Express session

We were storing the email id inside the userIdentified. Using this email id, we can read that in the server side and accordingly send the data. We will have the email, but will add one more property called name only to understand that we can add multiple properties.

```
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")
}))
```

In the home route, we are displaying the sign form and based on that we were seeing if that cookie existed. Everything was getting saved, but the profile we are not showing the users actual data. We were storing the email ID inside the userIdentified key. Using the userIdentified key we can read that in the server side. Let us add one more property “name”.

```
const user = [
{
    name: "John Doe",
    email: 'johnDoe@jasper.info',
    password: '123abc'
},
{
    name: "Anagram Singh",
    email:'abc@def.com',
    password: '654def'
}
]
```

The place where we are setting cookie,

```
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")
}))
```

We are only using the email in this. We will not store data such as password in the cookie. Using JSON.stringify(userObj) we will store the data in the cookie for now.

```
const jsonStr = JSON.stringify(userObj)
const expireDate = new Date('2021-04-30')
res.cookies('userIdentified', jsonStr, {expires: expireDate})
res.redirect('/profile')
return
```

We will go back to login screen and hit login button and we can see in the userIdentified cookie name, email and password is getting stored. But why to store data in cookie?

When we redirect to a particular URL, we cannot pass additional objects in your handlebars. It only works in render. We will be getting the userIdentified cookie value which will be storing the data and show it in the profile.

We could get the email ID from the cookie and then again search for user from the list and send it to the render function, but in such cases, we have to loop it again. We are storing the data directly in the cookie so that every time we go to the user profile, we already the data and will have access to the headers.cookie via server.

In req.headers.cookie we are getting a string and we have to convert the string into a javascript object so that we can have something like userIdentified.name or email. In order to have that we need to convert it into a Javascript object. We cannot pass the string directly, because then profile.hbs will not work.

So, how do we achieve that functionality?

There is a way to split this entire string using semi colon and convert it into a json object. This would be extra work and we have to write the code and logic for it. Here, we will use the **cookie parser**. It will convert the string into a object for us so that we don't have to convert it manually

Cookie parser → will convert the string into an object for me. We will install it and then as cookie-parser is a middleware, so we will call it in app.us() function.

We will install cookie-parser (`npm i cookie-parser`), then require it

```
const expHbs = require('express-handlebars');
const cookieParser = require('cookie-parser');
const express = require('express');
const app = express();
```

Cookie parser is imported and now we need to use it.

```
app.use(cookieParser())
```

We will still have to set it as req.cookie. Let us console.log req.headers.cookie and req.cookies

```
console.log(req.header.cookie)
console.log(req.cookie)
```

Parser means to give a set of data in specific format and parsing it or going through it. We need to import the cookie parser and then use it.

```
const expHbs = require('express-handlebars');
const cookieParser = require('cookie-parser');
const express = require('express');
const app = express();

const user = [
{
  name: "John Doe",
  email: 'johnDoe@jasper.info',
  password: '123abc'
},
{
  name: "Anagram Singh",
  email:'abc@def.com',
  password: '654def'
}
]

app.engine('hbs', expHbs({ extname: 'hbs' }))
app.set('view engine', 'hbs')
app.use(express.static('public'))

app.use(cookieParser())
app.use(express.urlencoded({ extended: true}))

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

      const jsonStr = JSON.stringify(userObj)
      const expireDate = new Date('2021-04-30')
      res.cookies('userIdentified', jsonStr, {expires: expireDate})
      res.redirect('/profile')
      return
    }
  }
  res.json("User not found")
}))
```

The cookie is stored and if we refresh now, in the terminal we can see req.header.cookie is coming as a string and req.cookie is coming as an object. Now, userIdentified is still containing a string, but we don't have to use split() or include().

