PES UNIVERSITY
(Established under Karnataka Act No.16 of 2013)
100-ft Ring Road, BSK III Stage, Bangalore – 560 085
Department of Computer Science & Engineering

Web Technologies-II(UE18CS353) UNIT II Reading Material

Unit II Lesson Plan:

- Predictive Fetch
- Multi-Stage Download
- Periodic Refresh and Fallback Patterns
- Submission Throttling
- Comet Techniques
- Http Long Polling
- Http Streaming
- Server Sent Events
- REST based Architecture
- SOAP vs REST

Predictive Fetch:

In the recent web applications, we have to think on how can you make the system respond quickly to user activity?

The application should respond to user actions quickly; ideally, it should feel instantaneous.
Many user actions require a response from the server.
Responses from the server can be noticeably latent due to data transfer and server processing overheads.

The solution to this is Pre-fetch content in anticipation of likely user actions. Pre-fetching attempts to remove the delay altogether for certain user actions.
The obvious motivation for instantaneous feedback is efficiency: the things can happen faster because the user's not sitting around waiting for results.
Here are some occasions when Predictive Fetch might be used:

The user's navigating a Virtual Workspace such as a large table. Pre-fetch the results of moving in each direction.
The user's converting between two currencies. Pre-fetch the major currency rates.
The user's reading some articles. Pre-fetch all stories in their favorite category.
google maps

Some experimentation with Google Maps suggests Predictive Fetch is used while navigating the map. The evidence is that you can slowly move along the map in any direction, and you won't see any part of the page going blank and refreshing itself. If you move quickly, you'll see the refresh behavior. Based on that observation, the map is apparently fetching content beyond the viewable area, in anticipation of further navigation

Key triggers for a pre-fetch:

1. User action
2. User history
3. Other users' action.

Decision factors:

How much information will be pre-fetched?

Pre-fetching comes at a cost. Anticipating the user's actions is a guessing game, and for each guess that goes wrong, some resources have been wasted. Designers must make a trade-off involving the likelihood that pre-fetched data will be used, the user benefit if it is used, and the overhead if it's not used. This could involve some fairly heavy user analysis combined with statistical methods.

Will it be the server or the browser that anticipates user actions?

The request for information will always come from the browser, but it's feasible for either the server or the browser to anticipate what the user will need next. If the browser is to decide, it can simply issue a request for that information. If the server is to decide, the browser can, for example, issue a periodic request for general information perhaps with some indication of current state and the server can then push down whatever information it decides might come in handy for the browser.

What information can be used to anticipate user actions?

It's likely you'll use some information about the present and the past to help predict the future. There's a lot of data you could potentially use to help decide what the user will do next:

User's profile

The user's profile and history should provide strong clues. For example, If the user always visits the Books area as soon as they log on, then the homepage should pull down Books content.

User's current activity

It's often possible to predict what the user will do next from their current activity. For example, If a user has just added an item to his shopping cart, he will likely be conducting a purchase; consider downloading his most recent delivery address.

Activity of other users

Sometimes, a rush of users will do the same thing at once. For example,If the user has just logged into a news portal while a major news event is in progress, system-wide statistics will inform the server that this user is probably about to click on a particular article.

Collaborative filtering

As an extension of the previous point, a more sophisticated technique is to correlate users based on information such as their profile and history. People are likely to behave similarly to those whose details are highly correlated. For example, if a user tends to look at "Sport" followed by "Weather," then the system should start pre-fetching "Weather" while the user's looking at "Sport."

Example:

- When user starts scrolling, we know that user is interested to get more content
    - At that point we should fetch new content
    - event : on window object -> onscroll
    - how much the user has scrolled :
        - document.body.scrollTop -> most broswers
        - document.documentelement.scrollTop -> IE
        - we need not fetch content as soon as user starts scrolling
            - set initial scrollAmount, and check if it is greater than 100
- In the php file
    - divide the content file into small parts (fetch one part at a time)
    - In the AJAX request, we need to mention which chunk of the content we need (offset)

```
<script>

        count=0;//send this with each request

        scrollAmount=200;

        var obj={

                xhr:new XMLHttpRequest(),

                getContent:function(){

        var src=document.body.scrollTop;
```

```
          if(src>scrollAmount){
                    obj.xhr.onreadystatechange=obj.showContent;

obj.xhr.open("GET","getChunk.php?count="+count++,true);
             obj.xhr.send();
          }
                   },
                   showContent:function(){
          //here this=obj.xhr
          if(this.readyState==4 && this.status==200){
             var res=this.responseText;
       document.getElementById("content").textContent+=res;
          }


                   }
          }
          window.onscroll=obj.getContent;
</script>
```

PHP file

```
   extract($_GET);
         $chunk=500;
         $pos=$count*$chunk;
                  // first req count=0, pos=0 start from the beginning
```

```
            //second req count=1 pos=500 go to 500th pos


    $file=fopen("content.txt","r");

    fseek($file,$pos); //go to position

    $data=fread($file,$chunk); //read 500

    echo $data;
```

- Additional things
    - if user stops scrolling for a while, we should still send requests
    - If end of file has 3 char, then don't fetch 500.We should not scroll endlessly

Multi Stage Download:

We have to think how can you optimize downloading performance?

Content can be time-consuming to load, especially when it contains rich images.Users rarely need to read all the content on the page.

Users sometimes browse around by exploring links between pages.

So the Solution is to Break the content download into multiple stages, so that faster and more important content will arrive first. Typically, the page is divided into placeholders (e.g., div elements) with the content for each placeholder downloaded independently. XMLHttpRequest Calls can be issued simultaneously or in a serial sequence, and the page will gradually be populated as the calls return.

The pattern applies when an application needs to download a lot of content from the server. By breaking up the call, you can deliver faster and more important data earlier on. You avoid bottlenecks that occur when a single piece of content blocks everything else from returning.

The initial download of a page might be very small, containing just a skeleton along with navigation links and enough context to help the user decide if it's worth sticking around. That's the critical content for a fresh visitor, and there's no need to make him wait for the rest of the page to load. The page then begins to download other components that should appear on the page .User leaves the page before all the components are downloaded.

User stays on the page for an extended period of time: Extra functionality is loaded in the background

Advantage is that You, as the developer, get to decide what is downloaded and at what point in time

When you first visit the page, it is a very simple page with minimal content

A series of requests is being fired off to fill in more content on the page

Within a few seconds, the page jumps to life as content from several different locations is pulled in and displayed on the UI.

Problems

- The Start and stop method of fetching data is very irritating.
- User has to perform an action before a fetch happens.
    - This is because, the fetch is a response to user action.
- Also, a small wait is required before each fetch is displayed.

Goal : make responses near real-time (0 delay).
`

Solution:

- Automatically fetch data, without user having to explicitly request for it.
- Mere navigating through a page should be enough for triggering requests.

Multistage Downloading Features:

- In force when **lots of data needs to be downloaded**.
  - Usually the "FIRST" load.
- **Order of the download**.
  - crucial : otherwise your site may die soon.
  - The most relevant and light data must arrive first so that the user can get started.
  - The heaviest ones should probably arrive later.
  - The advantage of this is that, if the user navigates away from the page fairly quickly, then the **heavy downloads are avoided.**
- Have indicators to **indicate to the user that more data is coming**.
  - (Progress bars or animated GIFs).
- Have **place holders** which can accept returned data readily.
  - elements or tables etc
- When will the Ads come??

Decision factors:

- How will the page be divided into blocks?

