# SAE3.02 - Développer des applications communicantes

---

## Présentation du projet

Dans le cadre de ce projet, nous avons développé une application web interactive mettant en œuvre un jeu de type clicker.

L'objectif de cette application est de créer une expérience utilisateur simple et engageante, tout en intégrant des fonctionnalités de stockage de données.

Une fois authentifié, l'utilisateur accède à une page de jeu où il doit cliquer sur une image centrale pour augmenter son score. Ce score, qui reflète le nombre de clics cumulés au fil des sessions, est enregistré  la base de données et est associé à chaque joueur pour créer un classement

Un classement dynamique des cinq meilleurs joueurs sera ensuite calculé , permettant aux utilisateurs de suivre les performances en direct.

On a aussi ajouté la fonctionnalité d'un chat en direct et d'un chat privé.

Afin d'avancer au mieux et de proposer un bon rendu, nous nous sommes organisé de cette manière :

**Attribution des tâches :**

| | |
|---|---|
| **Assistant et Responsable compte rendu** | Rohan ALAMELOU |
| **Responsable JavaScript** | Brice BERNARDIN |
| **Responsable PHP** | Bilel BOUGHLEM |

Cette structure a été décidée selon les facilités et les connaissances de chacun.

Brice a des qualités rapides en appréhension et à donc commencer le développement en JavaScript, Bilel ayant des qualités en PHP a commencer le développement de son côté et Rohan de son côté assiste du mieux qu'il peut les responsables dans le développement et sa progression.

Après avoir commencé à organiser et structurer le projet nous avons pu commencer.

Dans un premiers temps, lors du début du projet, nous avons décidé de ce plan, le fait d'avoir deux applications développées dans des langages différents nous a permis d'être polyvalent et d'avancer aisément.

## Technologie utilisée

Tout au long de ce projet nous avons choisi d'utiliser le JavaScript et le PHP. Le JavaScript dans un premier temps car notre intervenant nous a introduit au projets avec le JavaScript et le PHP car nous avons des qualités en PHP.

Ensuite pour développer le projet nous avons choisis d'installer XAMPP sur nos machines windows, ce choix a été fait pour le développement de

l'application en JavaScript car XAMPP nous confère une simplicité dans la gestion des services.

XAMPP nous donne un environnement complet pour développer les bases de données avec MYSQL et pour gérer un serveur WEB il y a Apache.
Les services présent sont parfait, car nous maîtrisons ces services d'où le choix de XAMPP

Donc MYSQL et Apache ont été choisis car ces services nous sont familiers, nous les maîtrisons.

Ensuite, débutant dans le JavaScript, Node.js s'est avéré être l'outil parfait pour nous répondant à nos attentes sur le développement backend de notre application en JavaScript. Le service Node.js nous a notamment servi pour le développement de fonctionnalités en temps réel comme le leaderboard et la manipulation de données en JSON.

Pour le développement en PHP, nous avons tout simplement utilisé une Debian, avec apache pour le serveur WEB. Le choix d'une Debian a été fait en rapport à notre aisance sur la manipulation de ce type de machine

En développant notre application, des idées de nouvelles fonctionnalités nous sont venues tout au long du projet.

## Voici nos objectifs finaux à atteindre :

- ☐ **Mettre en place le clicker**
- ☐ **Créer un classement en temps réel**
- ☐ **Permettre la connexion et l'inscription des joueurs**
- ☐ **Mettre en place un chat Générale**
- ☐ **Mettre en place un chat privée**

## Et voici les objectifs finaux atteintes :

- ☑ ~~Mettre en place le clicker~~
- ☑ ~~Créer un classement en temps réel~~
- ☑ ~~Permettre la connexion et l'inscription des joueurs~~
- ☑ ~~Mettre en place un chat Générale~~
- ☑ ~~Mettre en place un chat privée ( que sur la version JS)~~

## Erreur rencontrés

Lorsque l'on a voulu commencer la structure du site, nous avions pensé que mettre un fichier JS, un code JS par page html par rapport à leur fonction , était une bonne idée. Mais lors de cela nous avons rencontré des problèmes tels que la duplication de fichier inutiles, les fonctions risquent d'être dupliquées dans plusieurs fichiers.

Chaque fichier JavaScript ajouté à une page HTML entraîne une requête HTTP supplémentaire. Cela va ralentir le temps de chargement global, étant donné le nombre de fichier JS que l'on aurait créer

On a aussi rencontrés des problèmes de dépendances à cause de cela

Les dépendances croisées ont fait que l'on a rencontré des problèmes difficiles à résoudre d'ou le changement en un fichier "serveur.js"

Et notre projet devenait plus désordonné que structurer d'où le choix de créer "serveur.js" qui regroupe tous les codes et fonctions JavaScript.

