

1 Mise en place de l'environnement de développement

Installez en pont (@IP+100) la machine virtuelle debian11Node disponible à l'adresse :

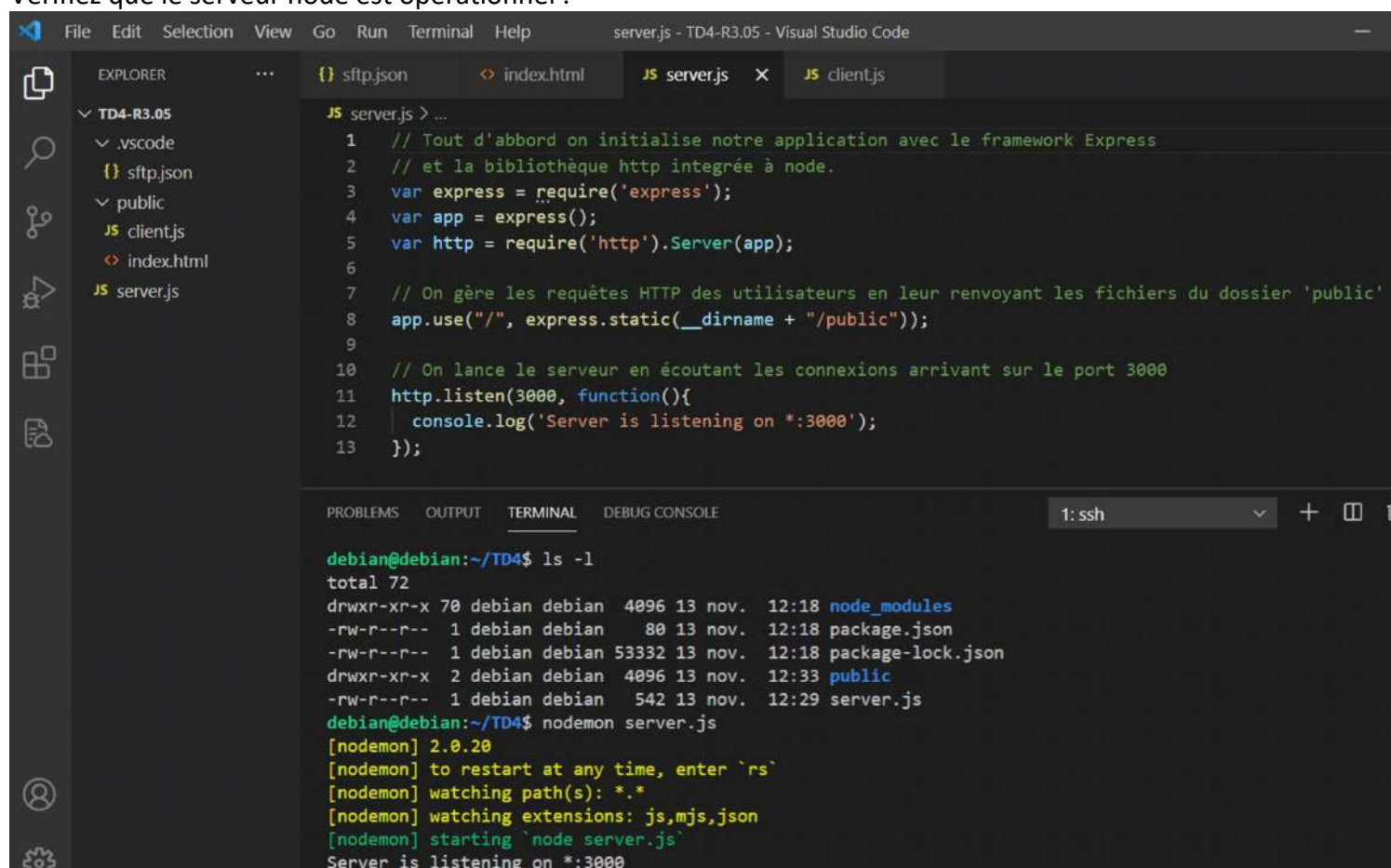
U:\VM\INFO\INFO1\Debian11_server

Dans la machine virtuelle en tant qu'utilisateur **debian** :

- Installez nodemon (npm install -g nodemon)
- Créez un répertoire TD4 et y installez express, socket.io (npm install express et npm install socket.io)
- Créez un répertoire public dans TD4 pour y placer les sources client.

Vous pouvez utiliser Visual Studio Code pour synchroniser directement vos sources (Remote SSH ou SFTP).

