



# Coding Bootcamp

## Sprint 3



# Temario



# Temario

- Autenticación.
- Autorización.
- ¿Cómo guardar contraseñas?
- Cookies y Sesiones.
- Passport.js.



# Autenticación



# Autenticación (1/2)

Es el **proceso** que determina si **alguien** (o algo) **es efectivamente quien dice ser quien es**. En el ámbito web, esto se traduce a un sistema de login de usuarios.

Para implementar un sistema de autenticación en una aplicación se necesita:

- a. Rutas y vistas que muestran los formularios de **registro** y **login** de usuarios.
- b. Método encargado de procesar el registro de usuarios.
- c. Método encargado de validar un email y password.
- d. Método encargado de “**proteger**” **una ruta**, es decir, inhabilitarla para usuarios que no han hecho login. Por ejemplo, las rutas que empiecen con `/admin` sólo deberían estar disponibles para usuarios logueados.
- e. Además se debe: **encriptar (hashear) contraseñas**, gestionar sesiones, crear migraciones para la base de datos...

En fin, no es una tarea sencilla y además se debe repetir para cada proyecto que requiera autenticación. Por suerte hay herramientas que simplifican la tarea.



# Autenticación (2/2)

Recordemos que la **seguridad** es uno de los atributos de calidad que solemos buscar en todo sitio web o aplicación, pero según el tipo de aplicación podemos darle más o menos prioridad a la seguridad. Por ejemplo, un banco no requiere el mismo tipo de seguridad que un juego como “El Solitario”.

En caso de ser necesario, se pueden agregar mecanismos adicionales de seguridad como:

- Two-factor Authentication ([link](#)) usando, por ejemplo:
  - Token físico (digital).
  - Token físico (analógico). Ej: “Tarjeta de coordenadas” de Banco Santander.
  - SMS de verificación.
- Cambios periódicos de contraseña (algo muy debatible).
- Control de direcciones IP.

En cualquier caso, siempre usen **HTTPS**.



# Autorización



# Autorización

Es el proceso que determina **a qué recursos puede acceder** determinado usuario.

Este proceso ocurre luego de que el sistema haya podido autenticar al usuario.

Un típico ejemplo de autorización es definir distintos tipos de **roles** que pueden tener los usuarios de una aplicación como, por ejemplo: lector, editor, administrador, etc. Luego, según el rol del usuario, el sistema determina a qué datos puede acceder y cuáles puede modificar.






¿Cómo guardar contraseñas?



# ¿Cómo guardar contraseñas? (1/2)

-  Las contraseñas en una base de datos jamás se deben guardar como “texto plano”. ¡Sería un problema de seguridad enorme!
- Las contraseñas deben encriptar o, mejor aún, *hashear*.
- La encriptación tienen un método inverso llamado desenscriptación. Para esto existe una “llave” o “clave” de encriptación y desenscriptación. Si bien guardar una contraseña encriptada es mejor que guardarla como texto plano, el hecho de que se pueda desenscriptar es peligroso.
- Las *funciones de hash* no requieren de una clave y no tienen una función inversa. Una vez que una contraseña es *hasheada*, no se puede volver para atrás.
- Cuando un usuario quiera loguearse a una aplicación, deberá ingresar su contraseña (texto plano). El sistema la *hashear*á y la comparará con el *hash* guardado en la BD.



# ¿Cómo guardar contraseñas? (2/2)

Existen varias funciones de hash disponibles, algunas son:

- MD5.
- SHA (SHA-1, SHA-256, SHA-521).
- BCrypt.

MD5 es una función de hash muy rápida. Es decir, una PC común y corriente puede calcular millones de hashes por segundo. Por lo tanto, no es recomendable su uso para contraseñas. En cambio, **BCrypt** es mucho más complejo y una PC demora mucho más en generar los hashes. Además, si la tecnología avanza y las PC se hacen más rápidas, BCrypt se puede configurar (de una forma muy sencilla) para complejizarse mucho más.

👉 La recomendación es usar **BCrypt** ([link](#)) y hay un [paquete en npm](#) para ello. ⚠ Si ese paquete no les anda (muy probablemente en Windows), prueben con [este otro](#).



