Module 0340: Express session

Tak Auyeung, Ph.D.

October 24, 2021

Contents

1 About this module

- Prerequisites:
- Objectives: This module introduces the concept of sessions, then demonstrates how it is implemented in Express.

2 Session as a concept

HTTP (HyperText Transport Protocol) is a connection-based protocol, but it uses one connection per request. Consecutive connections from the same tab of the same browser are not related as far as HTTP itself is concerned.

This poses a problem because how can an online merchant know which customer just clicked on "buy now" of an item? The entire concept of authentication also seem pointless because the authentication and identification of an account only lasts for one connection, subsequent clicks are logically not connected to the identified user.

One way to create the illustion that subsequent clicks (HTTP requests) are related is to add a parameter to the GET requests that identifies the continuity. However, this approach has several problems. First, if the continiuity ID is leaked, someone else can easily hijack an identity. Second, this requires all links to include the parameter which is tedious.

A *session* is the maintenance of the pretence of continuity between HTTP requests. As such, a session may include various kinds of information, including but not limited to user identity, shopping carts, and etc.

3 Express sessions

The Express framework include modules to handle sessions in a secure and convenient manner.

3.1 Under the hood

Like most web scripting environments, Express utilizes cookies to identify a session. Let us examine cookies.

A cookie has several parts. Of importance to us are the name, URL path, and a value. Typically, upon the response of the first request to a server, the server specifies a cookie. When the client (browser) receives the response, it creates the cookie on the client side (managed by the browser).

Once a cookie is created, the client (browser) transmits the cookie and its value every time a request is made to an URL that matches the URL path of the cookie. This is how continuity is maintained between HTTP requests to all the URL paths that match the URL path specification of a cookie.

However, a cookie can only contain a small amount of data, and it is not secure to store any sensitive data on the client side. As a result, the value of a cookie is only used to *identify* a session, but the actual data associated with a session is stored on the server side.

To this end, the `express-session` module is sufficient. However, this module implements the simplest "store", in-memory. This means that when a server (just the Express script) is restarted, all previous sessions would have been forgotten.

A more robust method that allow sessions to persist server reboots is to maintain data in a database (like MariaDB). This is implemented by the node module `express-mysql-session`.

3.2 Confusion between sandbox and production apps

This discussion is not only because of the use of sessions, but the use of databases, in general.

Unless precaution is made, a production app and a sandbox app using the same code base use the same database and the same tables in the database. This can be an issue because it is unwise to use production data to test code in the sandbox! Furthermore, it is also common practice to track several datasets (tables and their contents) for testing purposes.

Another limitation is that in most environments, both the production app and the sandbox app need to use the same database. Given this restriction, one way to keep the production and sandbox separated is to use a *prefix* for the tables.

Whether non-session tables in the database can be shared or not, session tables cannot be shared. Otherwise, sessions of the production app and sessions of the sandbox app can cross over.

When the credential object is used to create the "store" of Express sessions, it can use an optional member to determine the prefix of the tables used to store data related to sessions. We will examine how this can be done in the next section.

