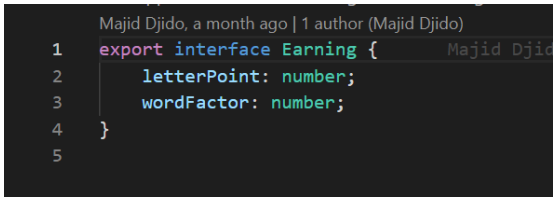
Table de matières

PROGRAMMATION TYPESCRIPT

Interface



Programmation asynchrone

- Un appelant fait appel à une fonction qui procède à un traitement asynchrone mais retourne de façon synchrone un objet container (la Promise) à l'appelant.
- L'appelant inscrit sur la Promise une "callback" de succès pour être informé quand le résultat est disponible (e.g. : promise.then(data => console.log(data)) et une callback d'erreur pour être informé de l'échec (e.g. : promise.catch(error => console.error(error)).
- Quand la fonction appelée obtient le résultat (ou une erreur), elle notifie la Promise qui passe alors à un état "resolved" (ou "rejected").
- La Promise déclenche alors toutes les fonctions de "callback" de succès (ou d'erreur) qui ont pu lui être transmises (via les méthodes .then et .catch).

```
this.router.get('/aiBeginners', async (req: Request, res: Response) => {
    await this.administratorService
        .getAllAiPlayers(AiType.beginner)
        .then((aiBeginners) => {
            res.status(StatusCodes.OK).send(aiBeginners);
        })
        .catch((error: Error) => {
            res.status(StatusCodes.NOT_FOUND).send('An error occurred while trying to get ai be
        });
});
```

Await oblige async et Promise<T> comme type de retour.

Une requête http retourne un observable qu'on peut convertir en Promise

Promise vs observable : Promise est bloquant (seulement si on decide de l'attendre avec await) et les observables non bloquants

ANGULAR

Notions de Template - Component – Service

Modules sont des conteneurs pour les components

Une classe marquée avec le décorateur @NgModule avec les propriétés suivantes :

- imports : d'autres NgModule dont les exports sont utilisés dans ce NgModule
- declarations : les components, directives ou pipes de ce NgModule
- providers : les services qui sont exposés par le NgModule et sont
- accessibles partout dans l'application. Très souvent défini plutôt directement dans chaque service
- exports : les déclarations qui sont visibles si le module est importé ailleurs
- bootstrap : le ou les composantes initiales. Devrait être seulement dans le root NgModule

Component : Élément central d'une application Angular, ils sont composés d'un Template (.html), d'une classe typescript (.ts), d'un fichier de style (.css)

On utilise le décorateur @Component avec les propriétés suivantes :

- selector : comment on fait référence au component. Se comporte comme une balise HTML régulière
- templateUrl : lien vers le gabarit (Template) HTML du component. Peut être aussi donné inline (pas pour LOG2990)
- styleUrls: tableau des liens des fichiers de style (CSS, SASS, etc)
- providers : les services qui sont utilisés par le component. Si onles déclare ici, chaque component aura une nouvelle instance du service lors de sa construction.

Service : classe en typescript qui est injecté dans les components pour effectuer certaines tâches pour celui-ci ou lui amener de l'information.

Elle fonctionne comme un singleton donc la même valeur est partagée par tous les composants du système

Injection de dépendances et gestion des dépendances par Angular

Plutôt que la classe crée elle-même ses dépendances à l'intérieur de son constructeur, elle est construite avec ses dépendances.
Avantages

- Code plus réutilisable. En utilisant des interfaces, on peut changer nos dépendances plus facilement : patron Stratégie
- Code plus testable : Une dépendance peut être remplacée par un mock pour faciliter les tests
- Code plus lisible : Il est facile de voir ce que la classe a besoin de sans fouiller dans le code
- Éviter les chaînes de paramètres

Directives structurelles (*ngIf, *ngFor, etc.)

```
<mat-card id="progress-bar-box">
    <mat-spinner *ngIf="isWaiting" mode="indeterminate"></mat-spinner>
</mat-card>
```

Code qui permet d'afficher le spinner tant qu'on est en attente.

```
<span
    [ngClass]="{
        red: room.state === 0 || room.state === 2,

        green: room.state === 1
    }"
    >{{ computeRoomState(room.state) }}</span      You, a month ago • ges
>
```

Code qui permet d'appliquer du rouge ou du bleu sur le statut de la room en fonction de son état (rouge si occupé ou terminé, vert si en attente).

```
<mat-radio-group [formControlName]="'levelInput'">
    <mat-radio-button color="primary" class="level-radio-box" *ngFor="let selectionValue of levelSelectionList" value="{{ selectionValue }}
        {{ selectionValue }}
    </mat-radio-button>
</mat-radio-group>
```

Code qui permet d'afficher les différentes sélections du niveau. value est la valeur de la sélection.

Data binding (interpolation, propriétés, événements, etc.)

```
<h2>Products</h2>

<div *ngFor="let product of products; index as productId">  DIRECTIVE
     <h3>
          <a [title]="product.name + ' details'"      PROPERTY BINDING
          [routerLink]="['/products', productId]">    ROUTING
          {{ product.name }}    INTERPOLATION
          </a>
     </h3>
     <p *ngIf="product.description"> Description: {{ product.description }} </p>  DIRECTIVE + INTERPOLATION
     <button (click)="share()"> Share </button>  EVENT BINDING
     <app-product-alerts [product]="product" (notify) ="onNotify($event)"> </app-product-alerts>
                         EMBEDDED COMPONENT    + PROPERTY/EVENT BINDING
</div>
```

The binding punctuation of `[]`, `()`, `[()]`, and the prefix specify the direction of data flow.

- Use `[]` to bind from source to view.
- Use `()` to bind from view to source.
- Use `[()]` to bind in a two way sequence of view to source to view.

```
<mat-paginator id="paginator" *ngIf="rooms.length !== 0" [length]="rooms.length" [pageSize]="pageSize" (page)="onPageChange($event)"></mat-
```

Le .ts dit à la vue (ici le paginator) le nombre d'éléments

Le .ts dit à la vue (ici le paginator) le nombre de page

La vue (ici le paginator) provoque sur le .ts un calcul de page onPageChange(…)

Two-way	Event and property	`<input [(ngModel)]="name">`

Lorsqu'on tape au clavier le texte se retrouve dans la variable name à l'intérieur du .ts et tout caractère ajouté a cette variable dans le .ts va s'afficher à la vue.