The trickiest decision is how to divide the page into blocks, each of which will be downloaded individually. Since the blocks are downloaded in parallel, the main advice is to create small blocks of initial content to ensure the initial download is quick, and to group together content that's likely to be ready at the same time. Also, too many blocks will give the initial load an ugly appearance and may have the undesirable effect of causing already displayed elements to move around. For a fresh visitor, one example would be to create blocks as follows:

- A block of general information about the web site; e.g., name, summary, and a small icon.
- A navigation block showing links within the site.
- A block of general information about the current page, such as a heading and summary text.
- One or more blocks of main page content.
- One or more blocks of auxiliary information; e.g., cross-site promotions, advertising, and legal notices.

- Next decision factor is How will the page be structured?

Ideally, this pattern should not affect the page appearance, but will require some thinking about the underlying HTML. In particular:

As new information loads, it's possible the browser application will automatically rearrange the page in order to accommodate new information. This can lead to the user clicking on the wrong thing, or typing into the wrong area, as items suddenly jump around the page. Ideally, information will not move around once it has been presented to the page. At least for images, it's worthwhile defining the width and height in advance, as CSS properties.

CSS-based layout is likely to play a big part in any solution, as it provides the ability to exercise control over layout without knowing exact details.

- Next factor to consider is What happens to the blocks while their content is being downloaded?

The entire DOM itself can be established on the initial load, so that divs and other structures are used as placeholders for incoming content. If that's the case, most of those elements can be left alone, but if the user is waiting for something specific that may take some time, it would be worth placing a Progress Indicator on the block where the output will eventually appear.

It's also possible to construct the DOM dynamically, so that new elements are added as new content arrives. This approach may be more work, but it may help to decouple the browser and the server. With the right framework in place, it means that the browser does not have to know exactly which items will be downloaded. The browser might ask the server to send all adverts, and the sender simply responds by sending down an XML file with three separate entries, which the browser then renders by adding three new divs.

- We also need to decide on Will the calls be issued simultaneously?

Let's say you're at a point where you suddenly need a whole lot of content. Should you issue several requests at once? Not necessarily. Keep in mind there are limits on how many outgoing requests the browser can handle at one time and consider what's best for the user. If there's a bunch of content that might not be used, consider deferring it for a bit with a JavaScript setTimeout call. That way, you can help ensure the user's bandwidth is dedicated to pulling down the important stuff first.

Advantages:

- Developer gets to decide what is downloaded and at what point in time
- User need not wait till the whole page gets loaded.

Disadvantages:

- page must work in its simplest form for browsers that don’t support Ajax technologies
- The basic functionality must work without any additional downloads
    - Solution : Graceful Degradation
        - browsers that support Ajax technologies :extensive interface
        - Other browsers: simple, bare-bones interface.

Example:

- html file has basic template
    - some images, text ,etc
      SCRIPT

```
<script>

var xhr=new XMLHttpRequest();

function init(){

  xhr.onreadystatechange=showContent;

  xhr.open("GET","content.txt",true);

  xhr.send();

}



.............

<body onload="init()">
```

show content :

```


function showContent(){

  if(xhr.readyState==4 && xhr.status==200){

      var res=xhr.responseText;

      var resArr=res.split(";");

      document.getElementById("secondary").innerHTML=resArr[0];

      document.getElementById("header").innerHTML=resArr[1];
```

```
    // display a loading icon

    document.getElementById("picture").innerHTML=resArr[3];

    document.getElementById("links").innerHTML=resArr[3];

    document.getElementById("audio").innerHTML=resArr[3];


    setTimeout(getPic,4000); //if user stays for 4 secs then load the
things



  }

}
```

Get the picture (AJAX request)

```
function getPic(){

  this.xhr.onreadystatechange=this.showPic;


  //using GET

  this.xhr.open("GET","img.txt",true);

  this.xhr.send();

}
```

Display the picture

```


function showPic(){

  if(xhr.readyState==4 && xhr.status==200){

    var res=this.responseText;
    document.getElementById("picture").innerHTML=res;

  }

}
```

Periodic Refresh

In today's world of web applications, we have to think how can the application keep users informed of changes occurring on the server?

The state of many web apps is inherently volatile. Changes can come from numerous sources, such as other users, external news and data, results of complex calculations, and triggers based on the current time and date.

HTTP requests can only emerge from the client. When a state change occurs, there's no way for a server to open connections to interested clients.

The Solution is that the browser periodically issues an XMLHttpRequest Call to gain new information; e.g., one call every five seconds. The solution makes use of the browser's Scheduling capabilities to provide a means of keeping the user informed of latest changes.

In its simplest form, a loop can be established to run the refresh indefinitely, by continuously issuing XMLHttpRequest Calls using the method

```
setInterval(callServer, REFRESH_PERIOD_MILLIS);
```

Here, the callServer function will invoke the server, having registered a callback function to get the new information. That callback function will be responsible for updating the DOM according to the server's latest report. Conventional web apps, even most of those using XMLHttpRequest Calls, operate under a paradigm of one-way communication: the client can initiate communication with the server, but not vice versa. Periodic Refresh fakes a back-channel: it approximates a situation where the server pushes data to the client, so the server can effectively receive new information from the browser. Indeed, as some of the examples show, the server can also mediate between users in almost real-time. So Periodic Refresh can be used for peer-to-peer communication too.

it's important to note that it's a serious compromise, for two key reasons:

- The period between refreshes would ideally be zero, meaning instant updates.
- There is a significant cost attached to Periodic Refresh.

<u>Decision Factors:</u>

How long will the refresh period be?

The refresh period can differ widely, depending on usage context. Broadly speaking, we can identify three categories of activity level:

- Real-Time interaction (milliseconds)
- Active monitoring (seconds)
- Casual monitoring (minutes)

To list a few of the real time applications for periodic Refresh patterns, they are Online Chat Applications, World headlines,New websites,Score boards,Customized Stock Portal.

Example:

- 4 files
    - Cricket scores html file
        - has scores of 3 teams
    - scores.txt
    - update.php
    - scores.php
- Cricket scores html file sends request to scores.php

- scores.php reads scores.txt and sends it back
- request can be sent every 2 seconds
- scores.txt can be static OR
  - or we can overwrite it everytime and see the change
  - or use update.php
    - this can update scores.txt with new scores
      CricketScores.html

```
<div id="status"> UPDATING ... <br></div>

<div id="team1" class="score"></div><br>

<div id="team2" class="score"></div><br>

<div id="team3" class="score"></div><br>
```

script

```
var xhr=new XMLHttpRequest();

function getScore(){

                xhr.onreadystatechange=showScore;

                xhr.open("GET","scores.php",true);

                xhr.send();

}

//showScore..........
```

PHP file

```


<?php

    $file=fopen("scores.txt","r");

    $data=fread($file,filesize("scores.txt"));

    echo $data;
```

```
?>
```

script :

```
function showScore(){

   if(xhr.readyState==4 && xhr.status==200){

      var res=xhr.responseText;

      scores=res.split(";")

      document.getElementById("team1").innerHTML=scores[0];

      document.getElementById("team2").innerHTML=scores[1];

      document.getElementById("team3").innerHTML=scores[2];


      document.getElementById("status").textContent+=" ...";


      setTimeout(getScore,2000);

         }

}
```

- setTimeout : arg is getScore not getScore()
    - if arg was getScore(), we are calling setTimeout with the return val of getScore
    - instead, call it with the object getScore
      update.php

```
<?php


         set_time_limit(0);

         for($i=0;$i<100;$i++){
                $file=fopen("scores.txt","w");

            $score=rand(0,100).";".rand(100,200).";".rand(200,300);

                fwrite($file,$score);

                fclose($file);

        sleep(6);

            }

?>
```

