Tagged: cookie, session, sessions

This topic contains 6 replies, has 5 voices, and was last updated by Yaapa 1 year, 1 month ago.

Viewing 7 posts - 1 through 7 (of 7 total)

- Author
  Posts
- November 21, 2013 at 7:58 pm #128

Yaapa
Keymaster

I am making a part of this chapter available in the form of a tutorial on **Express.js sessions** for those who haven't bought the book yet.

Sessions in Express.js

There are two broad ways of implementing sessions in Express – using cookies and using a session store at the backend. Both of them add a new object in the `request` object named `session`, which contains the session variables.

No matter which method you use, Express provides a consistent interface for working with the session data.

Cookie-based Session

Using the fact that cookies can store data in the users browser, a sessions API can be implemented using cookies. Express comes with a built-in middleware called `cookieSession`, which does just that.

Load the `cookieParser` middleware with a secret, followed by the `cookieSession` middleware, before the router middleware. The `cookieSession` middleware is dependent on the `cookieParser` middlware because it uses a cookie for storing the session data.

The `cookieParser` middleware should be initialized with a secret, because `cookieSession` needs to generate a signed **HttpOnly** cookie for storing the session data. If you don't specify a secret for `cookieParser`, you will need to specify the secret option of `cookieSsession`.

The following is the code for enabling sessions in Express using the `cookieSession` middleware.

```
app.use(express.cookieParser('S3CRE7'));
app.use(express.cookieSession());
app.use(app.router);
```

Although the `cookieSession` middleware can be initialized without any options, the middleware accepts an options object with the following possible options.

```
Option  Description
------------------
key     Name of the cookie. Defaults to connect.sess.
secret  Secret for signing the session. Required if cookieParser is not initialized with one.
cookie  Session cookie settings. Regular cookie defaults apply.
proxy   To trust the reverse proxy or not. Defaults to false.
```

Here is an example of initializing `cookieSession` with some options.

```
app.use(express.cookieSession({
  key: 'app.sess',
  secret: 'SUPERsekret'
}));
```

Once the session API is enabled, session variables can be accessed on the session object on the request object – `req.session`.

Cookie-based sessions work great for simple session data. However, it doesn't work well with large, complicated, and sensitive data because the session data is visible to the user, there is a limit in the size of cookies a browser can store, and multiple large size cookies can affect the performance of the website.

The limitations in cookie-based sessions can be overcome by sessions based on session stores.

Session Store-based Session

The `session` middleware provides a way for creating sessions using session stores. Like `cookieSession`, the session middleware is dependent on the cookieParser middleware for creating a signed HttpOnly cookie.

Initializing the `session` middlware is a lot like initializing `cookieSession` – we first load `cookieParser` with a secret, and load the `session` middleware, before the `router` middleware.

```
app.use(express.cookieParser('S3CRE7'));
app.use(express.session());
app.use(app.router);
```

The session middleware accepts an options object which can be used for defining the options of the middleware. The following are the supported options.

```
Option  Description
------------------
key     Name of the cookie. Defaults to connect.sid.
store   Instance of a session store. Defaults to MemoryStore. The session store may supports options o
secret  Secret for signing session cookie. Required if not passed to cookieParser().
cookie  Session cookie settings. Regular cookie defaults apply.
proxy   To trust the reverse proxy or not. Defaults to false.
```

Here is an example of initializing the session middleware with some options.

```
app.use(express.session({
  key: 'app.sess',
  store: new RedisStore,
  secret: 'SEKR37'
}));
```

The interface for accessing and working with the session variables remain the same – `req.session` – except now the session values reside on the backend.

Let's explore three popular session stores for Express.

MemoryStore

Express comes with a built-in session store called **MemoryStore**, which is the default when you don't specify one explicitly.

MemoryStore uses the application RAM for storing session data and works right out of the box, without the need for any database. Seeing how easily it is to set up, you might be tempted to make it the session store of your choice, but it is not recommended to do so because of the following reasons:

1. Memory consumption will keep growing with each new session.
2. In case the app is restarted for any reason; all session data will be lost.
3. Session data cannot be shared by other instances of the app in a cluster.

In fact, if you try to use MemoryStore in a production environment you will get the following warning.

```
Warning: connection.session() MemoryStore is not
designed for a production environment, as it will leak
memory, and will not scale past a single process.
```

Although MemoryStore is not a scalable solution and is not recommended on production, it makes for a good choice for getting to know the session API quickly and developing the application without the need for any database.

RedisStore

RedisStore is a popular third-party module which uses Redis for storing session data.

Install RedisStore in the application directory.

```
$ npm install connect-redis
```

Load the RedisStore module in the app and pass the instance of the Express object to it.

```
var express = require('express');
var RedisStore = require('connect-redis')(express);
```

With that, we are ready to use RedisStore as our session store – load the `session` middleware with its store option set to an instance of RedisStore.

```
app.use(express.session({ store: new RedisStore }));
```

RedisStore accepts a configuration object which can be used for specifying various aspects of the session store.

```
  port:6380,
  prefix:'sess'
}), secret: 'SEKR37' }));
```

Once you have set up RedisStore successfully, you can continue to work on the req.session object to create, read, update, and delete session variables as usual; only this time the data is stored on a Redis server – accessible by multiple instances of your app and persisting if your app is restarted.

You can get more information about RedisStore at https://github.com/visionmedia/connect-redis.

MongoStore

Another popular session store uses MongoDB for storing the data and is called MongoStore. The interface is very similar to RedisStore.