Text interpolation {{}} est utilisé pour afficher les propriétés de notre code dans la vue

Communication enfant parent se fait avec les patrons @Input() et @Output()

@Input() sert à un parent pour envoyer des infos à son enfant par exemple des initialiser un certain attribut de l'enfant

Code enfant

```
import { Component, Input } from '@angular/core'; // First, import Input
export class ItemDetailComponent {
  @Input() item = ''; // decorate the property with @Input()
}
```

Template enfant

```
<p>
  Today's item: {{item}}
</p>
```

Code parent

```
export class AppComponent {
  currentItem = 'Television';
}
```

Template parent

```
<app-item-detail [item]="currentItem"></app-item-detail>
```

@Output() sert à un enfant pour envoyer des infos à son parent sous forme d'event

Code enfant

```
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

Template enfant

```
<label for="item-input">Add an item:</label>
<input type="text" id="item-input" #newItem>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

Code parent

```ts
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

Template parent

```html
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

Form

```ts
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  form: FormGroup;

  // Init du form toujours dans le ngOnInit
  ngOnInit(): void {
    this.form = new FormGroup({
      // 'DefaultName' sera ecrit par défaut dans l'input
      // Les deux input sont requis (voir l. 24 du HTML)
      nameInput: new FormControl('DefaultName', [Validators.required]),
      descriptionInput: new FormControl('Default description', [Validators.required]),
    });
    // Le nom doit suivre ce pattern et avoir une taille entre 4 et 12
    this.form.controls.nameInput.setValidators([
      Validators.pattern('^([A-Za-z][A-Za-z][A-Za-z][A-Za-z])[A-Za-z0-9]*'),
      Validators.minLength(4),
      Validators.maxLength(12),
    ]);
  }

  // Appelée dans le HTML pour traiter les valeurs du form
  snapshotSettings(): void {
    console.log('Chosen name is ' + this.form.controls.nameInput.value);
    console.log(
      'Chosen description is ' + this.form.controls.descriptionInput.value
    );
  }
}
```

```
<mat-toolbar color="primary">
  <span class="fill-remaining-space">My Reactive Form with Material</span>
</mat-toolbar>

<div class="container">
  <form [formGroup]="form" class="form">
    <mat-form-field class="form-element">
      <input matInput placeholder="Name" formControlName="nameInput" />
    </mat-form-field>

    <mat-form-field class="form-element">
      <input
        matInput
        placeholder="Description"
        formControlName="descriptionInput"
      />
    </mat-form-field>

    <div class="form-element">
      <button
        mat-raised-button
        color="warn"
        class="button"
        [disabled]="!form.valid"
        (click)="snapshotSettings()"
      >
        Continuer
      </button>
    </div>
  </form>
</div>
```

NODE/EXPRESS

Node : Environnement d'exécution qui permet d'utiliser Javascript à l'extérieur d'un fureteur. Conçu pour des applications de type serveur et des outils de réseautique.

Express : C'est un middleware qui fait le lien entre deux composantes logicielles. Il offre une meilleure gestion des routes (Routing) (division des requetes entrantes en fonction de leur URL) pour des sites webs ou des Web APIs.

Router Express permet de regrouper un ensemble de routes qui traitent les requêtes du même domaine

HTTP

Principe :

Fonctionne sur le principe requête-réponse => Pas de requête pas de réponse.

Fonctions :

Format général : xxx(Url, body)

On utilise le module “http” et l’objet HttpClient sur le client pour envoyer et recevoir les réponses

- Ce module implémente la logique de communication HTTP pour nous
- Tous les méthodes prennent un URL comme premier paramètre et un
- objet de type any pour le body, si applicable
- Ex : GET et DELETE n’ont pas de body, mais POST, PATCH et PUT, oui
- Les méthodes retournent un Observable, donc vous devez appeler .subscribe() sinon la méthode ne sera jamais exécutée
- Gardez vos appels HTTP dans un service et pas dans les components

Le body et les paramètres sont dans le corps de la requête (req) s’il y a lieu

- **GET** : obtenir une ou plusieurs ressources
  - Devrait être une opération sans effets secondaires
  - En général, la requête ne devrait pas avoir un *body*
- **POST** : envoyer de l’information vers une ressource
  - L’information se trouve dans le *body* de la requête
- **PUT** : envoyer de l’information pour créer ou remplacer une ressource
  - Si la ressource existe déjà, elle est modifié
- **DELETE** : supprimer une ressource spécifié
  - Certaines implémentations de HTTP ne permettent pas un *body*
- **PATCH** : apporter des modifications partielles à une ressource existante
  - La difference entre PATCH et PUT est subtile, PATCH est à utiliser si on est certain de l’existance de la ressource

- 200 : OK (accompagné avec autre chose)
- 201 : CREATED
- 204 : NO CONTENT (OK, mais accompagné de rien)
- 304 : NOT MODIFIED (si vous accédez à une page dans la cache)
- 400 : BAD REQUEST (requête mal formulée)
- 403 : FORBIDDEN (bonne requête, mais serveur refuse)
- 404 : NOT FOUND
- 410 : GONE (ressource plus disponible)
- 500 : INTERNAL SERVER ERROR (erreur générique)
- 501 : NOT IMPLEMENTED (méthode HTTP non supportée sur cette route)

Fonctionnement :

```
this.validationState = await this.httpServer.validationPost(this.newPlayedWords, this.fileName).toPromise();
      You, seconds ago • Uncommitted changes
```

```
validationPost(newPlayedWords: Map<string, string[]>, fileName: string): Observable<boolean> {      MikeFolep
    this.wordsToValidate = [];
    for (const word of newPlayedWords.keys()) {
        this.wordsToValidate.push(word);
    }
    return this.http.post<boolean>(`${this.baseUrl}/game/validateWords/${fileName}`, this.wordsToValidate);
}
```

```
bindRoutes(): void {
    this.app.use('/api/docs', swaggerUi.serve, swaggerUi.setup(swaggerJSDoc(this.swaggerOptions)));
    this.app.use('/api/game', this.gameController.router);
    this.app.use('/api/admin', this.administratorController.router);
    this.app.use('/', (req, res) => {
        res.redirect('/api/docs');
    });
    this.errorHandling();
}
```

```
this.router.post('/validateWords/:fileName', (req: Request, res: Response) => {
    const isValid = this.wordValidator.isValidInDictionary(req.body, req.params.fileName);
    res.status(StatusCodes.OK).send(isValid);      UrsuleJoelle, 2 weeks ago • AQ et retraits co
});
```

SOCKET

Utilisé pour la communication bidirectionnelle.

Plus rapide que HTTP.

WebSocket nécessite un serveur : il n'y a pas de communication pair-à-pair (peer to peer). Tout passe par un serveur qui peut rediriger les messages.

Peut recevoir des données sans requête ex : sio.emit(…)

Depuis le client

Configuration client

```
export class ClientSocketService {
    socket: Socket;
    roomId: string;
    gameType: GameType;

    constructor(private gameSettingsService: GameSettingsService, private router: Router) {
        this.socket = io(environment.serverUrl);
```

Connexion

```
const startMessage = 'Connexion au serveur...';
this.waitBeforeChangeStatus(ONE_SECOND_DELAY, startMessage);
this.clientSocket.socket.connect();
```

(Envoyer/ recevoir) :