Run update.php on the browser using localhost and see the scores being updatd

xhr property : timeout

- when req is sent, and no response received in timeout seconds, event called ontimeout takes place
- This event can have a handler
    - either abort the request
    - or GRACEFUL DEGRADATION
        - either send request again (but this interval also matters)
        - or EXPONENTIAL BACKOFF
            - send the future requests after a time interval increasing at exponential rate
            - if the server is down, the exp val will tend to infinity
            - after the response comes back, set back to original time
    - These are called fallback patterns
      function getScore:

```
xhr.timeout=6000; //set timeout seconds

xhr.ontimeout=backoff; //call handler if this event takes place
```

To make the timeout happen, in scores.php make it sleep scores.php

```
<?php
        $file=fopen("scores.txt","r");
        $data=fread($file,filesize("scores.txt"));
        sleep(8);
        echo $data;
?>
```

Script

```

function backoff(){
                        console.log("request timeout");
                        //increase request interval
                        request_interval=request_interval*2;
                        //call getScore again
                        setTimeout(getScore,request_interval);
}
```

Now when the response is received, set back the interval to 2 seconds

```
function showScore(){
   if(xhr.readyState==4 && xhr.status==200){
     //reset request interval when it works
      request_interval=2000;
```

```
        .......
    }
}
```

- Output :
    - interval increases exponentially
    - to see the actual response (make it receive instead of timeout) , when the code is running, change the sleep(seconds) in the php file

Submission throttling

We should be thinking how can information be submitted to the server?

Information is often uploaded in bursts; e.g., a chat tool incurs many hits when a user becomes passionate about the topic, or a data entry tool incurs many hits when the user responds to some new information.

It's difficult for the server to cope with lots of messages at once.Browsers can only handle a limited number of pending XMLHttpRequest Calls at any moment.

Each message has overheads, such as packet headers and requires some processing at each stage of the browser/server round-trip.

So the solution would be instead of submitting upon each JavaScript event, retain data in a browser-based buffer and automatically upload it at fixed intervals. As with many applications of buffering and throttling, the purpose is to strike a balance between responsiveness and resources. In most cases, it would be ideal to respond to every keystroke and mouse movements, for example, like a desktop application, tooltips could come directly from the server as the user moves around. But that's not practical due to bandwidth considerations, and possibly server constraints too. So to ease bandwidth and to lessen server processing, call detail is accumulated and periodically uploaded.

<u>**Decision Factors:**</u>

- How long should the throttle period be?

Deciding on the throttle period requires some analysis of user needs. As the previous blog reader example demonstrates, there is a range of periods that might be required:

For background synchronization, the period can be a few minutes.Then for low-level interaction while the user types and mouses around, the period must be in the order of 100 milliseconds.

- When to send data to the server
    - Early Send : submit it every time
        - eg - CBT exams
    - Late Send :
        - throttle the submission
            - control the rate at which data is sent
        - eg : chat, don't send every char, send when user pauses typing
- Event : onkeypress
    - triggered only when character key is pressed (not for Ctrl etc)

Some of the real time applications for such pattern are Google Suggest, Gmail, any sort of Prototype Framework

Example:

```
<input type="text" id="search" onkeypress="obj.getTerm()"/>
```

script :

```
function Suggest(){

  this.getTerm=function(){
       //initiate send process only after 1 second

       setTimeout(this.sendSearch,1000);

       },

       ................

}

var obj=new Suggest();
```

- Adding a timer to Suggest
    - setTimeout returns a pointer
        - if this is passed to clearTimeout, this invocation is cancelled

```
this.timer=null;

this.getTerm=function(){

  if(this.timer){

                clearTimeout(this.timer);

        }

  this.timer=setTimeout(this.sendSearch,1000);

 },

 ........................
```

- Example
    - keypress triggers->getTerm
        - initially timer=null
            - setTimeout is called
        - but before 1 sec, type another key
        - keypress again triggers->getTerm
        - but this time timer != null
            - clearTimeout is called
            - setTimeout is deffered by 1 second

- now pause for more than 1 second
    - sendSearch is now called
      Obj is lost in sendSearch :

```
var temp=this;
this.sendSearch=function(){
   //here this = window
   //console.log(this) -> window Object
   //console.log(temp) ->instance of suggest object
   val=document.getElementById("search").value;
   console.log(val);
}
```

- *container* is where results will be displayed <div id="container">
    - Add this to suggest

```
function Suggest(){
.............
   this.xhr=new XMLHttpRequest();
   this.search=null;
   this.container=null;
   var temp=this;
}
```

- sendSearch :
    - get the search value
    - sent it to suggest.php

```
this.sendSearch=function(){
```

```
                                        //console.log("sendSearch");

                                        //here this = window

                                        //console.log(this) -> window
Object

        //val=document.getElementById("search").value;

                                        //console.log(val);

                                        //console.log(temp) ->instance of
suggest object


           temp.search=document.getElementById("search");

           temp.container=document.getElementById("container");


           temp.xhr.onreadystatechange=temp.showResult;

           temp.xhr.open("GET","suggest.php
term="+temp.search.value,true);

           temp.xhr.send();


//console.log(document.getElementById("search").value);

}
```

In the php file :

- Read the food.txt file
- compare the words with the "term" typed out
    - fgets -> read line by line
    - strncasecmp(haystack, needle, length)

  - can specify the (upper limit of the) number of characters from each string to be used in the comparison.
- return array of matches (JSON encode)

```
<?php
   extract($_GET);
   $file=fopen("food.txt","r");
   $res=array();

   while(!feof($file)){
           $line=trim(fgets($file)); // read each line
           if(strncasecmp($line, $term, strlen($term))==0){
//compare line and term
                   $res[]=$line;
           }
   }
   echo json_encode($res);

?>
```

Now when we get the response we want to

- create a new div for it
- Add it to container
- highlight if hovered on
- clicking on it should replace the text box with this value
- make container display block (visible)
  showResult :

```
if(this.readyState==4 && this.status==200){
            res=this.responseText;
             resO=JSON.parse(res);
            console.log("response "+resO);
            if(resO.length==0){
                temp.search.style.backgroundColor="red";
            }
        else{
        for(f in resO){
            temp.search.style.backgroundColor="white";
            itemDiv=document.createElement("div");
            itemDiv.innerHTML=resO[f];
            itemDiv.className="foodItem";
            temp.container.appendChild(itemDiv);
            itemDiv.onclick=temp.sendFood;
            }


}
temp.container.style.display="block";
```

sendFood :

```
this.sendFood=function(e){
```

```
            value_clicked_on=e.target.firstChild.nodeValue;

                //replace search box with this value

                temp.search.value=value_clicked_on;



}
```

- sendFood will be triggered on the event onclick (event e)
    - target.firstChild.nodeValue
        - *target* gives the div that was clicked on -> in this case <div class="foodItem">
        - it's *first child* is the text field firstChild: #text "Aloo gobi"
        - the actual text can be extracted by using *nodeValue*

- Issue
    - for every small change, we're sending a request to the server
    - cookies
        - again have to send it to the server every time
    - Solution : local storage
        - key value pairs
        - LS.getItem(key)
        - LS.setItem(key,value)
- Local storage
    - set item when you get a response from the php file
        - showResult :
            - localStorage.setItem("suggest.php?term="+temp.search.value,res);
    - Before sending a request, check if the item already exists in local storage
        - sendSearch :

```
if(localStorage.getItem("suggest.php?term="+temp.search.value)){

  console.log("from cache");
```

