- ⏱️ 11 minutes
- ✅ Completed

Configuring Sessions in Express

In this reading, you'll learn about HTTP sessions. You'll also learn about how to configure a session store as well as how to secure the session cookie.

Let's begin by revisiting how HTTP is stateless. Since HTTP is stateless, query string parameters and HTTP cookies provide ways to persist values across requests. However, storing data in query string parameters can become cumbersome and possibly look strange to end users. This is where session storage comes into play.

By the end of this article, you should be able to:

- Use the `express-session` middleware to configure an Express application to support sessions;
- Use the `req.session` object to store and retrieve a value;
- Configure the `express-session` middleware to use a production-ready session storage provider; and,
- Use the available security options (i.e. secure, httpOnly, domain, path, and expires) to secure a session's HTTP cookie.

Overview of sessions

Let's begin by revisiting how HTTP is a stateless protocol. This means that each HTTP request is independent from other requests that were executed before or after. Once the server has processed an incoming request and returned a response, it forgets about the client.

What are sessions?

Even though HTTP cookies allow the persistence of values across requests, cookies aren't an efficient way to store anything other than small amounts of data. Transmitting HTTP cookies between the server and client (and back again) can undesirably increase the amount of traffic on the server.

Most browsers don't reliably support more than 50 cookies per domain and the total size of all of the data in the cookies has to be less than 4093 characters of text. That may sound like a lot, but when you are storing data for a person's session in there, you will run out of room very quickly.

Sessions build upon the idea of an HTTP cookie. Instead of storing data in the cookie itself, a unique identifier known as the **session ID** is stored. This **session ID** is linked to an object stored on the server.

Why are sessions useful?

Sessions give you a way to identify a series of requests as being connected to the same client. Once you know that a request is connected to a known client session, you can associate the state (data) of that session without having to send that data to the client and rely upon them to send that data back to the server unaltered.

What are the drawbacks?

Although sessions are useful, there are still drawbacks to using sessions. Using sessions increases the overhead required to serve clients. Server affinity, the ability of a router to send a request to the same server over and over for a specific client, can be an issue depending on the session store that you're using.

Configuring Express to use sessions

Clone the starter project files at: starter files

Install the project's dependencies and run it.

```
npm install
npm start
```

The application is a simple website that contains three pages: "Home", "About", and "Contact".

Currently, when you browse from page to page, the server doesn't recognize or associate the page requests as being part of the same session.

Open up your web developer tools to view cookies under the "Application" tab. Click on "Storage > Cookies > `http://localhost:8080`" to view the cookies for your `localhost` domain. Currently, you shouldn't see any cookies listed.

Let's configure the application to use sessions!

Installing and configuring `express-session`

Install the `express-session` npm package.

```
npm install express-session
```

Since the `express-session` npm package handles your session cookies, you no longer need to install the `cookie-parser` package separately. If `cookie-parser` is configured separately, `express-session` and `cookie-parser` would both need to use the same `secret` value. You'll learn more about this `secret` value below.

Add the `express-session` middleware to the `app` module.

```
// ./app.js

const express = require('express');
const session = require('express-session');

const app = express();

app.set('view engine', 'pug');
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));

// Code removed for brevity.
```

Note: *Ideally the session `secret` option value is set from an environment variable. Using a literal value here is being done to keep this example as simple as possible.*

Configuration options

Let's take a closer look at the session middleware you configured in `app.js`:

```
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
```

```
  saveUninitialized: false,
}));
```

Notice the following keys that configure your session creation:

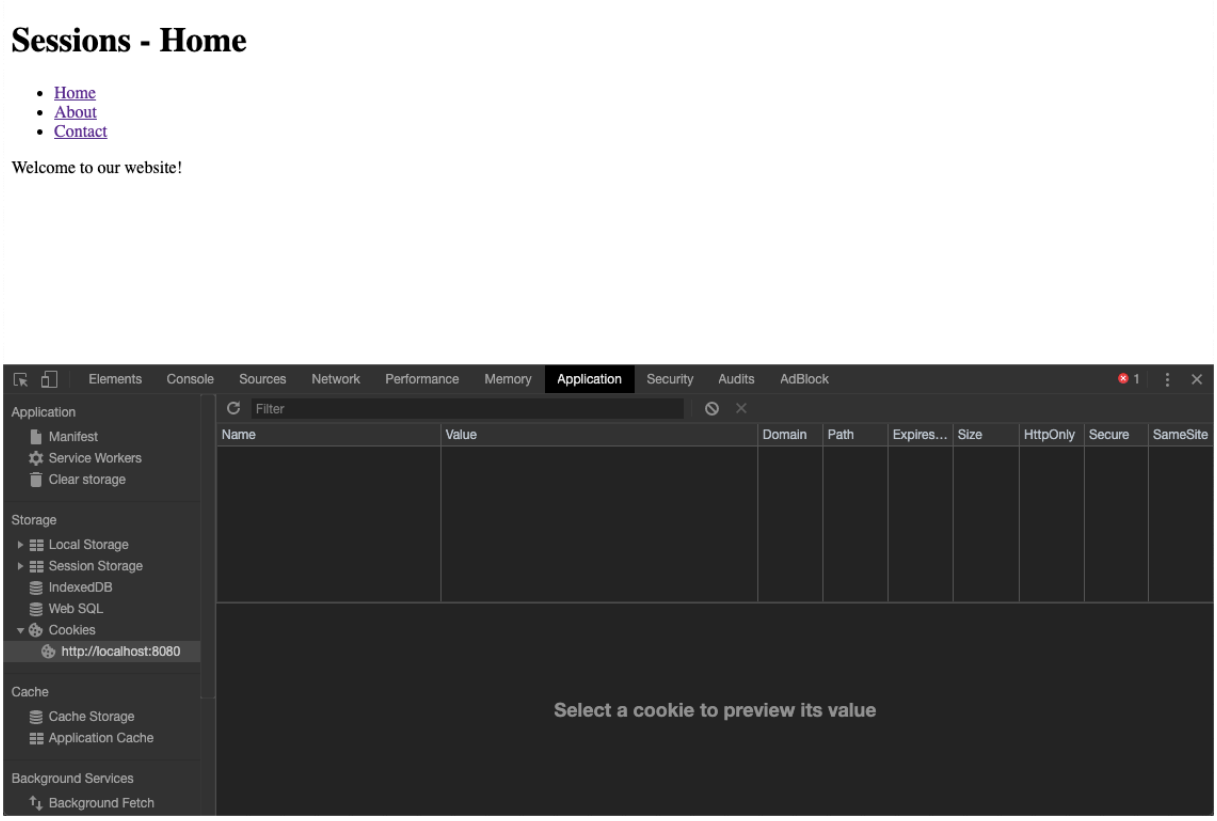
secret: This is the secret used to sign the **session ID** cookie. The `secret` value above was generated using the uuid npm package. The `uuid` package allows you to generate universally unique identifiers (UUIDs) from random cryptographically-strong values, a timestamp, or a user-supplied string.
resave: This option forces the session to be saved into the session *store*, even if the session was never modified during the request. You would typically want to set this option as `false` to prevent overwriting sessions during race conditions, which are undesired parallel requests. However if your session store sets an expiration date on stored sessions, then you likely need to set "resave" as `true`.
saveUninitialized: This forces an *uninitialized* session to be saved to the store. An uninitialized session is when a session is new but not modified. It's useful to set this option as `false` when creating login sessions, reducing use of server storage, or complying with permission laws to set cookies. Setting the option as `false` also prevents race conditions when multiple requests are made without a session.
Not setting the `resave` and `saveUninitialized` options results in the following warning in the console:

```
express-session deprecated undefined resave option; provide resave
option app.js:8:9
express-session deprecated undefined saveUninitialized option;
provide saveUninitialized option app.js:8:9
```

Another configuration option that you can set, but which is not listed in the example above, is the **name** of the cookie. By default, the **express-session** middleware uses the name `connect.sid`. Imagine if you have an application running on `localhost:8080` and a different application on `localhost:3000`. Since cookies are scoped to the general `localhost` domain, this means that cookies set for `localhost:8080` would appear in your `localhost:3000` cookies. This is why it's important to set a specific `name` property and separate each application's session cookies from each other.

Testing

In order to test whether your session cookie is properly created in the next step, begin by opening up your developer tools to view the cookies of the `localhost` domain. Remember to access the cookies by clicking Application > Storage > Cookies > `http://localhost:8080`.



As you can see, the session cookie isn't being created. This is because setting `saveUninitialized` to `false` prevents the session from being created until the session has been used. Let's use the `session` in your application! Keep your developer tools open to be able to view your cookies as they appear.

Setting and accessing session values

Let's use `session` to track each page that the user visits!

Create a middleware function that adds each request URL to an array of page visits stored in session.

```
// ./app.js

// Code removed for brevity.

app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));
```

```
app.use((req, res, next) => {
  // Attempt to get the `history` array from session.
  // If it's not initialized, then create an array
  // and assigned it back to session.
  let { history } = req.session;
  if (!history) {
    history = [];
    req.session.history = history;
  }

  // Construct the full URL for the current request.
  // Note: Using `req.get('host')` to get the hostname also
  // gives you the port number.
  const url =
`${req.protocol}://${req.get('host')}${req.originalUrl}`;

  // Add the URL to the beginning of the array.
  history.unshift(url);

  // Note: We don't need to update the `session.history`
property
  // with the updated array because arrays are passed by
reference.
  // Because arrays are passed by reference, when we get a
  // reference to the array in the above code
  // `let { history } = req.session;` and modify the array by
  // calling `history.unshift(url);` we're modifying the
original
  // array that's stored in session!

  next();
});

// Code removed for brevity.
```

After refreshing your page, you should now see a cookie in your developer tools.