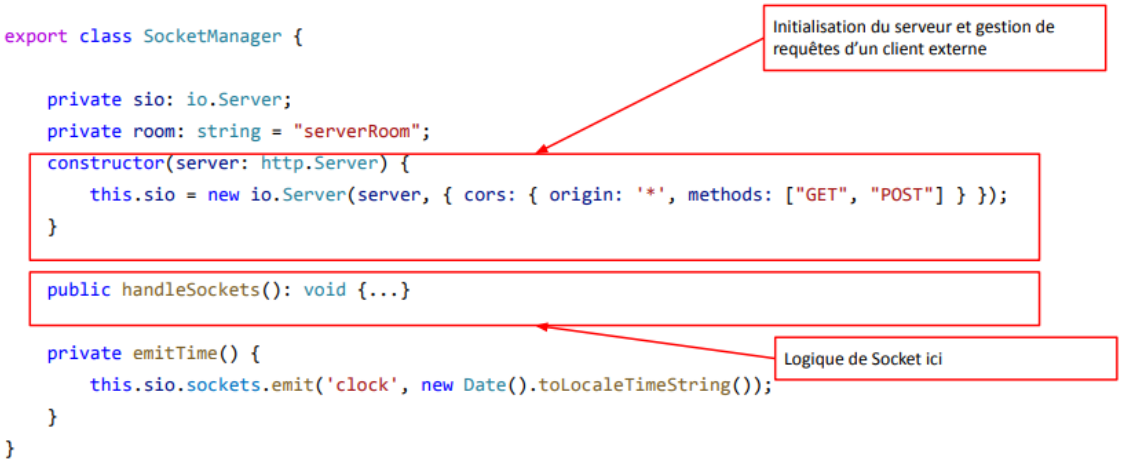
```
this.clientSocket.socket.emit('createRoom', this.gameSettingsService.gameSettings, this.clientSocket.gameType);
```

```
receiveRandomPlacement(): void {
    this.clientSocketService.socket.on('receiveCustomerOfRandomPlacement', (customerName: string, roomId: string) =>
        this.clientSocketService.socket.emit('newRoomCustomer', customerName, roomId);
    });
}
```

- Socket: **Socket**; socket = io(serverUrl);
  - socket.connect() appelé au moment où on veut créer les connexions socket-serveur, dans notre cas dans waitingRoomComponent; émet l'évènement « connection » (mot clé) sur le serveur

Depuis le serveur :

Configuration du serveur




```
export class SocketManager {

    private sio: io.Server;
    private room: string = "serverRoom";
    constructor(server: http.Server) {
        this.sio = new io.Server(server, { cors: { origin: '*', methods: ["GET", "POST"] } });
    }

    public handleSockets(): void {...}

    private emitTime() {
        this.sio.sockets.emit('clock', new Date().toLocaleTimeString());
    }
}
```

Quand le client se connecte au socket

```
handleSockets(): void {
    this.sio.on('connection', (socket) => {
```

Recevoir des évènements :

```
    socket.on('createRoom', (gameSettings: GameSettings, gameType: GameType) => {
        // Traitement
    });
```

Envoyer des évènements :

Broadcast -> A tous les clients connectés

```
    this.sio.emit('roomConfiguration', this.roomManagerService.rooms);
```

Code qui permet d'envoyer la situation des rooms à tous les clients

Broadcast -> A tous les clients connectés sauf l'émetteur

```
socket.on("hello", (arg) => {
    socket.broadcast.emit("hello", arg);
});
```

Envoi à tous les clients de la room sauf l'émetteur

```
socket.to(roomId).emit('yourGameSettings', this.roomManagerService.getGameSettings(roomId));
```

Code qui permet d'envoyer à ton adversaire ses paramètres de jeu complémentaires.

Envoi à tous les clients d'une room

```
this.sio.in(roomId).emit('goToGameView');
```

Code qui permet d'envoyer tous les clients (2) d'une room à la vue de jeu

- Socket.join(roomId); créée la room puis joint si elle n'existe pas encore; ou joint la room si elle existe déjà.
- Socket.leave(roomId) : enlève de la room (on peut être dans plusieurs room à la fois).
- this.sio.socketsLeave(roomId) : vide la room;
- NB : attributs d'un socket : id et port; roomId : string;

Tests :

Avantages des tests unitaires:

- Permettent de tester les unites dans le code
- Permettent de détecter et d'éviter les régréssions : on sait ou se trouve les problèmes quand les tests fail
- Réduction du couplage fort
- Coverture complète du code testé

Conception des tests

- Créer et initialiser tous les objets et variables nécessaires

- Créer seulement ce qu'on a vraiment besoin pour le test
- Remplacer le code hors de votre contrôle par des stub ou des mock
- Effectuer les manipulations requises sur ces objets ou variables
- Appeler que les manipulations nécessaires
- Effectuer les validations
- Limiter le nombre de validations par test
- S'assurer qu'on vérifie les bonnes choses

La stratégie des test utilisent les concepts de spy, mock et stub

Spy : Wrapper autour d'un object optionné (on peut vérifier l'exécution, modifier les params de fonctions …)

Stub : Remplacer le code par une version plus simple

Mock : Comme les stub mais rajoute des vérifications de comportements

Coté client

- Jasmine : librairie de test utilisée par défaut dans les projets Angular
- Karma : test runner qui permet de rouler les tests écrits avec Jasmineà

Sur jasmine, les spy et les mocks sont réunis sous le SpyObj. On peut spy sur des méthodes spécifiques de l'objet, changer les valeurs de retours(.and. retun…), vérifier l'appel(expect.toHaveBeenCalled) , appeler la vraie fonction (callThrough), appeler une fausse fonction (callFake) …

```
it('should on at event of receivePlayedWords', async () => {
    const easelSize = false;
    const isRow = true;
    service['newPlayedWords'].set('mAison', ['H8', 'H9', 'H10', 'H11', 'H12', 'H13']);
    const expectedResult: ScoreValidation = { validation: true, score: 7 };
    spyOn(service['httpServer'], 'validationPost').and.returnValue(of(true));
    const spyClear = spyOn(service['newPlayedWords'], 'clear');
    const validation = await service.validateAllWordsOnBoard(scrabbleBoard, easelSize, isRow);
    expect(validation.score).toEqual(expectedResult.score);
    expect(spyClear).toHaveBeenCalled();
    expect(validation.validation).toEqual(expectedResult.validation);
});
```

```
it('should emit an event if decision if false ', () => {
    const matDialogRefMock = jasmine.createSpyObj('MatDialogRef', ['afterClosed']);
    matDialogRefMock.afterClosed.and.callFake(() => {
        return of(true);
    });
    const matDialogMock = jasmine.createSpyObj('MatDialog', ['open']);
    matDialogMock.open.and.callFake(() => {
        return matDialogRefMock;
    });
    component.dialog = matDialogMock;
    const spyEmit = spyOn(component['clientSocketService'].socket, 'emit');
    const spyMessage = spyOn(component.sendMessageService, 'sendConversionMessage');
    component.confirmGiveUpGame();
    expect(spyMessage).toHaveBeenCalled();
    expect(spyEmit).toHaveBeenCalled();
});      Anthony Depachtere, a month ago • Enhance game UIX
```

Coté Server

- Mocha: librairie de test similaire à Jasmine. Utilise son propre runner, mais a
- besoin d'autres librairies pour certaines fonctionnalités
- Chai : librairie d'assertion qui rapproche Mocha à Jasmine
- Sinon : librairie de mock/spy
- Supertest : librairie de test pour les appels http

Dans le serveur l'utilisation des spy, stub, mock varie selon la librairie de test utilisé

Chai est généralement utilisé pour faire de l'assertion(chai.expect) mais on peut utiliser aussi les spy et ils fonctionnent un peu comme ceux de jasmine

```
it('should return the aiPlayers asked', (done) => {
    databaseService.start();
    const aiModel = AI_MODELS.get(AiType.beginner) as mongoose.Model<AiPlayer>;

    const spy = chai.spy.on(aiModel, 'find', () => {      MikeFolepe, 2 weeks a
        const player = new aiModel({
            aiName: 'Mike',
            isDefault: true,
        });
        player.save();

        // eslint-disable-next-line no-underscore-dangle
        return aiModel.findById(player._id);
    });
    adminService.getAllAiPlayers(AiType.beginner).then(() => {
        chai.expect(spy).to.have.been.called();
        chai.spy.restore(aiModel);
        done();
    });
});
```