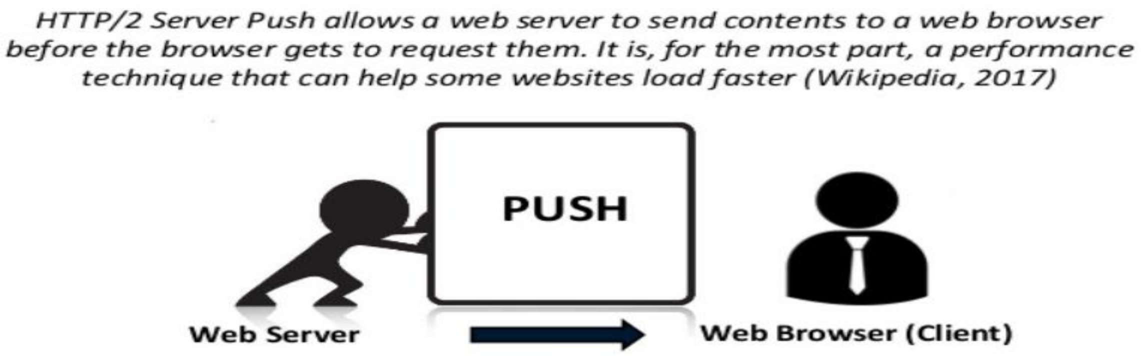
```
    var
cacheRes=localStorage.getItem("suggest.php?term="+temp.search.va
lue);

    console.log(JSON.parse(cacheRes));

}
```

Example of strncasecmp :

```
<?php

    $string = "true";

    if(strncasecmp($string, "Trudeau", 4)) { echo "True"; }

    else { echo "False"; }

?>

// o/p : true
```

Comet Techniques:

Comet is a web application model in which a long-held HTTPS request allows a web server to push data to a browser, without the browser explicitly requesting it. Comet is known by several other names such as Ajax Push, Reverse Ajax, Two-way-web, HTTP Streaming, HTTP server push, etc.
Developers can utilize Comet techniques to create event-driven Web apps. Comet is actually an umbrella term for multiple techniques for achieving client-server interaction.


HTTP/2 Server Push allows a web server to send contents to a web browser before the browser gets to request them. It is, for the most part, a performance technique that can help some websites load faster (Wikipedia, 2017)
PUSH
Web Server
Web Browser (Client)

There are various techniques to perform the server push and Comet techniques come in different flavours:

- Streaming
- Long polling
- Server Sent Events

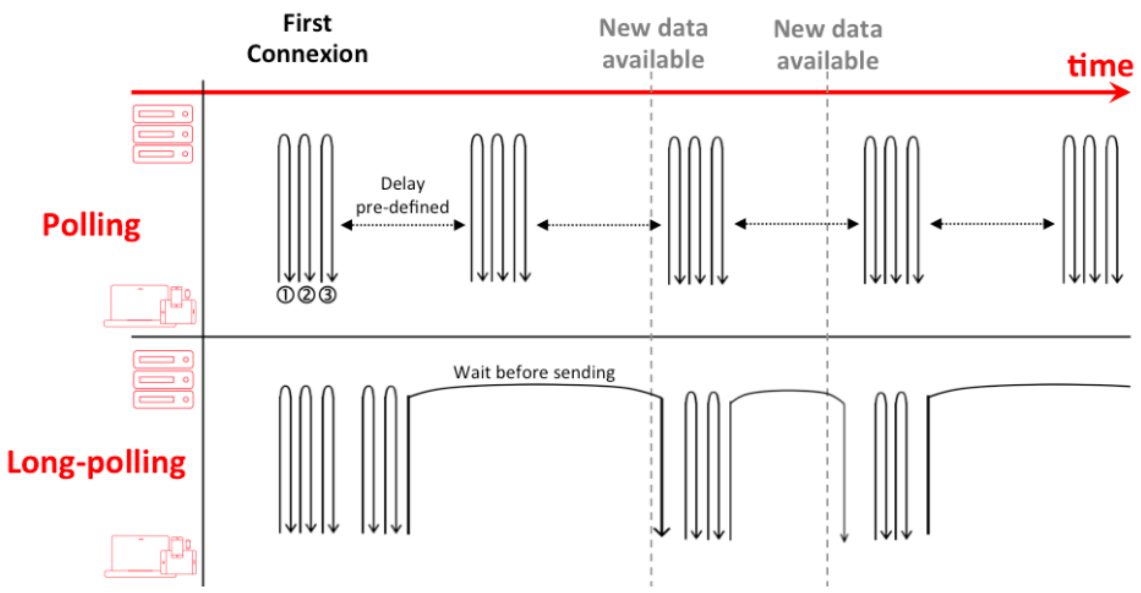
HTTP Long Polling:

Polling comes in two forms.

- Short polling and
- Long Polling.

Short Polling is also called as Periodic Refresh or client pull where there is an AJAX-based timer that calls at fixed delays

But long polling is based on Comet and the client makes an Ajax request to the server, which is kept open until the server has new data to send to the browser. Upon receiving the server response, the current HTTP Connection gets closed and the browser initiates a new long polling request in order to obtain the next data set. Long polling can be achieved using either Ajax or script tag techniques.



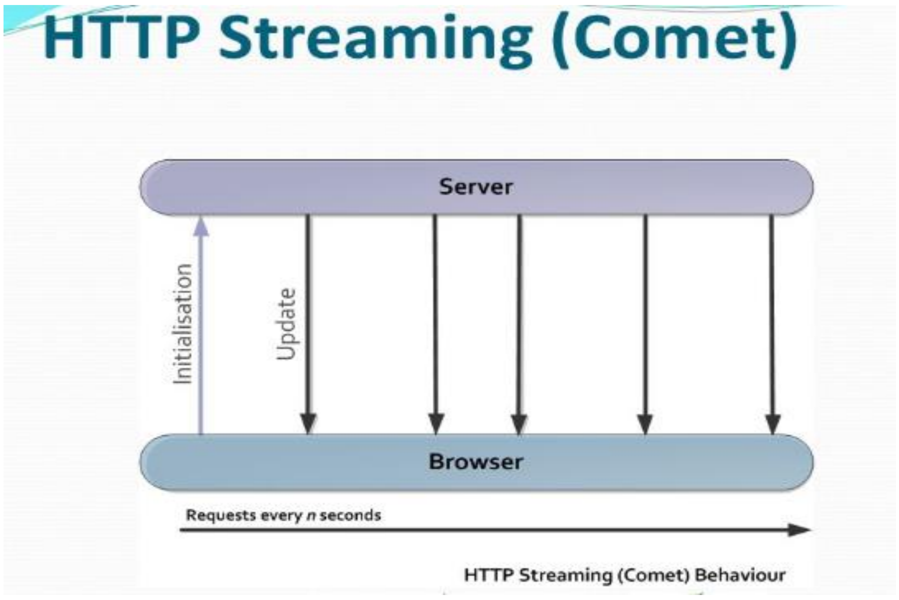
The worked example in the class for Long Polling using Iframes and XHR is attached over here.

LongPolling_Iframes.html

LongPolling_XHR.html

HTTP Streaming:

An application using streaming Comet opens a single persistent connection from the client browser to the server for all Comet events. These events are incrementally handled and interpreted on the client side every time the server sends a new event, with neither side closing the connection.



Specific techniques for accomplishing streaming Comet include the following:

Hidden iframe:

A basic technique for dynamic web application is to use a hidden iframe HTML element (an inline frame, which allows a website to embed one HTML document inside another). This invisible iframe is sent as a chunked block, which implicitly declares it as infinitely long. As events occur, the iframe is gradually filled with script tags, containing JavaScript to be executed in the browser. Because browsers render HTML pages incrementally, each script tag is executed as it is received. Some browsers require a specific minimum document size before parsing and execution is started, which can be obtained by initially sending 1–2 kB of padding spaces.

One benefit of the iframes method is that it works in every common browser. Two downsides of this technique are the lack of a reliable error handling method, and the impossibility of tracking the state of the request calling process.

