Document aide-mémoire pour examen de projet INF2990

Section 1 : MongoDB

Toutes les commandes MongoDB standard (non-Mongoose) utilisent la syntaxe **db.<collection>.<commande>**.

Les commandes de base (et les plus utiles) sont les suivantes :

Commande	Ce que ça fait	Exemple	Résultat
db.collection.find(<critères>, <projection>)	Retourne tous les documents correspondant aux critères de recherche. Utilisation la plus courante de MongoDB. Cette commande est détaillée extensivement plus bas.	db.joueurs.find({ $and : { sexe : "feminin" }, { score : { $gt : 100 } } }) --- équivalent à --- db.joueurs.find({ sexe : "feminin", score : { $gt : 100 } })	Retourne tous les joueurs féminins ayant un score supérieur à 100. Ici, le $and n'est pas nécessaire.
db.collection.findOne(<critères>)	Retourne le premier document (dans l'ordre de la base de données) correspondant aux critères de recherche.	db.osu.findOne({ nom : "Fudgyking", pays : "Canada"})	Retourne le premier joueur canadien de Osu enregistré nommé "Fudgyking".
db.collection.insert(<document>)	Insère un ou plusieurs nouveau(x) document(s) dans la collection. Pour en mettre plusieurs, on fait insert({...}, {...}, {...}, ...)	db.waifus.insert({ nom : "Amada Kokoro", type : "2D", affiliation : "AyyMD" })	Insère la waifu AyyMD 2D Amada Kokoro dans la collection de waifus.
db.collection.insertOne(<document>) / insertMany([<document>, <document>, ...])	Variante de insert() qui spécifie le nombre ajouté.	db.jeux.insertMany([{ nom : "Rocket League", genre : "Multiplayer" }, { nom : "Elite Dangerous", genre : "Spacesim" }])	Insère les jeux Rocket League et Elite Dangerous dans la collection de jeux. On remarque que les documents sont dans un array, encadré par **[...]**.
db.collection.update(<critères>, <document>)	Modifie un ou plusieurs documents correspondants aux critères de recherche dans la collection. On doit spécifier { multi : true } si on veut mettre à jour plusieurs documents à la fois (sinon équivaut à updateOne).	db.cpus.update({ $and : { marque : "Intel" }, { prix : { $gt : 500 } } }, { $set : { socket : "LGA2011" } }, { multi : true })	Trouve tous les CPUs Intel coûtant plus de 500$ et met leur socket à LGA2011, peu importe ce qu'il était auparavant (s'il n'existait pas, il est ajouté).
db.collection.updateOne(<critères>, <document>) / updateMany(<critères>, <document>)	Même principe que update, mais updateOne → multi : false et updateMany → multi : true	Voir ci-haut	Met à jour un nombre spécifique de documents correspondant aux critères.
db.collection.save(<document>)	Si un document avec le *id* spécifié existe déjà dans la collection, équivaut à *update()*. Sinon, équivaut à	db.cpus.save({ _id : "i7K", marque : "Intel", gamme : "Core", modele : "i7-7700K",	Puisque le document avec l'*id* "i7K" existait déjà dans la base de

	insert(). Si aucun *id* n'est spécifié, équivaut toujours à *insert()*.	prix : 470, socket : ''LGA1151'' })	données (avec le modèle ''i7-6700K''), il est remplacé par ce nouveau document.
db.collection.remove(<critères>)	Retire de la collection tous les documents correspondant aux critères spécifiés. Non, la méthode *delete()* n'existe pas!!	db.gpus.remove({ marque : ''AMD'', serie : ''300 series''})	Retire de la collection toutes les cartes graphiques série 300 de AMD.
db.collection.deleteOne(<critères>) / deleteMany(<critères>)	Même principe que updateOne/updateMany, mais agit comme *remove*().	db.curlingScores.deleteOne({ nom : ''1337H4X0R'' })	Retire le premier score du joueur ''1337H4X0R'', mais pas tous ses scores nécessairement.
db.collection.drop()	Supprime complètement la collection et son contenu (comme un DROP TABLE de SQL).	db.mtnDew.drop()	La production de Mtn Dew est arrêtée et toutes les sortes arrêtent d'exister. RIP Runo.
db.collection.count(<critères>)	Compte le nombre de documents correspondant aux critères de recherche, sans retourner ceux-ci; seulement leur nombre.	db.memes.count({ niveau : ''dank'' })	Retourne le nombre de dank memes en existence.
db.collection.aggregate(<opérations>)	Effectue une requête complexe comprenant des opérations spécifiques.	db.voitures.aggregate([{ $match : { marque : "Ferrari" } } , { $group : { _id: "$proprietaire", total: { $sum: "$cout" } } }, { $sort : { total : -1 } }])	Retourne la valeur totale des Ferraris de chaque propriétaire. On match d'abord la marque (Ferrari), puis on groupe par propriétaire (on le spécifie comme _id du groupement) et la valeur totale ($sum) de ses Ferraris. On trie ensuite la sortie en ordre décroissant (-1).
db.collection.distinct(<champ>, <critères?>)	Retourne toutes les valeurs distinctes du champ spécifié. On peut également spécifier certains critères pour filtrer la requête (comme un find), mais c'est optionnel.	db.gpus.distinct(''marque'')	Retourne [''AMD'', ''Intel'', ''Nvidia''], soit les trois valeurs distinctes du champ ''marque'' parmi tous les GPUs de la collection.

La commande find() en détails :

find() est de loin la commande la plus utilisée en MongoDB. Elle sert à faire une requête pour certains documents d'une collection à partir de notre base de données. On peut lui spécifier des critères de recherche simples (le champ suivi de sa valeur), mais on

peut aussi ajouter des modificateurs qui équivalent un peu à des WHERE de SQL. On peut aussi projeter les résultats de nos requêtes pour ne garder que certains champs dans les résultats (quoique ce ne soit pas nécessairement utile dans notre cas, à moins de vouloir obtenir des JSON « propres »). Pour faire cela, on spécifie un second argument { … } dans la requête, avec des 0 **ou** des 1 pour spécifier les champs que l'on veut garder (ou pas) dans notre sortie. La projection ne prend qu'un seul type d'argument (on décide lesquels inclure (1), **OU** lesquels exclure (0))!! La seule exception à cela est le champ **_id**, qui est toujours inclus par défaut mais peut être exclus avec des inclusions. Par exemple :

```
db.examen.find( { question : "MongoDB" }, { _id : 0, question : 1, enonce : 1, points : 1 } );
```

Ici, je veux obtenir toutes les questions (documents) portant sur MongoDB. Je veux avoir dans ma sortie seulement le type de la question, son énoncé et les points qu'elle vaut. J'exclus le champ _id car il ne m'intéresse pas (et c'est la seule exception à la règle pour mixer des 1 et des 0 !).

La méthode *find()*, tout comme toutes les autres se servant de critères de recherche, prend de nombreux opérateurs. Les opérateurs sont dénotés par le symbole **$**. En voici quelques-uns des plus utiles :

Opérateurs	Ce que ça fait	Exemple	Résultat
$eq, $gt, $gte, $lt, $lte, $ne	Opérateurs mathématiques (=, >, ≥, <, ≤, ≠)	`db.joueurs.find( { score : { $gt : 100 } } )`	Retourne tous les joueurs ayant un score supérieur à 100.
$in, $nin	Dans ou pas dans (pour des arrays)	`db.joueurs.find( { nom : { $in : [ "fireyoshiqc", "fudgyking" ] } } )`	Retourne les joueurs ayant pour nom l'un des éléments de l'array de noms spécifiés.
$or, $and, $not, $nor	Opérateurs booléens (ou, et, non, non-ou)	`db.joueurs.find( { $and : { score : { $lt : 420 } } , { score : { $gt : 100 } } } )`	Retourne tous les joueurs ayant un score supérieur à 100 mais inférieur à 420.
$exists	Détermine si le champ existe ou non	`db.joueurs.find( { tryhard : { $exists : true } } )`	Retourne tous les joueurs où le champ *tryhard* est défini (non-null).
$all, $elemMatch(<critères>), $size	Opérateurs d'array (contient toutes les valeurs, correspond à un ou plusieurs critères, est d'une certaine taille)	`db.joueurs.find( { bonus : { $all : [ "hardrock", "doublespeed" ] } } )`	Trouve tous les joueurs ayant joué avec au moins les bonus *hardrock* ET *doublespeed*.
$set, $inc, $dec, $unset	Opérateurs d'update (met un champ à une valeur ou ajoute le champ, incrémente/décrémente un champ, retire un champ)	`db.cpus.update( { modele : "i7-6700K" }, { $inc : { frequence : 0.2 }, $set : { modele : "i7-7700K" } } )`	Trouve le CPU de modèle i7-6700K, incrémente sa fréquence de 0.2 GHz et le renomme i7-7700K.

