# Authentication
# Session & Cookies

## User Authentication:

We cannot access our Gmail account with logging in. Once we access then the server takes care such that we don't have to log in again & again. The server identifies which user account is or are using the website. Even in AttainU portal, when we enter the credentials in the AttainU login page, the server will give us our own portal with our name and not someone else's.

### Conceptual Part of HTTP Request:

When we create a http request, either via script or form, from client to server or server to server, this http request can be of GET, POST, DELETE, PUT or any other type. Imagine this https request as a **package** or an **enclosure** or an **envelope** which is sent to the server via internet.

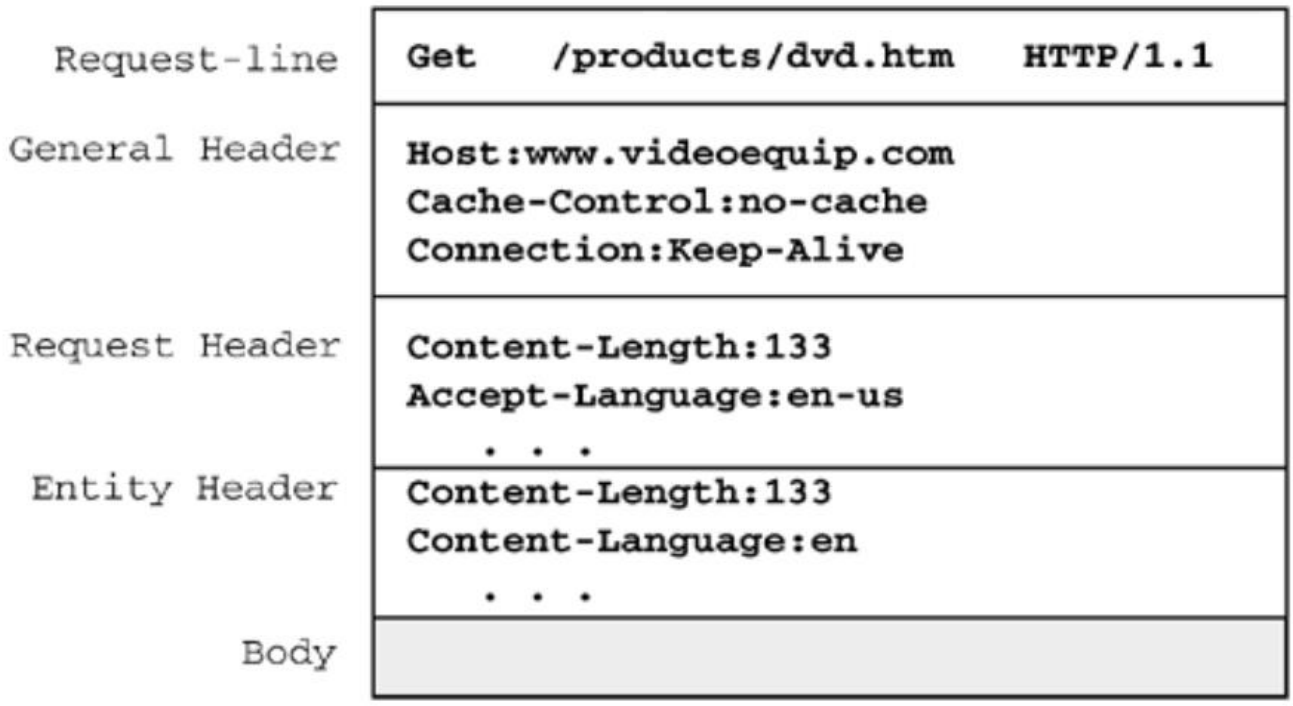
There is a **specific structure** of how this http is created.

- On high-level, on the top, there is a part called **Request Line**. It is a starting line of the request which determines what type of request it is and what URL it is going to hit. It might say POST or GET or a URL that will specify where this request will go. It will show the which version of http is being used.
- Then in the next part, there is a **Header Section**. In the http request there is always a header section which is like meta data for the request. It defines certain properties and value which defines the overall structure or content of the https request. This has information related to http request itself, what sought of data it is expecting, what time of encoding, technology and text type. It customizable and there are some standard headers as well.
- Finally, the body part. In case of POST request when we send form data, it gets into the body part of the https request. In the GET request, the body is generally empty. This has the actual content.

