level directory containing website content. By default, ServerRoot is set to "/etc/httpd" for both secure and non-secure servers.
    - PidFile: PidFile names the file where the server records its process ID (PID). By default the PID is listed in /var/run/httpd.pid.
    - Timeout: Timeout defines, in seconds, the amount of time that the server waits for receipts and transmissions during communications. Timeout is set to 300 seconds by default, which is appropriate for most situations.
    - KeepAlive: KeepAlive sets whether the server allows more than one request per connection and can be used to prevent any one client from consuming too much of the server's resources. (Default it's set to "off")
    - MaxKeepAliveRequests: This directive sets the maximum number of requests allowed per persistent connection (default = 100)
    - KeepAliveTimeout: KeepAliveTimeout sets the number of seconds the server waits after a request has been served before it closes the connection. (default = 15 seconds)
    - IfModule: <IfModule> and </IfModule> tags create a conditional container which are only activated if the specified module is loaded. Directives within the IfModule container are processed under one of two conditions. The directives are processed if the module contained within the starting <IfModule> tag is loaded. Or, if an exclamation point [!] appears before the module name, the directives are processed only if the module specified in the <IfModule> tag is not loaded.

The HTTP Status 401

- UNAUTHORIZED
- The request has not been applied because it lacks valid authentication credentials for the target resource.
- The server generating a 401 response MUST send a WWW-Authenticate header field containing at least one challenge applicable to the target resource.
- If the request included authentication credentials, then the 401 response indicates that authorization has been refused for those credentials.

Types of Web Attacks

- Please use this link :)
- Or if you're too lazy ...

XSS vs. CSRF

- Fundamental difference is that CSRF happens in authenticated sessions when the server trusts the user/browser... while XSS (Cross-Site scripting) doesn't need an authenticated session and can be exploited when the vulnerable website doesn't do the basics of validating or escaping input.

In case of XSS, when the server doesn't validate or escapes input as a primary control an attacker can send any input via any kind of request parameters or input fields which can be cookies, form fields or url params which can be written back toscreen , persisted in database or executed remotely. In case of CSRF, an example is a case when you are logged in into your banking site and at the same time logged into Facebook in another tab in same browser. An attacker can place a malicious link embedded in another link or zero byte image which can be like `yourbanksite.com/transfer.do?fromaccnt=youraccnt&toaccnt=attackersAccount&amt=2500`. Now if you accidentally click on this link in the background transfer can happen though you clicked from the Facebook tab.

This is because your session is still active in browser and browser has your session id. This is the reason the most popular CSRF protection is having another server supplied unique token generated and appended in the requested. This unique token is not something which is known to browser like session id. This additional validation at server (i.e whether the transfer request also contains the correct CSRF token) will make sure that the attacker manipulated CSRF attack in above example will never work.

Different Types of XSS

- Stored XSS (AKA Persistent or Type I):
    - Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5, and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, and never being sent to the server at all.

- Reflected XSS (AKA Non-Persistent or Type II):
    - Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser.

- DOM Based XSS (AKA Type-0):
    - DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the

page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write).

General Countermeasures/Precautions for protection:

- Sanitize input perfectly. If it is not what you are expecting, don't take it. Check for markup in the input values, especially <script> elements. Escape "quote" characters so that they are literally used and not interpreted.
- Do not run multiple queries in a single stroke.
- Give out very generic error messages to the client. Never send database errors to the front end.
- Avoid eval() as far as possible.
- Use Secure communication to avoid Man-in-the-middle attacks. SSL + Basic Authentication if you can.
- Use Http-only cookies as far as possible.
- As users, don't click on links to get to your wanted sites. Type them in the address bar.

The "Heartbleed" Vulnerability: (Refer Sir's PPT for a more picturesque explanation)

- The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).
- The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.
- What is leaked in practice?
    - secret keys used for X.509 certificates, user names and passwords, instant messages, emails and business critical documents and communication.

AJAX - Performance ;_;

- **The "2 simultaneous requests only" limitation with http 1.1**:
    - HTTP 1.1 says: At any point in time, there can only be 2 simultaneous open connections from each single client to a particular domain.