4 A complete example

Listing 1: A complete example using Express sessions

```
001 "use strict";
002 module.paths.unshift('/usr/lib/node_modules')
003 const fs = require('fs') // module to handle file system
004 const https = require('https') // module to handle HTTPS as a protocol
005 const express = require('express') // module to handle express framework
006 const asyncHandler = require('express-async-handler')
007 const app = express() // an express instance
008 // the following line gets the private key needed for SSL
009 const privateKey = fs.readFileSync('/var/local/ssl/selfsigned.key','utf8')
010 // the following line gets the certificate needed for SSL
011 const certificate = fs.readFileSync('/var/local/ssl/selfsigned.crt','utf8')
012 // the following line gets the port number Express listens to
013 const portNumber = fs.readFileSync('.port','utf8').trim()
014 //
015 async function delay(ms, value=undefined)
016 {
017   return new Promise(
018     (resolve, reject) =>
019     {
020       setTimeout(
021         () => { resolve(value) }, ms
022       )
```

```
023       }
024     )
025   }
026
027   async function epRootHandler(req, res)
028   {
029     if ('session' in req) // just checking, but session should be a part of the
030                           // 'req'uest object
031     {
032       // check whether there are parameters specified, and whether
033       // haveEnough is one of them
034       if ('query' in req && 'haveEnough' in req.query)
035       {
036         // if haveEnough is a parameter, is the value 1?
037         if (req.query.haveEnough==1)
038         {
039           // alright, the user has enough already, reset wait time
040           req.session.wait = 0
041         }
042       }
043       // what if there are no query parameters and the wait session
044       // variable is present?
045       else if ('wait' in req.session)
046       {
047         // wait a little before responding
048         await delay(req.session.wait * 1000)
049         req.session.wait++
050       }
051       // what if there are not query parameters, and there is no wait
052       // session variable?
053       else
054       {
055         // create it! this is the initialization of a session variable, which
056         // indirectly initializes a session
057         req.session.wait = 0
058       }
059     }
060     else
061     {
062       // this is bad! should never get here!
063       throw new Error("session is not initialized!")
064     }
065     res.write(`<!DOCTYPE html><html><head></head><body>`)
066     res.write("<h1>Got you sweating?</h1>")
067     // give the user a warning of the next wait time
068     res.write(`<p>The next refresh will wait ${req.session.wait} seconds`)
069     // the following specifies an anchor with a href to the same page
070     // but specifies a parameter of haveEhough=1
071     res.write('<p><a href="?haveEnough=1">I have enough!</a></p>')
072     // the following specifies an anchor with a href to the same page
073     // without any parameters
074     res.write('<p><a href="">I can wait longer.</a></p>')
075     res.write(`</body></html>`)
076     res.end()
077   }
078
079   const mdb = require("mysql-await")
```

```
080 const os = require("os") // needed for os.homedir() because ~ does not expand
081
082 // the following reads the JSON file that contains the credential and other
083 // configuration information of the database
084 let mdbSpecs =
085   JSON.parse( // decode JSON content
086     fs.readFileSync( // from the file
087       os.homedir()+"/.mysqlSecrets.json", // in the home folder
088       { encoding: "utf8" }
089     )
090   )
091 // the following specifies schema that is specific to the tables
092 // used by the database to track sessions
093 mdbSpecs.schema = { tableName: `s_${portNumber}_session` }
094
095 // the following creates and configures an object to represent
096 // a connection, but makes no attempt to connect
097 // this is technically not needed in this script
098 let mdbConnection = mdb.createConnection( mdbSpecs)
099 // the following actually communicates with MariaDB to try to
100 // make a connection, this takes time!
101 // this is also not needed in this script
102 mdbConnection.connect() // for non-session use
103
104 // the following brings in the session middleware callback
105 const session = require('express-session')
106 // the following brings in the mysql/MariaDB based session store handler
107 // and associate it with the session callback
108 const sessionDbStore = require('express-mysql-session')(session)
109 // the following associates the session store handler with a specific
110 // database using the credential stored in mdbSpects
111 let sessionStore = new sessionDbStore(mdbSpecs)
112
113 // the following specifies session handler as one of the middle ware
114 app.use(
115   session(
116     {
117       key: `s${portNumber}Cookie`, // use port number to distinguish
118                                    // production vs sandbox
119       secret: `${portNumber}`,     // different secrets, too
120       store: sessionStore,         // database store
121       resave: false,               // do no resave unchanged data
122       saveUninitialized: false, // uninitialized sessions are not stored
123       cookie:                      // session cookie properties
124       {
125         maxAge: 60*60*1000,        // expire in one hour (in milliseconds)
126         path: '/',                 // apply to all end-points
127         secure: true               // same site only
128       }
129     }
130   )
131 )
132
133 // specify end points and handlers for each end point
134 app.get('/', asyncHandler(epRootHandler))
135
136 // an object needed to create the HTTPS server
```

```
137 const credentials = { key: privateKey, cert: certificate }
138 // create the HTTPS server
139 let httpsServer = https.createServer(credentials, app)
140 // start the server
141 httpsServer.listen(portNumber)
```

You can download this script.

This is a fairly complex program. This program makes use of the GET parameters of a HTTP request as well as the concept of session to maintain continuity between clickes. At the lowest level, the continuity is maintained by the use of cookies that are stored on the client side.

4.1 Initialization

In addition to the "typical" iniitalization of an Express script, this program also makes use of the database. This initialization start with reading the file that contains credential to connect to the database.

Listing 2: Reading database access credential

```
082 // the following reads the JSON file that contains the credential and other
083 // configuration information of the database
084 let mdbSpecs =
085   JSON.parse( // decode JSON content
086     fs.readFileSync( // from the file
087       os.homedir()+"/.mysqlSecrets.json", // in the home folder
088       { encoding: "utf8" }
089     )
090   )
```

This step does not actually do anything with the database, it merely reads the content of a file

Listing 3: Specifying the name of the session table

```
091 // the following specifies schema that is specific to the tables
092 // used by the database to track sessions
093 mdbSpecs.schema = { tableName: `s_${portNumber}_session` }
```

Next, based on what is read from the credential file, a new member `schema` is added. The `schema` itself can have multiple members, but the one that we need here is `tableName`.

Because we are running both the sandbox and production apps on the same server, using the same database, the tables used to track session must have different names. In this case, the use of the backquote expands the value of `portNumer` within the string. This means that if the port number is 41220, then the actual name of the table is `s_41220_session`.