XMLHttpRequest

The XMLHttpRequest (XHR) object, a tool used by Ajax applications for browser–server communication, can also be pressed into service for server–browser Comet messaging by generating a custom data format for an XHR response, and parsing out each event using browser-side JavaScript; relying only on the browser firing the onreadystatechange callback each time it receives new data.The main point to remember here is that we have to catch the ready

State "3" to see the partial response. In PHP, The default output buffering has to be made to off in the php.ini file and in the code we have to explicitly mention the flow of buffer.

Ob_start()- To start the Output Buffering,

Ob_flush() – To flush the contents from Php buffer

flush() – To flush the contents from the Apache Server Buffer

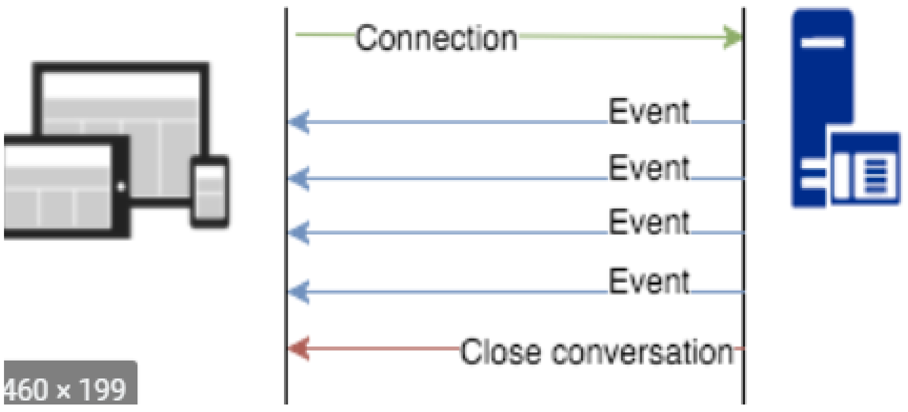
The example that is worked in the class is attached over here

STREAMING_XHR.html

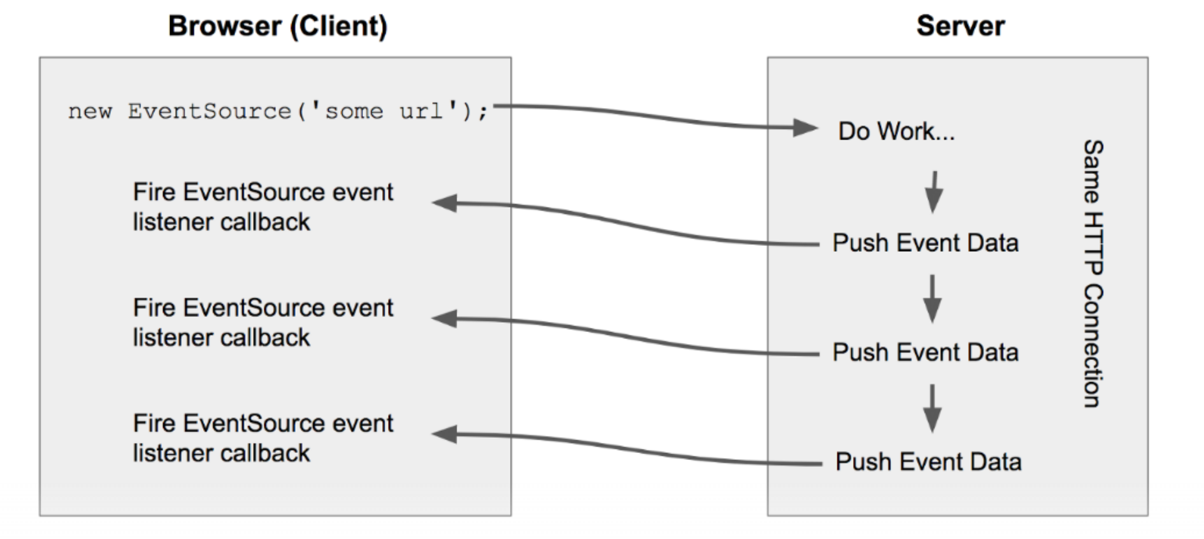
Click on this link to get the chat application using Streaming
https://drive.google.com/drive/folders/1vUi2BwrnJE_iLmMajnvySC7VU6Cjs6Wk

Server Sent Events:

The HTML 5 draft specification produced by the Web Hypertext Application Technology Working Group (WHATWG) specifies so called server-sent events, which defines a new JavaScript interface EventSource and a new MIME type text/event-stream. All major browsers except Microsoft Internet Explorer include this technology.

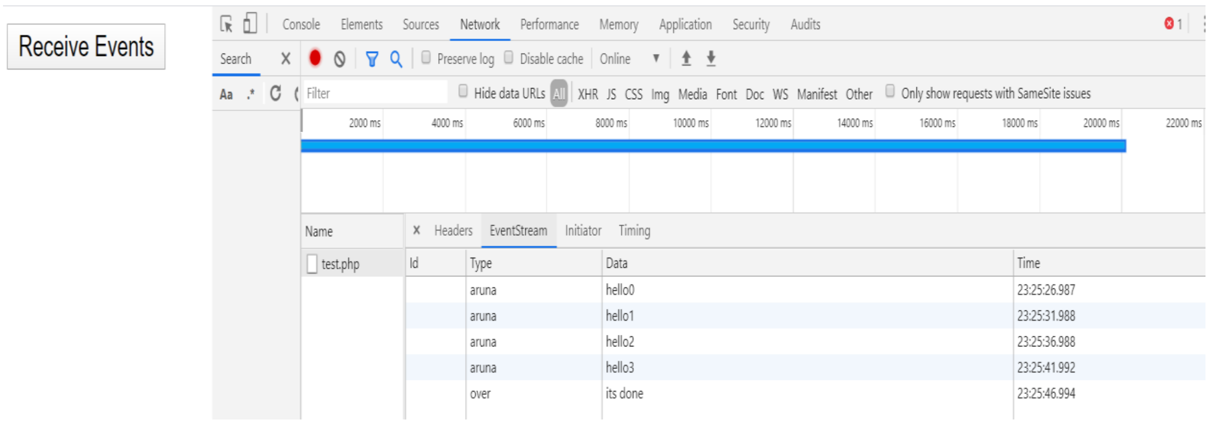


SSE is a mechanism that allows the server to asynchronously push the data to the client once the client-server connection is established. The server can then decide to send data whenever a new "chunk" of data is available. It can be considered as a one-way *publish-subscribe* model.It also offers a standard JavaScript client API named EventSource implemented in most modern browsers as part of the HTML5 standard by W3C..Edge and Opera Mini are lagging behind this implementation, the most important case for SSE is made for mobile browser devices where these browsers have no viable market share.


Browser (Client)
Server
new EventSource('some url');
Do Work...
Fire EventSource event listener callback
Push Event Data
Fire EventSource event listener callback
Push Event Data
Fire EventSource event listener callback
Push Event Data
Same HTTP Connection

Since SSE is based on HTTP, it has a natural fit with HTTP/2 and can be combined to get the best of both: HTTP/2 handling an efficient transport layer based on multiplexed streams and SSE providing the API up to the applications to enable push. So getting multiplexing over HTTP/2 out of the box. The client and server are informed when the connection drops. Maintaining unique Id with messages the server can see that the client missed n number of messages and send the backlog of missed messages on reconnect.

The output can be seen in the form of various events as per the screenshot below.


Receive Events
Console Elements Sources Network Performance Memory Application Security Audits
Search Preserve log Disable cache Online
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other Only show requests with SameSite issues
2000 ms 4000 ms 6000 ms 8000 ms 10000 ms 12000 ms 14000 ms 16000 ms 18000 ms 20000 ms 22000 ms
Name
test.php
Headers EventStream Initiator Timing

Id	Type	Data	Time
	aruna	hello0	23:25:26.987
	aruna	hello1	23:25:31.988
	aruna	hello2	23:25:36.988
	aruna	hello3	23:25:41.992
	over	its done	23:25:46.994

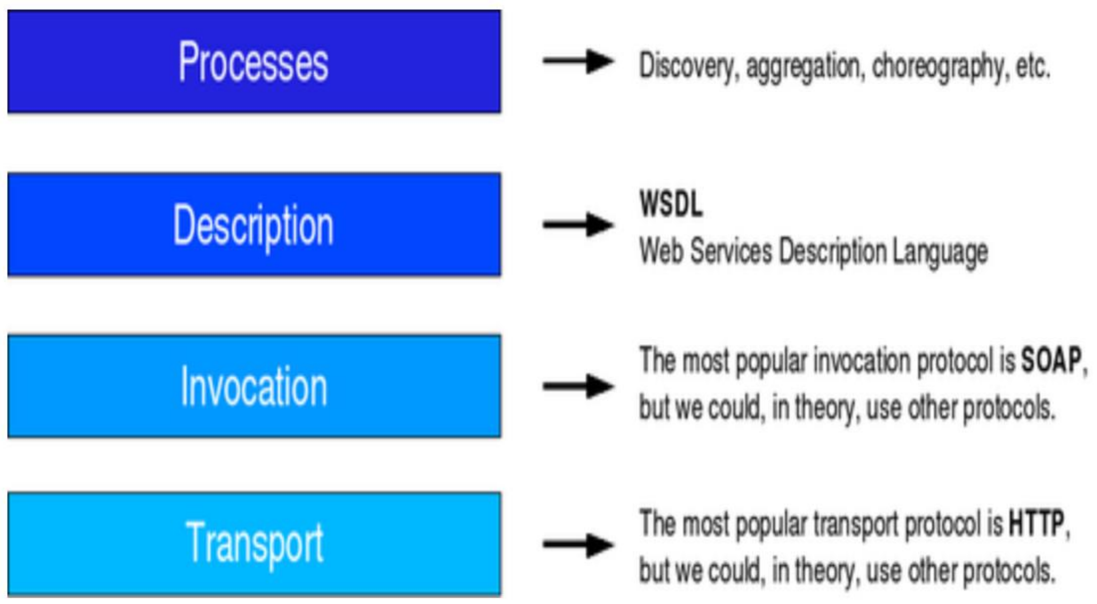
The example that is worked in the class is attached over here.

Server_Sent_Events.
html

Web Services:

A Web service is a software component designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (typically,WSDL for SOAP based services).

In simpler words, a Web Service is a legacy application that is web-enabled. It is a "service" built over the web.


Processes
Discovery, aggregation, choreography, etc.
Description
WSDL
Web Services Description Language
Invocation
The most popular invocation protocol is SOAP, but we could, in theory, use other protocols.
Transport
The most popular transport protocol is HTTP, but we could, in theory, use other protocols.

Web Services are basically of two types

1. RESTful : based on REST

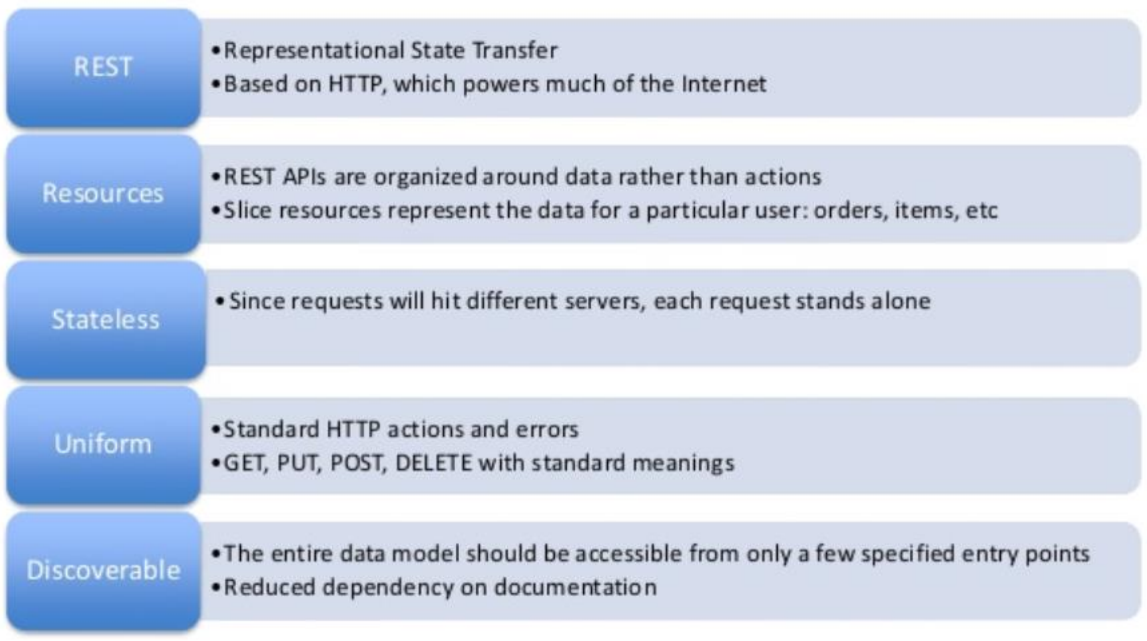
II. WS-*: Based on SOAP (WS stands for Web Service)

RESTFul Architecture:

REST stands for Representational State Transfer, a term coined by Roy Fielding in 2000. It is an architecture style for designing loosely coupled applications over HTTP, that is often used in the development of web services. ... REST defines 6 architectural constraints which make any web service – a true RESTful API.

1. Uniform interface
2. Client–server
3. Stateless

4. Cacheable
5. Layered system
6. Code on demand (optional)



REST	•Representational State Transfer •Based on HTTP, which powers much of the Internet
Resources	•REST APIs are organized around data rather than actions •Slice resources represent the data for a particular user: orders, items, etc
Stateless	•Since requests will hit different servers, each request stands alone
Uniform	•Standard HTTP actions and errors •GET, PUT, POST, DELETE with standard meanings
Discoverable	•The entire data model should be accessible from only a few specified entry points •Reduced dependency on documentation

Below is the link which very well explains the constraints

https://restfulapi.net/rest-architectural-constraints/

Since the web service we developed works over HTTP which is a web based Communication protocol, by default our web service satisfies most of the constraints. We have to create a uniform interface to access the resources within web page. Everything revolves around the concept of Resource and its representation.

Ex: An image is a resource. JPEG/PNG is a representation of the image.

Ex2: Data in a database is the resource. This resource can be represented as an html, XML or Json.

World wide web is a very best example for a fully restful architecture.

As part of the PHP, the rewrite rule engine has to be made on and the rule has to be written in the .htaccess file which means directory specific configuration.

Sample URL Mapping:

RewriteEngine On

RewriteRule ^getweather/([a-z]+)\.(xml|json) weatherproc.php?city=$1&format=$2 [NC,L]

RewriteRule ^update/([a-z]+)\.(xml|json) weatherproc.php?city=$1&format=$2 [NC,L]

There are few things which have to be made clear when we tell the RESTful web services works with all the HTTP Methods.