Vérifiez que le serveur node est opérationnel :



```

File Edit Selection View Go Run Terminal Help
server.js - TD4-R3.05 - Visual Studio Code

EXPLORER
TD4-R3.05
  .vscode
  {} sftp.json
  public
    JS client.js
    index.html
  JS server.js

JS server.js > ...
1 // Tout d'abord on initialise notre application avec le framework Express
2 // et la bibliothèque http intégrée à node.
3 var express = require('express');
4 var app = express();
5 var http = require('http').Server(app);
6
7 // On gère les requêtes HTTP des utilisateurs en leur renvoyant les fichiers du dossier 'public'
8 app.use("/", express.static(__dirname + "/public"));
9
10 // On lance le serveur en écoutant les connexions arrivant sur le port 3000
11 http.listen(3000, function(){
12   console.log('Server is listening on *:3000');
13 });

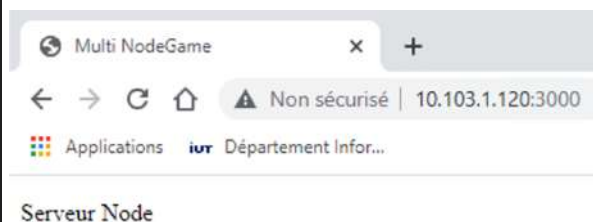
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
1: ssh
debian@debian:~/TD4$ ls -l
total 72
drwxr-xr-x 70 debian debian 4096 13 nov. 12:18 node_modules
-rw-r--r-- 1 debian debian 80 13 nov. 12:18 package.json
-rw-r--r-- 1 debian debian 53332 13 nov. 12:18 package-lock.json
drwxr-xr-x 2 debian debian 4096 13 nov. 12:33 public
-rw-r--r-- 1 debian debian 542 13 nov. 12:29 server.js
debian@debian:~/TD4$ nodemon server.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server is listening on *:3000
  
```

Et vérifiez l'accès à votre serveur depuis la machine hôte :



```

<!doctype html>
<html>
  <head>
    <title>Multi NodeGame</title>
    <script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
    <style>
    </style>
  </head>
  <body>
    <p>Serveur Node</p>
    <script src="client.js"></script>
  </body>
</html>
  
```



2 Implémentation de socket.io

Nous allons coder un client et un serveur pour gérer plusieurs joueurs. Le but est de pouvoir faire apparaître un nouveau joueur de façon aléatoire à chaque nouvelle connexion au serveur.

Ajoutez l'import du module dans index.html :

```
<!doctype html>
<html>
  <head>
    <title>Multi NodeGame</title>
    <script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
    <style>
    </style>
  </head>
  <body>
    <p>Serveur Node</p>
    <script src="/socket.io/socket.io.js"></script>
    <script src="client.js"></script>
  </body>
</html>
```

Et ajouter coté serveur :

```
var io = require('socket.io')(http);
```

Première étape : coté serveur

A chaque connexion on crée un nouveau joueur aléatoirement (canvas 800x600) avec une couleur aléatoire

```
var players = {};

io.on('connection', function (socket) {
  console.log('a user connected: ', socket.id);
  // creation d'un nouveau et ajout à la liste des joueurs
  players[socket.id] = {
    x: Math.floor(Math.random() * 700) + 50,
    y: Math.floor(Math.random() * 500) + 50,
    playerId: socket.id,
    c: 0xFFFFFF*Math.random()|0
  };
});
```

Deuxième étape : coté client

Faire une requête de connexion :

```
var config = {
  type: Phaser.AUTO,
  parent: 'phaser-example',
  width: 800,
  height: 600,
  physics: {
    default: 'arcade',
    arcade: {
      debug: false,
      gravity: { y: 0 }
    }
  },
  scene: {
    preload: preload,
    create: create,
    update: update
  }
};
```

```

var game = new Phaser.Game(config);

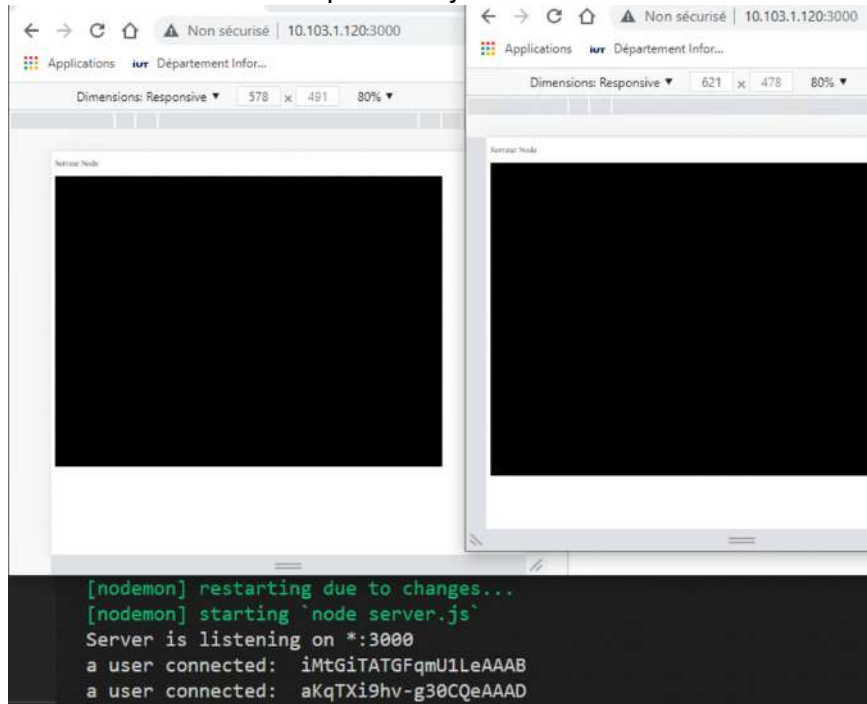
function preload() {
}

function create() {
    var self = this;
    this.socket = io();
}

function update() {
}