# Cookies y Sesiones

# Cookies 🍪 (1/3)



- Son pequeñas cantidades de **datos** que un sitio web guarda (y luego lee) en el **navegador** de un usuario (generalmente sin que éste lo sepa).
- Fueron diseñadas para que un sitio pueda guardar información relativa al “estado” de una aplicación. Por ejemplo, para guardar:
  - Items en un carrito de compras (mientras el usuario navega).
  - Datos del navegante como nombre, email, dirección, etc. (aunque no muy común).
  - Preferencias del navegante como colores o *layout* de una página. Ej: dark/light themes.
  - Páginas visitadas por un navegante (*tracking cookies*).
- Están asociadas a un dominio. Por lo tanto, un sitio web “A” no puede acceder a las cookies guardadas por un sitio web “B”.
- En cada *request*, el navegador adjunta las cookies existentes en los *headers*.

# Cookies 🍪 (2/3)



Otro de los usos más comunes para cookies es para guardar algún dato que permita **determinar si el navegante es un usuario logueado** en el sitio. De lo contrario, sería necesario pedirle al navegante que ingrese sus credenciales cada vez que quiera acceder a una página privada.

A este tipo de cookies generalmente se las conoce como ***authentication cookies***.

# Cookies 🍪 (3/3)



¿Qué dato podríamos guardar en una *authentication cookie*? 🤔 Deberíamos guardar algún dato que permita identificar al usuario.

Claramente no podemos guardar las credenciales sin encriptar, pero aunque lo hiciésemos, se daría el siguiente problema: si para cada *request* se debe enviar el username y password (encriptados), del lado del servidor habría que desencriptarlos y validarlos contra la BD... es decir, habría que acceder a la BD para cada *request*, lo cual es poco eficiente 🐢 (aunque es común hacerlo).

Lo mejor suele ser guardar un *token* 🎉 (en una cookie), que no es más que un string con ciertos datos, que le permita al servidor identificar al usuario, idealmente sin necesidad de hacer consultas a la BD.

# Sesiones (1/2)



Generalmente le decimos sesión al **intervalo de tiempo** mientras que un usuario permanece **logueado** en un sitio web.

La sesión comienza cuando el usuario se loguea en la aplicación. Aquí es cuando se crea la **authentication cookie**. La sesión se identifica con un **Session Id** y esto es lo que se suele guardar en la cookie. No se utiliza el User Id porque el usuario podría estar logueado en más de un equipo y, por lo tanto, tener **más de una sesión abierta**.

La sesión termina cuando el usuario se desloguea o cuando haya pasado determinada cantidad de tiempo. Para este último caso decimos que la sesión expiró. Aquí es cuando la cookie se destruye.




# Sesiones (2/2)



- Notar que HTTP es un protocolo *stateless* (sin estado). No hay realmente una “sesión” o “conexión permanente” entre el cliente y el servidor. Es más que nada una “ilusión” que se logra guardando datos de la sesión en el cliente y/o en el servidor.
- En el servidor, los datos de la sesión se puede guardar en memoria, en un archivo o en una base de datos.
- Notar que cuando se trabaja con Web APIs no hay cookies ya que no necesariamente hay un navegador en el proceso. Para cada llamado a la API en necesario adjuntar las credenciales de autenticación (contraseña o token).



# Passport.js

 Passport.js es una librería cuya documentación requiere una lectura en detalle. Las siguientes diapositivas son sólo un pequeño resumen.



# Passport.js (1/11)

Es un **middleware de autenticación** para Node.js. Su único objetivo es autenticar *requests*.

Es flexible, modular (fácil de agregar a nuestra aplicación) y soportar varias “**estrategias**” de autenticación como username/password, Facebook, Twitter y más.

Documentación: <http://www.passportjs.org>.

Instalación:

```
npm i passport
```



# Passport.js (2/11)

Para usar Passport con **username/password**, es necesario instalar este otro [paquete](#):

```
npm i passport-local
```

Además, será necesario instalar un [paquete](#) para gestionar sesiones:

```
npm i express-session
```

Notar que este paquete es totalmente independiente de Passport.  
Será el encargado de crear la cookie de autenticación.