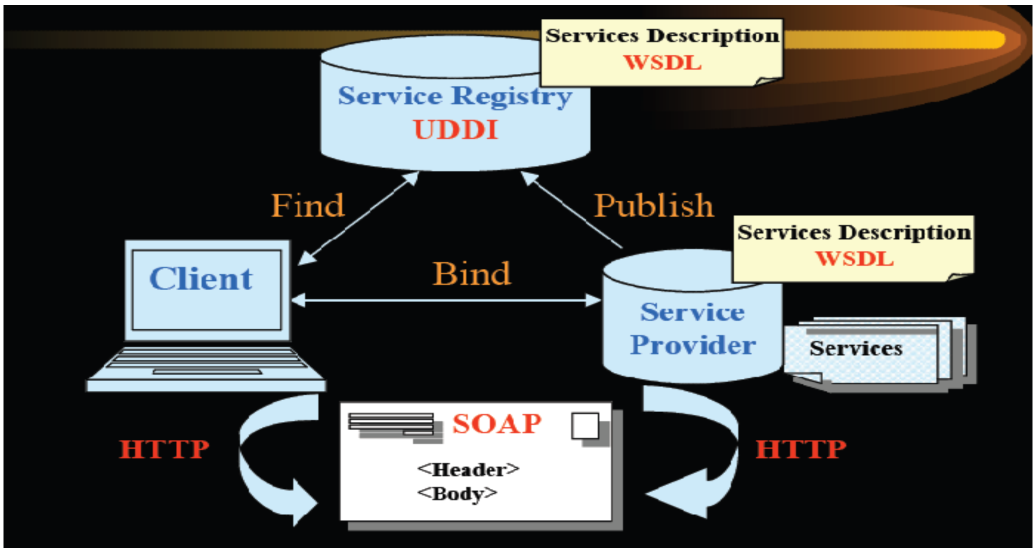
HTTP Verbs

- GET-Fetch
- PUT- Update/Insert
- DELETE – Delete
- POST – Append

The example weather web service developed in class is attached here.
Click on the below link

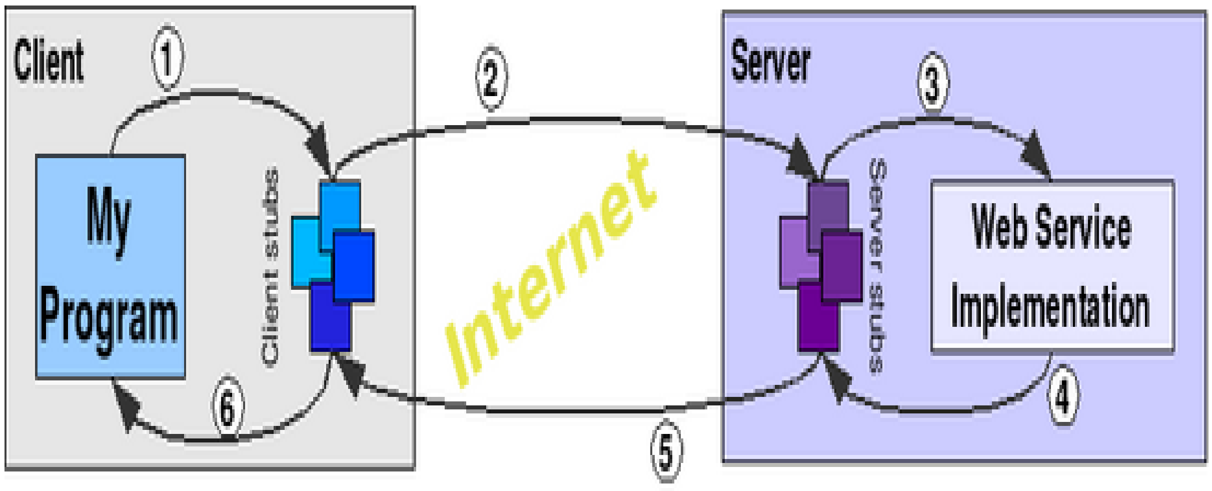
SOAP(Simple Object Access Protocol)

SOAP (Simple Object Access Protocol) is a message protocol that allows distributed elements of an application to communicate. SOAP can be carried over a variety of lower-level protocols, including the web-related Hypertext Transfer Protocol. There are various components that are involved in this Soap Protocol. A diagrammatic representation of them can be given as



Stub and Skeleton

Stub and skeleton are counterparts in a web service setup. Skeleton belongs to service provider side and stub belongs to receiver side. At lower level stub and skeleton communicate with each other. From client side the business objects communicates with stub objects and stub takes the responsibility form the message and invoke the web service. Once the invoking is done, at service provider side, skeleton is the parallel object for stub and it receives the request message and understands it and passes on the information to service side business objects.



A SOAP message is an ordinary XML document containing the following elements –

- Envelope − Defines the start and the end of the message. It is a mandatory element.
- Header − Contains any optional attributes of the message used in processing the message, either at an intermediary point or at the ultimate end-point. It is an optional element.
- Body − Contains the XML data comprising the message being sent. It is a mandatory element.
- Fault − An optional Fault element that provides information about errors that occur while processing the message.

I have attached a sample SOAP message here.

soap.xml

WSDL(Web Services Description Language):

- WSDL is an XML-based protocol for information exchange in decentralized and distributed environments.
- WSDL definitions describe 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 Universal Description, Discovery, and Integration (UDDI), an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.

The three major elements of WSDL that can be defined separately are –

- Types
- Operations
- Binding

A WSDL document has various elements, but they are contained within these three main elements, which can be developed as separate documents and then they can be combined or reused to form complete WSDL files.Some of the important elements in the file are

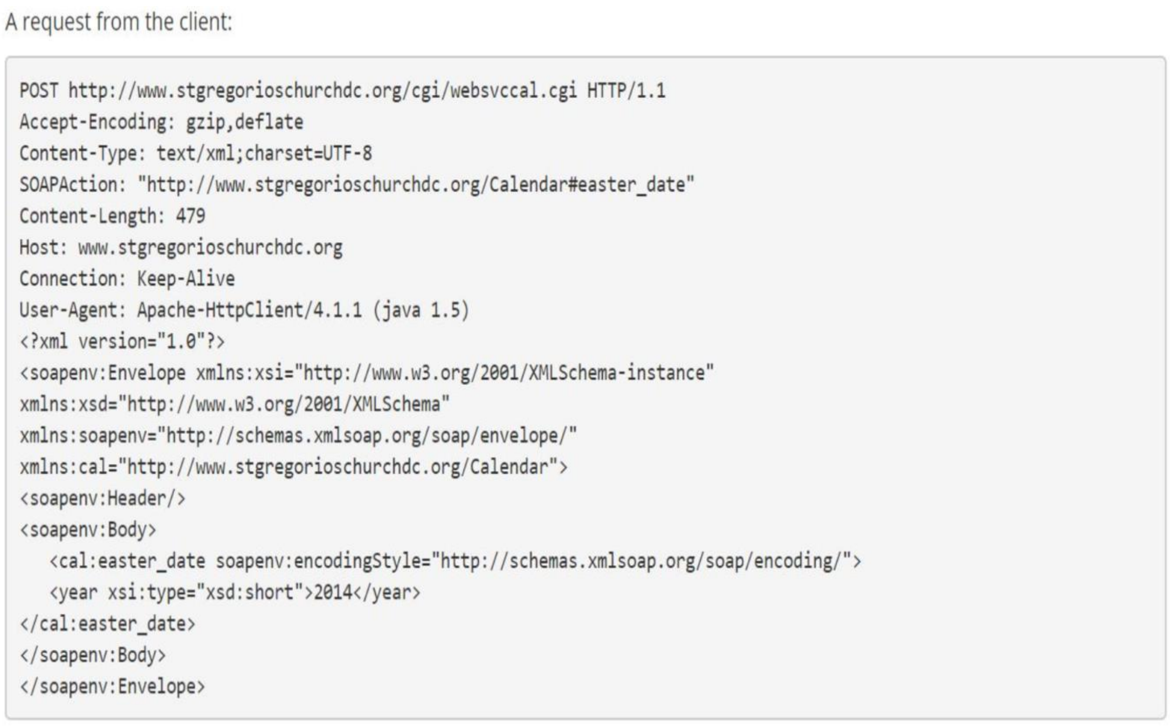
A WSDL document contains the following important elements –

Definition ,Data types ,Message ,Operation ,Port type, Binding ,Port, Service .