First, req.cookies.userIdentified will be key name, if this exists in the cookie, then we will create a userObj:

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

  if (req.cookies.userIdentified) {

    const userObj = JSON.parse(req.cookies.userIdentified)
    console.log(userObj)
    res.render('profile')
    return
  }

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

We are able to see the entire string. Then we have it converted to object using cookie-parser and when we use req.cookies, it converts it into a JSON so that we can use req.cookies.userIdentified. After this, we just convert the entire string into an object and pass it in the req.render in profile route

```
res.render('profile', userObj)
return
}
```

and then in the profile.hbs,

```
<h1> Name: {{name}}</h1>
<h1> email: {{email}}</h1>
```

Output:

Name: John Doe

Email: johnDoe@jasper.info

We have to display to username and email on the front-end. We are storing it in the cookie and so whenever the cookie is sent to the server, we have to parse it in the profile part and render it accordingly. Cookie-Parser, allows you to use property name then converts all the cookies from the semi colon separate values to use that dot notations.

For now, we are mocking how to identify a user.

Why don't we use this process in the production?

We can also change the value directly from the cookies list. We have not logged-in as John Doe and email Id: johnDoe@jasper.info. If we double click on it, we can edit it. now, instead of John Doe we will put Anagram Singh and do the same changes to the email ID as well. After these changes are made now if we refresh it would change to Anagram Singh. It can be over ride.

Even though we have logged in John Doe, in the front-end we can see name as Anagram Singh. This is a big PROBLEM. We will look into to how to solve this problem.

Stateful and Stateless

What we have did to achieve the authentication, whatever functionality we are doing right now is an example of **stateless authentication**.

In stateless, we set an expiry time of the token. When this particular time comes, the cookie will expire and be removed.

But in **stateful authentication**, we can decide or revoke the access in the server. Based on the user actions in the server, we are able to authentication or revoke the session or logout the user without any user action this is also called as **stateful authentication.**

Till the expiry time, the cookie is valid. In the server we don’t have ability to revoke or nullify the particular token or cookie. Now, instead of this if I used a server-side magic or the server decides whether a particular person will get access for two or three days or even hours or even seconds. This is **stateful authentication**.

In stateful, server will decide how long a cookie is valid.

How to achieve this functionality?

The idea is we want to use cookies but in a secure way, so that no one can change it. We don’t want to store the entire data in a plane string. In order to get this done, we have data-signing or encryption.

Data ➔ name, age, email.

There is an encryption algorithm and, in the server, we will be generating a secret token or a secret key. It can be a random string or a password. This will be the key which we use to encrypt this entire data. Let us say our secret key is ABC. So, before sending this data to the client, we will take this secret key, do some operations and generate a random hash string, which will contain the whole data in encrypted form. And this hash string will be shared to the front end to store as a cookie.

The benefit is, once the cookie is set with this unique token, for each subsequent request, this cookie will be sent for each subsequent request. We will do some algorithmic operations that only the server knows.

The server will take this token which is ABC do some operations which will spit out the exact data in the server.

For this exact same reason, we will be using **express session** package. This entire thing will do it and it will give me more ability as well.

We can actually get rid of **cookie parser** and the **express session** itself is capable of handling it for me.

We will install **npm i express-session** then require it,

```
const session = require('express-session');
app.use(session())
```

Now, it comes with some pre-configuration. But in order to over write it, we provide an object. In the example the first required parameter is secret. So, when we say secret, we are providing the ABC value. The express-session will use this secret key in order to do the encryption and decryption part.

We were storing string in our cookie, which can be edited. So, we will store an encrypted string and in the server side we will decode it and obtain the data. The production side, the secret kye is a long string, which increase the strength of the encryption. We will store data and save cookie to identify the user. Now, we will store cookie and data will be stored in encrypted format. For now, we will give a simple key as "Apple". ➔ **`secret: "Apple"`**

The next option is resave: which would be set to false. This is used or meaning, unless we don't modify the cookie in somewhere, express session will not save that cookie. This is relevant when you have the login and logout flow. When the user has actually signed in and verified the user email and password then we modify the session or data on that session only then will my expression session stories the cookie in the frontend. ➔ **`resave: false`**. If we put in true, no matter we have the authentication flow, express-server will save the cookie in the frontend.

