

Medii de proiectare și programare

2022-2023

Curs 13

Conținut

- WebSockets
- Autentificare REST
 - JWT
- Examen

HTTP

Limitările HTTP:

- *Stateless*:
 - Browserul deschide o conexiune pe portul 80.
 - Trimite o cerere HTTP serverului web.
 - Aplicația server decide ce va face cu cererea, procesează datele, generează răspuns HTML și îl trimite serverului web.
 - Serverul web adaugă antetele HTTP corespunzătoare răspunsului, îl trimite browserului și închide conexiunea.
- Site-urile web trebuie să păstreze informațiile despre utilizatori (ex. cookies).
 - Informația este transmisă între client și serverul web la fiecare cerere.
 - Informațiile adiționale măresc dimensiunea pachetelor transmise și pot face aplicațiile web vulnerabile (securitatea datelor).
- Comunicarea este întotdeauna inițiată de client și fiecare pereche cerere/răspuns este independentă de celelalte cereri/răspunsuri.

HTTP

- Aplicații *real-time*: burse de valori, vânzarea biletelor, trafic, citirea aparatelor medicale
- Metode:
 - *Polling*, browserul trimite regulat cereri HTTP și primește răspunsul imediat. Este o soluție bună dacă se cunosc intervalele la care mesajele devin disponibile, deoarece se pot sincroniza cererile clienților cu momentele când informațiile sunt disponibile pe server.
 - *Long-polling*, browserul trimite o cerere la server și serverul păstrează cererea deschisă pentru o perioadă prestabilită de timp. Dacă în acea perioadă se primește o notificare, serverul trimite clientului un răspuns care conține și notificarea. Dacă nu se primește nici o notificare, serverul trimite un răspuns pentru a încheia cererea clientului.
 - *Streaming*, browserul trimite o cerere completă dar serverul trimite și păstrează un răspuns incomplet ce este actualizat continuu și păstrat deschis pe termen nelimitat (sau o perioadă prestabilită de timp). Răspunsul este actualizat de fiecare dată când trebuie trimisă o notificare, dar serverul nu semnalează terminarea răspunsului, păstrând astfel conexiunea deschisă pentru mesaje ulterioare. Pot să apară probleme din cauza păstrării într-un buffer a răspunsurilor.

WebSockets

- WebSocket-urile sunt o **conexiune persistentă, bi-direcțională, full-duplex** de la un browser web la un server.
- După ce **conexiunea a fost stabilită**, ea rămâne **deschisă până când clientul sau serverul decide să o închidă**.
- Cât timp conexiunea este deschisă, **clientul și serverul își pot trimite mesaje unul altuia în orice moment de timp**.
- Programarea web devine astfel bazată pe eveniment, și nu doar inițiată de utilizator. Este *stateful*.
- O singură aplicație server care rulează știe de toate conexiunile, permițând comunicarea cu orice număr de conexiuni deschise în orice moment de timp.

Protocolul WebSocket

- În 2011, IETF a standardizat protocolul WebSocket sub denumirea de RFC 6455.
- De atunci, majoritatea browserelor Web implementează cod API pe partea de client care suportă protocolul WebSocket.
- S-au dezvoltat biblioteci în Java, .NET, Ruby, Objective C, JavaScript, etc. care implementează protocolul WebSocket.
- Browsere care suportă WebSockets: nativ în Chrome, Firefox, Opera și Safari (inclusiv Safari pentru dispozitive mobile), Internet Explorer.
- Orice browser care nu suportă WebSockets poate folosi soluția *polyfill* *Flash*.