This works because the sandbox and production apps use different port numbers.

Listing 4: Bringing in the `express-session` module

```
104 // the following brings in the session middleware callback
105 const session = require('express-session')
```

The `express-session` module is a "middle-ware" in the sense that it gets to read/parse/process a request before an Express end-point handler is eventually called. However, at this point, the module is simply loaded, but it is in no way connected to either the database or the Express framework.

To maintain modularity, `express-session` explicitly leaves out *how* information tracked by sessions is stored. This is because different server environment may have different methods to maintain session information.

This brings us to the following code.

Listing 5: Bringing in the `express-mysql-session` module

```
106 // the following brings in the mysql/MariaDB based session store handler
107 // and associate it with the session callback
108 const sessionDbStore = require('express-mysql-session')(session)
```

This code loads the `express-mysql-session` module, which is specifically developeed to interface with `express-session` and utilize a MySQL/MariaDB data to maintain session information.

In case you are wondering, internal to `express-mysql-session`, callback functions are utilized to handle asynchronous operations. This has zero impact, whatsoever, to any Express end-point handlers because all database operations related to the maintenance of sessions occur before and after the execution of Express end-point handlers.

In this step, the module of `express-mysql-express` is loaded, and it is linked to the `express-session` module. However, there is no database operation performed.

Here is the step that initializes database connection for session information maintenance.

Listing 6: Connecting the session module to the database

```
109 // the following associates the session store handler with a specific
110 // database using the credential stored in mdbSpects
111 let sessionStore = new sessionDbStore(mdbSpecs)
```

This step utilizes the object `mdbSpecs` to initialize the connection to the database. If the credential data is incorrect, this step fails. At this point, the connection to the database is made, and the modules are properly loaded and initialized. However, the Express framework is completely unaware of the session "middleware". This step creates an object that, in return, is used in the next step.

Listing 7: Adding session as "middleware" in Express

```
113 // the following specifies session handler as one of the middle ware
114 app.use(
115   session(
116     {
117       key: `s${portNumber}Cookie`, // use port number to distinguish
118                                    // production vs sandbox
119       secret: `${portNumber}`,     // different secrets, too
120       store: sessionStore,         // database store
121       resave: false,               // do no resave unchanged data
122       saveUninitialized: false, // uninitialized sessions are not stored
123       cookie:                      // session cookie properties
124       {
125         maxAge: 60*60*1000,        // expire in one hour (in milliseconds)
126         path: '/',                 // apply to all end-points
127         secure: true               // same site only
128       }
129     }
130   )
131 )
```

This is where the initialization completes from the perspective of setting up Express to handle sessions. This code is long because it also specifies many of the parameters related to how session cookies are created. These parameters are captured by the object (note the use of open brace `{` and close brace `}` to specify the object).

The `key` is used to name cookies uniquely on the client side. Imagine that when your professor is grading, the app of each student sets a cookie on the professor's browser. If the cookie name is not unique, then the browser can end up cross talking across the apps of different students, leading to massive confusion.

This is why `portNumber` is expanded and becomes a part of the identifier of the cookie.

The `secret`, on the other hand, do not need to be unique from the perspectives of making use cookies from different apps do not cause confusions. The `secret` is also a key of sorts, but it is the key of encryption. This parameter makes it difficult for a hacker to *spoof* cookies to attempt to hijack a session.

The `store` is where we utilize the rather long initialization process. This is where the session module understand how session information is maintained. The rest of the parameters are of less importance and therefore not explained in any further detail (the comments explain in a brief manner).

4.2 The end-point handler

The logic of the end-point handler is more complex than our previous sample code. First of all, it illustrates the use of nested statements where one statement becomes a part of another one. Secondly, it utilizes sessions and query parameters at the same time to determine what to do.

The only end-point is `/`, and the call-back (handler) is `epRootHander`.

Listing 8: The end-point handler, just the shell

```
027 async function epRootHandler(req, res)
028 {
077 }
```

An end-point handler actually has more than two parameters, but only two is in use in this example. `req` is an object representing the request, and `res` is an object representing the response.

The main logic to take into consideration of the session-maintained values (`wait`) and the request parameter (`haveEnough`) is from line 29 to 64.

The analysis of code start with the outermost layer because the outermost statement starts first. In this case, the following is the outermost logic.

Listing 9: The outermost conditional statement

```
029   if ('session' in req) // just checking, but session should be a part of the
030                         // 'req'uest object
031   {
059   }
060   else
061   {
064   }
```

The purpose of this statement is to make sure there is a session before continuing to process. The condition (also known as a boolean expression) 'session' **in** req checkes to see if `session` is a mmeber of `req`.