# Passport.js (3/11)

Luego, requerimos los módulos antes instalados:

```
const session = require("express-session");  
  
const passport = require("passport");  
  
const LocalStrategy = require("passport-local").Strategy;
```



# Passport.js (4/11)

Hay que decirle a Express que utilice el middleware `session`:

```
app.use(  
  session({  
    secret: "AlgúnTextoSuperSecreto",  
    resave: false, // Docs: "The default value is true, but using the default has been deprecated".  
    saveUninitialized: false, // Docs: "The default value is true, but using the default has been deprecated".  
  })  
);
```

Por detalles sobre las opciones de configuración, consultar la documentación de [express-session](#). Por ejemplo, se puede definir una **fecha de expiración** para la sesión. También se puede definir una **store** (donde se guardará la sesión en el servidor). Por defecto, se usa la `MemoryStore` (la memoria RAM del servidor).

Además, recordar que este middleware es independiente de Passport.



# Passport.js (5/11)

Es necesario especificarle a Express que utilice **Passport** con el siguiente middleware. Esto se utiliza para “inicializar” a Passport.

```
app.use(passport.initialize());
```

Además, dado que usaremos sesiones, también es necesario usar el siguiente middleware. Es importante que esto se ejecute luego del middleware `session` de la diapositiva anterior.

```
app.use(passport.session());
```



# Passport.js (6/11)

Es necesario especificarle a Passport la **estrategia** que usaremos. Por ejemplo, para usar login con username/password usamos la estrategia “local” que habíamos importado previamente:

```
passport.use(new LocalStrategy(  
  // Hay varias opciones de configuración  
  // que se pueden consultar aquí.  
  // Ver la documentación de Passport por detalles.  
));
```

Se tiene que especificar por lo menos una estrategia, pero se podrían haber especificado adicionales.





# Passport.js (7/11)

Es necesario especificarle a Passport **qué** es lo que debe **guardar en la sesión** de autenticación. Lo más común en estos casos es guardar el `id` del usuario.

A su vez, hay que especificarle a Passport **qué** debe **hacer cuando recibe la cookie**.

```
passport.serializeUser(function (user, done) {  
  done(null, user.id);  
});  
  
passport.deserializeUser(function (id, done) {  
  User.findById(id)  
    .then((user) => {  
      done(null, user);  
    })  
    .catch((error) => {  
      done(error, user);  
    });  
});
```

Los métodos `serializeUser` y `deserializeUser` son propios de Passport. También la función `done`.

En este ejemplo se está usando Sequelize (y su método `findById`) para determinar si el `id` que contiene la cookie corresponde a un usuario válido. Pero también se podría haber usado otro método, o ni siquiera haber usado Sequelize.



# Passport.js (8/11)

Una vez que Passport esté configurado, quedan dos puntos por definir:

- Proceso de login/registro.
  - Rutas de login y registro. Ej: `/login` y `/register`. Son dos GET y dos POST.
  - Vistas de login y registro (los formularios).
  - Controlador de login y registro. Ej: `authController.js`. Esto es opcional, aunque recomendable.
- Rutas que debe quedar protegidas (privadas).

Ej: todas las rutas relativas al Admin deben estar privadas. El resto, públicas.



# Passport.js (9/11)

Gracias a Passport, la ruta [POST] `/login` queda muy sencilla:

```
app.post(  
  "/login",  
  passport.authenticate("local", {  
    successRedirect: "/admin",  
    failureRedirect: "/login",  
  })  
);
```



# Passport.js (10/11)

Ejemplo de ruta [POST] `/register`

```
app.post("/register", async (req, res) => {  
  const [user, created] = await User.findOrCreate({  
    // Ver opciones en Sequelize.  
  });  
  if (created) {  
    req.login(user, () => res.redirect("/admin"));  
  } else {  
    res.redirect("back");  
  }  
});
```

Este código es simplemente un ejemplo o guía para implementar el controlador. En caso de no haber usado Sequelize, el código naturalmente sería diferente. Del mismo modo, las páginas a donde se quiera redirigir al usuario pueden ser distintas.



# Passport.js (11/11)

Para proteger un controlador, es necesario usar el método `isAuthenticated`.

```
app.get("/admin", (req, res) => {  
  if (req.isAuthenticated()) {  
    res.render("admin");  
  } else {  
    res.redirect("/login");  
  }  
});
```

Curiosamente, este método no está en la documentación oficial, 🤖 al menos a mayo de 2020. Ver [issue](#) en GitHub.

👉 Dado que sería engorroso hacer esta validación dentro de cada controlador, se recomienda crear un [middleware](#) que se encargue de la misma. Luego asignar dicho middleware a las rutas correspondientes.