Appunti su Autenticazione nelle Applicazioni Web

1. Autenticazione vs. Autorizzazione

- **Autenticazione**: verifica l'identità dell'utente (es. username e password).
- **Autorizzazione**: determina se l'utente ha i permessi per accedere a determinate risorse.

Entrambi sono fondamentali per la sicurezza delle applicazioni web.

2. Complessità della gestione di Autenticazione e Autorizzazione

- Implementare autenticazione e autorizzazione è complesso, richiede tempo e può introdurre errori.
- Spesso è meglio affidarsi a best practice e strumenti consolidati.

3. Sessioni e Cookie

- **HTTP è stateless**, quindi serve un meccanismo per mantenere lo stato dell'utente tra richieste diverse.
- **Sessioni**: dati temporanei scambiati tra client e server.
    - Identificate tramite un **Session ID**.
    - Il server salva i dati della sessione.
    - Il client riceve un cookie contenente il Session ID.
- **Cookie**: piccoli file salvati nel browser, usati per mantenere lo stato della sessione.
    - Devono essere sicuri (https, httpOnly).
    - Non devono contenere informazioni sensibili.

4. Autenticazione basata su Sessione

1. L'utente inserisce username e password.
2. Il server verifica le credenziali e genera un **Session ID**.
3. Il client riceve un cookie contenente il Session ID.
4. Ad ogni richiesta successiva, il client invia il cookie per essere riconosciuto.
5. Il server recupera i dati della sessione per autorizzare l'accesso.

5. Sicurezza nell'Autenticazione

- Usare sempre **HTTPS** e cookie **secure**.
- Mai memorizzare password in chiaro.
- Proteggere da attacchi **CSRF** e **XSS**.
- Usare librerie e framework affidabili.

6. Implementazione pratica con Passport.js e React

Login con Passport.js

1. Installare Passport.js: npm install passport passport-local
2. Configurare una strategia di autenticazione locale:

```
import passport from 'passport';
import LocalStrategy from 'passport-local';
passport.use(new LocalStrategy((username, password, callback) => {
  dao.getUser(username, password).then(user => {
    if (!user) return callback(null, false, { message: 'Invalid credentials' });
    return callback(null, user);
  });
}));
```

3. Salvare la sessione dell'utente:

```
passport.serializeUser((user, cb) => cb(null, user.id));
passport.deserializeUser((id, cb) => dao.getUserById(id).then(user => cb(null, user)));
```

Hashing delle password

- Usare funzioni di hashing sicure come scrypt:

```
import crypto from 'crypto';
const salt = crypto.randomBytes(16);
crypto.scrypt(password, salt, 32, (err, hashedPassword) => { /* store hashed password */ });
```

- Verificare la password confrontando gli hash:

```
crypto.timingSafeEqual(storedPassword, hashedPassword);
```

7. Protezione delle Rotte

- Usare req.isAuthenticated() per verificare se l'utente è autenticato.
- Creare un middleware per proteggere le API:

```
const isLoggedIn = (req, res, next) => {
  if (req.isAuthenticated()) return next();
  return res.status(401).json({ message: 'Not authenticated' });
};
```

- Applicarlo alle rotte protette:

```
app.get('/api/exams', isLoggedIn, (req, res) => { /* logica della route */ });
```

8. Logout

- Il server rimuove la sessione quando l'utente esegue il logout:

```
app.post('/api/logout', (req, res) => {
  req.logout(() => res.end());
});
```

9. Autenticazione con CORS

- Il server deve configurare le **opzioni CORS**:

```
app.use(cors({ origin: 'http://localhost:3000', credentials: true }));
```

- Il client deve inviare richieste con credenziali:

```
fetch(SERVER_URL + '/api/exams', { credentials: 'include' });
```