```

Vérifiez la connexion de plusieurs joueurs :



3 Mise en place du jeu multi-joueurs

Vérifiez coté serveur les coordonnées aléatoires des nouveaux joueurs :

```

[nodemon] starting `node server.js`
Server is listening on *:3000
a user connected: xarZT0DKrzb8DE7dAAAC
caractéristiques du joueur :
id : xarZT0DKrzb8DE7dAAAC
x,y : 210 , 444
couleur : #f2e76e
a user connected: WxbkrS9I7MH52HJVAAAD
caractéristiques du joueur :
id : WxbkrS9I7MH52HJVAAAD
x,y : 501 , 494
couleur : #2941f4

```

Mettre à jour la liste des joueurs pour le client et inversement :

```

// envoi des joueurs au nouveau joueur
socket.emit('currentPlayers', players);

// mise à jour vers les autres joueurs du nouveau joueur
socket.broadcast.emit('newPlayer', players[socket.id]);

```

Codez coté client (ajoutez dude.png dans assets):

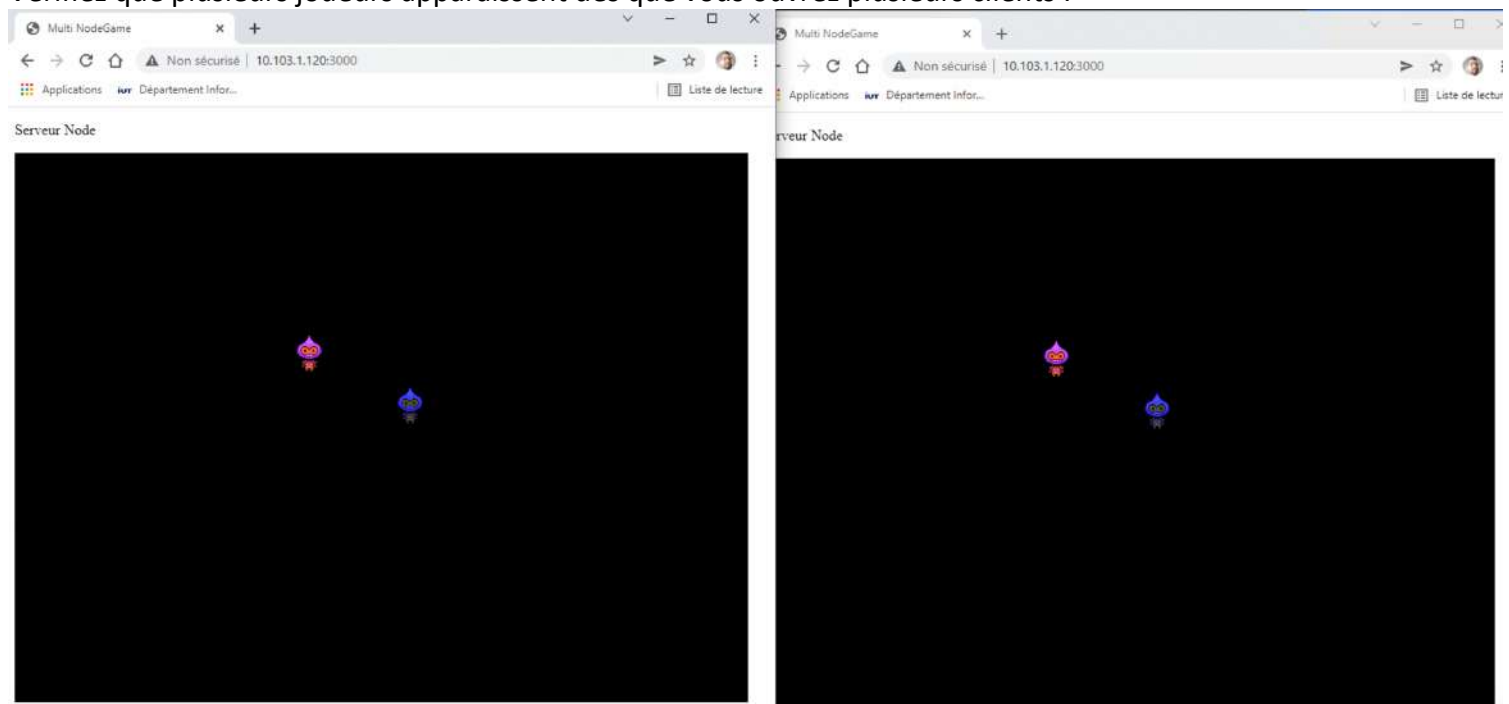
```
function preload() {  
  
    this.load.spritesheet('dude', 'assets/dude.png', { frameWidth: 32, frameHeight: 48 });  
}
```

```
this.otherPlayers = this.physics.add.group();  
this.socket.on('currentPlayers', function (players) {  
    Object.keys(players).forEach(function (id) {  
        if (players[id].playerId === self.socket.id) {  
            addPlayer(self, players[id]);  
        } else {  
            addOtherPlayers(self, players[id]);  
        }  
    });  
});  
this.socket.on('newPlayer', function (playerInfo) {  
    addOtherPlayers(self, playerInfo);  
});
```