Install MongoStore in the application directory.

```
$ npm install connect-mongo
```

Load the MongoStore module in the app and set an instance of it as the session store for the `session` middleware.

```
var express = require('express');
var MongoStore = require('connect-mongo')(express);
...
app.use(express.session({
  store: new MongoStore({
    db: 'myapp',
    host: '127.0.0.1',
    port: 3355
  })
}));
```

With that, session data will now be stored in MongoDB, but the session interface remains the same – the req.session object.

You can read more about MongoStore at https://github.com/kcbanner/connect-mongo.

Session Variables

Session variables are those variables, which you associate with a user session. These variables are independently set for each user and can be accessed on the session property of the request object – `req.session`.

While it might look like we are dealing a JavaScript object, it is not completely true; the session variables actually reside in the data store of the session and the JavaScript object only works as a proxy for those values. Operations on session variables are is basically working with JavaScript objects. The states of these objects are then updated on the session store.

Setting Session Variables

To set a session variable, attach it to the `req.session` object.

```
req.session.name = 'Napoleon';
```

In case the session variable contains illegal characters, use the substring notation to create it.

```
req.session['primary skill'] = 'Dancing';
```

Reading Session Variables

Session variables can be read from the `res.session` object using either the dot notation or the substring notation.

```
var name = req.session.name;
var primary_skill = req.session['primary skill'];
```

If you try to read an undefined property, you will get `undefined` as expected.

Updating Session Variables

Updating a session variable is just about updating the property in the `req.session` object or overwriting the existing property with a new value.

```
req.session.skills.push('Baking');
req.session.name = 'Pedro';
```

To delete a session variable just delete the property from the `req.session` object.

```
delete req.session.name
delete req.session['primary skill'];
```

Deleting a Session

We learnt that there are two broad ways of creating sessions – using `cookieSession` and `sessionStore`. Each session type has its own way of deleting the session.

Deleting a Cookie-based Session

To delete a cookie-based session just delete the session object from the request object, or set it to null.

```
delete req.session;
```

or

```
req.session = null;
```

Once the session object is deleted, the session cookie is also deleted from the browser, effectively destroying the session.

Deleting a Session Store-based Session

Session store-based sessions do not interpret a missing session object on the request object as the end of a session. If we delete the session object from the request object, it will be recreated from the session store, because the session store decides the state of the session, not JavaScript variables. This also the reason why these sessions are intact even after the app restarts.

Session store-based sessions has a method called `destroy()`, which is used for destroying sessions from the session store – the proper way of tearing down a session store-based session.

```
req.session.destroy();
```

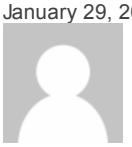
The `destroy()` method accepts an optional callback function to be executed after the session is cleared from the store.

```
req.session.destroy(function() {
  res.send('Session deleted');
});
```

This brings us to the end of the tutorial. Any queries or questions, ping me in the comments.

If you liked this tutorial, you will love my book "Express Web Application Development". Go buy it, it is your guide to mastering Express.

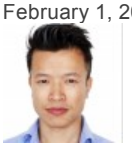
January 29, 2014 at 6:10 pm #137



modeef
Participant

Thanks, an interesting introduction. I have a quick question about this. Where is the best place to initialise a user's session? I.e. a user visits, they get a new session, I want to give them some default values… Also, where can I interact with the session at its start? Is there a similar way to detect when a session ends? Both of these events would be useful for logging, etc. Many thanks!

February 1, 2014 at 2:18 pm #139



Yaapa
Keymaster

You can initialize the session in the login route. Check if the user submitted the proper creds, and initialize the session accordingly.

You can check if a session has ended by looking at session variables, which are expected to be there. If they are not present, there is no session.

February 13, 2014 at 7:28 pm #151

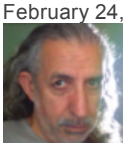
dstibrany
Participant

I'm a little baffled at how the following works:

```
req.session.name = 'Napoleon';
```

This is written in a synchronous manner, but wouldn't we have to do an async
call to store in mongo/redis/etc

February 24, 2014 at 3:58 pm #152



augusto.rene@lunanegra.net
Participant

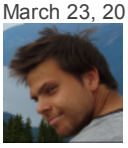
Thanks for the book is really grate.
I wonder about the TTL. I can't find a good and clear documentation for connect-mongo and ofte is confusing."expires", "originalMaxAge" and "cookie.expires".
It would be nice if you take a couple of examples how to configure the duration. I can see that MongoDB store the session like:
{
"_id" : "jvi555cr9lQGe3+kR0Q0c/wP",
"session" : "{\"cookie\":{\"originalMaxAge\":65000,\"expires\":\"2014-02-24T15:50:44.250Z\",\"httpOnly\":true,\"path\":\"/\"},\"user\":{\"firstName\":\"Testman\",\"lastName\":\"testsson\",\"email\":\"testman@mysite.se\",\"dbs\":[{\"dbName\":\"PeersDB\"}]}}",
"expires" : ISODate("2014-02-24T15:50:44.25Z")
}
After that time the document is deleted from the db.

In this case I wrote after the log-on
req.session.cookie.expires = 65000;

But I still wonder. Must I regenerate the session (I suppose if I don't the session will end after the time). Please tell me how wrong I have.
Thanks

Augusto

March 23, 2014 at 7:26 pm #153



shawninder
Participant

You give examples for using Redis and Mongo as stores for sessions. Would it be possible to use MySQL?

April 6, 2014 at 5:33 am #155

Yaapa
Keymaster

It would be definitely possible, but I am not aware of any implementation yet. The reason Redis and MongoDB are chosen is because they are faster and simpler to set up than MySQL, for data set like user sessions.

Viewing 7 posts - 1 through 7 (of 7 total)

You must be logged in to reply to this topic.