The second part is saveUninitialized. We see that if a new user uses the website, the express-session should not create automatically a session for that user. Only when we modify session variables, only then save the particular cookie in the server side and front end as well. **`saveUninitialized: false`**

Finally, cookie: which is an object. We will only provide maxAge. We are encrypting the data along with the key and generating the token and this toke is getting saved in the cookie in the front end. **`cookies: {maxAge: 900000}`**

```
app.use(session{
  secret: 'Apple',
  resave: false,
  saveUninitialized: false,
  cookie: {
    maxAge: 900000
  },
  store: myStore
})
```

We were doing cookie parse, but now it would be taken care by the express session. For each incoming req, express session would create a session for it meaning, for each incoming request, it will create a variable or stateful object which will track the user identification token or data. SO, when a particular user logs in, we can say, req.session is authenticated or logged-in. We are storing a Boolean value and an object.

```javascript
const expHbs = require('express-handlebars');
// const cookieParser = require('cookie-parser');
const session = require('express-session');
const express = require('express');
const app = express();

const user = [
{
  name: "John Doe",
  email: 'johnDoe@jasper.info',
  password: '123abc'
},
{
  name: "Anagram Singh",
  email:'abc@def.com',
  password: '654def'
}
]

app.engine('hbs', expHbs({ extname: 'hbs' }))
app.set('view engine', 'hbs')

app.use(express.static('public'))

// app.use(cookieParser())
app.use(session())
app.use(express.urlencoded({ extended: true}))

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

  console.log('IN PROFILE', req.session)

  if (false) {

    const userObj = JSON.parse(req.cookies.userIdentified)
    console.log(userObj)
res.render('profile', userObj)
return
}

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

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

  console.log('IN LOGIN', req.session)

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

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

      req.session.isLoggedIn = true
      req.session.user = userObj
      res.redirect('/profile')
      return
    }
    }
  res.json("User not found")
}))
```

When we hit login, we can see connect.sid is visible in the cookies list. So, express-session did the following. When we clicked on login button, the user details were verified and since it was true and then it set some variables in the session variable.

For IN LOGIN,

```
IN LOGIN Session {
 cookie: {
   path: '/'
   _expires: 2021-04-28T18:58:2.6242,
   originalMaxAge: 900000,
   httpOnly: true
 }
}
```

it created an object. But for the IN PROFILE,

```
IN LOGIN Session {
 cookie: {
   path: '/'
   _expires: 2021-04-28T18:58:2.65iZ,
   originalMaxAge: 900000,
   httpOnly: true
 },
 {
   1stLoggedIn: true,
   user: {
     name: 'John Doe',
     email: 'john.doe@gmail.com'
     password: '123abc'
   }
 }
}
```

The object created has changed. The cookie was logged in and the userObj also got logged in. What the express session did, is to encrypt it and using a secret key and generated specific token. All this data is available in the string we can see in the cookies list in the browser.

In the profile route,

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

  console.log('IN PROFILE', req.session)

  if (req.send.isLoggedIn === true) {

    res.sender('profile', req.session.user)
    return
}

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

We have logged in and even ID has been changed because the server had to restart. If we make any changes in the cookie, he will not be able to login. HE will have to re-login. This is stateful because, if we wanted to, req.session.destroy, will destroy the session. This control we have instead of giving it to the server. If we wanted to rechange it, by calling this particular method (resetMaxAge).

Now, how is expression.session actually referencing all that variables. In the bookshelf example, Desh Raj was maintaining a unique token and then the associated data. In the same express session is doing the same thing. Express session will generate a token which will be referring to whatever the data we have set in the login route or any other route. Express-session takes note of isLoggedIn as a variable that should be store and will make it true {isLoggedIn: true} and then sent this data back to the front end. A front end is only sending and receiving the token. The actual data is residing in the server rather than the front end. In previous case, we were storing data in the frontend, but now we are storing data in the server, and it is more secure.

When we click on login, we will login to the page and then if we make any changes in the index.js and save the file, then the server gets restarted. So, if we refresh now, we are redirected to the login page, because when the cookie was generated, server was running. So, the default configuration that the express-session uses is to store all the variables or token in the memory. So as the server got killed or restarted, it totally forgets about the data, as all this data is in the RAM or memory.

To solve this, we need to understand that by default express-session is using in-memory meaning it is storing everything in the RAM. The place were express-session stores or manages this entire thing, they call it as **store**. In this store, express-session stores everything. What will be that store?

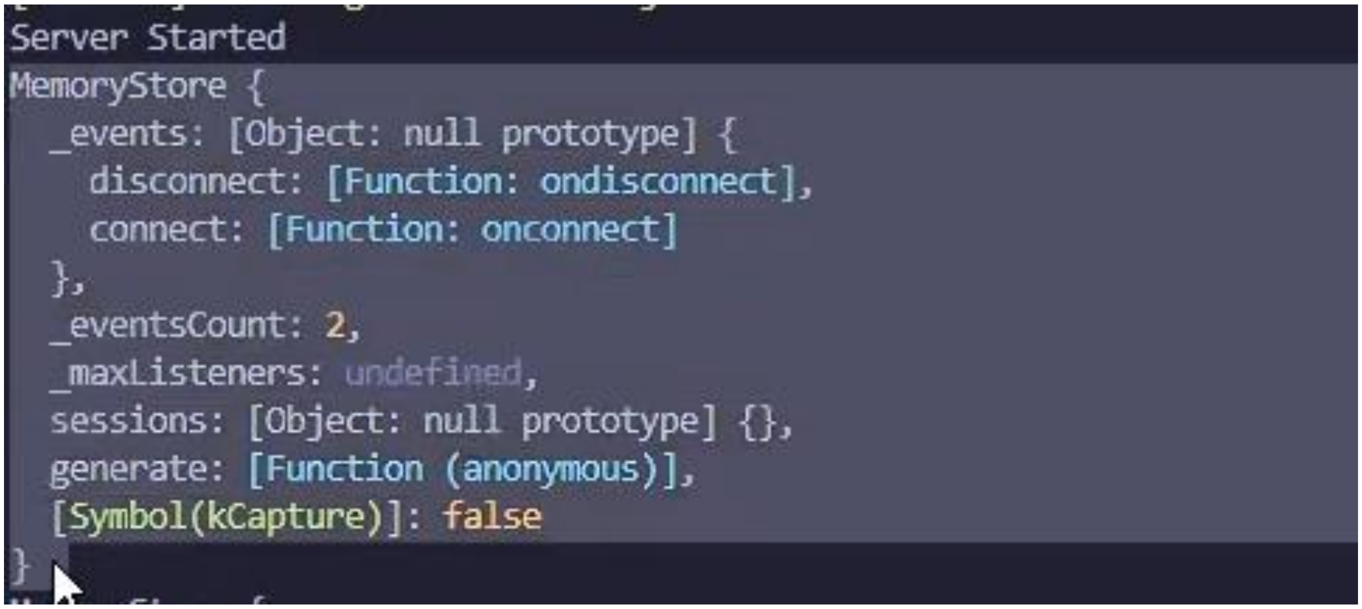
Store will have some token and relevant data. All this data is be managed internally by express-session. Right now, depending on what store you are using, whether a memory or file or database store, the store will save the data in that location. Currently, it is storing in my RAM (by default). If we use a file store, it will save it in the hard disk. If we use a database store, it will store the session data in the database.

```
const myStore = new session.MemoryStore()
const session = require('express-session');
```

Express-session is currently creating its own store and accessing it. So, we will create the memory store manually outside that and tell my request outside that. So,

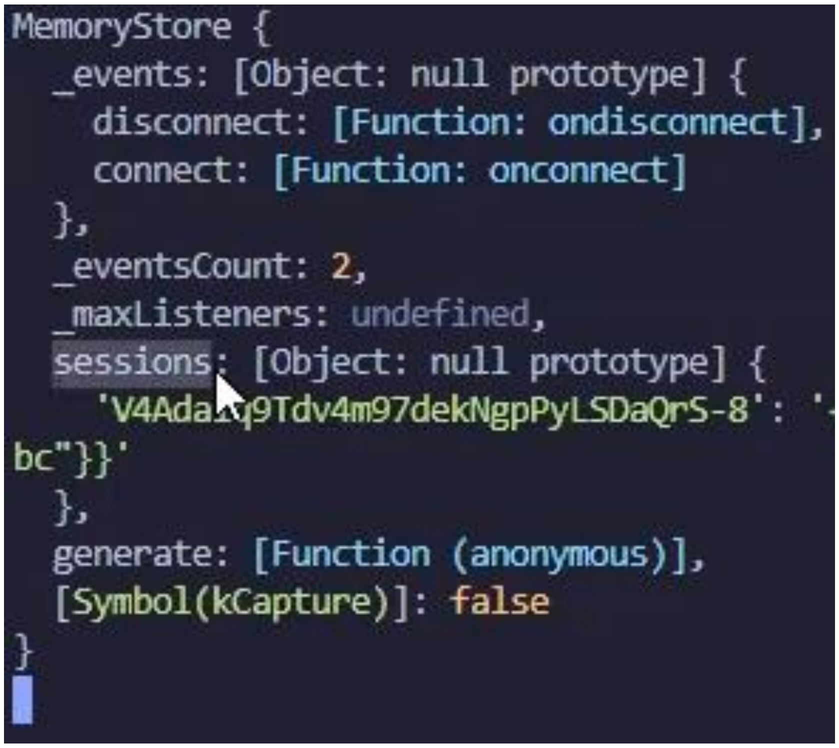
```
app.use(session{
  secret: 'Apple',
  resave: false,
  saveUninitialized: false,
  cookie: {
    maxAge: 900000
  },
  store: myStore
})
```

This will allow us to access the store. so now, if we console.log(myStore) in the profile route. Now when we login and redirected to the profile, let see what is in the store. In the store,



```
Server Started
MemoryStore {
  _events: [Object: null prototype] {
    disconnect: [Function: ondisconnect],
    connect: [Function: onconnect]
  },
  _eventsCount: 2,
  _maxListeners: undefined,
  sessions: [Object: null prototype] {},
  generate: [Function (anonymous)],
  [Symbol(kCapture)]: false
}
```

Initially there was nothing but there is a session variable maintained.



```
MemoryStore {
  _events: [Object: null prototype] {
    disconnect: [Function: ondisconnect],
    connect: [Function: onconnect]
  },
  _eventsCount: 2,
  _maxListeners: undefined,
  sessions: [Object: null prototype] {
    'V4AdaXq9Tdv4m97dekNgpPyLSDaQrS-8': '
bc"}}'
  },
  generate: [Function (anonymous)],
  [Symbol(kCapture)]: false
}
```

The id here in the terminal and the value in the cookies list for connect.sid is matching the same thing. This is exact thing which is actually getting matched. In this property, it key again has a json object in which we have a json string with the values as we have mentioned in the code.

This time when we login with the second user details, we are able to see another ID. If we go to edge browser's cookie, then connect.sid is matching it. when the login request came for John Doe, express session knew internally which object needs to be sued for him. and when we logged in using Anagram, express-session created another one for me and it got stored in the internal directory. When the user is getting redirected to the profile, based on the ID which is stored on the token basis, express-session is able to get the data. When the express-session goes through the cookie data, it finds the match, get the entire thing, passes it through the json.pass and give it to us. The same thing goes for Anagram Singh as well.

In the store we are actually using a memory store or a file store or a mongoDB store. So, all the store will be managed in the mongodb.