Clase 6

Webservices con express

Temario

1. Qué es Express
2. Instalación
3. Creación de servidores
4. Middlewares
5. Enrutamiento
6. Knex/Sequelize
7. **Sesiones, Cookies y JWT**

Sesiones, Cookies y JWT

Cada vez que visitamos una página, un servidor es el encargado de responder nuestra petición.

Cada solicitud que realizamos y la respuesta que recibimos, son totalmente independientes por naturaleza.

Sin embargo, **el uso de Sesiones, Cookies y JWT hace posible la existencia de un estado** entre las distintas peticiones que vamos realizando.

Es decir, **las peticiones HTTP carecen de estado** (stateless), pero **si se apoyan en Sesiones, Cookies y JWT pueden tener uno** (stateful) y con ello modificar lo que los visitantes ven, en función a sus cookies guardadas o token generado.

En esta sección veremos cómo trabajar con ellos.

- Qué son y cómo trabajo con sesiones
- Qué son y cómo trabajo con cookies
- Qué es y cómo trabajo con JWT

¿Qué son las cookies?

Las cookies son pequeños fragmentos de texto que los sitios web que visitas envían al navegador. Permiten que los sitios web recuerden información sobre tu visita, lo que puede hacer que sea más fácil volver a visitar los sitios y hacer que estos te resulten más útiles.

Las cookies se usan con diferentes propósitos:

- **Funcionalidad**: permiten acceder a funciones esenciales de un servicio. Por ejemplo, preferencias del usuario como el idioma, información relacionada con la sesión como un carrito de la compra, etc.
- **Seguridad**: ayudan a autenticar a los usuarios, prevenir el fraude y protegerte cuando interactúas con un servicio.
- **Analítica**: las cookies y otras tecnologías que se usan con fines analíticos ayudan a recoger datos que permiten a los servicios entender cómo interactúas con un servicio en particular. Esta información se usa para mejorar el contenido de los servicios y sus funciones, y así ofrecerte una mejor experiencia.
- **Publicidad**: ayudan a los servicios a mostrar/ocultar anuncios, personalizar anuncios, limitar el número de veces que se muestra un anuncio, etc.
- **Personalización**: mejoran la experiencia de usuario proporcionando contenido o funciones personalizadas según tu configuración. Por ejemplo, ayudan a los servicios a mostrarte funciones o resultados específicos de tu ubicación.

Las **cookies son datos que se almacenan en tu navegador**, y son enviados al servidor en cada petición que haces del cliente al servidor mismo.

Las cookies tienen varias **limitaciones**, por un lado **la cantidad de información que puedes almacenar** en cookies está limitada por el navegador, por otro lado, **el usuario podría alterar las cookies** como él prefiera, alterando así el funcionamiento de tu aplicación. Por último, **si algo sale mal** con las cookies, **no está en tu mano arreglarlo**, tendrías que pedirle a tus usuarios que ellos mismos arreglen el problema, ya que los datos están en su dispositivo.

¿Qué son las sesiones?

Las cookies y sesiones son **mecanismos a través de los cuales podemos identificar una petición**, con estas estrategias podemos almacenar información de nuestros usuarios que pueda ayudarnos como los productos que ha agregado a un carrito de compra, sus preferencias, si inició sesión o no, y mucho más.

Las sesiones **guardan la información en el servidor** y no en el cliente, y lo que **se envía entre cada petición es un identificador de sesión** para cada usuario, a través del cuál puedes obtener los datos que guardaste en el servidor.

La ventaja de que las sesiones se almacenen en tu servidor es que **puedes guardar mucha más información que en las cookies**, además puedes usar diferentes mecanismos de almacenamiento tales como archivos, RAM, base de datos, etc).

Además, las sesiones **no pueden modificarse desde el cliente**, ya que aunque el identificador de una sesión es guardado en una cookie, modificarlo significa que la información del usuario y el usuario ya no están conectados, y esto por sí mismo no presenta un riesgo de seguridad. Por otro lado, almacenar información sensible como el ID del usuario en una cookie puede prestarse a que se modifique y entonces sí exponga a tu servidor a una brecha de seguridad.

Por otro lado, es común que **las aplicaciones web encripten las cookies** para que estas no puedan ser leídas o modificadas en texto plano, para eso también hay distintas implementaciones con algoritmos de encriptación.

En general, la regla es, **no almacenes información sensible o de importancia para tu aplicación en cookies, usa sesiones.** Para todo lo demás, usa cookies.