Finalement, une fois le résultat d'un find obtenu, on peut lui appliquer de nombreuses méthodes pour ordonner et effectuer des opérations sur nos résultats. Puisqu'il s'agit toujours d'un tableau de JSON, on peut appliquer les méthodes standard sur les arrays. Les plus utiles, qui parlent d'elles-mêmes, sont **forEach(), map(), hasNext() et next()** (si on veut utiliser des itérateurs plutôt que des méthodes fonctionnelles). On a aussi des méthodes utilitaires, comme **count(), max(), min()** et **size()**. D'autres méthodes utiles sont **toArray()**, qui convertit le résultat en un array Javascript (pour le sauvegarder par exemple), **pretty()** (qui donne une sortie en JSON bien espacée et ordonnée) et bien sûr **sort()**.

sort() se comporte comme la variante à même la requête, c'est-à-dire qu'on lui spécifie un ou des champs, ainsi que l'ordre de tri. Par exemple, admettons le résultat d'une requête qu'on nomme *res* :

```
res.sort( { age : -1, nom : 1 } );
```

Ici, on va trier notre résultat en ordre décroissant d'âge (-1), mais aussi en ordre alphabétique ascendant (ABC…) de nom!

Section 2 : Socket.io

Socket.io est une librairie permettant de connecter plusieurs clients à un même serveur à l'aide de WebSockets. Il est parfait pour envoyer de l'information en temps réel et en simultané (comme le jeu de Scrabble).

La base de Socket.io est la distinction client/serveur. On a normalement un seul serveur, qui gère toutes les connexions des sockets côté client (socket.io-client). Le serveur est capable d'émettre (*emit*) de l'information à ses clients, ainsi que de réagir à certains signaux provenant des clients (*on*) pour leur donner une réponse ou effectuer un traitement. Il peut également séparer les clients en salles (*rooms*), et envoyer de l'information à seulement ceux-ci ou à un certain *namespace* (c'est-à-dire, seulement les clients se trouvant sur une certaine section du site ou de l'app web, par exemple). Maintenant que ces points sont traités, voici les principales fonctions de Socket.io :

Côté serveur (avec import * as io from 'socket.io')

On doit tout d'abord s'assurer d'avoir créé un serveur http à partir d'Express.

```
let server = http.createServer(application.app);
```

On peut ensuite passer ce serveur à notre *io* pour créer un socket serveur.

```
private sio : SocketIO.Server;
…
this.sio = io.listen(server);
```

Une fois le socket créé, on le fait attendre une connexion à l'aide de la méthode *on*.

```
this.sio.on('connection', (socket) => { … } );
```

À l'intérieur du callback, on ira placer toutes nos méthodes qui voudront se servir du socket client connecté, soit pour recevoir des informations ou en envoyer. On pourra par exemple y mettre :

```
socket.on('cwValidateName', (name: string) => {
    let validity = this.playerManager.validateName(name);
    socket.emit('wcNameValidated', validity);
});
```

Dans cet exemple, le socket serveur va se préparer à réagir à la réception d'un signal nommé *cwValidateName*. Sur réception de ce signal, il va lancer une méthode de callback à l'aide de ce qu'il est supposé recevoir dans le signal, c'est-à-dire un *name* de type *string* (la méthode côté client est montrée plus loin). Le serveur se chargera ensuite de valider le *name* reçu à l'aide de d'autres méthodes, puis émettra une réponse au socket client à l'aide de la méthode *emit*. Ici, il lui envoie simplement un booléen représentant la validité (ou non) du *name*, dans un signal nommé *wcNameValidated*. À son tour, le client devra écouter ce signal afin de savoir quoi faire s'il le reçoit.

Il est également possible d'envoyer un même signal à tous les clients connectés au serveur. Dans ce cas, plutôt que de faire appel au *socket*, on fera appel à *sio*, notre serveur Socket.io :

```
this.sio.sockets.emit('unMessage', msg);
```

La propriété *sockets* du serveur Socket.io nous donne l'ensemble de tous les sockets clients connectés au serveur. On peut également s'en servir pour envoyer des signaux à certaines *rooms*, ou à certains *namespaces* (on y reviendra plus loin). À noter qu'il est également possible d'envoyer un signal à tous les sockets clients excepté celui qui a lancé le signal (c'est utile pour éviter de dupliquer des messages de chat, par exemple). On se servira pour cela de la propriété *broadcast* :

```
socket.broadcast.emit('unMessage', msg);
```

À noter bien sûr que toutes les méthodes se servant de *socket* tout seul doivent être dans le callback mentionné plus haut.

Tel que dit précédemment, Socket.io permet de créer des *rooms*, c'est-à-dire des « salles » où les sockets seront réunis et pourront être traités comme un seul ensemble. C'est très utile pour faire des salles de jeux (*cough* Scrabble *cough*). Pour créer ou joindre une *room*, c'est très simple :

```
socket.join('maRoom');
```

C'est tout! Le serveur Socket.io créera une *room* identifiée par *maRoom* si elle n'existe pas déjà, et dans les deux cas elle y ajoutera le *socket* sur lequel est appelée la méthode *join*. Pour enlever un socket d'une *room*, c'est tout aussi simple :

```
socket.leave('maRoom');
```

Idem au *join*, si le socket se trouve bien dans *maRoom*, il se fera retirer de celle-ci, et si *maRoom* devient vide par la suite, la *room* sera supprimée.
Pour envoyer un signal à une *room* spécifique, on utilise la méthode *in* :

```
this.sio.sockets.in('maRoom').emit('unMessage', msg);
```

Bien sûr, on peut faire autre chose qu'un *emit* suite à un *in*; c'est toutefois l'utilisation la plus courante.
On peut également utiliser un principe similaire pour les *namespaces*, c'est-à-dire envoyer des messages seulement aux clients s'étant connecté eux-mêmes à ce *namespace* (on y reviendra dans la section client). On utilisera simplement la méthode *of* à la place :

```
this.sio.sockets.of('/jeu').emit('unMessage', msg);
```

Par défaut, le *namespace* global est ' / ' (donc quand on ne met pas de *of*, c'est comme si on s'en tenait au *namespace* /). On peut également combiner les *namespaces* et les *rooms* pour obtenir quelque chose comme suit :

```
this.sio.sockets.of('/jeu').in('maRoom').emit('unMessage', msg);
```

Cela enverra le message à tous les clients se trouvant dans *maRoom*, mais plus spécifiquement ceux s'étant connectés explicitement au *namespace /jeu*.
Il faudra bien évidemment gérer la déconnexion des clients, et pour faire cela, on utilise le même principe que pour leur connexion :

```
socket.on('disconnect', () => {
    setTimeout(() => {
        this.disconnectUser(socket);
    }, 4000);
});
```

Ici, toujours en étant à l'intérieur du callback de *on connection*, on écoute le signal *disconnect*. S'il est reçu, on effectue le traitement nécessaire (dans cet exemple, on attend 4 secondes avant de retirer le joueur du tableau informatif du Scrabble et remettre ses lettres dans la réserve à l'aide de la méthode *disconnectUser*, qui n'a rien à voir avec Socket.io). En somme, on aurait donc, par exemple, un serveur contenant ceci :

```
private sio = SocketIO.Server;
…
const application: Application = Application.bootstrap();
…
let server = http.createServer(application.app);

this.sio = io.listen(server);

this.sio.on('connection', (socket) => {

    this.sio.sockets.emit('wcNewPlayer', "Un joueur s'est connecté! Bonjour!");
```

```
    socket.on('cwValidateName', (name: string) => {
        let validity = this.playerManager.validateName(name);
        socket.emit('wcNameValidated', validity);
    });

    socket.on('cwJoinRoom', (roomID: string) => {
        socket.join(roomID);
    });

    socket.on('cwLeaveRoom', (roomID: string) => {
        socket.leave(roomID);
    });

    socket.on('disconnect', () => {
        setTimeout(() => {
            this.disconnectUser(socket);
        }, 4000);
    });
// Le code dont ceci est inspiré ne se servait pas de namespaces.
});
```

Côté client (import * as io from 'socket.io-client')

Il y a moins de choses du côté client. Le client est assez bête, dans le sens qu'il ne devrait que se connecter/déconnecter du serveur et écouter/envoyer des signaux. La procédure pour se connecter au serveur à partir d'un client est assez simple :

```
private client = SocketIO.Socket;
…
this.client = io.connect("http://localhost:3200");
```

L'adresse entrée ici est l'adresse du serveur sur lequel est créé Socket.io côté serveur (le serveur http mentionné plus haut). On peut également se connecter à un *namespace* en ajoutant celui-ci à la fin de l'adresse du serveur :

```
this.client = io.connect("http://localhost:3200/jeu");
```

Il faut faire attention avec les *namespaces*, car si celui spécifié n'existe pas, le client ne sera pas connecté au serveur du tout! Par contre, c'est utile pour permettre au client de joindre lui-même le *namespace* qu'il désire (par exemple en entrant l'adresse dans un *input*).
Le socket client se comporte ensuite exactement comme les sockets côté serveur, c'est-à-dire avec des *on* et des *emit*. Il n'a évidemment pas accès à *in* ou *of* ni aux broadcasts (envoyer des messages à tous les autres sockets) : il ne peut communiquer qu'avec le serveur. C'est le serveur qui devra transmettre les messages d'un client à un autre. Voici par exemple les deux signaux mentionnés dans la section *serveur*, mais du côté *client* :

```
constructor() {
    this.client = io.connect("http://localhost:3200");
    this.client.on('wcNameValidated', (nameValid: boolean) => {
        this.nameValid = nameValid;
    });
}

public validateName(name: string): void {
    this.client.emit('cwValidateName', name);
}
```