Protocolul WebSocket - Handshake

Pentru stabilirea unei conexiuni WebSocket, **clientul trimite o cerere HTTP pentru un *handshake* WebSocket:**



```
GET /echo HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Protocolul WebSocket - Handshake

- Dacă serverul acceptă cererea de actualizare a protocolului, va returna un răspuns **HTTP 101 Switching Protocols**:



HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Sec-WebSocket-Protocol: echo

- După ce serverul returnează răspunsul 101, **protocolul la nivel de aplicație se va schimba din HTTP în WebSockets**, care va folosi conexiunea TCP stabilită anterior. Din acest moment, ambii participanți pot trimite/primi mesaje în orice moment de timp.



WebSocket - URI

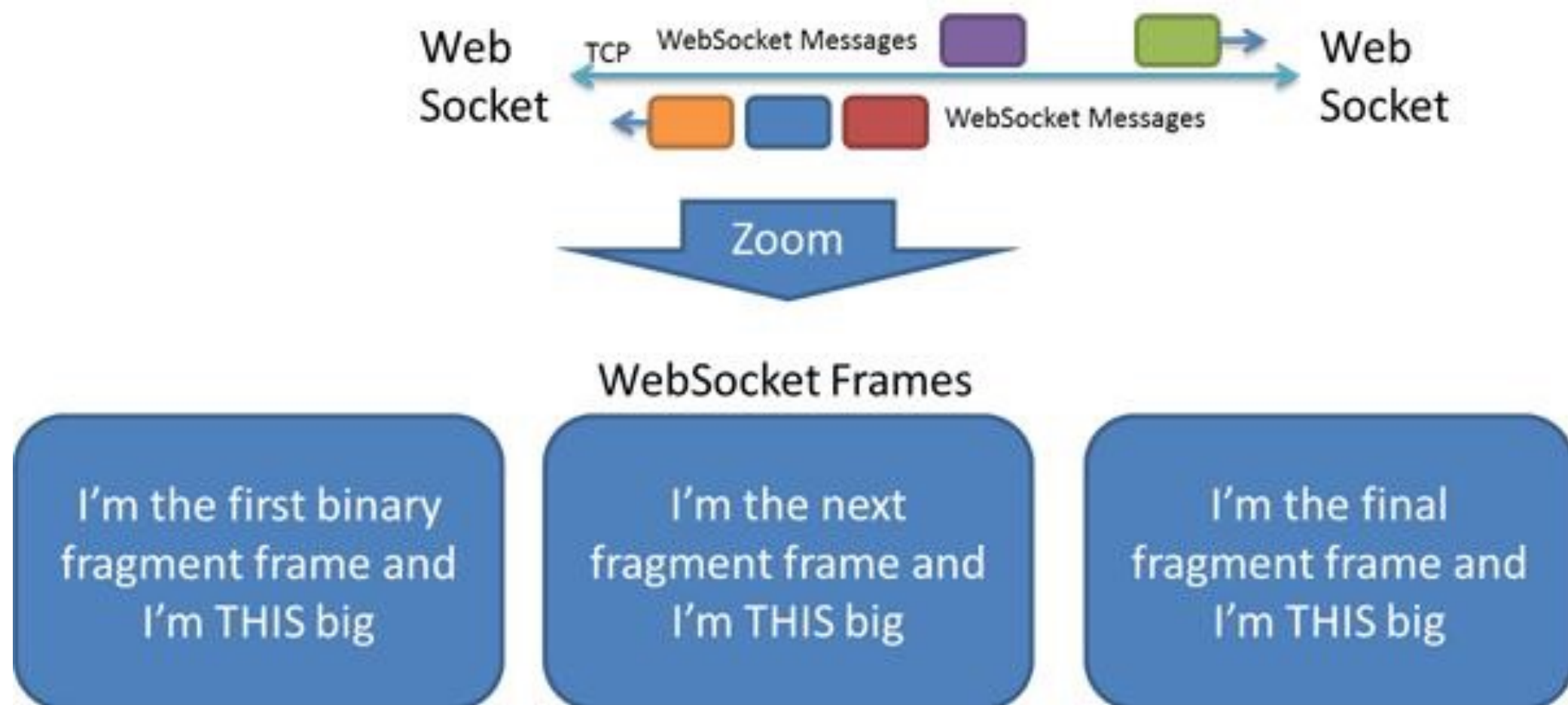
- Protocolul WebSocket definește două formate URI noi, similare cu formatele HTTP:
- `"ws:" "//" host [":" port] path ["?" query]` modelată după formatul "http:"
 - Portul implicit este 80.
 - Este folosit pentru conexiuni nesecurizate (necriptate).
- `"wss:" "//" host [":" port] path ["?" query]` modelată după formatul "https:"
 - Portul implicit este 443.
 - Este folosit pentru conexiuni securizate peste Transport Layer Security (TLS).

`ws://example.com`

`ws://example.com:8080/echo`

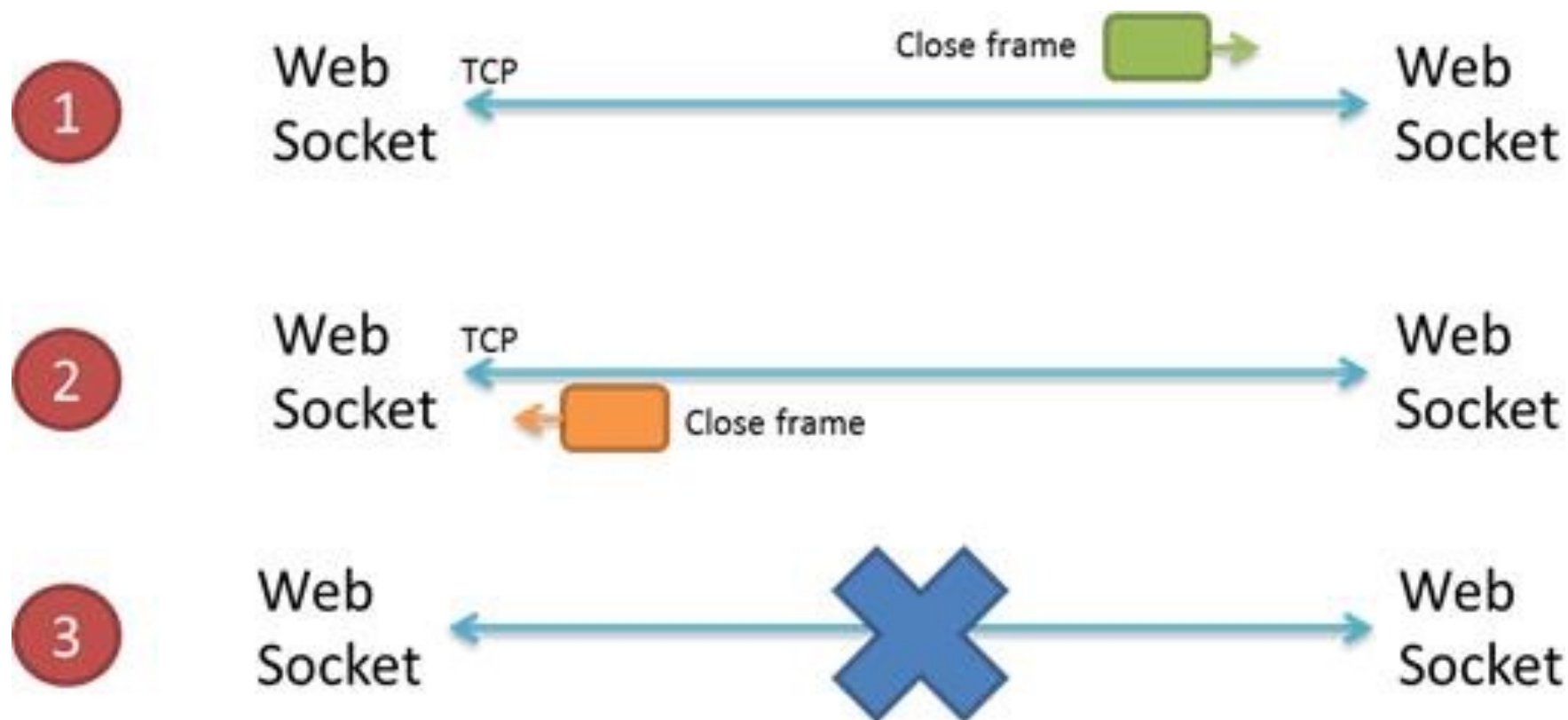
Mesaje WebSocket

- După o înțelegere (handshake) cu succes, aplicația și serverul web pot interschimba mesaje WebSocket.
- Un mesaj este compus dintr-o secvență de unul sau mai multe fragmente de mesaj/date numite "*frames*". Fiecare frame conține informații cum ar fi:
 - Dimensiunea/lungimea framelui.
 - Primul frame din mesaj va conține tipul mesajului (text sau binar).
 - Un flag (*FIN*) care indică dacă acesta este ultimul frame din mesaj.



Închiderea unui WebSocket

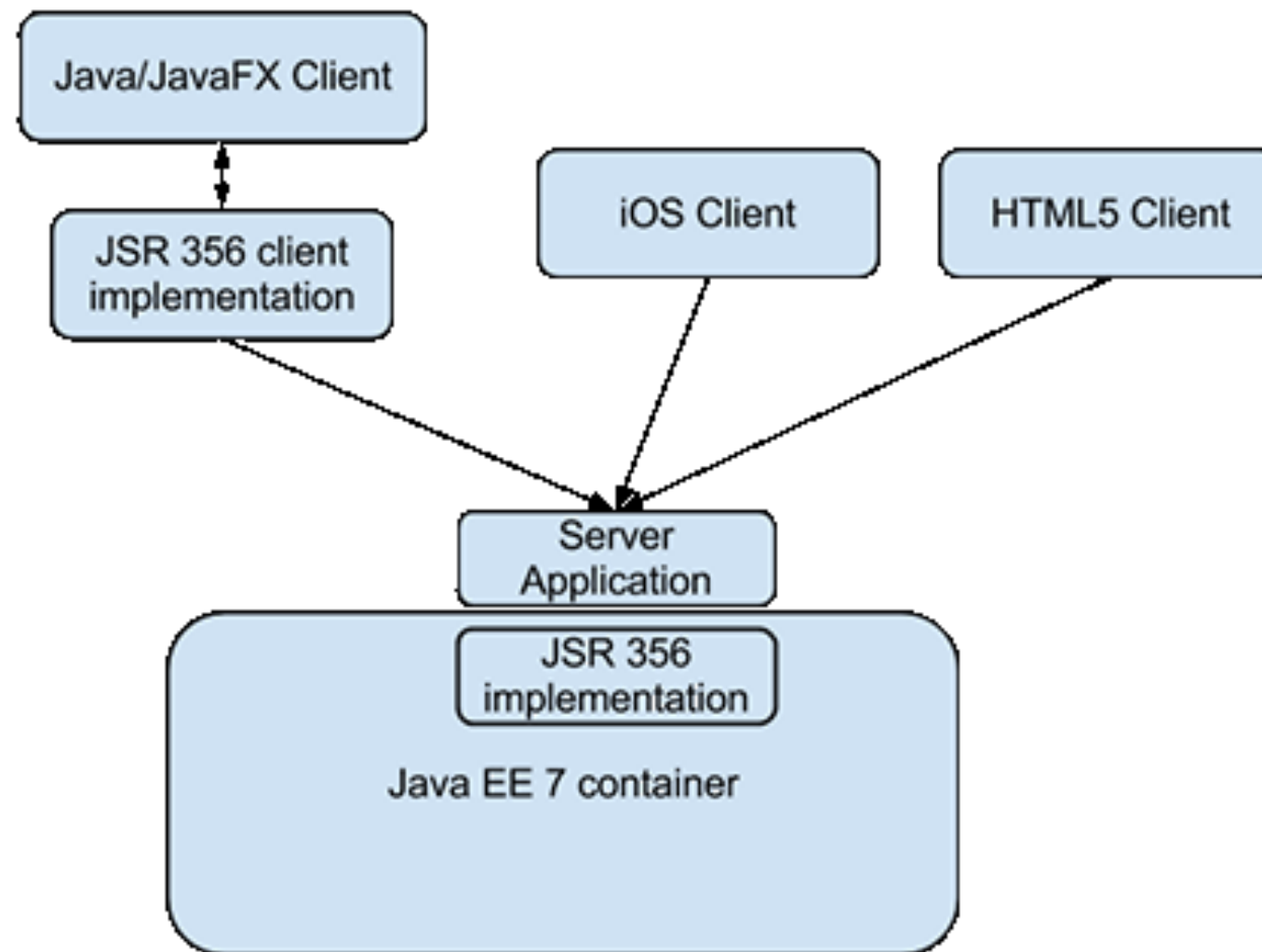
- Fiecare participant la conexiune (clientul sau serverul) poate iniția o cerere de terminare a înțelegerii (*handshake*).
- Un frame special – un frame *close* – este trimis celuilalt participant. Frame-ul *close* poate să conțină opțional un motiv pentru terminare și un status code.
- Protocolul definește un set de valori ce pot fi folosite pentru status code.
- Cel care trimite frame-ul *close*, nu trebuie să mai trimită alte mesaje după trimiterea acestuia.
- Când celălalt participant primește un frame *close*, el va răspunde cu propriul frame *close*. Poate să trimită mesajele netrimise anterior (din buffer) înainte de a trimite frameul *close*.



WebSockets API - Java

- **JSR 356**, Java API for WebSocket, specifică API-ul care poate fi folosit de dezvoltatorii Java când vor să integreze/folosească WebSocket-uri în aplicațiile lor, atât pe partea de server cât și pe partea de client Java.
- Fiecare implementare a protocolului WebSocket care pretinde să respecte JSR 356 trebuie să implementeze acest API.
- Dezvoltatorii pot dezvolta aplicațiile lor bazate pe WebSocket independent de implementarea WebSocket.
- JSR 356 face parte din standardul Java EE 7.
- Toate serverele de aplicații care respectă Java EE 7 vor avea/conține o implementare a protocolului WebSocket care respectă standardul JSR 356.

WebSockets API - Java



- Un client Java poate folosi o bibliotecă client care respectă JSR 356 pentru tratarea problemelor (eng. *issues*) specifice protocolului WebSocket.
- Alți clienți (iOS, HTML5) pot folosi alte implementări (care nu sunt bazate pe Java) dar care respectă specificația RFC 6455 pentru a comunica cu aplicația server.

WebSockets API - Java

- JSR 356 folosește adnotări și injectare.
- Sunt suportate **două modele de programare**:
 - *Bazate pe adnotări*. Se folosesc clase POJO adnotate, cu ajutorul cărora dezvoltatorii pot trata evenimentele ce apar în ciclul de viață al unui WebSocket.
 - *Bazate pe interfețe*. Dezvoltatorii pot trata evenimentele prin implementarea interfeței *Endpoint* care conține metodele corespunzătoare evenimentelor ce pot să apară.
- **Evenimentele din ciclul de viață:**
 - Un client inițiază conexiunea trimițând o cerere de tip HTTP handshake.
 - Serverul răspunde cu un handshake.
 - Conexiunea este stabilită. Din acest moment conexiunea este simetrică, bi-direcțională.
 - Ambii participanți (clientul și serverul) pot să trimită și să primească mesaje.
 - Unul dintre participanți închide conexiunea.
- Majoritatea evenimentelor din ciclul de viață pot fi mapate la metode Java, indiferent de abordarea folosită (adnotări sau interfețe).

Java WebSockets API - Anotări

- Un endpoint care acceptă cereri WebSocket (serverul) poate fi orice POJO (Plain Old Java Object) annotat cu `@ServerEndpoint`.
- Această anotare specifică containerului (web) că clasa annotată ar trebui considerată un endpoint WebSocket.
- Elementul anotării specifică calea către endpoint-ul WebSocket.

```
@ServerEndpoint("/hello")
```

```
public class MyEndpoint { }
```

```
@ServerEndpoint("/hello/{userid}")
```

```
public class MyEndpoint { }
```

- valoarea variabilei `{userid}` poate fi obținută în metodele corespunzătoare ciclului de viață folosind anotarea `@PathParam`.

Java WebSockets API - Adnotări

- Un endpoint care inițiază o conexiune WebSocket (clientul) poate fi un POJO adnotat cu `@ClientEndpoint`.
- `ClientEndpoint` nu accepta variabile de cale (ex. {userid}) deoarece nu așteaptă cereri.

`@ClientEndpoint`

```
public class MyClientEndpoint {}
```

- Inițierea unei conexiuni WebSocket, folosind abordarea bazată pe adnotări:

```
javax.websocket.WebSocketContainer container =  
javax.websocket.ContainerProvider.getWebSocketContainer();
```

```
container.connectToServer(MyClientEndpoint.class,  
new URI("ws://localhost:8080/hello"));
```

- Clasele adnotate cu `@ServerEndpoint` sau `@ClientEndpoint` sunt numite endpoint-uri adnotate.

Java WebSockets API - Adnotări

- După ce a fost stabilită o conexiune WebSocket se creează un obiect de tip `Session` și metoda adnotată cu `@OnOpen` va fi apelată.
- Metoda poate avea diferiți parametri:
 - Un parametru de tip `javax.websocket.Session` specificând sesiunea (obiectul Session) creată.
 - Un obiect de tip `EndpointConfig` care conține informații despre configurația endpoint-ului.
 - Zero sau mai mulți parametri de tip String adnotați cu `@PathParam`, care referă variabilele de cale din calea endpoint-ului(server).

`@OnOpen`

```
public void myOnOpen (Session session) {  
    System.out.println ("WebSocket opened: "+session.getId());  
}
```

Java WebSockets API - Adnotări

- O instanță de tip **Session** este validă atâta timp cât conexiunea WebSocket nu este închisă.
- Clasa **Session** conține metode care permit dezvoltatorilor să obțină mai multe informații despre conexiune.
- Conține metoda **getUserProperties()** care returnează un dicționar **Map<String, Object>** ce păstrează date specifice aplicației.
- Permite dezvoltatorilor să păstreze informații specifice în obiectul de tip **Session**, informații care ar trebui păstrate între diferite apeluri ale metodei.

Java WebSockets API - Adnotări

- Când unul dintre **participanți primește un mesaj**, este apelată metoda adnotată cu **@OnMessage**.
- Metoda poate avea **parametrii**:
 - Un parametru de tip **javax.websocket.Session**.
 - Zero sau mai mulți parametri de tip string adnotați cu **@PathParam**, care referă valorile variabilelor de cale (server).
 - Mesajul care poate fi de tip text, binar sau pong.
- Pentru fiecare tip de mesaj este permisă câte o metodă adnotată cu **@OnMessage**. Parametrii permiși pentru specificarea conținutului depind de tipul mesajului primit.

@OnMessage

```
public void myOnMessage (String txt) {  
    System.out.println ("WebSocket received message: "+txt);  
}
```

Java WebSockets API - Anotări

- Dacă metoda anotată cu `@OnMessage` returnează ceva, valoarea returnată va fi transmisă celuilalt participant la conexiune.

`@OnMessage`

```
public String myOnMessage (String txt) {  
    return txt.toUpperCase();  
}
```

- Metodă alternativă de a trimite mesaje folosind o conexiune WebSocket:

```
RemoteEndpoint.Basic other = session.getBasicRemote();  
other.sendText ("Hello, world");
```

- Metoda `getBasicRemote()` din clasa `Session` returnează o reprezentare a celuilalt participant la conexiunea WebSocket, un `RemoteEndpoint`. Acea instanță `RemoteEndpoint` poate fi folosită pentru trimiterea diferitor tipuri de mesaje (text, binar, pong).
- Obiectul de tip `Session` poate fi obținut/păstrat în metodele corespunzătoare ciclului de viață. (ex. metoda anotată cu `@OnOpen`)

Java WebSockets API - Adnotări

- Metoda `@onclose` este apelată când se închide conexiunea WebSocket.
- Poate avea **parametrii**:
 - Un obiect de tip `javax.websocket.Session`. Acest parametru nu mai poate fi folosit după ce conexiunea WebSocket este închisă (imediat după terminarea execuției metodei).
 - Un obiect de tip `javax.websocket.CloseReason` care descrie motivul închiderii conexiunii WebSocket (ex. închidere normală, eroare de protocol, serviciu supraîncărcat, etc.).
 - Zero sau mai mulți parametri de tip String adnotați cu `@PathParam`, referind variabilele de cale din URL asociat endpoint-ului (server).

`@OnClose`

```
public void myOnClose (CloseReason reason) {  
    System.out.println ("Closing a WebSocket due to  
    "+reason.getReasonPhrase());  
}
```

- In cazul **aparitiei unei erori**, va fi apelată **metoda adnotată cu `@OnError`**.

Java WebSockets API - Mesaje

- Orice obiect Java poate fi transmis sau primit ca și mesaj WebSocket.
- Trei tipuri diferite de mesaje:
 - *Text*
 - *Binare*
 - *Pong* (specifice conexiunii WebSocket).
- Pentru mesaje de tip text sunt permisi următorii parametrii:
 - **String** care reprezintă tot mesajul
 - Un tip primitiv Java sau clasa asociată care reprezintă mesajul convertit la acel tip
 - **String** și o valoare booleană care reprezintă o parte din mesaj (mesajul poate avea mai multe părți)
 - **Reader** pentru a primi întregul mesaj ca și un stream (blochează execuția până la terminarea citirii de pe stream)
 - orice obiect Java pentru care participantul la conexiune are un *text decoder* (**Decoder.Text** sau **Decoder.TextStream**).

Java WebSockets API - Mesaje

- Parametrii permisi pentru **mesajele binare**:
 - **byte[]** sau **ByteBuffer** pentru a primi tot mesajul
 - perechea (**byte[]**, **boolean**) sau (**ByteBuffer**, **boolean**) pentru a primi mesajul în părți.
 - **InputStream** pentru a citi tot mesajul dintr-un stream (*blocking*).
 - orice obiect pentru care participantul are un *binary decoder* (**Decoder.Binary** sau **Decoder.BinaryStream**).
- Parametrii permisi pentru **mesaje pong**:
 - **PongMessage** pentru tratarea mesajelor de tip pong.
- Orice obiect Java poate fi convertit într-un mesaj de tip text sau binar folosind un convertor (eng. *encoder*).
- Mesajul în format text/binar va fi transmis celuilalt participant, unde va fi reconvertit (eng. *decoded*) într-un obiect Java.
- Pentru transmiterea mesajelor WebSocket se folosește adesea formatul XML sau JSON. În acest caz convertirea/reconvertirea se reduce la transformarea unui obiect Java într-un XML/JSON și invers.

Java WebSockets API - Mesaje

- Un convertor este o clasă care implementează interfața `javax.websocket.Encoder`, iar un reconvertor este o clasă care implementează interfața `javax.websocket.Decoder`.
- Participanții la conexiunea WebSocket trebuie să știe posibili convertori/reconvertori.
- Lista convertorilor/reconvertorilor poate fi transmisă ca și elemente ale adnotărilor `@ClientEndpoint` și `@ServerEndpoint`.

```
@ServerEndpoint(value="/endpoint", encoders = MessageEncoder.class,  
decoders= MessageDecoder.class)  
public class MyEndpoint {  
    ...  
}
```


Java WebSockets API - Messages

```
class MessageEncoder implements Encoder.Text<MyJavaObject> {  
    @Override  
    public String encode(MyJavaObject obj) throws EncodingException {  
        ...  
    }  
}
```

```
class MessageDecoder implements Decoder.Text<MyJavaObject> {  
    @Override  
    public MyJavaObject decode (String src) throws DecodeException {  
        ...  
    }  
  
    @Override  
    public boolean willDecode (String src) {  
        // return true if we want to decode this String into  
        // a MyJavaObject instance  
    }  
}
```

Java WebSockets API - Messages

- Interfață **Encoder** are următoarele subinterfețe:
 - **Encoder.Text** - convertirea obiectelor Java în mesaje de tip text
 - **Encoder.TextStream** - adăugarea obiectelor Java la un *character stream*
 - **Encoder.Binary** - convertirea obiectelor Java în mesaje binare
 - **Encoder.BinaryStream** - adăugarea obiectelor Java la un *binary stream*
- Interfață **Decoder** are 4 subinterfețe:
 - **Decoder.Text** - convertirea mesajelor text într-un obiect Java
 - **Decoder.TextStream** - citirea unui obiect Java dintr-un *character stream*
 - **Decoder.Binary** - convertirea unei mesaj binar într-un obiect Java
 - **Decoder.BinaryStream** - citirea unui obiect Java dintr-un *binary stream*

Abordarea bazată pe interfețe

- Clasa `javax.websocket.Endpoint`:

- se redefinesc metodele `onOpen`, `onClose`, și `onError`:

```
public class MyOwnEndpoint extends javax.websocket.Endpoint {  
    public void onOpen(Session session, EndpointConfig config) {...}  
    public void onClose(Session session, CloseReason closeReason) {...}  
    public void onError (Session session, Throwable throwable) {...}  
}
```

- Pentru interceptarea mesajelor trebuie înregistrat un obiect de tip `javax.websocket.MessageHandler` în definirea metodei `onOpen`:

```
public void onOpen (Session session, EndpointConfig config) {  
    session.addMessageHandler (new MessageHandler() {...});  
}
```

Abordarea bazată pe interfețe

- Interfața **MessageHandler** are două subinterfețe:
 - **MessageHandler.Partial** - apariția unui mesaj parțial.
 - **MessageHandler.Whole** - apariția unui mesaj complet.

```
public void onOpen (Session session, EndpointConfig config) {  
    final RemoteEndpoint.Basic remote = session.getBasicRemote();  
    session.addMessageHandler (new MessageHandler.Whole<String>() {  
        public void onMessage(String text) {  
            try {  
                remote.sendString(text.toUpperCase());  
            } catch (IOException ioe) {  
                // handle send failure here  
            }  
        }  
    });  
}
```

WebSockets

- Exemple:
 - Broadcast
 - Whiteboard
- Referințe
 - RFC 6455
 - <https://tools.ietf.org/html/rfc6455>
 - Johan Vos, *JSR 356, Java API for WebSocket*,
 - <http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html>
 - Brian Raymor, *WebSockets: Stable and Ready for Developers*,
 - <https://msdn.microsoft.com/en-us/hh969243.aspx>

Autentificare REST

- *Autentificarea* (eng. *Authentication*): verificarea credențialelor trimise de client (utilizator, parolă)
- *Autorizarea* (eng. *Authorization*): verificarea dreptului de a accesa anumite resurse
- Antetul HTTP *Authorization*
- Constrângerea REST *stateless*: la fiecare cerere HTTP trebuie transmise informațiile necesare
- Abordări:
 - HTTP Authentication (RFC 2617)
 - Basic Authentication
 - Digest Authentication
 - OAuth 2.0

Autentificare REST

- **HTTP Basic Authentication**

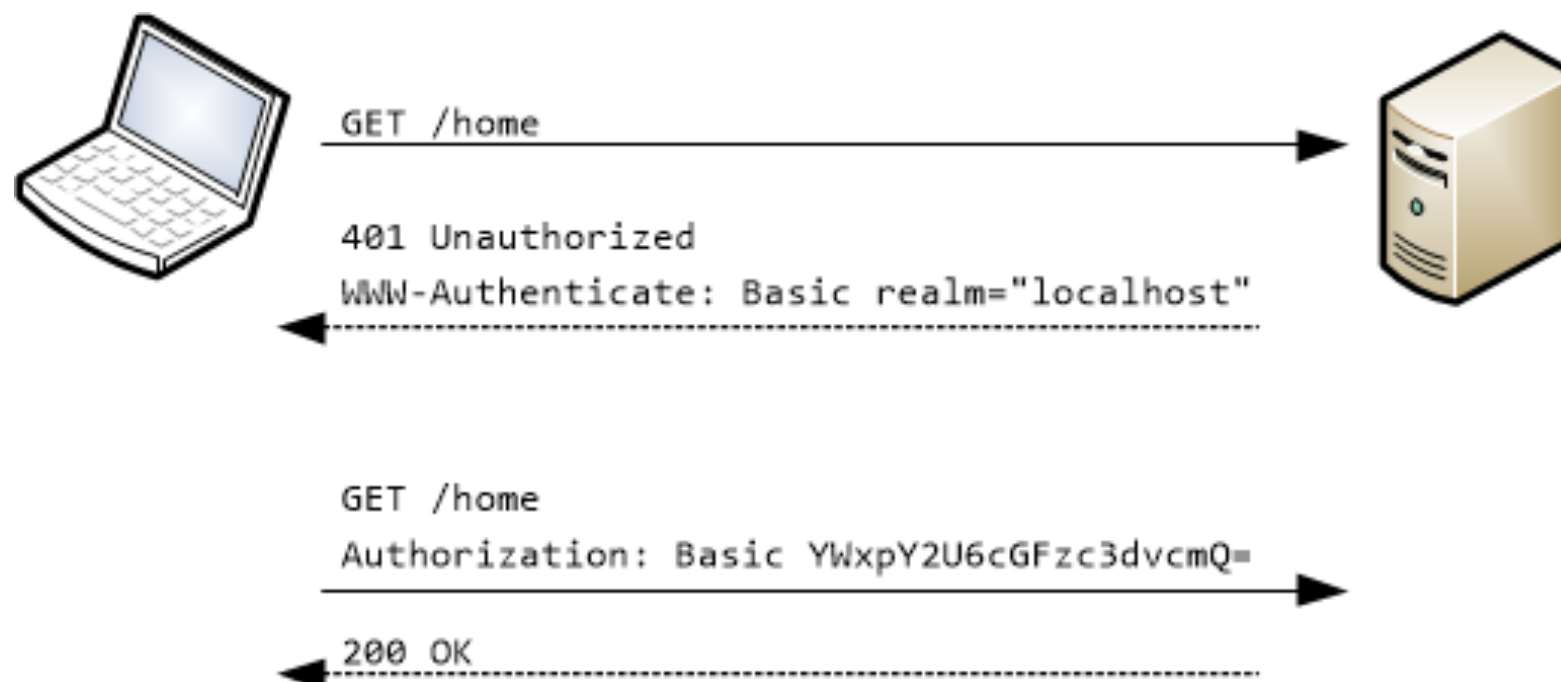
- Folosirea antetului HTTP Authorization în care se transmit username-ul și parola codificate în base64 la fiecare cerere:

GET / HTTP/1.1

Host: example.org

Authorization: Basic Zm9vOmJhcG==

- Credențialele sunt codificate, dar nu sunt criptate!
- Informațiile pot fi obținute ușor.
- Se recomandă folosirea doar cu SSL/TLS



Autentificare REST

- **HTTP Digest Authentication**

- Folosirea antetului HTTP Authorization în care se transmit username-ul, parola și alte informații codificate.
- HMAC (hash based message authentication)
- Se transmite o versiune criptată a parolei
- Se codifică și alte informații despre resursa dorită

Exemplu: utilizator “ion”, parola “ionpass”,

GET /users/ion/account

```
digest = base64encode(hmac("sha256", "ionpass",  
                             "GET+/users/ion/account"))
```

GET /users/ion/account HTTP/1.1

Host: example.org

Authorization: hmac ion:[digest]

- Parola nu poate fi criptată pe server, pentru a putea reconstrui digest
- Se recomandă folosirea unui “secret” cunoscut de client și server

Autentificare REST

- **HTTP Digest Authentication**

- **Avantaj:** un hacker nu poate modifica cererea (ar trebui să modifice valoarea *digest* și nu cunoaște “secret”-ul)
- **Dezavantaj:** un hacker poate executa cererea de oricâte ori
 - **Soluție:** se transmit mai multe informații în *digest*:
 - Data curentă
 - **Nonce** (**N**umber used *once*). La cereri subsecvente valoarea nonce este diferită.

```
digest = base64encode(hmac("sha256", "secret",  
    "GET+/users/ion/account+30may201912:59:24+123456"))
```

```
GET /users/ion/account HTTP/1.1
```

```
Host: example.org
```

```
Authorization: hmac ion:123456:[digest]
```

```
Date: 30 may 2019 12:59:24
```

Autentificare REST

- **OAuth**

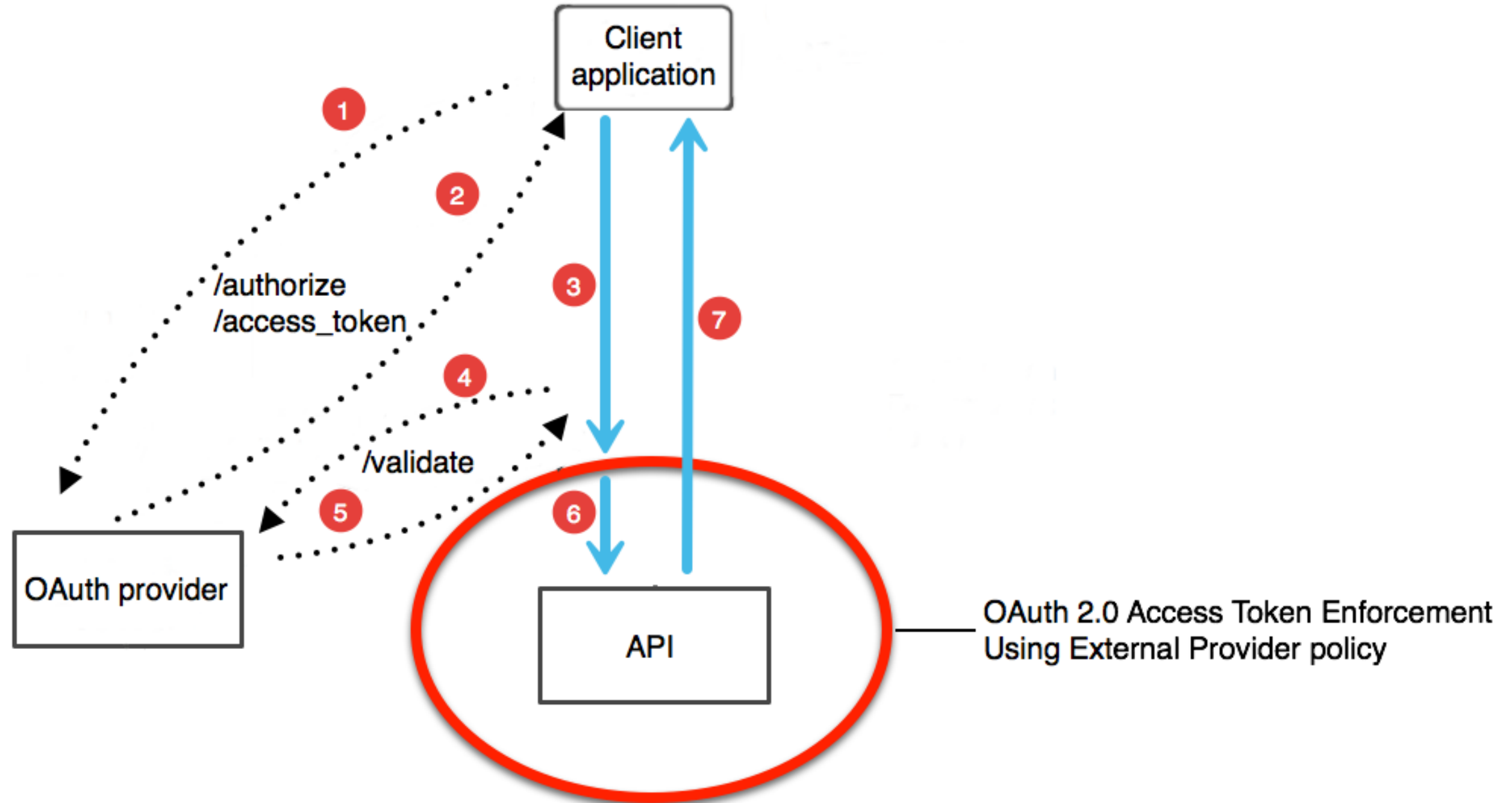
- OAuth 1.0 decembrie 2007
 - Sigur, dar dificil de implementat (criptografie, interoperabilitate)
- OAuth 2.0 octombrie 2012
 - Mai ușor de implementat
 - Compromisuri la nivel de securitate
 - Cel mai folosit

- **Flux:**

- Aplicația client se înregistrează la un **furnizor**
- Furnizorul transmite clientului un “secret” unic
- Clientul include “secret”-ul la fiecare cerere
- Dacă cererea nu este formată corect, are date lipsă sau nu conține secretul corect, ea va fi respinsă.

Autenticare REST - OAuth

OAuth Dance



JSON Web Token (JWT)

- **JSON Web Token** este un standard (RFC 7519) care definește un mod *compact* și *independent* de a transmite informații între participanți ca și un object JSON.
 - *Compact*: are dimensiuni reduse.
 - poate fi transmis prin URL, parametrii POST, antet HTTP
 - este transmis rapid
 - *Independent*: un token JWT conține toată informația necesară despre o entitate pentru a evita interogări multiple ale bazei de date.
 - cine primește tokenul nu trebuie să facă alte cereri la server pentru a-l valida.
- Informația conținută într-un token JWT poate fi verificată și se poate avea încredere în ea pentru că este **semnată digital**.
- JWT poate fi semnat folosind:
 - un *secret* (folosind HMAC)
 - cheie publică/privată folosind RSA, ECDSA, etc

JWT

- Aplicabilitate JWT
 - Autentificare/Autorizare
 - Schimb de informații între participanți (client/server)
 - Securizat
- Structura unui token JWT
 - 3 părți separate prin '.': **aaa.bbb.ccc**
 - Header (**aaa**)
 - Payload (**bbb**)
 - Semnătura digitală (**ccc**)

Header JWT

- Conține informații despre tipul de token și algoritmi de criptare folosiți pentru conținut.
- Are **două părți**:
 - tipul tokenului (JWT)
 - algoritmul de criptare (ex. HMAC SHA256, RSA)

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Payload JWT

- Conține informații despre entitate ce pot fi verificate (identitatea utilizatorului, permisiunile, etc), eng. *claims*
- 7 *claims* rezervate și recomandate (dar nu obligatorii):
 - *iss* (issuer), *sub* (subject) -utilizatorul, *aud* (audience) - pentru cine a fost creat, *exp* (expiration time), *nbf* (not before time), *iat* (issued at time), *jti* (JWT ID) -identificator unic, poate fi folosit o singură dată

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

- Poate conține și alte informații adiționale (eng. *custom claims*). Poate fi folosit orice nume, care nu este rezervat prin specificație.

Semnătura digitală JWT

- Este folosită pentru a verifica expeditorul și pentru a verifica dacă mesajul a fost modificat pe parcurs.
- Se codifică Base64 header-ul și payload-ul și se criptează folosind algoritmul specificat în header (împreună cu secretul/cheie publică-privată)

HMACSHA256 (

```
base64UrlEncode(header) + "." +  
base64UrlEncode(payload) ,  
secret)
```

- Înainte de folosirea unui token JWT trebuie verificată semnătura lui.

JWT.io

ALGORITHM

HS256



Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp1hcm1uZXNjdSBWYXNpbGUiLCJpYXQiOiE1MTYyMzkwMjJ9.J3H8MqVValMLWo1ycEOUBNax3glXeURR2XwU5Z9F9Zc
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

PAYLOAD: DATA

```
{  "sub": "1234567890",  "name": "Marinescu Vasile",  "iat": 1516239022}
```

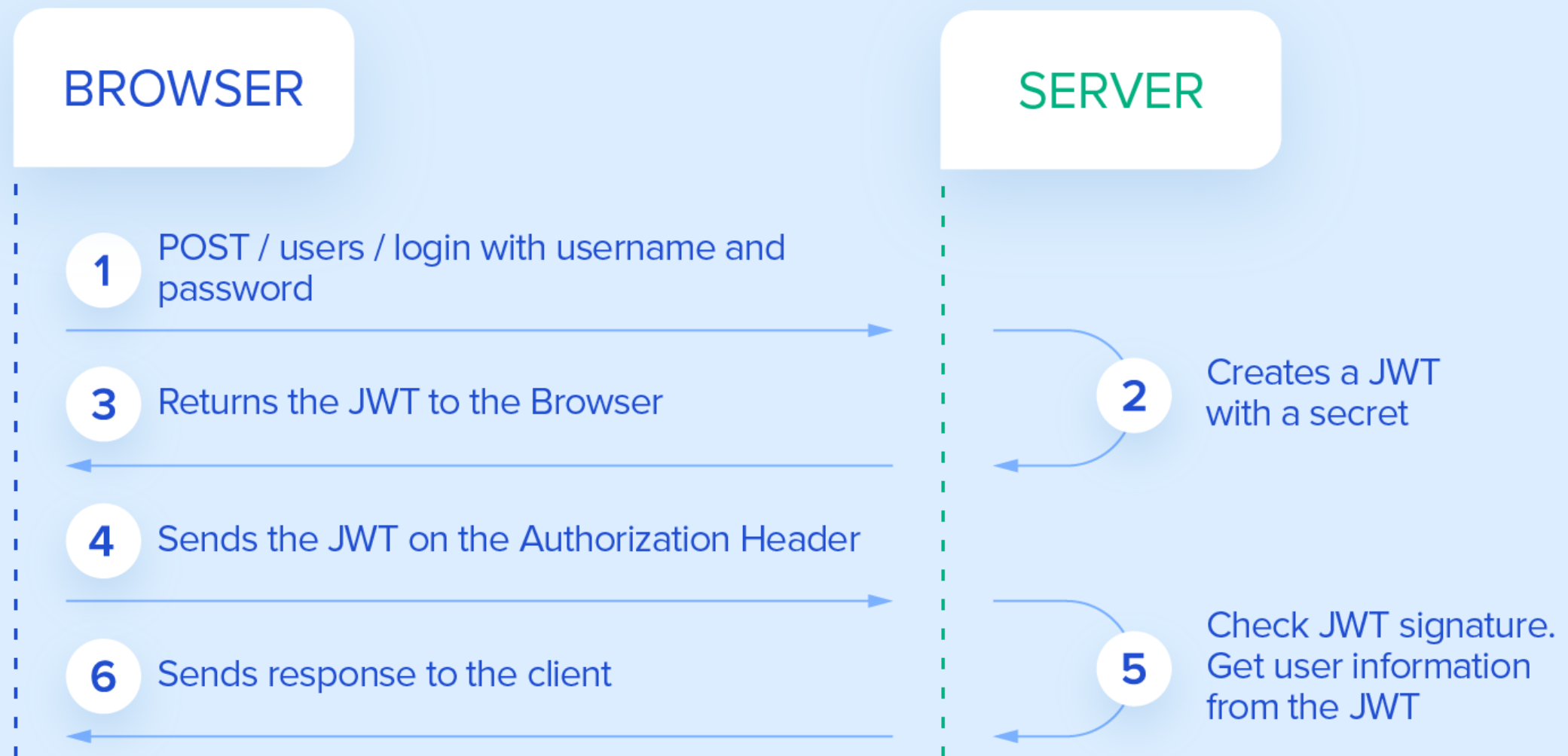
VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secretul meu  
) ☒ secret base64 encoded
```

✔ Signature Verified

SHARE JWT

JWT



Referințe

- RFC 7235 - Access Authentication Framework
<https://tools.ietf.org/html/rfc7235#section-2>
- RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication
<https://tools.ietf.org/html/rfc2617>
- RFC 6749 - OAuth2 standard
<https://tools.ietf.org/html/rfc6749>
- Guy Levin, *RESTful API Authentication Basics*,
<https://blog.restcase.com/restful-api-authentication-basics/>
- Dejan Milosevic, *REST Security with JWT using Java and Spring Security*
<https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java>
- Bruno Krebs, *Implementing JWT Authentication on Spring Boot APIs*
<https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/>

Examen

- **Calcul media finală:**
 - **10%** Teme lab (4 teme)- TL
 - **30%** Media lab - Lab (Lab ≥ 4.5 - examen in sesiunea normala)
 - **10%** Quizzuri - $MQ = (Q1 + Q2) / 2$
 - **50%** Nota examen, (NE ≥ 4.5)

Media finală = $0.1 * TL + 0.3 * Lab + 0.5 * NE + MQ$

Promovat: Media finală ≥ 4.5 !

Examen - Structura

- 2 întrebări teoretice - 15 min - 2p
- Dezvoltarea unei aplicații client server - 2 h - 8p
 - Puteți reutiliza codul sursă (teme laborator, alte exemple, etc).
 - Puteți folosi resurse de pe Internet (pentru erori, alte probleme).
 - **NU AVEȚI VOIE SĂ DISCUȚAȚI CU ALTE PERSOANE (viu grai, chat, telefon, email, social media, etc)**
 - Denumirile interfețelor, claselor, atributelor, etc. trebuie modificate conform enunțului problemei!
 - Contează arhitectura și proiectarea (interfete)!
 - Jurnalizare, fișiere de configurare, conectare la baza de date, ORM
 - 2 servicii REST (respectarea regulilor de formare a URL-ului)
 - Observer distribuit (notificare)
 - Pentru a fi evaluată, soluția dezvoltată trebuie să compileze!!!
 - Se punctează doar funcționalitățile care se execută!