## Possibilité d'améliorations

Après la finition de notre application nous avons pu penser à ce que l'on pourrait améliorer si l'on souhaite continuer de la développer plus tard.

- Ajouter un système d'ajout d'amis
- Ajouter la création d'un compte et une gestion d'utilisateurs
- permettre de reset le score individuelle et calculer un CPS (clique par seconde)

## Conclusion

Le projet nous a permis de travailler sur le développement d'une application web interactive tout en consolidant nos compétences techniques et organisationnelles.

Au cours de cette réalisation, nous avons su exploiter nos compétences complémentaires en JavaScript et PHP, tout en utilisant des outils adaptés tels que XAMPP et Node.js pour répondre aux exigences du projet. Malgré des difficultés, comme la gestion initiale des fichiers JavaScript ou les dépendances croisées, nous avons su nous adapter et améliorer la structure en centralisant les fonctionnalités dans un fichier principal, serveur.js.

En conclusion, ce projet a été une excellente opportunité de mettre en œuvre nos connaissances, de développer notre capacité à résoudre des problèmes techniques, et de collaborer efficacement en équipe. Il nous a également ouvert la voie à de nouvelles perspectives d’apprentissage et de perfectionnement pour nos futurs développements.

# ANNEXE : Explication du code serveur.js

Voici ici une explication du code principale de notre code en JavaScript, le code qui gère le Backend.

## package.json

Le fichier package.json et l'installation des dépendances que l'on a besoin a été l'une des premières choses à faire pour assurer le bon fonctionnement de notre code en JS.

Pour créer ce fichier nous avons utilisé la commande "**`npm init`**" dans le dossier du projet qui nous permettra de créer notre fichier selon les informations que l'on rentre.

fichiers package.json :

```
{
  "name": "clicker_app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.0.1",
    "bcryptjs": "^2.4.3",
    "clicker_app": "file:",
    "express": "^4.21.0",
    "express-session": "^1.18.0",
    "mysql": "^2.18.1",
    "mysql2": "^3.11.3",
```

```
    "socket.io": "^4.8.0"
  }
}
```

Nous avons donc ajouter des dépendances nécessaires, il y a :
**bcrypt** : Utilisé pour le hachage des mots de passe.

**bcryptjs** : Version alternative de bcrypt, écrite en JavaScript pur.

**express** : Framework de serveur web.

**clickerapp** : qui fait référence à un fichier locale donc pas besoin de l'installer à l'aide de npm

**express-session** : Middleware pour gérer les sessions dans Express.

**mysql** : Bibliothèque pour interagir avec les bases de données MySQL.

**mysql2** : Une version plus récente et plus performante que mysl.

**socket.io** : Permet la communication en temps réel avec WebSockets.

Ces dépendances sont nécessaires pour notre projet pour la protection des mot de passe utilisateur, la gestion d'un serveur WEB simplement et la gestion des sessions utilisateurs dans un serveur WEB. La gestion d'une base de donnée et enfin la gestion en temps réel de client à serveur.

## Création du fichier du serveur.js

Le fichier serveur.js est essentiel pour l'authentification des utilisateurs, la mise à jour des scores, et la communication en temps réel grâce à Socket.io

La configuration de ce fichier débute avec l'importation des modules que l'on a besoin :

```
const express = require('express');
const mysql = require('mysql2');
const bodyParser = require('body-parser');
const http = require('http');
const socketIo = require('socket.io');
const path = require('path');
const bcrypt = require('bcryptjs');
const session = require('express-session');
```

On initialise maintenant l'application et le serveurs :

```
const app = express();
const server = http.createServer(app);
const io = socketIo(server);
```

On configure ensuite les session utilisateurs, on les paramètre dans cette section Par exemple, la balise secret sert à gérer les cookies de chaque session utilisateurs.

```
Configuration des sessions utilisateurs :
app.use(session({ secret: 'votre_secret',
unique resave: false,
saveUninitialized: true
}));
```

Elle permet de conserver des informations sur chaque utilisateur connecté, comme leur nom d'utilisateur, et d'y accéder tout au long de leur navigation sur le site.

