HTTP(REST), WebSocket Frontend

Speaker:

Maxym Komarnytsky

HTTP(HyperText Transfer Protocol) - is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes.

HTTPS(Hypertext Transfer Protocol Secure) - the communication protocol is encrypted using Transport Layer Security (TLS), or, formerly, its predecessor, Secure Sockets Layer (SSL). It is used for secure communication over a computer network.

Versions

HTTP/0.9 (1991)

HTTP/1.0 (1996)

HTTP/1.1 (1997)

HTTP/2.0 (2015)

HTTP/3.0 (2018)

HTTPS (2000)

HTTP session

- An HTTP session is a sequence of network request-response transactions.
- An HTTP client initiates a request by establishing a (TCP) connection to a particular port on a server (typically port 80).
- An HTTP server listening on that port waits for a client's request message.
- The server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own.
- The body of this message is typically the requested resource, although an error message or other information may also be returned.

Request message

- a request line (e.g., GET /assets/img/home/logo.png HTTP/1.1, which requests a resource called /assets/img/home/logo.png from the server.)
- request header fields (e.g., Accept-Language, Content-Type: text/html).
- an empty line
- an optional message body

Request methods

- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT
- PATCH

Safe methods

- GET
- HEAD
- TRACE
- OPTIONS

Not change the state of the server, only for information retrieval

Methods with side effect

- POST
- PUT
- DELETE
- PATCH

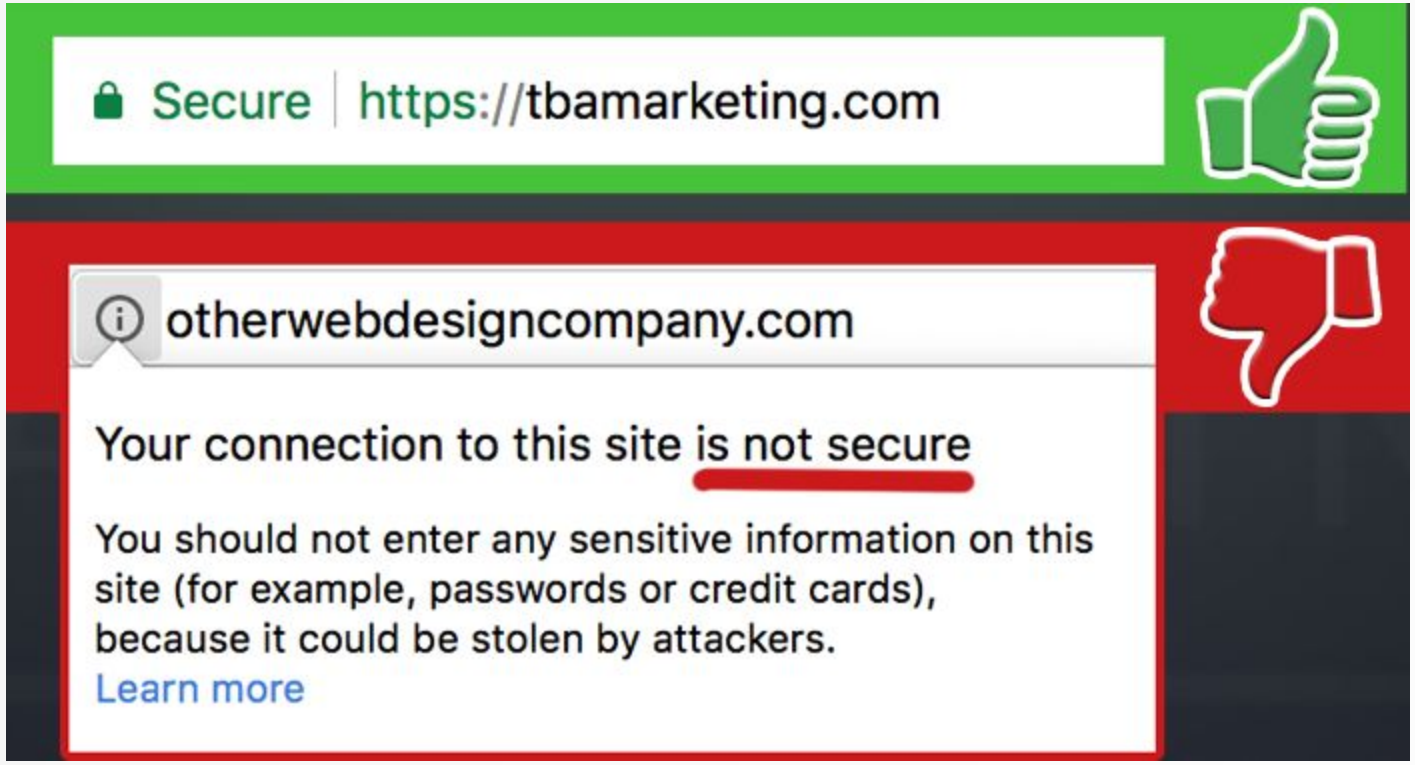
May cause side effects either on the server, or external side effects