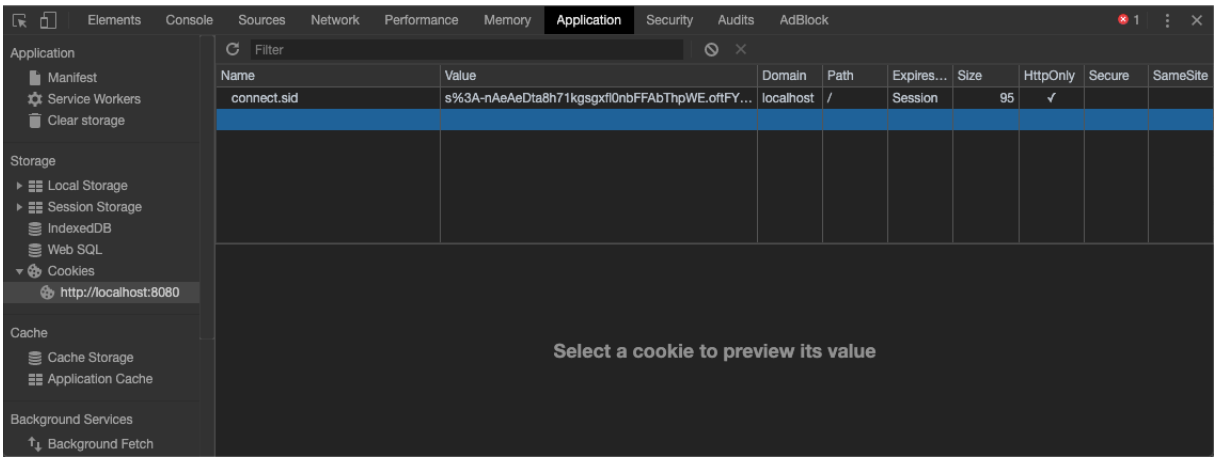
Sessions - Home

- Home
- About
- Contact

Welcome to our website!

History

- http://localhost:8080/



Notice how the code accessed the `session` attribute of the `req` object. In order to store or access session data, you use the `req.session` property.

In the example above, you attempted to get the `history` array from session. If the array was not already initialized, you created an array and assigned it to the "history" property of the `req.session` object.

Now you'll want to update each of the route handlers to pass the `req.session.history` array to the views.

```
app.get('/', (req, res) => {
  res.render('index', {
    title: 'Home',
    history: req.session.history,
  });
});

app.get('/about', (req, res) => {
  res.render('about', {
    title: 'About',
    history: req.session.history,
  });
});

app.get('/contact', (req, res) => {
  res.render('contact', {
    title: 'Contact',
    history: req.session.history,
  });
```

```
});
```

Then update the `layout.pug` view to render the history array.

```
doctype html
html(lang="en")
  head
    title Sessions - #{title}
  body
    h1 Sessions - #{title}
    ul
      li: a(href='/') Home
      li: a(href='/about') About
      li: a(href='/contact') Contact
    block content
    footer
      h2 History
      ul
        for item in history
          li: a(href=item)= item
```

Now when you browse from page to page, your history is tracked by the server using `session`. The middleware function below (that you have already added to your `app.js`) takes care of generating the `history` array of visited links. In each of your GET routes, your `req.session.history` array is passed into the view as a `history` attribute. Your views then use the `history` attribute to render a list of visited links in the footer of each page.

Note that browsing history is specific to each user session. If you open a private, Incognito tab, you'll see that the private client gets its own array of visited pages.

Session store configuration

The default in-memory store that you have been using (`MemoryStore`) is meant only for local development. There are many available session store options. For example, `connect-pg-simple` and `connect-session-sequelize` are two PostgreSQL compatible session store options.
Let's install and configure the `connect-pg-simple` session store!

Install the npm package.

```
npm install connect-pg-simple
```

Next, create a database and a database user. You can use the `psql` terminal commands below.

```
create database session_example;
create user session_example_app with encrypted password '«a strong
password for the session_example_app user»';
```

```
grant all privileges on database session_example to
session_example_app;
```

Then create the `session` table in the `session_example` database. You can run the following command from the root of your project to create the `session` table.

```
psql -U session_example_app session_example < node_modules/connect-
pg-simple/table.sql
```

Or you can run the following SQL statements in `psql` against the `session_example` database:

```
CREATE TABLE "session" (
  "sid" varchar NOT NULL COLLATE "default",
  "sess" json NOT NULL,
  "expire" timestamp(6) NOT NULL
)
WITH (OIDS=FALSE);

ALTER TABLE "session" ADD CONSTRAINT "session_pkey" PRIMARY KEY
("sid") NOT DEFERRABLE INITIALLY IMMEDIATE;

CREATE INDEX "IDX_session_expire" ON "session" ("expire");
```

Update the npm `start` script by setting `PGUSER`, `PGDATABASE`, and `PGPASSWORD` environment variables. These variables are used by the `pg` npm package, which is used by the `connect-pg-simple` package to communicate to the PostgreSQL database. Make sure to set the `PGPASSWORD` to the password you've given to the `session_example_app` user.

```
"scripts": {
  "start": "PGUSER=session_example_app
PGDATABASE=session_example PGPASSWORD=password nodemon app.js"
}
```

An alternative way to update the npm `start` script is by defining the `DATABASE_URL` environment variable. The `DATABASE_URL` is used by `connect-pg-simple` to determine how to connect to the PostgreSQL database containing the `session` table.

```
"scripts": {
  "start":
"DATABASE_URL=postgresql://session_example_app:password@localh
ost:5432/session_example nodemon app.js"
}
```

Now update the `app` module as follows.

```
// ./app.js

const express = require('express');
const session = require('express-session');
const store = require('connect-pg-simple');

const app = express();

app.set('view engine', 'pug');
app.use(session({
  store: new (store(session))(),
```

```
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));

// Code removed for brevity.
```

Test your app again, and it should behave as it did before, but now the session data is stored in the database!

In `psql`, connect to the `session_example` database and run the following query:

```
select * from session;
```

You should see a `sid` session ID, a session cookie, and a `history` array of visited urls stored in your PostgreSQL database, like the example below.

```
sid: 2kseXpBRl9h1ZHoBndzEyu_cem9l0GN4
sess:
{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"pat
h":"/"},"history":["http://localhost:8080/","http://localhost:8080/"
,"http://localhost:8080/","http://localhost:8080/","http://localhost
:8080/contact","http://localhost:8080/about","http://localhost:8080/
","http://localhost:8080/about","http://localhost:8080/contact","htt
p://localhost:8080/about"]}
expire: 2020-04-11 12:19:13
```

Securing the session HTTP cookie

As you see in your database query, you have a `cookie` object with the following keys of `originalMaxAge`, `expires`, `httpOnly`, and `path`.

A session cookie has a default configuration of:

```
{
  httpOnly: true,
  maxAge: null,
  path: '/',
  secure: false
}
```

A secure session cookie should be configured like so:

```
{
  httpOnly: true,
  maxAge: «time in milliseconds»,
  path: '/',
```

```
  secure: true
}
```

It's important to configure cookies as secure HTTP cookies because it's possible for sessions to be hijacked. Insecure cookies allow hackers to make requests to the web application through another user's session.

Setting the cookie's `httpOnly` property to `true` prevents JavaScript on the page from accessing the cookie. This helps prevent cross-site scripting (XSS) attacks by making the `httpOnly` cookies inaccessible through the `document.cookie` API.
Setting the cookie's `secure` property requires that HTTPS is used. HTTPS uses a digital certificate for authentication known as an SSL certificate to make a web application more secure. Using `secure` cookies and HTTPS prevents anyone from being able to *sniff* or hijack the cookie as requests and responses are passed between the client and the server.
In production, the `secure` property should be set to `true`. If your Express application is behind a proxy you would need to configure it to trust the proxy with `app.set('trust proxy', 1)`.
Appropriately setting a `maxAge` on the cookie keeps the cookie from living longer than it needs to. Have you ever logged into a bank website and noticed how quickly the website will log you out if you're inactive? This is to prevent someone from stealing or accessing your information when sitting down at your computer while you're away. Expiring sessions as quickly as possible helps to protect your users from accidentally allowing someone to access the web application using their identity.
`path` - Defaults to `/`

Additional session configuration cookie options

You can also configure session cookies with the following options that are not set by default:

`domain`
Since no domain is set by default, this causes browsers to interpret the current domain as the cookie's domain. This is connected to why setting cookies in a `localhost` domain (i.e. `localhost:8080`) would set cookies in a different `localhost` domain (i.e. `localhost:3000`).
`expires`
Instead of setting this property directly, you can use `maxAge` instead. Note that `maxAge` takes in an integer number representing milliseconds to calculate when the cookie should expire.

What you have learned

You now know what HTTP sessions are as well as how to configure and access the session store in your Express application. You also know how to configure cookie options to secure the session cookie.

Now that you have learned about sessions, you can implement sessions in your own applications!