I have attached a sample WSDL File here.

WSDL_wsdl

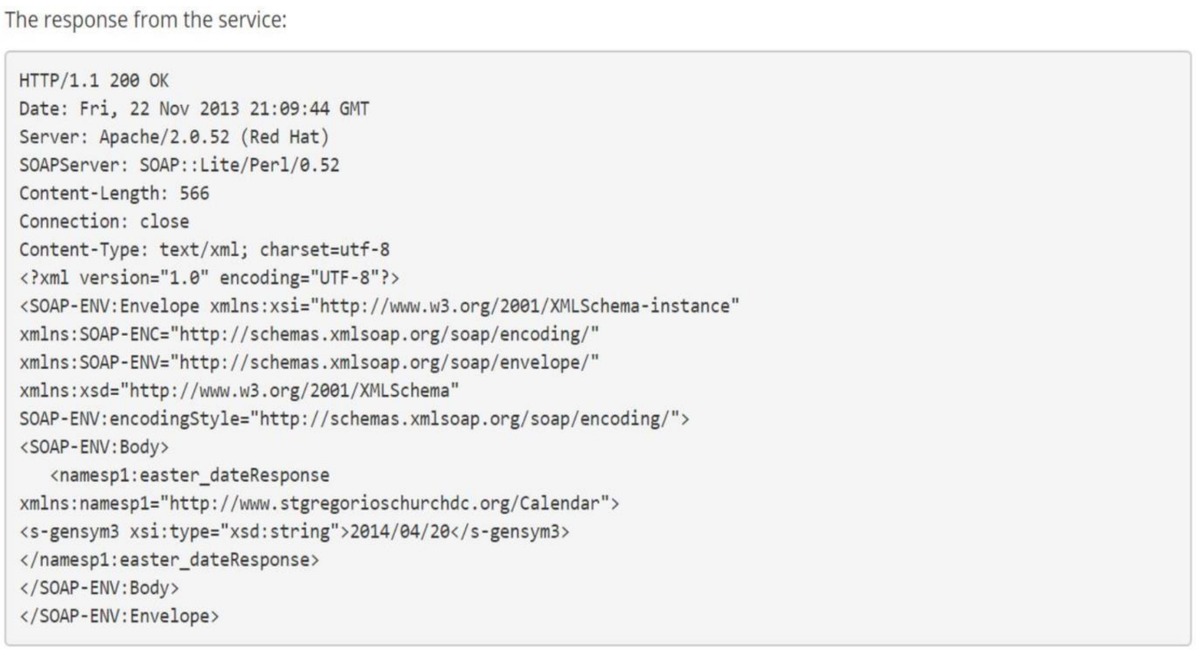
A Soap Request:



A request from the client:

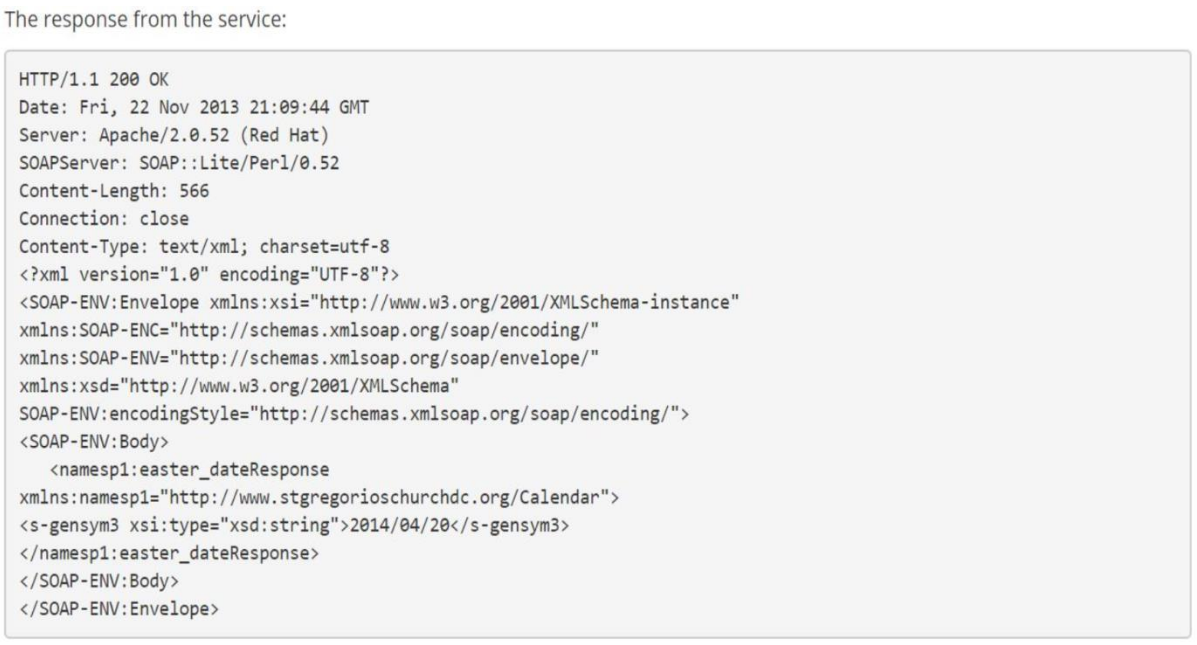
```
POST http://www.stgregorioschurchdc.org/cgi/websvccal.cgi HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://www.stgregorioschurchdc.org/Calendar#easter_date"
Content-Length: 479
Host: www.stgregorioschurchdc.org
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
<?xml version="1.0"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cal="http://www.stgregorioschurchdc.org/Calendar">
<soapenv:Header/>
<soapenv:Body>
   <cal:easter_date soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <year xsi:type="xsd:short">2014</year>
</cal:easter_date>
</soapenv:Body>
</soapenv:Envelope>
```

A Soap Response:



The response from the service:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 21:09:44 GMT
Server: Apache/2.0.52 (Red Hat)
SOAPServer: SOAP::Lite/Perl/0.52
Content-Length: 566
Connection: close
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
   <namesp1:easter_dateResponse
xmlns:namesp1="http://www.stgregorioschurchdc.org/Calendar">
<s-gensym3 xsi:type="xsd:string">2014/04/20</s-gensym3>
</namesp1:easter_dateResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```


The response from the service:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 21:09:44 GMT
Server: Apache/2.0.52 (Red Hat)
SOAPServer: SOAP::Lite/Perl/0.52
Content-Length: 566
Connection: close
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
   <namesp1:easter_dateResponse
xmlns:namesp1="http://www.stgregorioschurchdc.org/Calendar">
<s-gensym3 xsi:type="xsd:string">2014/04/20</s-gensym3>
</namesp1:easter_dateResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP vs REST

SOAP is designed to break traditional monolithic applications down into a multi-component, distributed form without losing security and control. In contrast, REST is a model of distributed computing interaction based on the HTTP protocol and the way that web servers support clients. REST over HTTP is almost always the basis for modern microservices development and communications. RESTful APIs uses HTTP requests to GET, PUT, POST and DELETE data.

REST/HTTP is simple, flexible, lightweight, and offers little beyond a way of exchanging information. SOAP can ride on HTTP as well, but it connects the elements of a complex set of distributed computing tools (the Web Services and SOA framework) as well as application components, and this forms a part of a total service-oriented framework.