On voit ici que le client émet le signal *cwValidateName* lorsque la méthode *validateName* est appelée (à partir de son component parent, par exemple). Ce signal était *reçu* par le serveur *(on)*. Or, il écoute le signal *wcNameValidated*, qui lui était émis *(emit)* par le serveur, et s'attend à recevoir un booléen *nameValid* (voir le code côté serveur pour référence), qu'il va ensuite assigner à une variable de classe.
Finalement, on peut fermer explicitement un socket client en utilisant la méthode *close()* ou *disconnect()* :

```
this.client.close();
```

Cela aura pour effet de lancer explicitement le signal *disconnect*, qui pourra être reçu et traité côté serveur.
C'est pas mal tout pour Socket.io! Il faut simplement se rappeler que les sockets peuvent s'envoyer toutes sortes d'informations, donc pas seulement des *string* et autres types primitifs, mais aussi des interfaces, des JSON, etc. Il faut toutefois s'assurer de les traiter en conséquence de chaque côté (si un booléen est *émis*, un booléen doit être *reçu*, peu importe que ce soit client vers serveur ou serveur vers client).

Section 3 : Les services REST et NodeJS + Express

On parlera d'un API REST ou d'un service RESTful lorsqu'on traite les communications client-serveur via les protocoles http standards, par exemple GET, PUT, POST, DELETE, etc. On regroupera souvent ces pages sous forme d'un sous-ensemble de pages distinct, par exemple *localhost:3000/api/…* Puisqu'on se sert d'Express dans le projet, on voudra se servir de son routeur intégré, très facile à utiliser. On peut également se servir directement de *this.app()*, l'instanciation de base d'Express :

```
public app: express.Application;
…
this.app = express();
```

Une fois l'application Express construite, on voudra configurer notre pile de *middlewares*. Un *middleware* est une façon pour l'application de passer toutes les requêtes http, avant de les router, dans une certaine séquence de traitements. Il se sert de la méthode use(…). La configuration standard de la pile de *middlewares* du projet de Sudoku est la suivante, par exemple :

```
this.app.use(logger('dev'));
this.app.use(bodyParser.json());
this.app.use(bodyParser.urlencoded({ extended: true }));
this.app.use(cookieParser());
this.app.use(cors());
this.app.use(express.static(path.join(__dirname, '../../client')));
this.app.use('/api', rest.default);
```

Les deux lignes les plus importantes de cette pile de middlewares sont les deux dernières. Celle contenant *express.static* est une façon d'indiquer à Express que l'on désire servir de façon statique les fichiers contenus dans le répertoire *client*, c'est-à-dire là où se trouve notre application web. La suivante est un middleware personnalisé, qui constitue en fait la base de tout le routage de l'application. Puisqu'il s'agit d'une méthode use(…), on indique par-là que toutes les requêtes débutant par */api* seront gérées par *rest.default*, qui est en fait un routeur Express défini dans un autre fichier, importé à l'aide de la ligne suivante :

```
import * as rest from './api/rest.api'; // Le fichier se nomme rest.api.ts.
```

Le routeur constitue le gros de l'application Express REST. C'est lui qui s'occupera de gérer les requêtes http en provenance du client de façon appropriée. Toutefois, avant de s'y attarder, on s'intéressera à ce qui vient après (le middleware */api* est un *end*, il ne fait pas de *next*). En effet, chaque middleware de la pile, excepté */api*, fait un appel interne à *next()*, ce qui a pour effet de passer au prochain middleware correspondant. On remarquera que les middlewares jusqu'à *express.static* n'ont pas d'URI comme premier paramètre (contrairement à *'/api'*, *rest.default*), ce qui veut dire qu'ils vont traiter toutes les requêtes.
Si on veut adopter une structure appropriée dans notre application, il faut faire bien attention d'appeler *next()* au bon moment, car les middlewares sont considérés du haut vers le bas : si le premier middleware ne fait pas appel à *next()*, les subséquents ne seront pas appelés. Voici un exemple :

```
this.app.use("/system", (req, res, next) => {
    if (req.body.id === "admin") {
        res.send("Wow, un vrai hacker!");
    } else {
        next();
    }
});
```

```
this.app.use("/system", (req, res, next) => {
    res.send("Aww des chatons!");
});

this.app.use("/system", (req, res, next) => {
    res.send("Git gud mon ptit gô!");
});
```

Ici, on a trois middlewares ayant le même point d'entrée, c'est-à-dire l'URI /system. Le troisième middleware ne sera jamais appelé, peu importe le résultat du premier middleware! En effet, le second ne fait jamais appel à next(), et constitue donc un cul-de-sac pour toutes les requêtes faites sur l'URI /system.

Pour en revenir au contenu général de l'application Express maintenant (on parlera du routeur après), la suite des middlewares est l'établissement des vues (views). Les vues sont une façon d'accéder visuellement à des informations sur le serveur, à partir de pages qui sont générées à la volée à l'aide d'un outil comme *Jade* ou *PUG*. Voici un exemple de vues dans une application Express :

```
private views(): void {
    this.app.set("view engine", "pug");
    this.app.set("views", path.join(__dirname, '../app/api/views'));
}
```
(Oui, ici mon VSCode a décidé de me permettre de copier du code comme faut).

On assigne ici à Express le moteur de vue PUG (*view engine*) à l'aide de la méthode *set*. On lui spécifie également où se trouvent nos pages qui devront être générées (des fichiers .pug). Voici un exemple de fichier PUG, appelé *log.pug* :

```
h1 Tableau de bord du serveur de Sudoku
h3= 'Nombre de sudokus faciles disponibles : ' + easy
h3= 'Nombre de sudokus difficiles disponibles : ' + hard
ul
  each entry in list
    li= entry
  else
    p Il n'y a pas encore d'entrées dans le tableau de bord!
```

Ce code simple permet d'afficher le tableau de bord du Sudoku, qui est accessible via l'API.
On termine finalement l'application Express avec des middlewares de gestion d'erreurs, et, si on le désire, une façon de servir l'application (Angular 2) directement sur l'adresse du serveur :

```
this.app.get('*', (req, res) => {
    res.sendFile(path.join(__dirname, '../../client/index.html'));
    res.redirect('/');
});
```

On remarque ici qu'il s'agit d'un traitement pour une requête GET. Ce traitement, puisqu'il est placé plus bas que le middleware du routeur *rest.default* (voir page précédente), « attrapera » toutes les requêtes GET ne passant pas par la route */api*, c'est-à-dire les URLs normaux que l'utilisateur entrera pour accéder au site (lorsque vous accédez à un site, vous faites une requête GET via votre navigateur). Puisqu'on a mis un caractère joker comme route (*) pour attraper tout URL mal formulé, on voudra envoyer au client connecté notre page principale de l'application Angular, soit *index.html*, et le rediriger à la racine de cette page (/), soit la page d'accueil du Sudoku dans le cas présent.
Bon! Attaquons-nous au routeur maintenant. Le routeur Express est responsable de répondre aux requêtes des clients qui se connectent à notre serveur. Ainsi, il répond aux diverses requêtes http telles que GET, POST, PATCH, PUT, DELETE, etc. En se servant de ce routeur, on pourra créer un API REST, qui saura effectuer toute la communication éventuelle entre nos clients et le serveur.
Pour utiliser le routeur Express, il suffit d'y faire référence :

```
const router = express.Router();
```

Comme dit précédemment, on peut également se servir directement de this.app.*méthode*. Il y a toutefois une utilité à se servir d'un routeur : on peut avoir des sous-routes bien séparées. Voici par exemple l'API REST de l'application Sudoku :

```
router.get('/log', (req, res) => {
    res.render("log", …);
});

router.get('/getSudoku/easy', (req, res) => {
    …
    sudokuManager.getLogger().logEvent("DEMANDE", …);
    res.send({ grid: easySudoku.grid, difficulty: easySudoku.difficulty });
});

router.get('/getSudoku/hard', (req, res) => {
    …
    sudokuManager.getLogger().logEvent("DEMANDE", …);
    res.send({ grid: hardSudoku.grid, difficulty: hardSudoku.difficulty });
});

router.get('/getHighscores', (req, res) => {
    database.getHighscores()
        .then((highscores) => res.send(highscores))
        .catch((error) => res.send(error));
});

router.post('/validateSudoku', (req, res) => {
    let result = sudokuManager.verifySudoku(req.body);
    res.send(result);
});

router.post('/validateName', (req, res) => {
    res.send(nameManager.validateName(req.body.name));
});

router.delete('/removeName', (req, res) => {
    res.send(nameManager.removeName(req.body.name));
});

router.put('/addScore', (req, res) => {
    database.addScore(req.body.name, req.body.time, req.body.difficulty)
        .then((added) => res.send(added))
        .catch((error) => res.send(error));
});
// Certaines méthodes ont été masquées afin de n'afficher que les choses pertinentes.
```