Chai.spy.on ici modifie le comportement de la methode find

Il faut restore les spy dans le server si non on ne peut plus les redéfinir par la suite et il y' a une erreur

Sinon lui permet d'utilser les spy, mock et stub avec plus de versatilité

Les spies de sinon permettent de suivre le comportement d'une fonction (sinon.spy(---)) par exemple si elle a été appelé etc..

```
const spy = sinon.spy(adminService, 'getAllAiPlayers');      Paul M

const result = await adminService.updateAiPlayer(players[0]._id, {
    aiBeginner: beginner,
    aiType: AiType.beginner,
});
expect(result[0].aiName).to.equal(beginner.aiName);
expect(spy.calledWith(AiType.beginner)).to.equal(true);
spy.restore();
```

Stub de sinon :

Contrôler le comportement d'une méthode à partir d'un test pour forcer le code à suivre un chemin spécifique. Les exemples incluent le fait de forcer une méthode à générer une erreur afin de tester la gestion des erreurs.

Lorsque vous souhaitez empêcher l'appel direct d'une méthode spécifique (peut-être parce qu'elle déclenche un comportement indésirable, tel qu'un XMLHttpRequest ou similaire).

```
beforeEach(() => {
    const app = Container.get(Application);
    administratorService = Container.get<AdministratorService>(AdministratorService);
    expressApp = app.app;
});

it('should return the beginner Ais', (done) => {
    const stubOnGet = sinon.stub(administratorService, 'getAllAiPlayers').returns(Promise.resolve(aiPlayers));
    chai.request(expressApp)
        .get('/api/admin/aiBeginners')
        .end((err, response) => {
            expect(stubOnGet.called).to.equal(true);
            expect(response.status).to.equal(StatusCodes.OK);
            expect(response.body).to.deep.equal(aiPlayers);
            stubOnGet.restore();
            done();
        });
});
```

Ici par exemple on stub sur la methode getAiPlayers de la dépendance adminService pour éviter d'éxecuter le réel code

Façon de créer un stub dès le début :

```
let roomManagerService: SinonStubbedInstance<RoomManagerService>;
```

```
roomManagerService = createStubInstance(RoomManagerService);
```

Base de données

SQL : base de données relationnelles.

NoSQL : non relationnelles; tout mélangé; fonctionne en document et en collection.

Cas de MongoDB : C'est une BD NoSQL;

MongoDB utilise le format BSON qui est un JSON en binaire

Un document est juste une donnée : possède un id généré par mongo.

Collection : ensemble de documents.

• L'attribut _id est réservé pour définir la clé primaire d'un document • Si ce n'est pas fourni, MongoDB en génère un automatiquement • Toujours le premier attribut d'un document et forcément unique

Mongoose (dans notre cas) /MongoDBClient. : import à faire pour utiliser MongoDB. « mongoose » peut-être utilisé directement pour manipuler la bd ou le mettre dans un attribut comme dans notre cas (this.database).

Code de la page Admin :

```
export class AdministratorService {
    private newAiPlayer: AiPlayerDB;
    private dictionaries: Dictionary[];

    async getAllAiPlayers(aiType: AiType): Promise<AiPlayerDB[]> {
        const aiModel = AI_MODELS.get(aiType) as mongoose.Model<AiPlayer>;
        const aiPlayers = await aiModel.find({}).exec();
        return aiPlayers;
    }

    async addAiPlayer(aiPlayer: AiPlayer, aiType: AiType): Promise<AiPlayerDB> {
        const aiModel = AI_MODELS.get(aiType) as mongoose.Model<AiPlayer>;
        const aiToSave = new aiModel({
            aiName: aiPlayer.aiName,
            isDefault: aiPlayer.isDefault,
        });
        await aiToSave.save().then((ai: AiPlayerDB) => {
            this.newAiPlayer = ai;
        });
        return this.newAiPlayer;
    }

    async deleteAiPlayer(id: string, aiType: AiType): Promise<AiPlayerDB[]> {
        const aiModel = AI_MODELS.get(aiType) as mongoose.Model<AiPlayer>;
        await aiModel.findByIdAndDelete(id).exec();
        return await this.getAllAiPlayers(aiType);
    }

    async updateAiPlayer(id: string, object: { aiBeginner: AiPlayer; aiType: AiType }): Promise<AiPlayerDB[]> {
        const aiModel = AI_MODELS.get(object.aiType) as mongoose.Model<AiPlayer>;
        await aiModel.findByIdAndUpdate(id, { aiName: object.aiBeginner.aiName }).exec();
        return await this.getAllAiPlayers(object.aiType);
    }

    getDictionaries(): Dictionary[] {
        this.dictionaries = [];
        const files = fileSystem.readdirSync('./dictionaries/', 'utf8');
        for (const file of files) {
            const readFile = JSON.parse(fileSystem.readFileSync(`./dictionaries/${file}`, 'utf8'));
            const isDefault = file === 'dictionary.json';
            const dictionary: Dictionary = {
                fileName: file,
                title: readFile.title,
                description: readFile.description,
                isDefault,
            };
            this.dictionaries.push(dictionary);
        }
        return this.dictionaries;
    }

    updateDictionary(dictionary: Dictionary): Dictionary[] {
        const readFile = JSON.parse(fileSystem.readFileSync(`./dictionaries/${dictionary.fileName}`, 'utf8'));
        readFile.title = dictionary.title;
        readFile.description = dictionary.description;
        fileSystem.writeFileSync(`./dictionaries/${dictionary.fileName}`, JSON.stringify(readFile), 'utf8');
        return this.getDictionaries();
    }

    deleteDictionary(fileName: string): Dictionary[] {
        fileSystem.unlinkSync(`./dictionaries/${fileName}`);
```

Syntaxe des opérateurs

- Opérateurs de modification :
  - **$set** : change un document ou un champ d'un document
  - **$inc** : incrémente la valeur du champ par le paramètre fourni
  - **$mul** : multiplie la valeur du champ par le paramètre fourni
  - **$rename** : change le nom d'un champ spécifique
  - **$unset** : retire le champ spécifié du document

- Opérateurs logiques :
  - **$eq** : effectue une comparaison d'égalité
  - **$gt** : effectue une comparaison de type strictement plus grand
    - **$gte** : effectue une comparaison de type plus grand ou égal
  - **$lt** : effectue une comparaison de type strictement plus petit
    - **$lte** : effectue une comparaison de type plus petit ou égal
  - **$in** : effectue une comparaison avec les éléments fournis dans un tableau.
    - Ex : find(sigle :{$in: ["LOG8480","LOG4420"] }) retourne les cours avec le sigle LOG8480 ou LOG4420.
  - **$nin** : l'inverse de **$in** : retourne les documents qui ne correspondent pas aux critères.

Requêtes : modification de données

- Les méthodes **insertOne()** et **insertMany()** ajoutent des documents d'une collection.
  - Le premier argument est le ou les documents à ajouter
  - Si **_id** n'est pas défini dans le document, MongoDB en génère un.
- Les méthodes **updateOne()** et **updateMany()** modifient des documents d'une collection.
  - L'option **upsert** permet d'ajouter le document si aucun n'est trouvé.
- Les méthodes **deleteOne()** et **deleteMany()** suppriment des documents d'une collection.
  - Prend en paramètre le critère identifiant le(s) document(s) à retirer (similaire à find)
  - La méthode **findOneAndDelete()** retourne le document supprimé après la suppression

```javascript
db.restaurants.count({"grades.0.grade":"C"})
db.restaurants.find({"grades.0.grade":"C"}).count()
```

```javascript
db.restaurants.save({"_id" : 1, "test" : 1});
```

Avant de commencer, il faut voir à quoi ressemble un document de notre collection *restaurants*. Utilisons la fonction "*findOne()*".

```javascript
db.restaurants.findOne()
```

Update

Pour commencer, nous allons ajouter un commentaire sur un restaurant (opération **$set**) :

```javascript
db.restaurants.update (
   {"_id" : ObjectId("594b9172c96c61e672dcd689")},
   {$set : {"comment" : "My new comment"}}
);
```

Filtrage

Commençons par un type simple de requête : le filtrage. Nous allons pour cela utiliser la fonction "find" à appliquer directement à la collection. L'exemple ci-dessous récupère tous les restaurants dans le quartier (borough) de Brooklyn.

```javascript
db.restaurants.find( { "borough" : "Brooklyn" } )
```

6 085 restaurants sont retournés.

Pour compter, il suffit d'ajouter la fonction "count"

```javascript
db.restaurants.find( { "borough" : "Brooklyn" } ).count()
```

En effet, nous sommes en *Javascript* : le résultat d'un *find* est une liste dont on peut compter les documents avec la méthode de liste "count"

Maintenant, nous cherchons parmi ces restaurants ceux qui font de la cuisine italienne. Pour combiner deux "clés/valeurs", il suffit de faire le document motif donnant quels paires clés/valeurs sont recherchés :

```javascript
db.restaurants.find(
    { "borough" : "Brooklyn",
      "cuisine" : "Italian" }
)
```

Ensuite, on va chercher les restaurants présents sur la 5° Avenue. La clé "street" est imbriquée dans l'adresse ; comment filtrer une clé imbriquée (obtenue après fusion) ? Il faut utiliser les deux clés, en utilisant un point "." comme si c'était un objet.

```javascript
db.restaurants.find(
    { "borough" : "Brooklyn",
      "cuisine" : "Italian",
      "address.street" : "5 Avenue" }
)
```

Pourquoi ne pas chercher le mot "pizza" dans le nom du restaurant ? Pour cela, il faut utiliser les expressions régulières avec "/xxx/i" (le i pour "Insensible à la casse" - majuscule/minuscule).

```javascript
db.restaurants.find(
    { "borough" : "Brooklyn",
      "cuisine" : "Italian",
      "address.street" : "5 Avenue",
      "name" : /pizza/i }
)
```

Il ne reste maintenant que 2 restaurants Italiens dans le quartier de Brooklyn dans la 5° Avenue, dont le nom contient le mot "pizza".

Projection

Et si nous ne gardions dans le résultat que le nom ? C'est ce que l'on appelle une projection. Un deuxième paramètre (optionnel) de la fonction *find* permet de choisir les clés à retourner dans le résultat. Pour cela, il faut mettre la clé, et la valeur "1" pour projeter la valeur (on reste dans le concept "clé/valeur").

```javascript
db.getCollection('restaurants').find(
    {"borough":"Brooklyn",
     "cuisine":"Italian",
     "name":/pizza/i,
     "address.street" : "5 Avenue"},
    {"name":1}
)

{"_id" : ObjectId("594b9173c96c61e672dd074b"), "name" : "Joe'S Pizza"}
{"_id" : ObjectId("594b9173c96c61e672dd1c16"), "name" : "Gina'S Pizzaeria/
Deli"}
```

L'identifiant du document "_id" est automatiquement projeté, si vous souhaitez le faire disparaître il suffit de rajouter la valeur "0" : {"name":1, "_id":0}

Et si nous regardions les résultats des inspections des commissions d'hygiène ? Il faut pour cela regarder la valeur de la clé "*grades.score*".

```javascript
db.getCollection('restaurants').find(
    {"borough":"Brooklyn",
     "cuisine":"Italian",
     "name":/pizza/i,
     "address.street" : "5 Avenue"},
    {"name" : 1,
     "grades.score" : 1}
)
```

```javascript
db.getCollection('restaurants').find(
    {"borough":"Manhattan",
     "grades.score":{$lt : 10}
    },
    {"name":1,"grades.score":1, "_id":0})
```

Déploiement

But :

Déployer en général sert à héberger notre projet (client+serveur+base de données) dans un environnement de production pour qu'il soit accessible et utilisable par tous sans plus avoir besoin d'exécuter du code local. (Généralement par le client pour qui on a developpé tout le produit).

Dans notre cas :

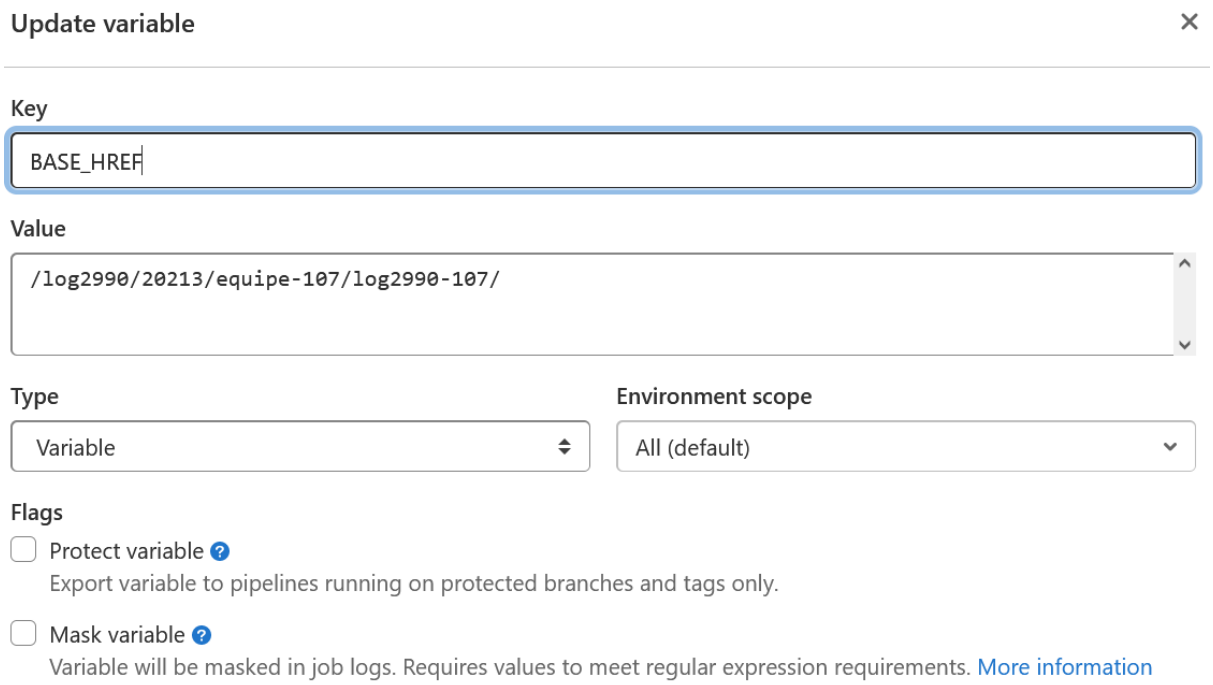
- L'environnement de déploiement du client est gitlabPages.
- Serveur --> amazon AWS
- Base de données --> mongoDB.

Déploiement du client sur gitlabPages :

Mettre à jour cette variable sur gitlab pour le déploiement automatique du client sur gitlab pages.

La valeur sera utilisée pour générer le lien.

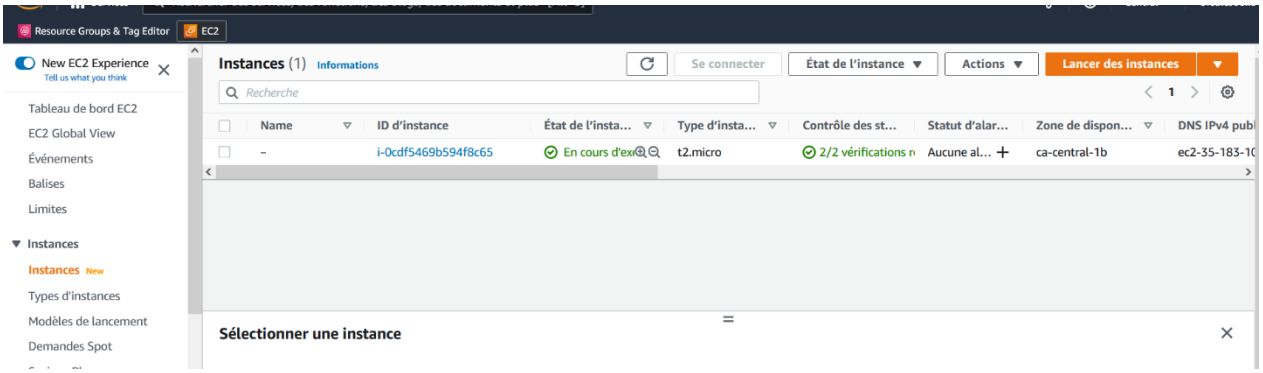
Notre URL : http://polytechnique-montr-al.gitlab.io/log2990/20213/equipe-107/log2990-107/



Serveur --> amazon AWS

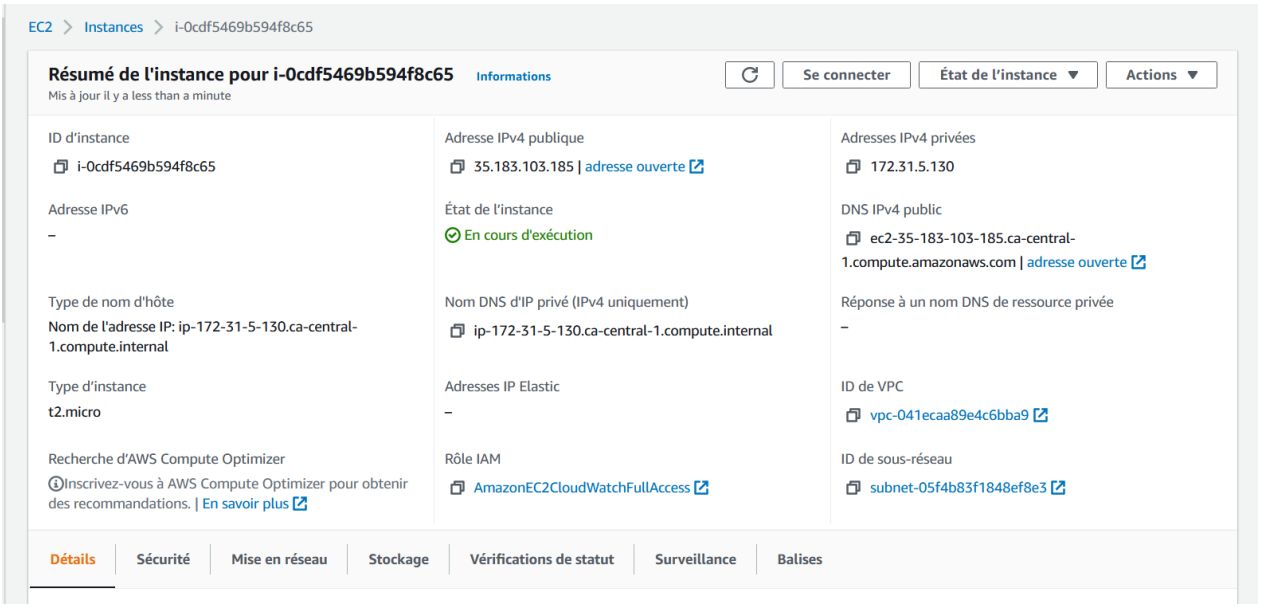
Machine d'hébergement sur AWS

- Liste des instances (machines). Dans notre cas on en a créé qu'une seule.



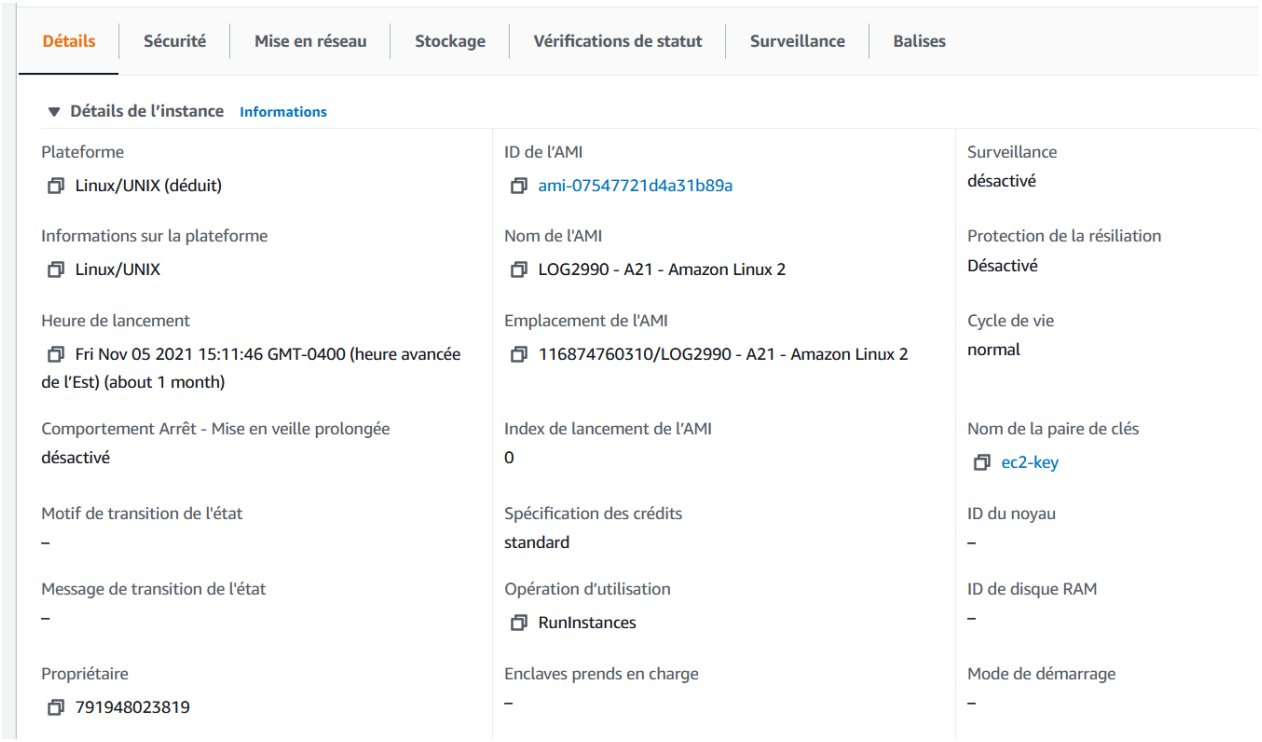
➢ Caractéristiques de l'instances --> résumées :

Elles ont pour la plupart été configurées lors de la création de la machine, selon les standards donnés par les chargés dans le fichier deploiement.txt. Sauf les adresses IP privées et publics qui est elles donc générées par AWS chaque fois qu'on redémarre la machine.

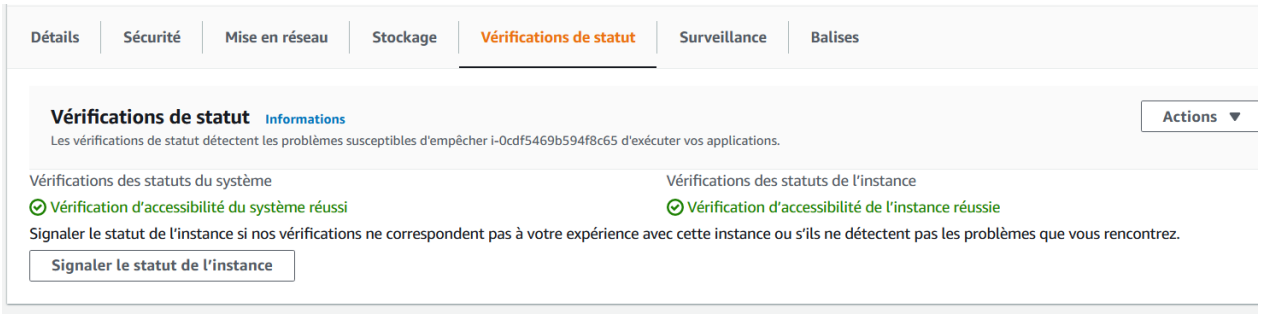


les onglets au bas de l'image ci-dessus donnent chacun des caractéristiques plus détaillées. (adresse IP, rôle IAM, stockage, nombre de coeurs CPU, l'AMI ...). elles sont pas très importantes; cest juste des configurations; ci-dessous uniquement la capture pour le premier onglet, "details";

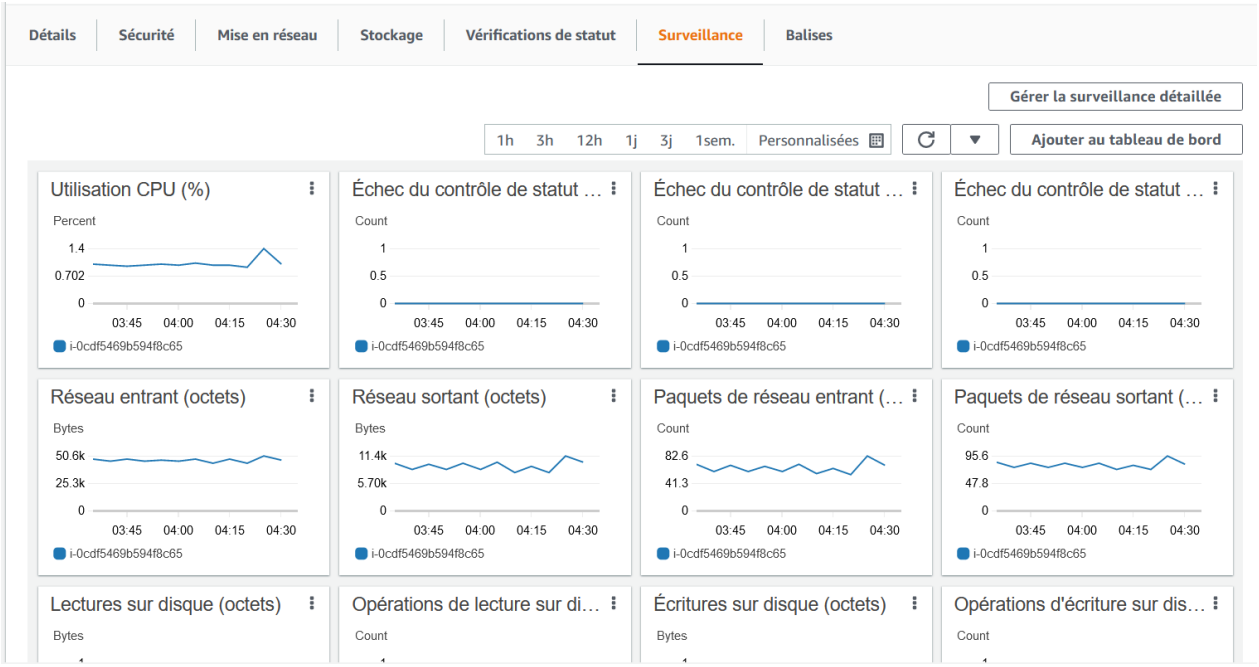
L'AMI aussi "LOG2990 - A21 - Amazon linux 2" a été spécifié par les chargés dans la documentation. (je ne connais pas exactement son rôle). je peux checker rapidement.



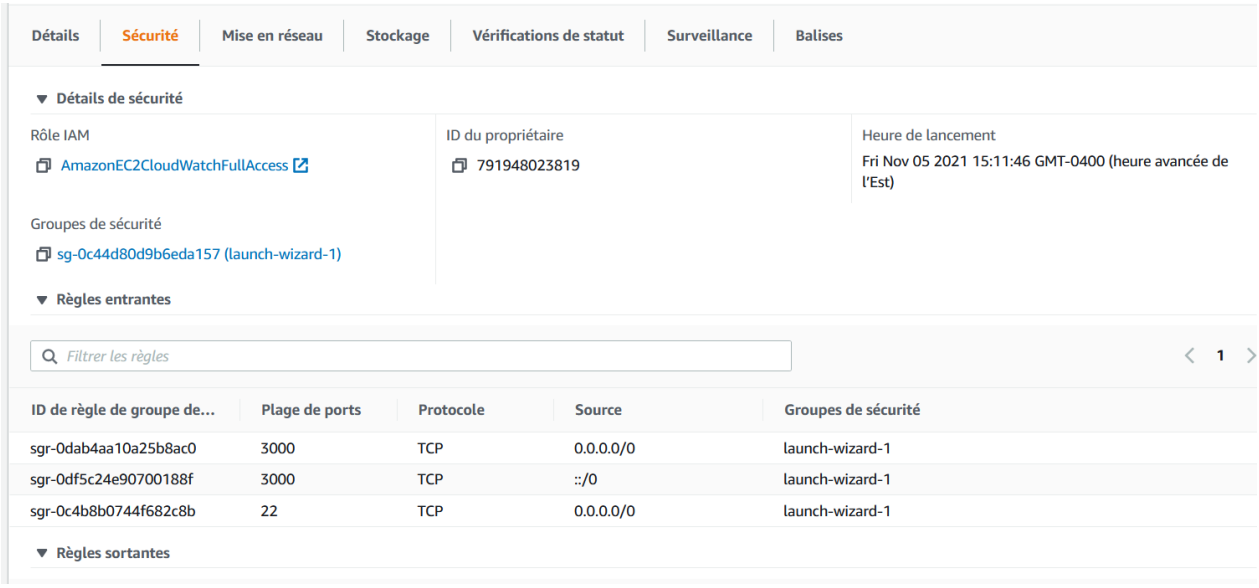
➢ "Vérification de statut" indique juste l'accessibilité de l'instance.