¿Qué ocurre con los clientes que no soportan cookies y quieren consumir una API?

Tradicionalmente, el **principal tipo de cliente** (para visitar aplicaciones web) ha sido **el navegador web**, que tiene un soporte completo para cookies.

Pero las APIs hoy en día son usadas también por **clientes HTTP mucho más simples** (consola, postman, apps, servidores, etc), que **no soportan cookies de forma nativa.**

La autenticación basada en cookies es conveniente para los navegadores, pero más allá de ellos, para otros tipos de clientes tiene más sentido un enfoque basado en tokens (ya que estos pueden ser transportados a través de parámetros, encabezados o como parte del cuerpo de las peticiones HTTP).

Es decir, si una API admite tokens entonces aumentará el rango de clientes que puede atender, por lo que resulta más conveniente si la API se debe usar más allá de los navegadores web.

Hoy en día es posible el uso de cookies en las aplicaciones móviles nativas, pero en resumen: **los JWT tienen una ventaja sobre las cookies sólo por el hecho de que su uso es más común.** Siguiendo este enfoque, podemos tener más recursos de aprendizaje, SDKs, información sobre las vulnerabilidades más conocidas, etcétera.

¿Qué es JWT?

JWT (JSON Web Token) es un estándar que está dentro del documento RFC 7519.

En el mismo se define un mecanismo para poder propagar entre dos partes, y de forma segura, la identidad de un determinado usuario, además con una serie de claims o privilegios.

Estos privilegios están codificados en objetos de tipo JSON, que se incrustan dentro de del payload o cuerpo de un mensaje que va firmado digitalmente.

En la práctica, un token JWT es una cadena de texto que tiene tres partes codificadas en Base64, cada una de ellas separadas por un punto, por ejemplo:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Podéis decodificar tokens JWT en: https://jwt.io. Podemos ver el contenido del token sin necesidad de saber la clave con la cual se ha generado, aunque no podremos validarlo sin la misma.

Partes de un token JWT

1. **Header**: encabezado dónde se indica, al menos, el algoritmo y el tipo de token, que en el caso del ejemplo anterior era el algoritmo HS256 y un token JWT.

2. **Payload**: donde aparecen los datos de usuario y privilegios, así como toda la información que queramos añadir, todos los datos que creamos convenientes.

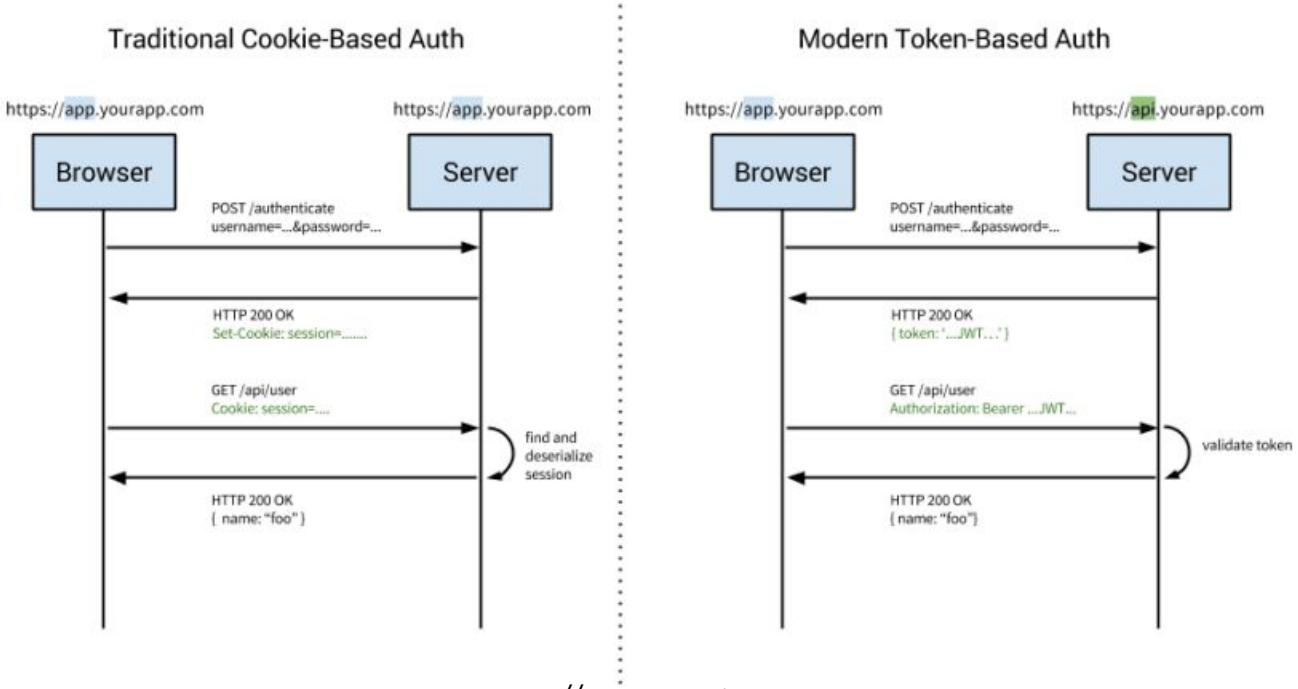
3. **Signature**: una firma que nos permite verificar si el token es válido, y aquí es donde radica el quid de la cuestión, ya que si estamos tratando de hacer una comunicación segura entre partes y hemos visto que podemos coger cualquier token y ver su contenido con una herramienta sencilla, ¿dónde reside entonces la potencia de todo esto?