- Browsers do not obey this strictly. For example, firefox, chrome have the limit set at 6.
- Workarounds:

  a) The client must decide which of the requests is most important. Also, the client must have its own queue so that requests are not dropped by the browser. We build a practical solution to this. Check the request manager example we did in class. (prqueue.js, prqueue.html, reqmanager.js and reqclient.js)

  (if order of responses is important, this can still cause prbs. We will then need to add an additional property called "depends" to each JSON request object and do logic on that)

  b) Use sub-domains on the server.

  c) Split up resources over many servers and use the "Access-control-allow-origin" header. But server side cost starts going up.

  d) Increase max connections from browser side, but we are just mimic-ing the fly-over method of traffic control in Bangalore. In firefox this can be done by typing "about:config" in the address bar and then modifying the **network.http.max-persistent-connections-per-server** field.

  e) Send Requests in batches. This will require code to pack and unpack requests. Disadvantages? (The speed of the slowest request determines the speed of the batch)

- **Status-Code 200 does not mean All is OK**:
  - If server script encounters an error, some server environments return a 200 as status. So merely checking for 200 and readyState == 4 may not be enough at all. Also, if a server script times out while executing (maybe it got caught accidentally in an infinite loop), 200 can be returned.
  - One of the simplest ways of introducing the first level of error check is to check for the MIME type of the returned value. If it is not what we expect, then we might want to take alternate action. Otherwise we have to analyze data a little bit to make sure it is ok, before using it completely.

- **Techniques to reduce size of markup/javascript being transferred**:
  - Markup:
    - Remove indentation, unnecessary comments.
    - Remap color values appropriately - Big words can use hex codes while small words can be retained.
    - Be careful with removing quotes around attributes, short-closing some tags (dangerous with <script>, <textarea>)

- CSS Optimizations:
    - Remove all whitespaces, Comments
    - Remap colors to shortest values.
    - Use Short-hand notations as much as possible
      (ex: instead of saying “font-size: 26px; font-family: Arial; font-weight: bold;” you can say “font: bold 26px Arial;”)

- Javascript Optimizations:
    - A single line can have the entire script. Therefore you normally have two versions of the file. One for the developer and one for the end-user (who need not understand the code at all)
    - Remove all comments. They are useless to the end user.
    - Remove all whitespaces. Be careful with ';' character.
    - Perform code optimizations. (x=x+1 can be replaced with x++;)
    - You can replace meaningful variables with s, x, y etc using a tool and then deliver that to the user.
    - Remap built-in objects to save space.
      ex: w=window, n=navigator ap = appname;

```
if(w.n.ap == "Netscape")
{
}
```

    - Minimise DOM access as much as you can. Try to store DOM elements in variables.

- Use External Javascripts:
    - The first download will be a little slow. But the browser can cache the script and if requested by subsequent pages, can use the cached copy of the script. Use need-based script downloads.

- **GET vs POST requests to save performance**:
    - Try to use "GET" as the method as often as you can. Experiments have shown that POST is actually a two step process (headers are sent first and then the body) unlike GET which takes only ONE TCP packet.
    - The YAHOO Mail team found that all browsers except Firefox perform POST as a two-step process – headers are sent first, server responds with 100 (continue) and then the body is sent by the browser. Firefox performs POST in one step.

- **Effective use of Caching**:
    - Predictive Fetch pattern needs to be implemented judiciously. (Can prefetch and cache both on client and server side).
    - External Javascripts can be cached. This will slow down the first download/execution but the browser can cache downloaded JS and

subsequent page downloads (which use the same JS) will be significantly faster because the JS files are already there with the browser.

- Use the “Expires” header judiciously to cache information. (But this means the browser can arbitrarily choose to persist the data even after the expiry date.). Hence you will need to use the URL manipulating method to force a new fetch for the cached content. The “Expires” header is very effective in caching static data received through AJAX. Alternately, use the “Cache-control” header (http 1.1). This is much better than the “Expires” header.
- POST responses are NOT cached in spite of the "Expires" header being set by the server (Or for that matter the Cache-control header). You have to MANUALLY cache the POST responses.
- Manually cache wherever possible. This way, dependency on browser is much reduced (cross browser issues are eliminated too). Before you make new requests, check for cached data in the browser’s memory and use it if you can.