Toutes les routes présentées ici débutent en fait par */api*. On aurait donc, par exemple, */api/getSudoku/hard*. Cela est possible grâce au *this.app.use('/api', rest.default*) effectué précemment : on assigne toutes les routes débutant par */api* au routeur *rest*. On remarque que la plupart des routes font usage de la méthode ***send*** : celle-ci sert à envoyer une **réponse** au client qui a effectué la requête http. Ainsi, lorsqu'un client demande un sudoku facile par exemple, le routeur Express se chargera de lui envoyer une réponse contenant une grille de Sudoku facile, générée à partir des classes côté serveur. Il faut envoyer une réponse au client pour lui faire savoir que sa requête s'est bien rendue, sans quoi il pourrait recevoir une erreur de *timeout* (aucune réponse). Toutefois, on voudra éviter d'envoyer des réponses trop chargées pour des requêtes qui ne sont pas des GET : on contrevient à l'esprit d'un API REST (dit RESTful) si on fait cela. En effet, chaque requête devrait être traitée de la façon appropriée :

Requête	Ce que ça veut dire	Méthode Express	Ce qu'on veut	Ce qu'on ne veut pas
ALL	Ceci n'est pas une vraie requête http! C'est propre à Express!!	router.all() ou this.app.all()	On veut gérer tous les types de requête passant par cette route	On veut éviter de s'en servir trop souvent…
GET	Obtenir des informations, des données ou une page web,	router.get() ou this.app.get()	On veut une réponse qui contient ce que l'on demande : une grille de Sudoku, une page web…	On ne veut rien modifier sur le serveur!
POST	Envoyer quelque chose au serveur à partir du client.	router.post() ou this.app.post()	On veut ajouter quelque chose au serveur; un document dans une base de données par exemple. On peut aussi s'en servir pour la validation.	On ne veut pas retourner quelque chose d'autre qu'un id, un booléen ou une erreur! La requête GET existe pour ça…
PUT	Modifier quelque chose sur le serveur (souvent, une ressource).	router.put() ou this.app.put()	On veut modifier une entrée déjà existante sur notre serveur, par exemple un document d'une base de données. On peut aussi s'en servir pour ajouter, comme POST, mais de façon idempotente (si l'entrée existe déjà, on ne fait rien).	On ne veut pas retourner quelque chose d'autre qu'un id, un booléen ou une erreur! La requête GET existe pour ça…
DELETE	Supprimer quelque chose sur le serveur.	router.delete() ou this.app.delete()	On veut supprimer une entrée déjà existante sur notre serveur (ex. document de BD).	On ne veut pas de retour significatif (comme POST et PUT), et on ne veut surtout pas ajouter quelque chose sur le serveur!
PATCH	Modifier une structure sur le serveur (requête assez rare).	router.patch() ou this.app.patch()	On veut explicitement modifier une partie de notre modèle de données (exemple, le schéma des scores de Sudoku).	On ne veut pas de retour significatif et on ne veut rien changer en termes de quantité de données sur le serveur! Seulement leur structure…

Ne pas traiter correctement les requêtes http avec Express peut générer des erreurs (ou on peut en générer soi-même…). Généralement, dans le cas de requêtes plus complexes ou risquant d'échouer, on voudra modifier le callback de nos routes :

```
router.get('/log', (err, req, res, next) => {
    if (err) {
        res.status(500).send("T'as tout cassé mon Roger!");
    }
    res.render("log", …);
});
```

Ici, si le callback contient bel et bien une erreur (err !== undefined), le serveur Express lancera une erreur 500 à notre client, ce qui signifie « Erreur serveur »! Toutefois, puisque nos applications ont déjà des middlewares de gestion d'erreurs, ce n'est pas absolument nécessaire à moins de vouloir envoyer une erreur spécifique à notre client. Au cas où, voici une petite liste des codes de réponses http les plus communs :

Code	Nom complet	Quand, comment, pourquoi?
200	OK	Tout va bien! La requête a bien été traitée et une réponse a été reçue. Code le plus commun.
201	CREATED	Une ressource a été créée sur le serveur. Elle est généralement accessible via la réponse.
204	NO CONTENT	Requête bien traitée mais la réponse est vide.
304	NOT MODIFIED	La requête a bien été traitée mais la ressource demandée n'a pas changé depuis le dernier appel à cette requête. Survient souvent lors du chargement d'images, de CSS, etc.
400	BAD REQUEST	Votre requête est mal écrite! Ne devrait pas arriver sauf si vous avez des erreurs côté client.
401	UNAUTHORIZED	La ressource nécessite une authentification (ceci n'est pas traité dans le cours).
403	FORBIDDEN	Comme 401, mais est totalement inaccessible (le client ne devrait jamais pouvoir accéder à la ressource).
404	NOT FOUND	Erreur la plus commune. La ressource demandée est introuvable sur le serveur (mauvais chemin d'accès, route non gérée, etc.).
500	INTERNAL SERVER ERROR	Probablement que votre serveur vient de crasher. Survient aussi lorsqu'il y a une erreur lors de l'exécution du traitement d'une requête sur le serveur et que le code est envoyé explicitement.
503	SERVICE UNAVAILABLE	Vous venez de crasher le serveur. Pour vrai.

C'est pas mal tout pour Express! Passons maintenant à notre meilleur ami Angular.

Section 3 : Angular 2 (ou 4)

Angular est un framework de développement d'applications côté client. Sa principale fonctionnalité est le data binding, et la séparation des objets en composants et en services. Angular peut affecter le DOM en temps réel, donc mettre à jour la page sans la recharger à partir du serveur (on parlera d'AJAX).

Le décorateur @Component

Un component Angular est une façon de faire le lien entre des données et des méthodes en TypeScript (fichier .ts) et le DOM d'une page html (et son CSS). Il est déclaré dans un fichier TypeScript à l'aide du décorateur **@Component**. Voici un exemple de décorateur pour le *SudokuGridComponent* :

```
@Component({
    selector: 'sudoku-grid',
    templateUrl: '/assets/templates/sudokuGrid.component.html',
    providers: [SudokuService, StopwatchService, InputService]
})
```

Le décorateur comprend trois choses importantes : un sélecteur, un *template* et des *providers*. Le sélecteur est l'identifiant du component dans le DOM; c'est avec ce nom qu'on pourra insérer le component dans notre HTML (ici : **<sudoku-grid></sudoku-grid>**). Le *template* est le contenu du component : c'est ce qu'il contiendra et affichera à l'intérieur de sa balise HTML (définie par le sélecteur). On peut soit écrire *template : `…du code html …` si* on veut écrire du HTML directement, ou bien *templateUrl : 'chemin d'accès du fichier'* si on veut utiliser un fichier HTML écrit séparément (ce qui est préférable dans presque tous les cas). Finalement, les *providers* sont les **services** injectés dans le component : ils servent à fournir de l'information au component et à effectuer des calculs pour lui.
Les components peuvent implémenter un certain nombre d'interfaces afin d'effectuer des actions à des moments précis. Les plus utiles sont **OnInit** et **AfterViewInit**. Ces interfaces sont implémentées avec les méthodes **ngOnInit** et **ngAfterViewInit**, toutes deux publiques. ngOnInit devra contenir le code que l'on veut exécuter dès l'instanciation du component (par exemple l'ouverture d'un dialogue), alors que ngAfterViewInit contiendra ce que l'on veut faire une fois que le component est pleinement affiché à l'écran (il n'est plus en train de se charger). Toutefois, ce sont bien sûr des méthodes facultatives.

Le data-binding avec les components

Les components servent principalement à afficher des informations dynamiques sur notre application web. Pour faire cela, on utilisera ce qu'on appelle du data-binding : on *bind* des variables du component avec des éléments du DOM. La façon la plus simple de faire un binding est en utilisant l'*interpolation* {{ ... }} :

```
<p>Niveau de difficulté : {{sudokuService.difficulty}} </p>
```

Cette forme très simple de data-binding affiche directement le contenu d'une variable ou la valeur de retour d'une fonction. Ici, on verra par exemple sur le site Niveau de difficulté : Facile. Il s'agit d'un binding uni-directionnel du component vers le DOM.

On peut également identifier des éléments du DOM afin d'utiliser leurs propriétés à même la page, sans passer par du TypeScript. On appelle cela une *template variable* :

```
<md-sidenav #sidenav mode="above" opened="false">
...
<button md-button class="list-button" (click)="showHighscores(); sidenav.toggle()">
```

Ici, l'élément *md-sidenav* est identifié par *#sidenav*. Il est plus tard appelé à même la page à l'aide de *sidenav.toggle()*, car il s'agit lui aussi d'un component (de Angular Material) avec méthodes et des variables.
On voit également dans cet extrait de code un exemple de *template statement*, c'est-à-dire une exécution effectuée suite à un événement du DOM. Ici, l'événement *click* est le déclencheur : lorsque le bouton sera cliqué, on montrera les records et on fermera le *sidenav*. *showHighscores()* est une méthode du component qui se sert du template où elle se trouve (*GameAreaComponent* dans ce cas-ci).

On peut également avoir du binding dans l'autre direction, c'est-à-dire du DOM vers notre component, ou vers un élément HTML. Pour faire cela, on se servira du *property binding* [...] :

```
<table [style.border-color]="isDarkTheme ? 'white' : 'black'">
```

Ici, en fonction de la valeur de la variable *isDarkTheme* du component *SudokuGridComponent*, on modifiera le CSS de l'élément *table* à l'aide de *[style.border-color]*. Le *property binding* est très utile pour effectuer des modifications au CSS ou à d'autres éléments HTML. Un exemple particulièrement intéressant, si on fait du CSS propre, est l'utilisation de [ngClass] :

```
<div [ngClass]="{'strikethrough' : player.hasQuitAfterGameEnd, 'bold' : player.name ===
this.activePlayerName}">
    {{player.name}}
</div>
```

ngClass est une façon simple d'assigner des classes de CSS en fonction de certaines variables, booléens ou méthodes. On peut voir ici que les classes *strikethrough* et *bold* (deux classes CSS personnalisées) sont assignées à l'élément *div* contenant le nom du joueur si celui-ci a quitté la partie ou si celui-ci est le joueur actif, respectivement.

Le dernier type de data-binding est le bidirectionnel, aussi appelé *Banana in a Box* [(...)]. Ce type de data-binding fait le lien dans les deux directions : si la valeur est modifiée dans le component, elle sera modifiée en conséquence dans le DOM, mais si elle est modifiée via le DOM (un input field, par exemple), elle sera modifiée à son tour dans le component! C'est donc une façon de donner un contrôle direct sur une variable à l'utilisateur de l'application. L'exemple le plus courant de data-binding bidirectionnel est l'utilisation de [(ngModel)] sur les inputs et les forms :

```
<input [(ngModel)]="getPlayer().name" placeholder="name"/>
```

Ici, le texte entré dans l'élément *input* est lié de façon bidirectionnelle avec le nom du joueur dans le component. Ainsi, si on commence à taper dans le input, le nom du joueur est mis à jour en temps réel dans le component. À l'inverse, si on modifiait le

nom du joueur via le component alors que l'input n'est pas encore envoyé, le joueur verrait son nom tapé dans l'input changer à mesure que le code le fait!

La communication entre les components

Les components Angular ont la particularité de pouvoir communiquer entre eux assez facilement. Pour faire cela, on utilise les décorateurs **@Input** et **@Output**, ainsi que la classe **EventEmitter**. Par exemple, si dans le détail d'un component j'ai :

```
<dew-machine [mtnDew]="color" (giveDew)="receiveDrink($event)"></dew-machine>
```

Je devrai avoir dans mon component les décorateurs suivants :

```
@Input() mtnDew: DewColor;
@Output() giveDew = new EventEmitter<DewColor>();
```

Ici, mtnDew est bel et bien une variable utilisable dans mon component *DewMachineComponent*. De même, *giveDew* est un émetteur d'événement : si je l'appelle, il émettra un événement qui pourra être reçu par la méthode *receiveDrink($event)* du component dans lequel se trouve le *DewMachineComponent* (on pourrait supposer *MLGComponent*).

Les événements et le @HostListener

Les événements sont cruciaux en Angular. Ils permettent à l'application de réagir à toutes sortes d'interactions de l'utilisateur (des clics de souris, des touches de clavier, des événements custom comme *giveDew*...). Reprenons l'exemple précédent et voyons comment on pourrait gérer l'événement *giveDew* dans *MLGComponent* :

```
public receiveDrink(event: any): void {
    this.dewTank.refill(event.color);
}
```

Comme on peut le voir, c'est assez simple! Il suffit d'identifier l'événement par *$event* dans la parenthèse de la méthode réceptrice, et on pourra ensuite le traiter comme on voudra à l'intérieur de la méthode (pour autant qu'on se serve des bonnes propriétés). C'est cool, mais qu'est-ce qu'on fait si on veut recevoir un événement qui ne survient pas explicitement sur un component, comme une touche de clavier? On se sert de **@HostListener** :

```
@HostListener('window:keydown', ['$event'])
public keyboardInput(event: KeyboardEvent): void {
    // Player can switch camera view
    if (event.key === "ArrowLeft" || event.key === "ArrowRight") {
        this.switchCamera();
    } else { // For space bar
        this.gameController.onKeyboardDown(event);
    }
}
```

Avec @HostListener, on peut spécifier dans notre code directement quel élément du DOM on souhaite « écouter » pour des événements. Ici, on écoutera la fenêtre du navigateur, *window*! On pourra alors intercepter et traiter tous les événements de type *keydown* (touche du clavier qui se fait enfoncer) tant que le component est actif. Voici des exemples avec la souris :

```
@HostListener('window:mousedown', ['$event'])
public onMouseDown(event: MouseEvent): void {
    this.gameController.onMouseDown(event);
}

@HostListener('window:mouseup', ['$event'])
public onMouseUp(event: MouseEvent): void {
    this.gameController.onMouseUp(event);
}
```

```
@HostListener('window:mousemove', ['$event'])
public onMouseMove(event: MouseEvent): void {
    this.gameController.onMouseMove(event);
}
```

Les ViewChilds (@ViewChild)

Les ViewChilds sont une façon extrêmement simple pour un component parent d'exécuter des méthodes ou de modifier des propriétés de ses enfants (les enfants d'un component sont placés à l'intérieur de celui-ci, dans le HTML). On n'a qu'à se servir du décorateur **@ViewChild** :

```
@ViewChild(RackComponent) private rackChild: RackComponent;
@ViewChild(BoardComponent) private boardChild: BoardComponent;
@ViewChild(InfoComponent) private infoChild: InfoComponent;
@ViewChild(ChatComponent) private chatChild: ChatComponent;
```

Cet exemple provenant du Scrabble est parfait pour expliquer le fonctionnement des ViewChild. En effet, ces lignes sont contenues dans *GameComponent*, un component qui englobe 4 enfants : le rack de lettres, la grille de jeu, le panneau informatif et la boîte de chat. Voici un exemple d'utilisation des méthodes d'un ViewChild :

```
public disableRack(): void {
    if (this.rackChild.isActive()) {
        this.rackChild.toggleActiveState();
        this.rackChild.deselectLetter();
    }
}
```

La méthode *disableRack()* est appelée dans le *template* du *GameComponent* via un événement intercepté lors de de la mise en focus du component de chat. Cette méthode aura alors pour effet de rendre inactif le contrôle du rack, ce qui est fait de façon très simple grâce aux ViewChilds. C'est aussi idéal pour propager des modifications s'effectuant au component parent directement, par exemple l'application d'un thème sombre.

Les directives structurelles (NgIf, NgFor)

Angular inclut certaines directives pour nous rendre la vie plus facile. À l'aide des directives ***ngIf** et ***ngFor**, on peut afficher, cacher, et même répéter des éléments du DOM en fonction de nos variables de component :

```
<div *ngFor="let round of roundsCompleted;">
    <div id="rounds">
        <img *ngIf="round === false" [src]="isDarkTheme ? './assets/textures/curling-manche-blanc-
icon.png' : './assets/textures/curling-manche-gris-icon.png'" height=30vh>
        <img *ngIf="round === true" [src]="isDarkTheme ? './assets/textures/curling-manche-jaune-
icon.png' : './assets/textures/curling-manche-bleu-icon.png'" height=30vh>
    </div>
</div>
```

On a ici un bel exemple d'une combinaison des deux. Le *ngFor se trouvant dans la balise du premier *div* va faire en sorte que celui-ci va se répéter pour chaque *round* se trouvant dans *roundsCompleted* (1, 2, 3…). Les *ngIf à l'intérieur vont faire s'afficher une image différente en fonction de si *round* (la variable de parcours du *ngFor !!!) est true ou false!
Il y a également moyen d'obtenir l'index du *ngFor lors de son parcours afin de l'utiliser dans notre HTML :

```
<md-card id="chatCard" *ngFor="let msg of msgList; let i = index">
```

Il faut faire attention avec les *ngIf, car on pourrait avoir certains problèmes si on évalue une condition changeante à l'intérieur (sa valeur n'est plus la même lorsque le DOM n'est pas chargé et lorsqu'il est chargé, par exemple).