Response message

- a status line which includes the status code and reason message (e.g., HTTP/1.1 200 OK, which indicates that the client's request succeeded.)
- response header fields (e.g., Content-Type: text/html)
- an empty line
- an optional message body

Status codes

- Informational 1XX (101 Switching Protocols)
- Successful 2XX (200 OK)
- Redirection 3XX (301 Moved Permanently)
- Client Error 4XX (404 Not Found)
- Server Error 5XX (500 Internal Server Error)


Secure | https://tbamarketing.com
otherwebdesigncompany.com
Your connection to this site is not secure
You should not enter any sensitive information on this site (for example, passwords or credit cards), because it could be stolen by attackers.
Learn more

REST

REST(Representational State Transfer) - is a software architectural style that defines a set of constraints to be used for creating Web services

- Give every “thing” an ID
- Link things together
- Use standard methods
- Resources can have multiple representations.
- Communicate statelessly.

XMLHttpRequest

XMLHttpRequest is a built-in browser object that allows to make HTTP requests in JavaScript.

- Historical reasons: we need to support existing scripts with XMLHttpRequest.
- We need to support old browsers, and don’t want polyfills (e.g. to keep scripts tiny).
- We need something that fetch can’t do yet, e.g. to track upload progress.

XMLHttpRequest example

GET EXAMPLE

```
// create XMLHttpRequest object
let xhr = new XMLHttpRequest();

// Initialize xhr(method, URL, [async, user, password])
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts');

// response format, json, document, text
xhr.responseType = 'json';

//Opens connection and sends the request
xhr.send();

// This will be called after the response is received
  xhr.onload = function() {
    let responseObj = this.response;

    if (this.status ==  200) {
      console.log(responseObj.length);
      console.log(responseObj[0].title);
    }
  };
```

POST EXAMPLE

```
let xhr = new XMLHttpRequest();

xhr.open('POST', 'https://jsonplaceholder.typicode.com/posts');
xhr.setRequestHeader("Content-type", "application/json; charset=UTF-8");
xhr.responseType = 'json';

let body = JSON.stringify({
  title: 'test',
  body: 'test',
  userId: 1
});

xhr.send(body);

xhr.onload = function() {

  let responseObj = this.response;
  if (this.readyState == 4 && this.status == 201) {
    console.log(responseObj);
    console.log(responseObj.title);
  }
};
```

EVENTS FOR RESPONSE

```
// Triggers periodically during the download, reports how much downloaded.
xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }

};

//when the request couldn't be made, e.g. network down or invalid URL
xhr.onerror = function() {
  alert("Request failed");
};
```

READY STATES

```
xhr.abort(); // terminate the request
xhr.timeout = 10000; // timeout in ms, 10 seconds

UNSENT = 0; // initial state
OPENED = 1; // open called
HEADERS_RECEIVED = 2; // response headers received
LOADING = 3; // response is loading (a data packed is received)
DONE = 4; // request complete

xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // loading
  }
  if (xhr.readyState == 4) {
    // request finished
  }
};
```

Synchronous request

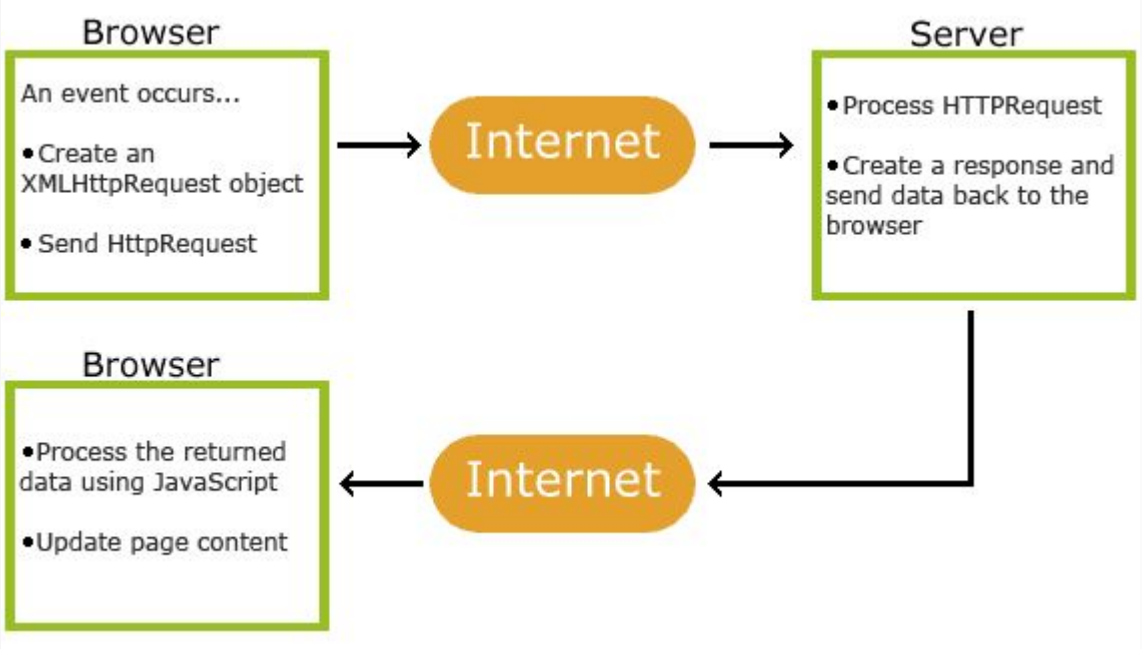
```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt',
false);

try {
  xhr.send();
  if (xhr.status != 200) {
    alert(`Error ${xhr.status}: ${xhr.statusText}`);
  } else {
    alert(xhr.response);
  }
} catch(err) { // instead of onerror
  alert("Request failed");
};
```

AJAX Examples

```
$.ajax(
  "https://jsonplaceholder.typicode.com/users/1",
  {
    success: function(data) {
      console.log(data);
    },
    error: function(error) {
      if (error) {
        console.log(error);
      }
    }
  })
```

```
$.ajax({
    method: "PATCH",
    url: "https://jsonplaceholder.typicode.com/posts/1",
    data: { body: 'foo'}
  })
  .done(function( response ) {
    console.log(response);
  });
```

How AJAX Works



Fetch

Method **fetch()** is the modern way of sending requests over HTTP.

PUT Method using fetch()

```
fetch('https://jsonplaceholder.typicode.com/posts/1', {
    method: 'PUT',
    body: JSON.stringify({
      id: 1,
      title: 'foo',
      body: 'bar',
      userId: 1
    }),
    headers: {
      "Content-type": "application/json; charset=UTF-8"
    }
  })
  .then(response => response.json())
  .then(json => console.log(json))
```

GET Method using fetch()

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

HttpClient

HttpClient in @angular/common/http offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers

```
//GET
getUsers(): Observable<any> {
   return this.http
     .get(this.baseUrl + '/users', this.options)
     .catch(this.handleError);
 }
}

// POST
postPosts(param: any): Observable<any> {
 const body = JSON.stringify(param);
 return this.http
   .post(this.baseUrl + '/posts', body,this.options)
   .catch(this.handleError);
}

// PUT
putPosts(param: any): Observable<any> {
 const body = JSON.stringify(param);
 return this.http
   .put(this.baseUrl + '/posts/1', body, this.options)
   .catch(this.handleError);
}
```

```
// PATCH
patchPosts(param: any): Observable<any> {
 const body = JSON.stringify(param);
 return this.http
   .patch(this.baseUrl + '/posts/2', body, this.options)
   .catch(this.handleError);
}

// DELETE
deletePosts(): Observable<any> {
 return this.http
   .delete(this.baseUrl + "/posts/1", this.options)
   .catch(this.handleError);
}
```

WebSocket

The WebSocket protocol, provides a way to exchange data between browser and server via a persistent connection

```
let socket = new WebSocket(url);

socket.onopen = function(e) {
  alert("[open] Connection established, send -> server");
  socket.send("My name is John");
};

socket.onmessage = function(event) {
  alert(`[message] Data received: ${event.data} <-
server`);
};
```

```
socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Connection closed cleanly,
code=${event.code} reason=${event.reason}`);
  } else {
    alert('[close] Connection died');
  }
};

socket.onerror = function(error) {
  alert(`[error] ${error.message}`);
};
```

Events

- open – connection established,
- message – data received,
- error – websocket error,
- close – connection closed.

```
socket.send(data)
```

WebSocket data

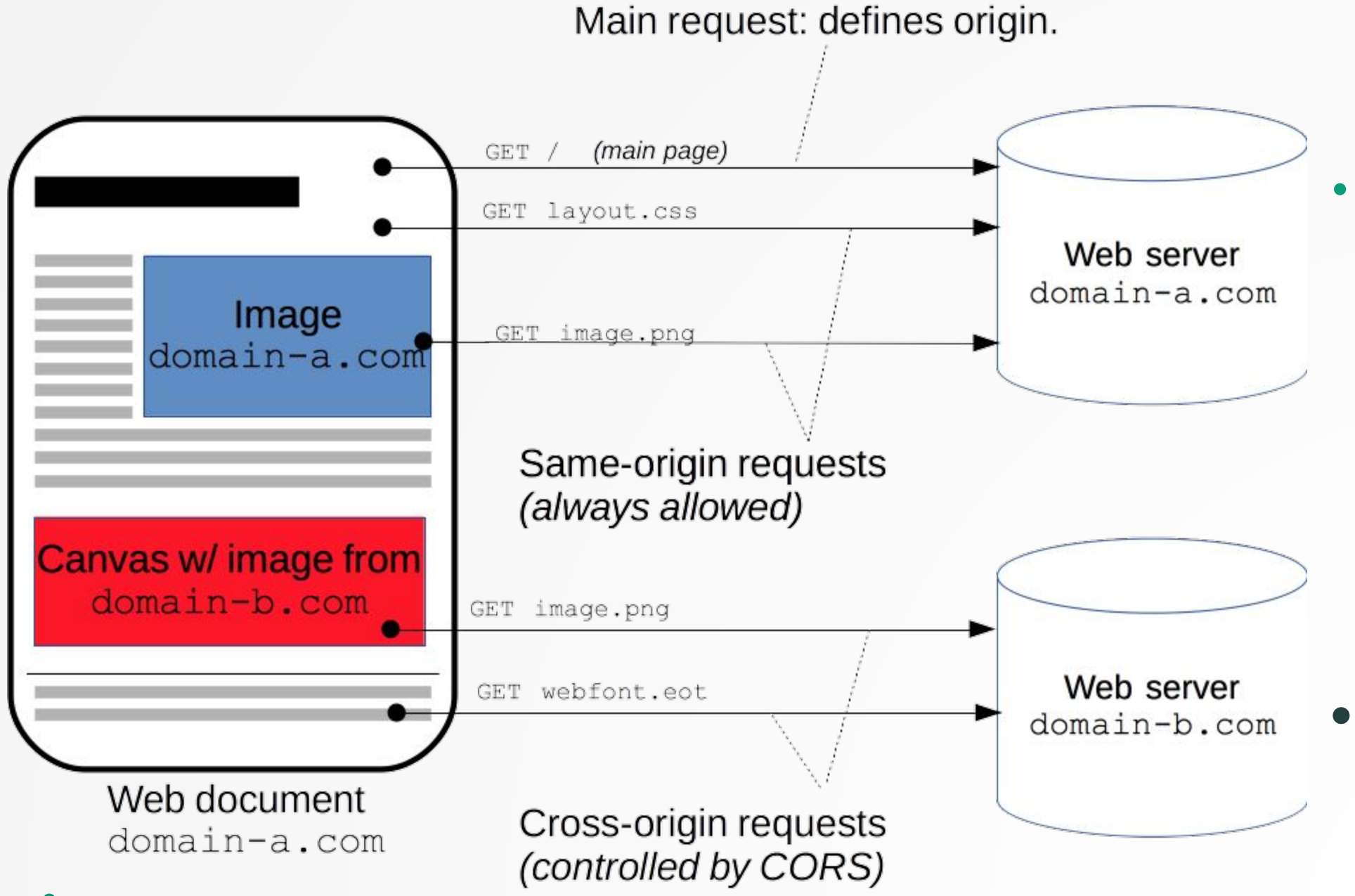
WebSocket communication consists of “frames” that can be sent from either side

- “text frames” – contain text data that parties send to each other.
- “binary data frames” – contain binary data that parties send to each other.
- “ping/pong frames” are used to check the connection, sent from the server, the browser responds to these automatically.
- “connection close frame” and a few other service frames.

CORS

Cross-origin resource sharing (CORS) is a mechanism to allows the restricted resources from another domain in web browser.

- Access-Control-Allow-Origin: The origin that is allowed to make the request, or * if a request can be made from any origin
- Access-Control-Allow-Methods: A comma-separated list of HTTP methods that are allowed
- Access-Control-Allow-Headers: A comma-separated list of the custom headers that are allowed to be sent
- Access-Control-Max-Age: The maximum duration that the response to the preflight request can be cached before another call is made


Main request: defines origin.
GET / (main page)
GET layout.css
GET image.png
Image
domain-a.com
Web server
domain-a.com
Same-origin requests
(always allowed)
Canvas w/ image from
domain-b.com
GET image.png
GET webfont.eot
Web server
domain-b.com
Web document
domain-a.com
Cross-origin requests
(controlled by CORS)