Ajout du joueur en cours et les autres

```
function addPlayer(self, playerInfo) {  
    self.perso = self.physics.add.sprite(playerInfo.x, playerInfo.y, 'dude',4);  
    self.perso.setTint(playerInfo.c);  
    self.perso.setCollideWorldBounds(true);  
}  
  
function addOtherPlayers(self, playerInfo) {  
    const otherPlayer = self.add.sprite(playerInfo.x, playerInfo.y, 'dude',4);  
    otherPlayer.setTint(playerInfo.c);  
    otherPlayer.playerId = playerInfo.playerId;  
    self.otherPlayers.add(otherPlayer);  
}
```

Vérifiez que plusieurs joueurs apparaissent dès que vous ouvrez plusieurs clients :



Attention veillez à bien gérer la déconnexion (coté serveur puis client) pour faire disparaître les joueurs quand vous fermez les clients :

```
// when a player disconnects, remove them from our players object
socket.on('disconnect', function () {
  console.log('user disconnected: ', socket.id);
  delete players[socket.id];
  io.emit('disconnection', socket.id);
});

this.socket.on('disconnection', function (playerId) {
  self.otherPlayers.getChildren().forEach(function (otherPlayer) {
    if (playerId === otherPlayer.playerId) {
      otherPlayer.destroy();
    }
  });
});
});
```

4 Gestion des déplacements des joueurs

Maintenant il ne reste plus qu'à gérer les déplacements des joueurs :

Coté serveur :

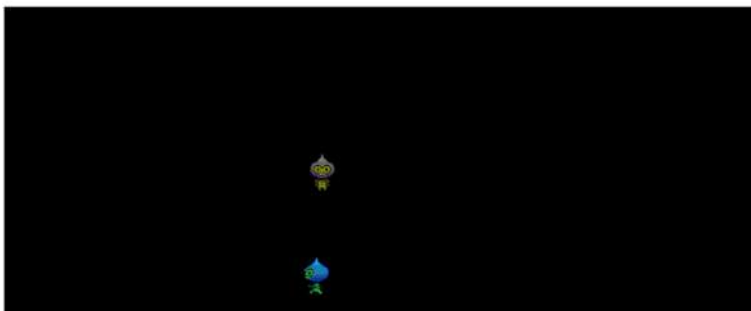
```
socket.on('playerMovement', function (movementData) {
  players[socket.id].x = movementData.x;
  players[socket.id].y = movementData.y;
  // envoi un message aux autres joueur que le joueur s'est déplacé
  socket.broadcast.emit('playerMoved', players[socket.id]);
});
```

Coté client traitez les déplacements des autres joueurs

```
this.socket.on('playerMoved', function (playerInfo) {
  self.otherPlayers.getChildren().forEach(function (otherPlayer) {
    if (playerInfo.playerId === otherPlayer.playerId) {
      otherPlayer.setPosition(playerInfo.x, playerInfo.y);
    }
  });
});
```

Et ajouter la gestion du déplacement du joueur dans update (pensez aux animations du sprite) :

Serveur Node



Rq : pensez à mettre à jour la position du joueur vers le serveur :

```
// envoi des info au serveur
this.socket.emit('playerMovement', { x: this.perso.x, y: this.perso.y});
```

5 Prolongements

Vous pouvez prolonger en réalisant un petit jeu :

Le jeu consiste à récupérer des étoiles qui apparaissent aléatoirement. Chaque joueur à son propre score. Le gagnant est celui qui obtient le meilleur score à la fin d'un chronomètre déclenché à la première apparition d'un joueur.