Angular HTTP

Angular implémente une façon simple de faire des requêtes http à partir de notre client. C'est très utile pour communiquer avec un API REST…
Les requêtes http se font en ajoutant d'abord l'objet http d'Angular à notre constructeur de component ou de service :

```
constructor(private http: Http) { }
```

Ensuite, on peut se servir de l'objet http pour effectuer divers types de requêtes :

```
this.http.get(url);

this.http.post(url, body);

this.http.put(url, body);

this.http.patch(url, body);

this.http.delete(url);
```

L'URL correspond au chemin que l'on veut emprunter pour notre requête. Ça pourrait par exemple être http://localhost:3000/api/getSudoku/easy . Le body, quant à lui, contient ce que l'on veut envoyer au serveur dans le cas d'une requête POST, PUT ou PATCH. Il s'agit habituellement d'un objet au format JSON, dénoté par des *curly braces*.
Par exemple :

```
this.http.post("http://localhost:3000/api/validateName", { "name": name })…
```

Si on veut écouter la réponse du serveur (habituellement, on veut avoir une réponse!), on a deux choix : utiliser *subscribe* ou bien transformer notre requête en une *Promise* :

```
…).subscribe((res) => { validName = res.text() === "true" });
```

La façon subscribe, ici.

```
let postPromise = new Promise((resolve, reject) => {
    this.http.post(this.HOST_NAME + this.SERVER_PORT + "/api/validateName", { "name": name })
        .toPromise().then(res => {
            if (res.text() === "true") {
                validName = true;
                resolve(true);
            } else {
                resolve(false);
             }
        }).catch(() =>  reject());
    return validName;
});
```

Et la façon promesse ici.
Bien sûr, la méthode avec la promesse n'est utile que dans le contexte où vous auriez à vous servir d'une promesse. Toutefois, les requêtes http Angular sont asynchrones, et il est donc très bénéfique d'utiliser des promesses si on veut effectuer un traitement suite à notre requête, plutôt que de tout enfourner dans le callback du *subscribe*.
Dernière chose, on voudra convertir notre réponse *res* en texte ou en *json*, selon ce que nous renvoie le serveur :

```
res.text(); // Donne le texte de notre réponse, si celle-ci n'est pas un JSON.
res.json(); // Donne notre réponse en JSON, si on veut accéder à ses champs.
```

Les services et @Injectable

Finalement, les services sont l'équivalent en Angular du Modèle dans l'architecture MVC. Lorsqu'on déclare un service, on doit lui ajouter le décorateur **@Injectable** pour signaler à Angular que l'on désire se servir des méthodes du service à partir d'un component (et potentiellement à partir du DOM). On parlera alors d'un *provider* :

```
@Injectable()
export class GameController { … }
```

Dans *GlComponent* :

```
@Component({
    selector: 'my-gl',
    templateUrl: "/assets/templates/gl.component.html",
    providers: [GameController]
})
```

Rendre des services injectables est surtout utile dans la mesure où cela permet à Angular de donner les bons paramètres aux constructeurs de services interdépendants. Par exemple, le constructeur de *GameController* se sert du *HighscoresService*, un autre injectable, et n'est jamais explicitement appelé (c'est Angular qui s'en charge); le tout sans erreurs :

```
constructor(private highscoresService: HighscoresService) { }
```

C'est tout pour Angular, je crois!

Section 4 : Three.js

Three.js est une librairie Javascript permettant de réaliser des applications web graphiques avec WebGL. Elle inclut toutes sortes d'outils afin de créer une scène, ajouter de l'illumination, des objets 3D, etc.

La scène et le moteur de rendu

Three.js se sert du principe de **scène** couramment utilisé en infographie. Une scène est une représentation structurelle de l'espace en 3D que l'on désire afficher à l'écran. Elle est essentielle pour afficher quoi que ce soit à l'aide de Three.js, car il s'agit d'un argument pour le moteur de rendu WebGL (*renderer*). Il faut d'ailleurs initialiser ce moteur de rendu avant toute autre chose, car c'est lui qui sera responsable d'afficher nos objets 3D à l'écran :

```
private renderer: THREE.WebGLRenderer;
…
this.renderer = new THREE.WebGLRenderer();
this.renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(this.renderer.domElement);
```

Ici on crée le moteur de rendu WebGL et on lui affecte la taille de la fenêtre d'affichage. On l'attache ensuite au corps de notre document (le *domElement* du renderer est un élément de type *canvas*, utilisé en HTML5 pour de l'affichage graphique).
Par la suite, on peut créer une scène vide, à laquelle on rajoutera des éléments éventuellement :

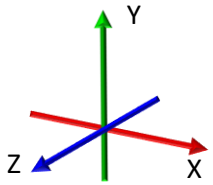
```
this.scene = new THREE.Scene();
```

On a alors l'essentiel pour afficher quelque chose à l'écran, moins une chose : une caméra. Pour l'instant, on va supposer qu'on a une caméra d'un type quelconque contenue dans *this.camera* ; on reviendra sur les caméras un peu plus loin :

```
public update(): void {
    window.requestAnimationFrame(() => this.update());
    this.renderer.render(this.scene, this.camera);
}
```

Ceci constitue une boucle de jeu très basique. Elle sera appelée une première fois dans le code à quelque part pour démarrer l'animation, puis s'appellera elle-même à chaque fois que le navigateur sera prêt à afficher un nouveau *frame* (image) d'animation. En général, WebGL se synchronise avec la fréquence d'affichage de l'écran : faire rouler l'application sur un écran 60 Hz appellera la méthode *update* au maximum 60 fois par seconde, en autant que la carte graphique de la machine exécutante puisse suivre.

L'espace 3D et les caméras



Three.js, de par son nom, utilise un moteur de rendu en 3D, c'est-à-dire qu'il peut afficher des objets ayant des dimensions sur trois axes (x, y, z). Par défaut, le référentiel qu'utilise Three.js respecte la règle de la main droite, c'est-à-dire qu'on a des vecteurs unitaires orientés comme ceux présentés à droite. Il faut faire attention, car contrairement à certains autres moteurs où l'axe z est vers le haut, l'axe z est sur le côté en Three.js (et l'axe y est vers le haut).

Tout en Three.js est positionné à l'aide de vecteurs 3D. Le *Vector3* est donc « l'unité de base » de Three.js, avec chaque unité sur les axes valant théoriquement 1 mètre :

```
let position = new THREE.Vector3(1, 2, 3);
```

Ceci est un exemple de vecteur à trois dimensions, qui définit le point (x, y, z) = (1, 2, 3) à partir de l'origine (0, 0, 0). Il faut savoir que tout en Three.js est par défaut créé, placé, etc. à l'origine (0, 0, 0).

On a besoin d'une caméra pour afficher notre espace 3D à l'écran, et celle-ci peut être de deux types : perspective ou orthographique. Une caméra perspective affiche le monde de la même façon que nous le voyons, c'est-à-dire que les objets plus près ou plus loin sont déformés; ils se dirigent vers un point de fuite. C'est généralement le type de caméra le plus souhaitable :

```
private initPerspectiveCamera(containerRect: ClientRect): void {
    // Camera creation
    let aspectPerspective = containerRect.width / containerRect.height;
    this.cameraPerspective = new THREE.PerspectiveCamera(FOV, aspectPerspective, NEAR, FAR);
}
```

Une caméra perspective est créée à l'aide de quatre arguments : un champ de vision (FOV, Field Of View), le ratio d'affichage de l'écran (*aspectPerspective*, qui ici est calculé à partir de la largeur du canvas sur sa hauteur), le plan de coupe proche (near clipping plane) et le plan de coupe lointain (far clipping plane). Ce sont tous des chiffres (type *number*).

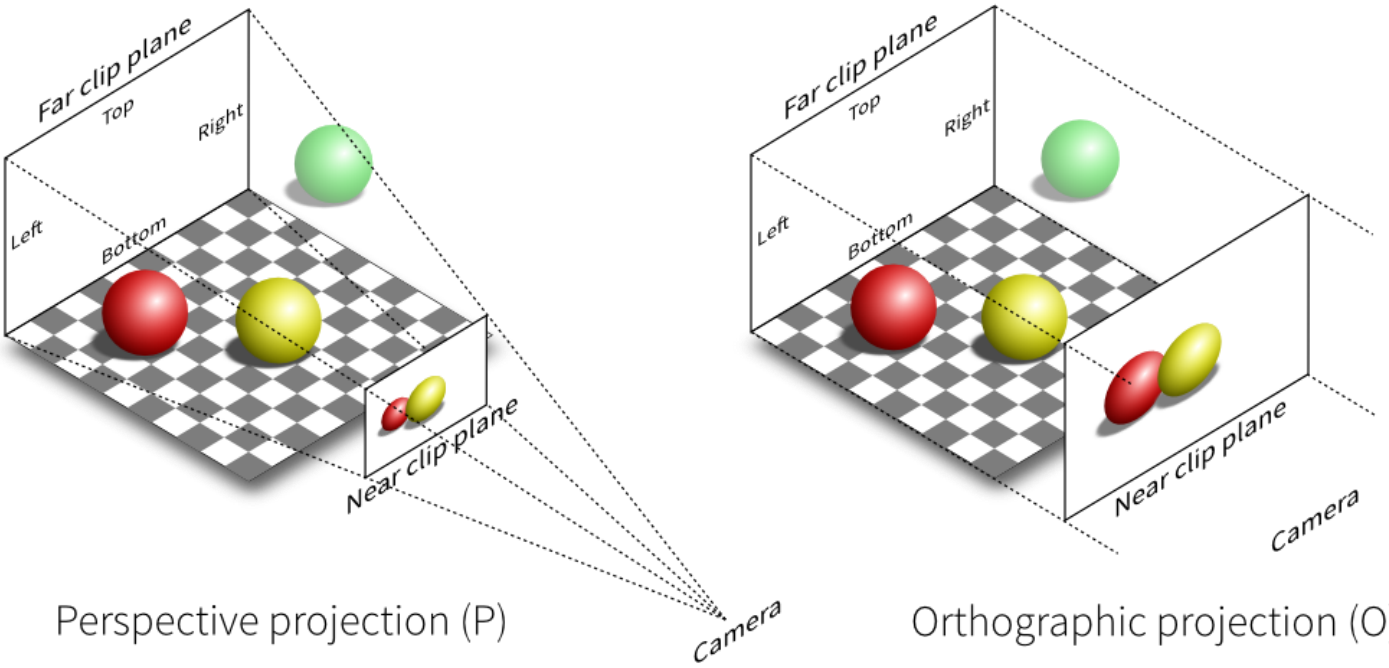
- Le FOV équivaut à l'angle vu par la caméra et représenté à l'écran. Plus il est petit, plus l'affichage paraît gros et proche, et plus il est grand, plus l'affichage s'étire sur les côtés pour afficher davantage de la scène. Un FOV souhaitable pour un écran standard est entre 40 et 70 degrés.
- Le ratio d'affichage est une façon de faire en sorte que les choses soient bien affichées à l'écran, et pas étirées en hauteur ou en largeur. Un exemple serait 1920/1080 = 16/9 = 1.7 périodique.
- Le plan de coupe proche est la distance avant laquelle les choses trop proches de la caméra ne seront pas affichées à l'écran. C'est utile dans la mesure où on veut éviter d'avoir de toutes petites choses qui apparaissent immenses lorsqu'elles sont très près de la caméra. Ça simule la façon dont l'être humain ne peut pas voir des choses qui sont trop près de son nez. Il est calculé en mètres (near = 1 voudrait dire que tous les objets à moins de 1 mètre ne seront pas affichés)
- Le plan de coupe lointain est la distance après laquelle les choses trop loin de la caméra ne seront pas affichées à l'écran. C'est utile pour sauver en performances : ça ne sert à rien d'afficher à l'infini si notre scène ne fait pas plus de 200 mètres de profondeur. Il est calculé en mètres (far = 200 voudrait dire que tous les objets à plus de 200 mètres ne seront pas affichés).

Une caméra orthographique quant à elle affichera tous les objets de la même façon, peu importe qu'ils soient proches ou loin de la caméra. Ils ne seront donc jamais étirés, et on perdra beaucoup l'effet de « profondeur » amené par la caméra perspective. C'est toutefois très utile pour faire des vues en plongée, où on veut une représentation exacte des distances, par exemple. Ça revient à « plaquer » notre image 3D sur un plan 2D.

```
private initOrthographicCamera(containerRect: ClientRect): void {
    // Camera creation
    this.cameraOrthographic = new THREE.OrthographicCamera(LEFT, RIGHT, TOP, BOTTOM, NEAR, FAR);
```

Une caméra orthographique se sert de 6 plans, définis par des nombres, afin de délimiter la zone qu'elle affiche. Il faut donc la voir comme une « boîte » :

- LEFT est la distance en mètres du plan de coupe à gauche par rapport au centre visé par la caméra.
- RIGHT est la distance en mètres du plan de coupe à droite par rapport au centre visé par la caméra.
- TOP est la distance en mètres du plan de coupe en haut par rapport au centre visé par la caméra.
- BOTTOM est la distance en mètres du plan de coupe en bas par rapport au centre visé par la caméra.
- NEAR est la distance en mètres du plan de coupe avant lequel les objets ne seront pas affichés, par rapport à la position de la caméra.
- FAR est la distance en mètres du plan de coupe après lequel les objets ne seront pas affichés, par rapport à la position de la caméra.



La caméra perspective possède un avantage important par rapport à la caméra orthographique : elle peut viser précisément un point de la scène à l'aide de la méthode *lookAt* :

```
this.cameraPerspective.lookAt(new THREE.Vector3(1, 2, 3));
```

Cet appel fera en sorte que la caméra perspective « pointera » vers la position (x, y, z) = (1, 2, 3) à partir de sa position actuelle (elle effectuera une rotation autour de l'axe des Y, puis des X ou des Z). C'est super pour suivre des objets comme des pierres de Curling…
Bien sûr, on pourra vouloir déplacer la caméra, et possiblement la faire tourner sur soi-même. Pour faire cela, on utilisera les mêmes principes que pour les objets 3D (*this.camera.position.set…*), détaillés dans la section suivante…

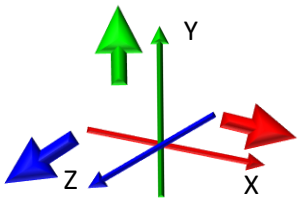
Les objets 3D, les positions et les rotations

En Three.js, tous les objets qu'on ajoute à une scène sont de type *Object3D* (ou *Group* si on ne veut pas être deprecated). Ils possèdent tous une position et une rotation, qui sont des vecteurs 3D. On peut modifier ces vecteurs afin de faire se déplacer ou tourner nos objets. ATTENTION : La position et la rotation des objets, accédée directement, est une propriété READONLY. Il faut donc utiliser des méthodes, ou affecter des valeurs aux composantes de ces vecteurs; on ne peut pas faire *this.object.position = unVector3* directement. Voici toutefois certaines méthodes appropriées :

Translations (ajouter unChiffre à sa position sur l'axe spécifié)

```
this.object.translateX(unChiffre);
this.object.translateY(unChiffre);
this.object.translateZ(unChiffre);
```

Les translations se font dans la direction positive des axes.



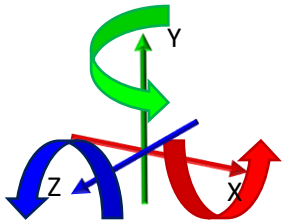
Positionnement (spécifier une position)

```
this.object.position.set(unX, unY, unZ);
// équivaut à
this.object.position.x = unX;
this.object.position.y = unY;
this.object.position.z = unZ;
```

Rotations (tourner l'objet de desRadians selon l'axe spécifié)

```
this.object.rotateX(desRadians);
this.object.rotateY(desRadians);
this.object.rotateZ(desRadians);
```

Les rotations se font selon la règle de la main droite.



Rotation spécifique (spécifier une rotation exacte)

```
this.object.rotation.set(xRads, yRads, zRads);
// équivaut à
this.object.rotation.x = xRads;
this.object.rotation.y = yRads;
this.object.rotation.z = zRads;
```

Les géométries, les matériaux, les mesh et les textures

Les objets 3D en Three.js sont composés de trois éléments de base : une géométrie, un mesh et au moins un matériau.
Une géométrie est un ensemble de points (de sommets) définissant une forme en trois dimensions. Les différentes géométries prennent différents paramètres lors de leur création, notamment des dimensions et un nombre de faces (surtout pour les géométries plus rondes). Voici par exemple les géométries pour un anneau, un disque (un cercle), et une boîte (un prisme):

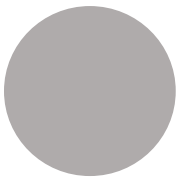
```
let whiteCenterGeometry: THREE.Geometry = new THREE.CircleGeometry(radius, faces);
let redRingGeometry: THREE.Geometry = new THREE.RingGeometry(inner, outer, faces);
let rinkGeometry: THREE.Geometry = new THREE.BoxGeometry(width, height, length);
```

Dans chacun des cas, on obtient une géométrie ayant les dimensions spécifiées (en mètres), ainsi que certains attributs supplémentaires comme le nombre de faces (plus il est grand, plus il y a de sommets créés pour représenter la « courbure »).

On a ensuite besoin d'un matériau. Un matériau représente le type de surface qui sera utilisé pour « combler » l'espace entre nos sommets, c'est-à-dire donner des faces à notre objet. Il en existe plusieurs types, dont les principaux vus en infographie :

Matériau basique non-affecté par la lumière.

```
let basic: THREE.Material = new THREE.MeshBasicMaterial({attributs});
```



Les principaux attributs du BasicMaterial sont *color* et *map*. Color sert à assigner une couleur fixe à notre matériau, alors que map sert à assigner une texture qui sera mappée (d'où le nom) sur notre objet (elle sera étirée pour le couvrir entièrement). ON pourrait avoir par exemple :

```
let material = new THREE.MeshBasicMaterial({ color: 0xFFFFFF, map: texture });
```

Matériau de Lambert.

```
let lambert: THREE.Material = new THREE.MeshLambertMaterial({attributs});
```

Un matériau de Lambert reflète la lumière d'une certaine façon sur chacune de ses faces. Il a les mêmes attributs principaux que le matériau basique.

Matériau de Phong.

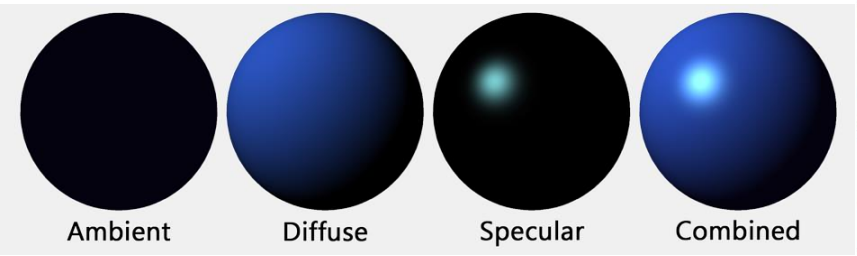
```
let phong: THREE.Material = new THREE.MeshPhongMaterial({attributs});
```

Un matériau de Phong reflète la lumière de façon réaliste, avec une réflexion diffuse et spéculaire qui est « smooth ». Il a certains attributs supplémentaires par rapport au Basic et au Lambert : *emissive*, *emissiveIntensity*, *envMap*, *reflectivity*, *shininess* et *specular* :

- *Emissive* et *emissiveIntensity* déterminent la couleur et l'intensité de la lumière émise par le matériau. Cette lumière sera toujours présente, même si l'objet est placé dans le noir complet.
- *envMap* est une texture d'environnement qui pourra être reflétée par le matériau. On pourra penser entre autres à la Skybox.
- *Reflectivity* détermine l'intensité de la réflexion de la envMap.
- *Specular* et *shininess* déterminent la couleur et l'intensité de la lumière spéculaire reflétée par le matériau.

Les variables ayant rapport à des couleurs sont définies comme *color* (ex : #FFFFFF), et la *envMap* est une THREE.CubeTexture. Les autres ne sont que des nombres.

Exemple de lumière ambiante, diffuse, spéculaire, combinée et d'une envMap.



Matériau standard.

```
let standard: THREE.Material = new THREE.MeshStandardMaterial({attributs});
```

Un matériau standard ressemble à un matériau de Phong, à l'exception près qu'il est affecté par la lumière de façon physiquement réaliste (plutôt qu'avec des calculs de vecteurs normaux). Il perd d'ailleurs les attributs *reflectivity*, *shininess* et *specular* pour plutôt avoir *envMapIntensity*, *metalness* et *roughness*.

- *envMapIntensity* équivaut à *reflectivity* pour Phong.
- *Metalness* détermine à quel point le matériau est « métallique » : à 0.0, il agit comme quelque chose de doux et non-métallique (du bois par exemple), à 1.0, il agit purement comme un métal très réfléchissant.
- *Roughness* détermine à quel point le matériau reflète la lumière diffuse de façon spéculaire. À 0.0, le matériau a un fini miroir : il reflète toute la lumière de façon spéculaire. À 1.0, c'est l'inverse : aucune réflexion spéculaire, purement diffuse.

Metalness et *roughness* sont définies entre 0.0 et 1.0.

Finalement, une fois qu'on a une géométrie et un matériau, on peut les ajouter à un mesh. Un mesh est une représentation dans l'espace 3D de notre objet, donc littéralement une forme en 3D composée avec nos sommets de géométrie et notre matériau. Voici un exemple :

```
let redRingGeometry: THREE.Geometry = new THREE.RingGeometry(inner, outer, 40);
let redRingMaterial: THREE.Material = new THREE.MeshStandardMaterial({
    side: THREE.DoubleSide,
    metalness: 0.6,
    roughness: 0.2,
    envMap: this.reflectTexture,
    envMapIntensity: 1.0,
    map: this.redIce
});
let redRing: THREE.Mesh = new THREE.Mesh(redRingGeometry, redRingMaterial);
```

Ce qu'on obtient dans *redRing* est la représentation visuelle de notre objet 3D! On peut à présent l'ajouter directement à scène, ou bien modifier sa position et/ou sa rotation avant de le faire. Ajoutons-le directement dans cet exemple :

```
this.scene.add(redRing);
```

C'est aussi simple que ça. Il sera maintenant affiché dans notre scène pour autant que la caméra soit en mesure de le voir (se rappeler que par défaut, les objets 3D sont positionnés en (0, 0, 0) si on ne modifie pas explicitement leur position).

Pour ce qui est des textures, comme celles qu'on a appliqué aux matériaux, il suffit d'une simple ligne pour les charger dans une variable :

```
let texture: THREE.Texture = new THREE.TextureLoader().load(path);
```
On peut répéter les textures sur une surface afin d'éviter de les étirer, surtout si on a une texture de type « tuile » (qui se répète sans rebords visibles) :

```
this.redIce.wrapS = this.redIce.wrapT = THREE.RepeatWrapping;
this.redIce.repeat.set(2, 2);
```

Ici, on répète la texture sur les deux axes (s et t) jusqu'à deux fois lorsqu'on l'appliquera sur un matériau (comme dans l'exemple de mesh).

Les lumières

Afin de voir notre scène dans Three.js, on doit l'illuminer. Les deux types de lumière les plus courants sont la lumière ambiante et la lumière « spotlight » :

```
let ambient: THREE.Light = new THREE.AmbientLight(couleur, intensité);
this.scene.add(ambient);
```

Cette lumière est ambiante : elle éclairera de façon identique tous les objets, avec une certaine couleur et une certaine intensité de lumière.

```
let spotlight: THREE.Light = new THREE.Spotlight(couleur, intensité, distance, angle, pénombre, decay);
this.scene.add(spotlight);
```

Cette lumière est directionnelle : elle éclairera dans une certaine direction les objets, et provoquera des zones plus claires et plus sombres sur ceux-ci.

- La *distance* est un attribut définissant la distance maximale où la lumière se rend : plus loin que ça et elle n'éclaire rien. Par défaut, ce paramètre est à 0 et la lumière éclaire à l'infini.
- L'*angle* est l'angle d'ouverture du cône de lumière du spotlight (par défaut, il est de PI/2).
- La *pénombre* détermine l'atténuation de la lumière du spotlight, et va de 0.0 à 1.0. À 0.0, le spotlight est très net (éclairé | sombre). Jusqu'à 1.0, il est de plus en plus fou (éclairé | pénombre floue | sombre).
- Le *decay* est le facteur avec lequel se réduit la lumière au fur et à mesure qu'elle se rend à la distance maximale établie. Un facteur de 1, par défaut, définit une réduction linéaire. 2 est plus physiquement correct.

Le spotlight peut cibler un objet pour le suivre. Il suffit de le spécifier via son attribut *target* (ici, *lightTarget* est un autre Object3D déjà ajouté à la scène) :

```
spotlight.target = lightTarget;
```

Sinon, par défaut, un spotlight pointe vers le bas.

Le raycasting

La dernière chose importante en Three.js est le raycasting. Il s'agit d'une technique permettant de projeter la position de la souris à l'écran sur un objet 3D, afin de vérifier par exemple si l'utilisateur clique sur l'objet 3D. On peut créer un raycaster comme n'importe quel autre objet Three.js :

```
this.raycaster = new THREE.Raycaster();
```

On peut ensuite vérifier l'intersection avec n'importe quel objet de notre scène, soit via son nom (attribut *name*, défini auparavant ex : this.object.name = ''patate''), soit via son id si on le connait :

```
public checkIntersectIce(mouse: THREE.Vector2): THREE.Intersection[] {
    this.raycaster.setFromCamera(mouse, this.camera);
    let intersects = this.raycaster.intersectObject(this.scene
                    .getObjectByName("rink").getObjectByName("whiteice"), true);
    return intersects;
}
```

Ici, on prend en paramètre la position de la souris à l'écran (la méthode sera montrée après). On place ensuite le raycaster à l'emplacement de la caméra, afin de lui donner un point de référence, à l'aide de la méthode *setFromCamera*, qui prend la position de la souris et la caméra active. On vérifie ensuite l'intersection avec un objet de notre scène : si on le connaît déjà (il est sauvegardé comme un *this*... dans notre classe), on peut s'en servir directement, sinon, on peut aller chercher notre objet via son nom dans le graphe de scène (ici, je fais la séquence scène → objet « rink » → objet « whiteice ») pour aller chercher mon prisme blanc représentant la glace, qui est un enfant de l'objet *rink*, qui lui, est ajouté à la scène. Je spécifie le second attribut à true pour indiquer à Three.js que je veux que ce soit récursif (il va me donner tous les points d'intersection entre le rayon projeté de la souris et la glace blanche).

Je peux ensuite me servir des intersections pour effectuer un traitement : s'il n'y a pas d'intersection, ça veut dire que la souris ne se trouvait pas vis-à-vis la glace sur l'écran :

```
private mouse: THREE.Vector2;
...
public onMouseMove(event: MouseEvent): void {
    this.mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    this.mouse.y = - (event.clientY / window.innerHeight) * 2 + 1;
    let intersects = GameEngine.getInstance().checkIntersectIce(this.mouse);
    if (intersects.length > 0) {
        let intersectionPoint = intersects[0].point;
        let distance = intersectionPoint.x;
        …
    }
}
```

Cet exemple a été simplifié afin d'illustrer l'essentiel. Ici, sur mouvement de la souris (cette méthode est appelée à l'aide d'un @HostListener Angular, plus haut dans la hiérarchie de classes), on prend note de sa position à partir du *MouseEvent* passé en paramètre. Cela donne alors une valeur normalisée entre 0 et 1 (on doit inverser la position en y, attention!). On peut ensuite constater ou non l'intersection à l'aide de la méthode donnée en exemple auparavant, puis effectuer un traitement si le tableau d'intersections retourné n'est pas vide.

C'est tout! Bon examen!