<https://stormpath.com/blog/everything-you-ever-wanted-to-know-about-node-dot-js-sessions>

# All You Ever Wanted to Know About Sessions In Node.js

Want to [build a Node.js web app](https://stormpath.com/blog/build-nodejs-express-stormpath-app) that wows and delights with secure, persistent login? Sounds like a job for sessions! In this post, you’ll learn what sessions are and how to implement them in your next Node.js web app, step-by-step. The samples use [Express.js](http://docs.stormpath.com/nodejs/express/latest/), but don’t worry if that’s not your cup of tea – the same principles hold true for most web frameworks.

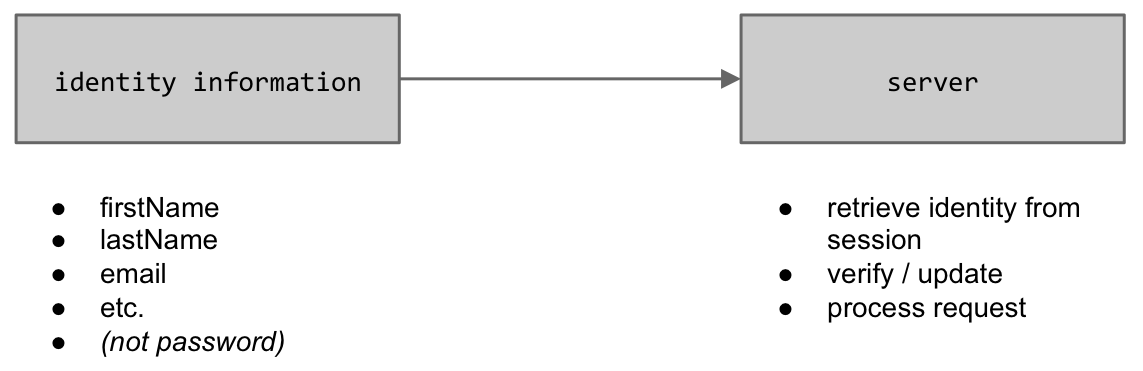
Content for this post came from our [Node.js Authentication screencast](https://www.youtube.com/watch?v=yvviEA1pOXw&list=UUJI9gByFSXE1ABDRcLQjWgQ). As it turns out, a lot of people want to know more about sessions, and in particular, how they relate to authentication! Check out the video for a broader walkthrough on implementing sessions and much more in a working demo app.

Otherwise, let’s get started!

### What’s In A Session Anyway?

When a user first logs in or registers for your site, you know who they are because they just submitted their information to your server. You can use that information to create a new record in your database, retrieve an existing one, or better yet – [use Stormpath](https://stormpath.com/). Simple!

But how do you keep them [authenticated](https://stormpath.com/product/authentication/) when they do something crazy like reload the page? Magic, that’s how! Also known as sessions.



That being said, a ‘session’ is a squishy, abstract term for keeping users logged in. We care more about the actual mechanism for persisting authentication; namely, cookies. The most delicious part of user management.

Cookies allow you to store a user’s information inside a file on their browser. The browser then sends that info back on every request, allowing your application to identify the user and customize their experience. Which is objectively way better than asking for a username and password on every request.

They’re pretty simple too. Let’s say a user has a cookie on their browser, and this is the HTTP request sent when they load a new page:



The request is broken into two parts:

1. The headers section (the top box), which contains metadata about the request
2. The body, which holds the actual payload from the user

Headers get our attention today because that’s where the cookies are. Notice the “Cookie” header in our request – that’s where the user’s info is stored. They’re simply strings separated with semicolons, nothing fancy (yet).

### Set-Cookie

Of course, it doesn’t do much good to read cookies unless you can set your own. Like requests, HTTP responses have headers and a body. Tell the browser to store your cookie with the aptly named “Set-Cookie” header in your response.



For instance, the “Set-Cookie” header might set the cookie value to a string like “session=[r@rdegges.com](mailto:r@rdegges.com)”. Just like that, the user’s browser will store and pass along a cookie the next time they visit your site. Your server can grab the email and pull their profile information from your database if needed.

### Client-Sessions

Now for the fun part: adding the session code to your app. Preferably with a well-vetted library like [Mozilla’s client-sessions](https://github.com/mozilla/node-client-sessions). It’s not the default option in Express.js, but we like the security options, it’s easy to use, and the documentation is excellent.

Install client-sessions with npm install client-sessions and import the library.



|  |  |
| --- | --- |
| 1  2 | var session = require('client-sessions'); |

Next, add session handler middleware to your app.js file and set these [basic configuration options](https://github.com/mozilla/node-client-sessions#usage).



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | app.use(session({    cookieName: 'session',    secret: 'random\_string\_goes\_here',    duration: 30 \* 60 \* 1000,    activeDuration: 5 \* 60 \* 1000,  })); |

1. The secret is a random, high-entropy string you create to encrypt the cookie. We need to take this step because the browser is an inherently untrusted environment; anyone with access can open it up and see what’s stored in there. Client-sessions will encrypt and decrypt all the cookie values, so you don’t have to worry about prying eyes.  
   This is a big part of why we recommend using a library to manage sessions. It’s never a good idea to roll your own crypto, and unencrypted cookies are a non-starter.
2. The duration defines how long the session will live in milliseconds. After that, the cookie is invalidated and will need to be set again. You probably experience this behavior daily on sites that deal with secure data. Your banking portal, for instance (hopefully).
3. Finally, activeDuration allows users to lengthen their session by interacting with the site. If the session is 28 minutes old and the user sends another request, activeDuration will extend the session’s life for however long you define. In this case, 5 minutes.  
   In short, activeDuration prevents the app from logging a user out while they’re still using the site.

### Store User Data in the Session

For the cookie to be useful, it needs to store information about the user. You can do this by slightly modifying your registration and login routes. For example, this login route checks the database for the user record and verifies the password but does not store anything in the cookie.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | app.post('/login', function(req, res) {    User.findOne({ email: req.body.email }, function(err, user) {      if (!user) {        res.render('login.jade', { error: 'Invalid email or password.' });      } else {        if (req.body.password === user.password) {          res.redirect('/dashboard');        } else {          res.render('login.jade', { error: 'Invalid email or password.' });        }      }    });  }); |

To change that, just add req.session.user = user; to the password check like below:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | app.post('/login', function(req, res) {    User.findOne({ email: req.body.email }, function(err, user) {      if (!user) {        res.render('login.jade', { error: 'Invalid email or password.' });      } else {        if (req.body.password === user.password) {          // sets a cookie with the user's info          req.session.user = user;          res.redirect('/dashboard');        } else {          res.render('login.jade', { error: 'Invalid email or password.' });        }      }    });  }); |

In addition to redirecting the user after a successful authentication, /login now saves their data on the cookie using the “Set-Cookie” header. Because client-sessions does the heavy lifting, those values will also be encrypted.

### Retrieve User Data from the Node Session

You can take advantage of the newly personalized cookie to make your templates more customized. For instance, if your app has a dashboard page like the sample, you could incorporate that in your view for a more customized look. If this is the original route:



|  |  |
| --- | --- |
| 1  2  3  4 | app.get('/dashboard', function(req, res) {    res.render('dashboard.jade');  }); |

Modify it to check for a session, lookup the user in your database and expose the user’s profile fields as variables for the template:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | app.get('/dashboard', function(req, res) {    if (req.session && req.session.user) { // Check if session exists      // lookup the user in the DB by pulling their email from the session      User.findOne({ email: req.session.user.email }, function (err, user) {        if (!user) {          // if the user isn't found in the DB, reset the session info and          // redirect the user to the login page          req.session.reset();          res.redirect('/login');        } else {          // expose the user to the template          res.locals.user = user;            // render the dashboard page          res.render('dashboard.jade');        }      });    } else {      res.redirect('/login');    }  }); |

### Session Middleware

That approach works fine for a few pages, but you probably don’t want to rewrite the session logic for every single route in a more substantial app. Instead, make it into a global middleware function.

Here’s what that would look like in Express.js:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | app.use(function(req, res, next) {    if (req.session && req.session.user) {      User.findOne({ email: req.session.user.email }, function(err, user) {        if (user) {          req.user = user;          delete req.user.password; // delete the password from the session          req.session.user = user;  //refresh the session value          res.locals.user = user;        }        // finishing processing the middleware and run the route        next();      });    } else {      next();    }  }); |

Most of the logic here should be familiar from the ‘/dashboard’ route before. Pay particular attention to the delete req.user.password; line though, as it’s new and important. The password might be encrypted in the session, but that’s still no reason to leave it in the cookie. Go ahead and delete it!

We’re almost there, but we still need a middleware function that will check if the user is logged in and redirect them if not.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function requireLogin (req, res, next) {    if (!req.user) {      res.redirect('/login');    } else {      next();    }  }; |

Coming full circle, we can update /dashboard to call the requireLogin function and get rid of the now unnecessary session code.



|  |  |
| --- | --- |
| 1  2  3  4 | app.get('/dashboard', requireLogin, function(req, res) {    res.render('dashboard.jade');  }); |

The global middleware we added checks for a session on every request and sets req.user to user if the user is logged in. Because we explicitly call requireLogin, we also ensure the dashboard page is only available to logged-in users.

### Secure Cookies on the Client

There are a few more steps to properly secure the session. The first is simply to make sure your app resets the session when a user logs out. Something like this would do the trick:



|  |  |
| --- | --- |
| 1  2  3  4  5 | app.get('/logout', function(req, res) {    req.session.reset();    res.redirect('/');  }); |

Next, make sure to use SSL so your application only communicates with the browser over an encrypted channel. With SSL in place, there are a few additional security options to set on client-sessions:

1. httpOnly prevents browser JavaScript from accessing cookies.
2. secure ensures cookies are only used over HTTPS
3. ephemeral deletes the cookie when the browser is closed. Ephemeral cookies are particularly important if you your app lends itself to use on public computers.

To recap, here’s the updated configuration:

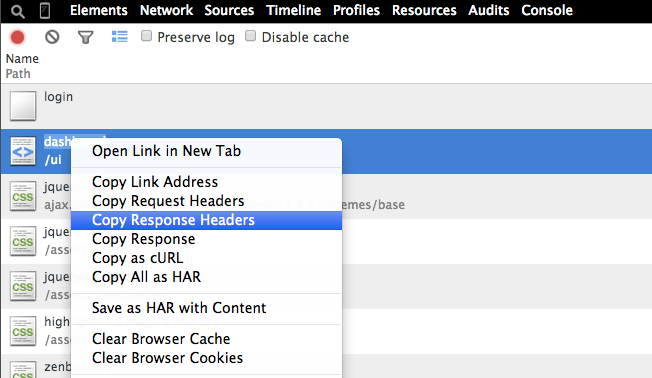


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | app.use(session({    cookieName: 'session',    secret: 'eg[isfd-8yF9-7w2315df{}+Ijsli;;to8',    duration: 30 \* 60 \* 1000,    activeDuration: 5 \* 60 \* 1000,    httpOnly: true,    secure: true,    ephemeral: true  })); |

### Cookies in the wild

I can’t think of a better way to wrap up than looking at real sessions in action. Here’s how:

1. Open up your browser
2. Go to the login page of a site that uses sessions, like [Stormpath!](https://api.stormpath.com/login)
3. Open the network tab on your developer tools and login
4. Right-click on the page you just logged into in the network tab and click ‘Copy Response Headers’
5. Paste into a text editor and take a look at the “Set-Cookie” header!



Assuming you chose a site with good security, you should see an encrypted string instead of your personal data.

And there you have it! Node.js sessions in a nutshell. Cool stuff, eh?