**Middleware pour parser les requêtes**

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(express.static(path.join(__dirname, 'public')));
```

Ce bloc nous permet de traiter les données envoyées par le formulaire et les données en JSON.

## Connexion à la base de données

Ensuite nous avons ici le code qui va nous permettre de se connecter à la base de données, la liaison entre la base de données et le site.

```
const db = mysql.createConnection({
    host: 'localhost',
    user: 'clicker_user',
    password: 'password',
    database: 'clicker_game'
});

db.connect(err => {
    if (err) throw err;
    console.log('Connecté à la base de données MySQL');
});
```

## Les routes

### Route de base avec `GET /`

```
app.get('/', (req, res) => {
res.redirect('/login');
});
```

Cette route est importante, c'est la route de base qui va être utilisée lors utilisateur accède à la racine du site.
Elle redirige automatiquement l'utilisateur vers la page de connexion

### Route d'inscription avec `GET /register`

```
app.get('/register', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'register.html'));
});
```

Cette route va nous permettre d'accéder à la page register.html, qui sert a comme son nom l'indique créer un utilisateur.

### Route du traitement de l' inscription avec `POST /register`

```
app.post('/register', (req, res) => {
    const { username, password } = req.body;

    // Validation de base
    if (!username || !password) {
        return res.status(400).send('Veuillez fournir un nom d'utilisateur
et un mot de passe.');
    }
```

```
    // Vérifier si l'utilisateur existe déjà
    const checkUserSql = 'SELECT * FROM users WHERE username = ?';
    db.query(checkUserSql, [username], (err, results) => {
        if (err) {
            console.error(err);
            return res.status(500).send('Erreur lors de la vérification de
l\'utilisateur.');
        }

        if (results.length > 0) {
            // Si un utilisateur avec ce nom d'utilisateur existe déjà
            return res.status(409).send('Nom d'utilisateur déjà pris.
Veuillez en choisir un autre.');
        } else {
            // Si l'utilisateur n'existe pas, hacher le mot de passe et
l'insérer
            const hashedPassword = bcrypt.hashSync(password, 10);

            const insertUserSql = 'INSERT INTO users (username, password)
VALUES (?, ?)';
            db.query(insertUserSql, [username, hashedPassword], (err,
result) => {
                if (err) {
                    console.error(err);
                    return res.status(500).send('Erreur lors de
l\'inscription de l\'utilisateur.');
                }
                res.redirect('/login'); // Redirige vers la page de
connexion après l'inscription
            });
        }
    });
});
```

Pour plus de compréhension, ce traitement pour créer un compte se passe en plusieurs étapes :

**1- Validation de base** : Elle vérifie que le `username` et le `password` sont fournis. Si non, elle retourne un message d'erreur (code 400).

**2- Vérification de l'existence de l'utilisateur** : Elle exécute une requête pour vérifier si un utilisateur avec ce nom existe déjà dans la base de données (`SELECT * FROM users WHERE username = ?`).

- Si l'utilisateur existe déjà, elle retourne une réponse avec le code 409 pour indiquer que le nom d'utilisateur est déjà pris.

**3 - Création du compte** (Si l'utilisateur n'existe pas)

- Le mot de passe est haché (crypté) avec `bcrypt.hashSync` pour plus de sécurité.

- Une requête `INSERT INTO` est envoyée pour ajouter le nouvel utilisateur avec le nom et le mot de passe haché dans la base de données.

Après l'inscription réussie, l'utilisateur est redirigé vers la page de connexion

### Route de la page de connexion `GET /login`

```
app.get('/login', (req, res) => {
    res.sendFile(path.join(__dirname, 'public',
'login.html'));
});
```

Ce bloc est une route nous renvoyant à la page de connexion login.html

### Route du classement `GET /leaderboard`

```
app.get('/leaderboard', (req, res) => {
    res.sendFile(path.join(__dirname, 'public',
'leaderboard.html')); // Assurez-vous que le chemin est
correct
});
```

### Route pour le chat privé `GET /private-chat`

```
app.get('/private-chat', (req, res) => {
    if (req.session.username) {
        res.sendFile(path.join(__dirname, 'public', 'private-chat.html'));
    } else {
        res.redirect('/login');
    }
});
```

Cette route nous permet d'accéder et de créer une page de chat privée, elle contient une simple conditions qui est celle d'être connecté afin de pouvoir créer la page

### Route pour obtenir le nom d'utilisateur connecté GET /getUsername

```
app.get('/getUsername', (req, res) => {
    const currentUser = req.session.username; // Récupérer
l'utilisateur de la session
    if (currentUser) {
        res.json({ username: currentUser }); // Retourner le
nom d'utilisateur au client
    } else {
        res.status(401).json({ error: 'Utilisateur non
```

```
connecté' });
    }
});
```

Cette route nous permet d'obtenir le nom d'utilisateur qui est connecté depuis la session ouverte, il va donc récupérer le nom d'utilisateur stocker dans req.session.username puis le retourner s'il est connecté et envoyer un message d'erreur sinon.

### Route de connexion POST /login

```
app.post('/login', (req, res) => {
    const { username, password } = req.body;
    const sql = 'SELECT * FROM users WHERE username = ?';
    db.query(sql, [username], (err, results) => {
        if (err) throw err;
        if (results.length > 0) {
            const user = results[0];
            if (bcrypt.compareSync(password, user.password)) {
// Vérifiez le mot de passe
                req.session.username = username; // Mettre à
jour la session
                res.redirect('/game'); // Redirige vers le jeu
après connexion réussie
            } else {
                res.send('Nom d'utilisateur ou mot de passe
incorrect'); // Message d'erreur
            }
        } else {
            res.send('Nom d'utilisateur ou mot de passe
incorrect'); // Message d'erreur
        }
    });
});
```

Cette route importante pour la connexion des utilisateurs va gérer les connexions en vérifiant son nom d'utilisateur et son mot de passe.

Voici les étapes de fonctionnement de notre algorithme :

**1- Récupération des données** : Elle extrait `username` et `password` depuis le corps de la requête.

**2- Vérification de l'utilisateur** :

- Elle exécute une requête pour chercher l'utilisateur dans la base de données (`SELECT * FROM users WHERE username = ?`).

- Si l'utilisateur existe (`results.length > 0`), le mot de passe saisi par l'utilisateur est comparé avec celui stocké dans la base de données à l'aide de `bcrypt.compareSync`.

    - Si le mot de passe est correct, le nom d'utilisateur est enregistré dans la session (`req.session.username`) et l'utilisateur est redirigé vers la page du jeu (`/game`).
    - Si le mot de passe est incorrect, un message d'erreur est renvoyé.

- Si l'utilisateur n'existe pas, un message d'erreur est également envoyé.

**Route pour obtenir le score de l'utilisateur connecté GET /getScore.**

```
app.get('/getScore', (req, res) => {
    const currentUser = req.session.username; // Récupérer
l'utilisateur de la session
    if (currentUser) { // Vérifiez si l'utilisateur est
connecté
        db.query('SELECT score FROM users WHERE username = ?',
[currentUser], (err, results) => {
            if (err) throw err;
            const score = results[0] ? results[0].score : 0;
// Si pas de résultats, score est 0
            res.json({ score }); // Retournez le score au
client
        });
    } else {
        res.status(401).json({ error: 'Utilisateur non
connecté' }); // Renvoie une erreur si pas
connecté
    }
});
```

Cette route est celle qui permet à l'utilisateur connecté d'avoir son score, voici les étapes de fonctionnement de notre algorithme :

**1 - Vérification de la connexion** : Elle vérifie si l'utilisateur est connecté via req.session.username.

- Si l'utilisateur est connecté, une requête est exécutée pour récupérer son score (`SELECT score FROM users WHERE username = ?`).

- Si l'utilisateur a un score, il est retourné sous forme de réponse JSON.

- Si aucun score n'est trouvé pour cet utilisateur (par exemple, si c'est un nouvel utilisateur), la route retourne un score de `0`.

- Si l'utilisateur n'est pas connecté, elle renvoie un message d'erreur JSON avec le code 401.

### Route pour accéder à la page de jeu GET `/game`

```
app.get('/game', (req, res) => {
    if (req.session.username) { // Vérifiez si l'utilisateur
est connecté
        res.sendFile(path.join(__dirname, 'public',
'game.html'));
    } else {
        res.redirect('/login'); // Redirige vers la page de
connexion si pas connecté
    }
});
```

Cela nous permet simplement d'aller à la page du jeu, ce code vérifie simplement si l'utilisateur est connecté.

### Gérer le clic POST `/click`

```
app.post('/click', (req, res) => {
```

```
    const currentUser = req.session.username; // Récupérer
l'utilisateur de la session
    if (currentUser) { // Vérifiez si l'utilisateur est
connecté
        // Mettre à jour le score de l'utilisateur
        const sql = 'UPDATE users SET score = score + 1 WHERE
username = ?';
        db.query(sql, [currentUser], (err) => {
            if (err) {
                console.error(err);
                return res.status(500).json({ error: 'Erreur
lors de la mise à jour du score' });
            }

            // Récupérer le score mis à jour
            db.query('SELECT score FROM users WHERE username =
?', [currentUser], (err, results) => {
                if (err) {
                    console.error(err);
                    return res.status(500).json({ error:
'Erreur lors de la récupération du score' });
                }

                const score = results[0].score; // Récupérer
le score mis à jour
                res.json({ score }); // Retourner le score au
client
            });
        });
    } else {
        res.status(401).json({ error: 'Utilisateur non
connecté' });
    }
});
```

Cette route gère la fonction principale de notre jeu, l'action d'un clic de l'utilisateur qui va augmenter son score.

Pour procéder à cela voici les étape de fonctionnement de notre code

**1-** La condition de base qui se répète, **Vérification de la connexion de l'utilisateur.**

- **Si l'utilisateur est connecté, une requête SQL est exécutée pour incrémenter le score de cet utilisateur de 1 après un click**

**2-** Après l'incrémentation, une autre requête récupère le score mis à jour pour confirmation.

**3-** Si l'incrémentation et la récupération du score réussissent, la route retourne le score mis à jour.

En cas d'erreur dans le processus, un message d'erreur JSON est renvoyé.

## Classement `GET /api/leaderboard`

```
app.get('/api/leaderboard', (req, res) => {
    const sql = 'SELECT username, score FROM users ORDER BY
score DESC';
    db.query(sql, (err, results) => {
        if (err) {
            console.error(err);
            return res.status(500).json({ error: 'Erreur lors
de la récupération du classement' });
```

```
        }
        res.json(results); // Envoie les résultats au format
JSON
    });
});
```

Cette route fait référence au leaderboard, il permet de récupérer le classement global utilisateurs, triés par score de manière décroissante.

Voici les étapes du fonctionnement de ce bloc :

**1-** La requête SQL récupère les noms d'utilisateur et leurs scores, triés par score en ordre décroissant.

**2-** Si la requête réussit, le classement est renvoyé en réponse JSON.

**3-** En cas d'erreur, un message d'erreur JSON est retourné avec le statut 500.

### Déconnexion GET /logout

```
app.get('/logout', (req, res) => {
    req.session.destroy(err => {
        if (err) {
            return res.status(500).send('Erreur lors de la
déconnexion');
        }
        res.redirect('/login'); // Redirige vers la page
de connexion après déconnexion
    });
});
```

Cette route permet tout simplement de gérer la déconnexion d'un utilisateur

### Socket.IO pour le chat privé

```
io.on('connection', (socket) => {
    console.log('Un utilisateur est connecté');

    socket.on('private message', (data) => {
        const { sender, receiver, message } = data;

        // Enregistrer le message dans la base de données
        const insertMessageSql = 'INSERT INTO private_messages
(sender, receiver, message) VALUES (?, ?, ?)';
        db.query(insertMessageSql, [sender, receiver,
message], (err) => {
            if (err) {
                console.error(err);
                return; // Ignore les erreurs pour le moment
            }

            // Émettre le message aux utilisateurs concernés
            io.emit('private message', { sender, receiver,
message });
        });
    });

    socket.on('disconnect', () => {
        console.log('Un utilisateur s\'est déconnecté');
    });
});
```

Cette section gère le chat privée avec Socket.IO

**Rappel :**

Socket.IO permet de gérer la communication en temps réel entre un client (application web) et un serveur Node.js. Elle utilise le protocole WebSocket pour établir une connexion bidirectionnelle, permettant au serveur et au client d'échanger des messages instantanément, sans attendre que le client envoie une nouvelle requête.

Pour gérer et stocker les messages dans la base de données le code fonctionne comme suit :

**1-** Lorsqu'un message privé est envoyé, l'application le reçoit dans **`socket.on('private message', ...)`**.

**2-** Le message est ensuite stocké dans la base de données à l'aide de **`INSERT INTO private_messages`**.

**3-** Après enregistrement, le message est émis aux utilisateurs concernés via **`io.emit`**

**Route pour obtenir la liste des utilisateurs**

```
app.get('/users', (req, res) => {
    const sql = 'SELECT username FROM users'; // Sélectionne
uniquement les noms d'utilisateur
    db.query(sql, (err, results) => {
        if (err) {
            console.error(err);
            return res.status(500).json({ error: 'Erreur lors
de la récupération des utilisateurs' });
```

```
        }
        res.json(results); // Retourne la liste des
utilisateurs sous forme de JSON
    });
});
```

Cette route récupère simplement tous les utilisateurs de la tables users, puis la liste est envoyée sous forme de JSON.

### Route pour récupérer les messages privés entre deux utilisateurs

```
app.get('/private-messages', (req, res) => {
    const currentUser = req.session.username; // Utilisateur
connecté
    const otherUser = req.query.otherUser; // Utilisateur
cible dans la conversation

    if (!currentUser || !otherUser) {
        return res.status(400).json({ error: 'Utilisateur non
connecté ou destinataire manquant' });
    }

    const sql = `
        SELECT * FROM private_messages
        WHERE (sender = ? AND receiver = ?) OR (sender = ? AND
receiver = ?)
        ORDER BY timestamp ASC
    `;

    db.query(sql, [currentUser, otherUser, otherUser,
currentUser], (err, results) => {
        if (err) {
            console.error(err);
            return res.status(500).json({ error: 'Erreur lors
```

```
de la récupération des messages privés' });
        }
        res.json(results); // Retourne les messages sous forme
de JSON
    });
});
```

Cette route permet de récupérer tous les messages privés échangés entre l'utilisateur connecté (`currentUser`) et un autre utilisateur (`otherUser`).

Elle vérifie ensuite que `currentUser` et `otherUser` existent, sinon elle renvoie une erreur `400`.

Puis elle sélectionne tous les messages dans lesquels l'utilisateur actuel est soit l'expéditeur, soit le destinataire et les trie par ordre chronologique (`ORDER BY timestamp ASC`).

### Route pour obtenir le nom d'utilisateur actuel

```
app.get('/current-user', (req, res) => {
    if (req.session.username) {
        res.json({ username: req.session.username }); //
Renvoie le nom d'utilisateur actuel
    } else {
        res.status(401).json({ error: 'Utilisateur non
connecté' });
    }
});
```

Cette route nous renvoie le nom du user qui est actuellement connecté, si le user n'est pas connecté une réponse avec une erreur est renvoyée.

## Socket.IO pour le chat en temps réel

```
io.on('connection', (socket) => {
    console.log('Un utilisateur est connecté');

    socket.on('new message', (data) => {
        const { username, message } = data;
        io.emit('new message', { username, message }); //
Diffuse le message à tous les utilisateurs
    });

    socket.on('disconnect', () => {
        console.log('Un utilisateur s\'est déconnecté');
    });
});
```

Cette section utilise Socket.IO pour gérer les connexions en temps réel. Lorsqu'un utilisateur se connecte, un message est affiché dans la console. Quand le serveur reçoit un nouveau message (new message), il le diffuse instantanément à tous les utilisateurs connectés, un message s'affiche dans la console.

### Route pour le message de bienvenue et le classement avec (/welcome-message)

```
app.get('/welcome-message', (req, res) => {
    const currentUser = req.session.username;

    if (!currentUser) {
        return res.status(401).json({ error: 'Utilisateur non
connecté' });
    }
```

```
    const sql = `
        SELECT username, score,
            (SELECT COUNT(*) + 1
             FROM users AS u2
             WHERE u2.score > u1.score) AS ranking
        FROM users AS u1
        WHERE username = ?
    `;

    db.query(sql, [currentUser], (err, results) => {
        if (err) {
            console.error(err);
            return res.status(500).json({ error: 'Erreur lors
de la récupération des données utilisateur' });
        }

        if (results.length > 0) {
            const { username, ranking } = results[0];
            res.json({ message: `Bienvenue ${username}`,
ranking });
        } else {
            res.status(404).json({ error: 'Utilisateur non
trouvé' });
        }
    });
});
```

Ce bloc vérifie encore si un utilisateur est connecté, si l'utilisateur est connecté, une requête SQL est lancée, qui va permettre de déterminer le classement de l'utilisateur actuel basé sur les scores.

Elle compte le nombre d'utilisateurs ayant un score supérieur et ajoute 1 pour donner le rang exact.

Ce qui est renvoyé est un message de bienvenue personnalisé avec le classement de l'utilisateur

Et enfin nous démarrons le serveur sur le bloc 3000 grâce à ce bloc :

```
server.listen(3000, () => {
    console.log('Serveur démarré sur le port 3000');
});
```

Sur server.js, nous avons donc configuré le serveur backend de notre clicker. Pour ce faire nous avons utilisé Node.js, Express, MySQL, Socket.IO afin de gérer tout cela.

Le code du jeu est fourni avec des commentaires en compléments.