Firma de un Token JWT

La firma se construye de tal forma que vamos a poder verificar que el remitente es quien dice ser, y que el mensaje no se ha modificado por el camino.

De esta forma, si alguien modifica el token por el camino, por ejemplo, inyectando alguna credencial o algún dato malicioso, entonces podríamos verificar que la comprobación de la firma no es correcta, por lo que no podemos confiar en el token recibido y deberíamos denegar la solicitud de recursos que nos haya realizado, ya sea para obtener datos o modificarlos.

Autenticación basada en Cookies y basada en Tokens



Fuente: https://programacionymas.com

Trabajando con Cookies 1/2

1. Instalamos el middleware de Cookies

```
npm install cookie-parser --save
```

2. Cargamos el módulo cookie-parser en app.js

```
const cookieParser = require("cookie-parser");
```

3. Cargamos la función middleware de cookies

```
app.use(cookieParser("ClaveSuperSecreta"));
```

4. Creamos una ruta para setear 2 cookies **GET /cookies/set**

```
// app.js
. . .
// Ruta para asignar las cookies
app.get("/cookies/set", (req, res) => {
 const date = new Date();
 date.setHours(date.getHours() + 5);
 res.cookie("customCookie", "Cookie value", {
   secure: false,
   httpOnly: true,
   expires: date,
   sameSite: "strict",
 });
 res.cookie("customSignedCookie", "Cookie value signed",
{
   signed: true,
   httpOnly: true,
   expires: date,
   sameSite: "strict",
 });
 res.send("Cookies set!");
});
. . .
```

Trabajando con Cookies 2/2

5. Creamos una ruta para obtener los valores de las cookies **GET /cookies**

6. Creamos una ruta **GET /protected** que muestra un texto si la cookie ha sido seteada y otro en caso contrario.

7. Creamos una ruta **GET /cookies/delete** para eliminar las cookies

```
// app.js
. . .
// Ruta para eliminar las cookies
app.get("/cookies/delete", (req, res) => {
 res.clearCookie("customCookie");
 res.clearCookie("customSignedCookie");
 res.send("Cookies removed!")
});
. . .
```

```
// app.js
. . .
// Ruta para obtener los valores de las cookies
app.get("/cookies", (req, res) => {
 // Cookies que no han sido firmadas
 console.log("Cookies: ", req.cookies);
 // Cookies que han sido firmadas
 console.log("Signed Cookies: ", req.signedCookies);
 res.json({
   customCookie: req.cookies.customCookie,
   customSignedCookie: req.signedCookies.customSignedCookie,
 });
});
// Ruta protegida, necesita que la variable haya sido
configurada
app.get("/protected", (req, res) => {
 if (req.cookies.customCookie) {
   res.send("Cookie has been set!");
 } else {
   res.send("The Cookie doesn't exist!")
 }
});
. . .
```

Trabajando con Sessions 1/2

1. Instalamos el middleware de Sessions

```
npm install express-session --save
```

2. Cargamos el módulo express-session en app.js

```
const session = require("express-session");
```

3. Cargamos la función middleware de session

```
// app.js
. . .
app.use(session({
 secret: "ClaveUltraSecreta",
 resave: false,
 saveUninitialized: false
}));
. . .
```

4. Creamos una ruta para setear 1 variable de sesión **GET /sessions/set**

```
// app.js
. . .
// Ruta para asignar la session
app.get("/sessions/set", (req, res) => {
 req.session.isSessionSet = true;
 res.send("isSessionSet set!");
});
. . .
```

Trabajando con Sessions 2/2

5. Creamos una ruta para obtener los valores de la variable de sesión **GET /sessions**

6. Creamos una ruta **GET /protected-by-session** que muestra un texto si la variable de sesión ha sido seteada y otro en caso contrario.