The header is divided into 3 part.

- General Header
- Request Header
- Response/Entity Header

Figure 17-1. Structure of a request message




In our server.js file we have two routes, one is the home route (home.hbs) and the other is profile route (profile.hbs). In the network tab, we can see everything related to your http request-response cycle.

In localhost:3000 → home route would get rendered. We receive a particular form and we can see a list of details in the network tab. When we select the localhost, it will show header-part as General, we can see all the details.

- In the request header, the **Accept** property what the type of files can the browser accept. Depending on what type of data is being sending further types can be added to **Accept** property.
- **Accept Encoding** specifies what type of encoding is there. ‘gzip’ shrinks the 400 kb to 1kb. A client can accept deflate and br encoding.

- Accept Language show what languages are set. By default, browser sets it to English, but if we set to another language, then the server will know in which language it needs to be served.
- Response Header, these are set by the server itself. When a request is passed into the server-side and server response with some data, then some more headers as being created under Response Header.
- The complete size of file is being mentioned in this. the content-type is text/html type of data. All these headers are sent automatically by the server itself.
- In date header, it tells when was a particular request is being passed.
- x-powered by is added by the express.

We can add, mutate or change request. Browser add provisional headers and the text type would be set as text/javascript. We can see the code in the response.

This structure is always maintained. In any type of request, the structure will be maintained.

In the post office example, we have created a letter and divided into 3 parts, request, header and body. In the header, there are again 3 parts; general, request and response header. We create this structure in the frontend and fill the general and request header. The response header is empty. if we have anybody or text, it would be in the body and will be sent to the server.

When it is passed in the server and handled accordingly, the server will keep request headers as it is but it will also add extra data in the response header, change the body part and will send it back to you.

What we see inside the headers is the final http request after the entire cycle is over. Until then we don't have any such details.

Inject.js part comes from chrome extension and it will have it's own request headers. We can see provisional header, which means these are not set automatically, but the browser adds a little bit of provisional headers and from wherever it is fetching from text/javascript. So, it is sending a javascript data. We can see the code in the response and in the preview, it will be a code itself and will be executed in the browser javascript engine. Based on the content type, browser takes the decision.

## Cookies:

In the book store example, we were doing something like stateful to stateless. Keeping that in mind, we will be moving forward.

In this entire structure, there is a header called **cookies**. Using this header, we can store some information in it to achieve the token-wala functionality from the book store example. We can see this in the **Request Property** of the Network Tab.

Cookie is one of many headers inside the request header section which specifies what sought of data get sent automatically. If the structure is getting maintained then everything inside it will also get maintained. This cookie header is getting sent to the server for each individual request. Once we send the cookie, whatever is stored in the cookie is getting sent to the server for each request.

So, the idea is to use the cookie header to specify certain bits of data, so that on the server side we can get the data. If a website sets certain type of cookie, it will always be getting send, like all the headers are getting sending.

The other way of seeing the list of cookies is going to Application tab and in the storage part, we can see the cookies. In this we can see, '_ga'; Google Analytics; this is how Google is tracking what we are doing. These cookies tell the particular website or server on how the user is interacting.

We will be using the cookie header to specify certain data so that in my server side I can get the data and verify the user.

## Code Examples:

On the home route we are rendering the home view (a form) and another route profile, which will render the profile view. When we go to the home page and we will be typing in some data then we can see that user's data, but when it goes back to the home page and without login also if he says profile, it would not show the same data. When user has logged-in and closes the browser and reopens it, then there should not be a need to login again. We want to achieve this.

IN our case the route is profile. Unless the users do not login, we don't want to show the profile page. Unless we don't click on the submit button, user cannot see the profile route. A POST route will not handle it well, because unless we don't click on the submit button, POST route will not be seen by the user. If we login enter localhost:3000/profile, it will not show the profile page. We will define app.get as

```
app.get('/profile', (res, req) => {
    res.render('Profile')
})
```

In this we will be showing the profile route. We want to protect the profile.

But if we put a <a> tag in the home route and link it to the profile page, that we are trying to protect, then once we click on the link, it will take us directly to the profile page without asking us to first log-in.

We want to user to be logged in to access the profile page. We want to have such behavior for such.

In the login page, user will send its credentials in the login route and then if the user is available and correct then it will be redirected to the profile page. For that, we will create a POST route. we will put a make-up collection of users for now as an array and assign it to a variable **users**.

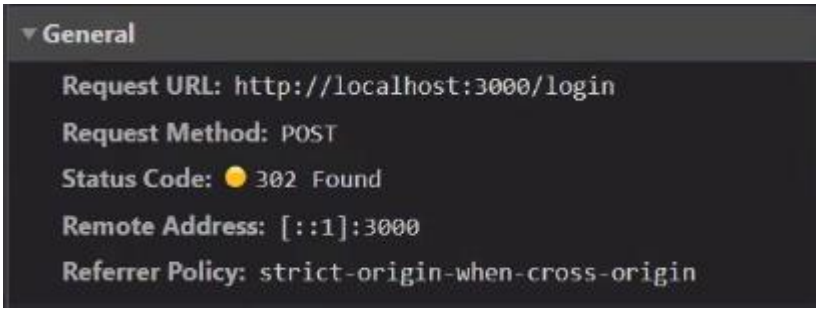
We will use a **`for loop`** with index=0 and inside it we will write,

Now, we will de structure from request.body

```
const user = [{
    email: 'johnDoe@jasper.info',
    password: '123abc'
},
{
    email:'abc@def.com',
    password: '654def'
}]

app.post('/login', (req, res => {
    const {email, password} = req.body
    for (let index = 0; index <user.length; index++){
        const userObj = user[index];

        if(userObj.email === email && userobj.password === password){
            res.redirect('/profile')
            break
        }
    }
    res.json("User not found")

}))
```

We cannot set header after they are sent to client, so instead of break we will use return, because after the entire loop is executed, **`res.json("User not found")`** is getting executed. But even now, using the a-link we are able to go the profile page using the <a> tag in the home.hbs page.

I'm receiving some data from the login part, so we will set some header and it will crerate a post request. In this we can see in browser → Network Tab → Header,




At the bottom, we will notice form data;

Form Data

email: 'johnDoe@jasper.info',
password: '123abc'

There is a way to inform the frontend that we want to set this cookie. So, we can use

```
app.post('/login', (req, res => {
    const {email, password} = req.body
    for (let index = 0; index <user.length; index++){
        const userObj = array[index];

        if(userObj.email === email && userobj.password === password){
            res.cookies('userIdentified', userObj.email)
            res.redirect('/profile')
            return
        }
    }
    res.json("User not found")
}))
```

Now when we click on login, we can see userIdentified mentioned in the cookies list along with the user data in the value. We have sent this directive through the server and the browser acted accordingly.

In the response header, we have Set-Cookie. When we called the function res.cookie, it internally sets a header inside the response area, browser will see set-cookie and the browser will set the cookie accordingly.

Once I have set this cookie, it will be sent every time to all the request. When we click on the GET request, we can see the userIdentified along with the details. And all the cookies are sent to the server every time. browser is sending the cookie. We have not enabled the server to use this cookie yet, but it is still being sent.

In the GET route we can put the check, that if it has userIdentified cookie that means it has logged in and then we can be rendering.

```
app.get('/profile', (res, req) => {
    console.log(req.cookies)
})
```

This will show undefined, that means cookies is not directly accessible. Cookies is available in the request path. Cookies is present in the request header. let us see what is in req.header → we can see an object that is similar to request headers in the networks tab.

Now that we have established all the data is being sent to the terminal, if I say **`request.header.cookie`** and log it, we can all the cookies being logged in the terminal. The same thing found in the browser is coming as text format in the terminal. We cannot use **`request.header.cookie.userIdentified`** it will show undefined because this cookie is a very big string. So we can use split() function based on semicolon. But we can use **`include('userIdentified')`**. So, we can say,

```
app.get('/profile', (res, req) => {
    console.log(req.header.cookie)

    if (req.headersSent.cookie.includes('userIdentified') === true) {
        res.render('profile')
        return
    }

    res.redirect('/')
})
```

Before testing it out, let us delete userIdentified from the cookies list. If it is not deleted then that means the user has logged-in and will be able to go to profile page.

Now, from the home page, let us try to access profile from the <a> link. It is not opening and neither is it going to the page via profile route.

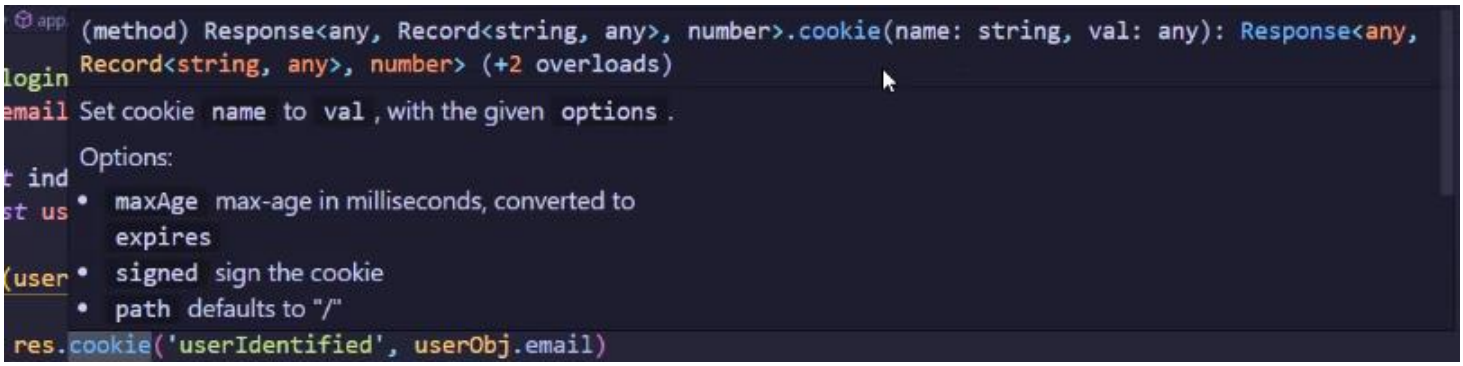
```
const user = [{
    email: 'johnDoe@jasper.info',
    password: '123abc'
},
{
    email:'abc@def.com',
    password: '654def'
}]

app.get('/profile', (res, req) => {
    console.log(req.header.cookie)

    if (req.headers.cookie.includes('userIdentified') === true) {

        res.render('profile')
        return
    }

    res.redirect('/')
})

app.post('/login', (req, res => {
    const {email, password} = req.body
    for (let index = 0; index <user.length; index++){
        const userObj = array[index];

        if(userObj.email === email && userobj.password === password){

            res.cookies('userIdentified', userObj.email)
            res.redirect('/profile')
            return
        }
    }
    res.json("User not found")
}))
```

If the user enters a different email id, based on it, the server will override.

So, for how long will this cookie be stored? We do not know the validity of the cookie. As you see, there are different parameters, and in the domain we can see the cookie value. Path by default is valid for every sub path. In expries/Max-Age it is

mentioned as Session meaning that if we close the browser then the cookie also will be removed. So, we need to set Expires/Max-Age.

The place where we are setting the cookie we can see that it accepts three parameter,




The third one is optional.

First parameter – key name

Second Parameter – value that we want to set

Third Parameter – is an object

Here, there is a property called maxAge, in which we can define how long we want the cookie to be available. We use 900000 ms = 15 mins approx.

```
app.post('/login', (req, res => {
  const {email, password} = req.body
  for (let index = 0; index <user.length; index++){
      const userObj = array[index];

      if(userObj.email === email && userobj.password === password){

        res.cookies('userIdentified', userObj.email, { maxAge: 900000})
          res.redirect('/profile')
          return
      }
  }
  res.json("User not found")
}))
```

In the login page, it would override the cookie and instead of Session, it is showing some time. if we create a new Date(‘’) and pass the string we can see it as 22:55 meaning from the current time till 30th April, the cookie will get expired and browser will remove it automatically.

```
app.post('/login', (req, res => {
  const {email, password} = req.body

  for (let index = 0; index <user.length; index++){
    const userObj = array[index];

    if(userObj.email === email && userobj.password === password){

       const expireDate = new Date('2021-04-30')
       res.cookies('userIdentified', userObj.email, { expires: expireDa
te} )
       res.redirect('/profile')
       return

     }
  }

  res.json("User not found")

}))
```

***Cookies are stored in the client side however; the session is stored in the server side.***

We can define a middleware to do the check in all the places, you want to save the route. The size of the cookie is limited, it can store only 4KB of data, not more than that.