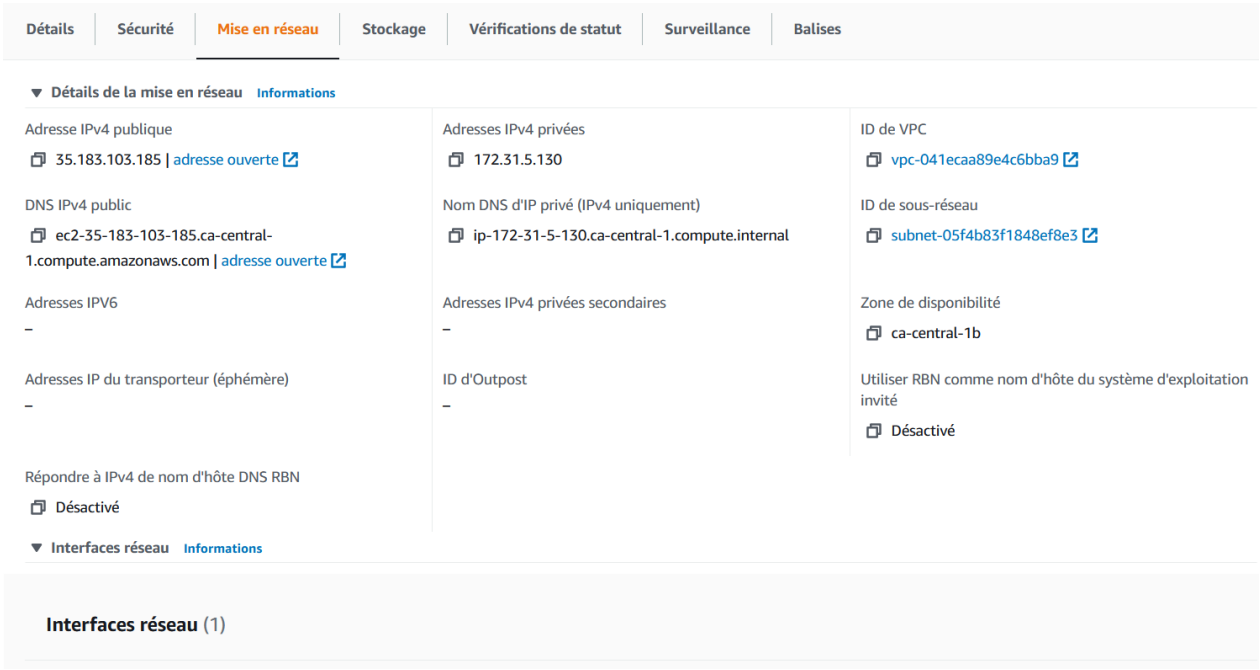
➢ Surveillance : Indique les métriques d'utilisation de la machine.



➢ Sécurité: remarquer qu'elle indique les ports



➢ Mise en réseau :



Déploiement automatique du serveur sur notre machine AWS :

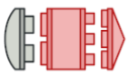
Type	↑ Key	Value	Protected	Masked	Environments
Variable	BASE_HREF	**********	✗	✗	All (default)
Variable	EC2_HOST	**********	✗	✗	All (default)
Variable	EC2_PEM_FILE_CONTENT	**********	✗	✗	All (default)
Variable	EC2_USER	**********	✗	✗	All (default)
Variable	GITLAB_DEPLOY_TOKEN_...	**********	✗	✓	All (default)
Variable	GITLAB_DEPLOY_TOKEN_...	**********	✗	✗	All (default)
Variable	GITLAB_REPO_URL	**********	✗	✓	All (default)
Variable	SERVER_PORT	**********	✗	✗	All (default)

Add variable | Reveal values

NB : De EC2 HOST à SERVER_PORT uniquement.

Patrons structurels

Les patrons structurels vous guident pour assembler des objets et des classes en de plus grandes structures tout en gardant celles-ci flexibles et efficaces.



Adaptateur
Adapter

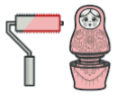
Permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

Pont
Bridge

Permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies — abstraction et implémentation — qui peuvent évoluer indépendamment l'une de l'autre.

Composite
Composite

Permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.

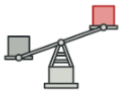


Décorateur
Decorator

Permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.

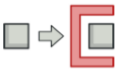
Façade
Facade

Procure une interface qui offre un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.



Poids mouche
Flyweight

Permet de stocker plus d'objets dans la RAM en partageant les états similaires entre de multiples objets, plutôt que de stocker les données dans chaque objet.

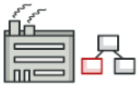


Procuration
Proxy

Permet de fournir un substitut d'un objet. Une procuration donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.

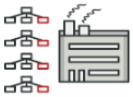
Patrons de création

Les patrons de création fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code.



Fabrique
Factory Method

Définit une interface pour la création d'objets dans une classe mère, mais délègue aux sous-classes le choix des types d'objets à créer.

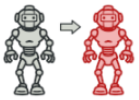


Fabrique abstraite
Abstract Factory

Permet de créer des familles d'objets apparentés sans préciser leur classe concrète.

Monteur
Builder

Permet de construire des objets complexes étape par étape. Ce patron permet de construire différentes variations ou représentations d'un objet en utilisant le même code de construction.



Prototype
Prototype

Permet de créer de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.

Singleton
Singleton

Permet de garantir que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

Patrons comportementaux

Les patrons comportementaux s'occupent des algorithmes et de la répartition des responsabilités entre les objets.

Chaîne de responsabilité

Chain of Responsibility

Permet de faire circuler une demande dans une chaîne de handlers. Lorsqu'un handler reçoit une demande, il décide de la traiter ou de l'envoyer au handler suivant de la chaîne.

Commande

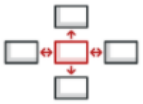
Command

Prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.

Itérateur

Iterator

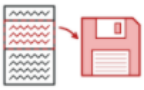
Permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, etc.).



Médiateur

Mediator

Permet de diminuer les dépendances chaotiques entre les objets. Ce patron restreint les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.



Mémento

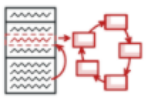
Memento

Permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation.

Observateur

Observer

Permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.



État

State

Modifie le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

Stratégie

Strategy

Permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

Patron de méthode

Template Method

Permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.

Visiteur

Visitor

Permet de séparer les algorithmes et les objets sur lesquels ils opèrent.