7. Creamos una ruta **GET /sessions/delete** para eliminar la variable de sesión.

```
// app.js
. . .
// Ruta para eliminar la variable de la session
app.get("/sessions/delete" , (req, res) => {
 delete req.session.isSessionSet
 res.send("Session variable removed!" );
});
. . .
```

```
// app.js
. . .
// Ruta para obtener los valores de la session
app.get("/sessions", (req, res) => {
 console.log("Sessions: ", req.session);
 // Cookies que han sido firmadas
 res.json({
   isSessionSet: req.session.isSessionSet
 });
});
// Ruta protegida, necesita que la variable haya sido
configurada
app.get("/protected-by-session" , (req, res) => {
 if (req.session.isSessionSet) {
   res.send("isSessionSet has been set!" );
 } else {
   res.send("The session variable doesn't exist!" );
 }
});
. . .
```

Trabajando con JWT Token 1/2

1. Instalamos la librería de JWT

```
npm install jsonwebtoken --save
```

2. Cargamos el módulo jsonwebtoken en app.js

```
const jwt = require("jsonwebtoken");
```

3. Declaramos una clave para firmar y verificar el token y creamos una ruta para obtener un token.

```
// app.js
. . .
const JWT_SECRET = "ClaveMegaSecreta";
app.get("/jwt/set", (req, res) => {
 const token = jwt.sign({ data: "jwt value" },
JWT_SECRET, {
   expiresIn: "5m",
 });
 res.json({ token: token });
});
. . .
```

4. Creamos un middleware para obtener los datos del token jwt

```
// app.js
. . .
// Middleware para obtener el data de JWT
const isAuth = (req, res, next) => {
 if (!req.headers.authorization) {
   return res.status(401).json({
     message: "Authorization Header missing",
   });
 }
 let authorization = req.headers.authorization;
 let token = authorization.split(" ")[1];
 let jwtData;
 try {
   jwtData = jwt.verify(token, JWT_SECRET);
 } catch (error) {
   console.log(error);
   return res.status(401).json({
     message: "Invalid Token.",
   });
 }
 req.data = jwtData.data;
 next();
};
. . .
```

Trabajando con JWT Token 2/2

5. Creamos una ruta para obtener los datos del token si es válido **GET /jwt**

6. En la terminal guardamos el token en una variable:

```
export JWT_TOKEN=${TOKEN}
```

7. En la terminal llamamos a nuestro endpoint **GET /jwt** con el token obtenido en el punto 4

```
// app.js
. . .
app.get("/jwt", isAuth, (req, res) => {
 res.json({ data: req.data });
});
. . .
```

```
curl -XGET http://localhost:3000/jwt -H "Authorization: Bearer ${JWT_TOKEN}"
```

Ejercicios

1. Crear un endpoint **GET /login** que renderice la vista login.html que puedes encontrar en los materiales de esta clase.
2. Crear un endpoint **POST /login** que reciba dos campos en el body, un username y un password, si el username es “foo” y el password es “bar”, setear una variable de sesión que indique que el usuario está logueado, además tiene que hacer un redirect a la ruta /home.
3. Crear un endpoint **GET /home** que renderice una vista home.html con el mensaje “User logged” y un botón para hacer logout si el usuario ha hecho login, en caso contrario, tiene que hacer un redirect a /login
4. Crear un endpoint POST **/logout** que elimine la variable de sesión de login y haga un redirect a /login
5. Crear un endpoint **POST /api/token** que reciba dos campos en el body, un username y un password, si el username es “foo” y el password es “bar”, devuelva un json con un token JWT que contenga el username y con un tiempo de validez de 5 minutos.
6. Crear un endpoint **GET /api/protected** que espere un JWT token en la cabecera Authorization, lo valide y si es válido devuelva en un json el username que contenga.
7. Probar el endpoint anterior pasados 5 minutos

ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas