

 Review the assignment due date

LOG210 - Lab 0: tutoriel sur les technologies

Cet exercice, sous forme de tutoriel, a l'objectif de vous apprendre les technologies utilisées pour le laboratoire de LOG210. On vous propose d'ajouter une fonctionnalité au **Jeu de dés**, un squelette de code que votre vrai projet de LOG210 doit suivre. Le squelette est une application minimaliste permettant d'intégrer correctement plusieurs technologies (interface utilisateur, serveur web avec couches logicielles, etc.). En apprenant avec le squelette, vous pouvez aller plus vite, sans nécessairement tout comprendre au début. Vous pouvez vous concentrer sur la méthodologie d'analyse et de conception qui est le sujet principal de LOG210.

Ce travail est individuel, soumis dans un dépôt privé, pour que chaque personne puisse comprendre et contribuer efficacement dans son équipe. ⚠ Les points pour ce laboratoire sont dans le volet de travail de nature individuelle. **L'évaluation de ce travail déterminera en partie si vous passez le seuil pour la note minimale pour l'ensemble des éléments évalués individuellement.** Alors chaque point est très important!

Préalables

Dans les cours préalables à LOG210, vous devriez avoir déjà vu:

- Les dépôts de code source en *git* (ici, c'est GitHub)
- La programmation orientée objet (ici, c'est TypeScript)
- Les tests (ici, c'est Jest et SuperTest)

Date de remise

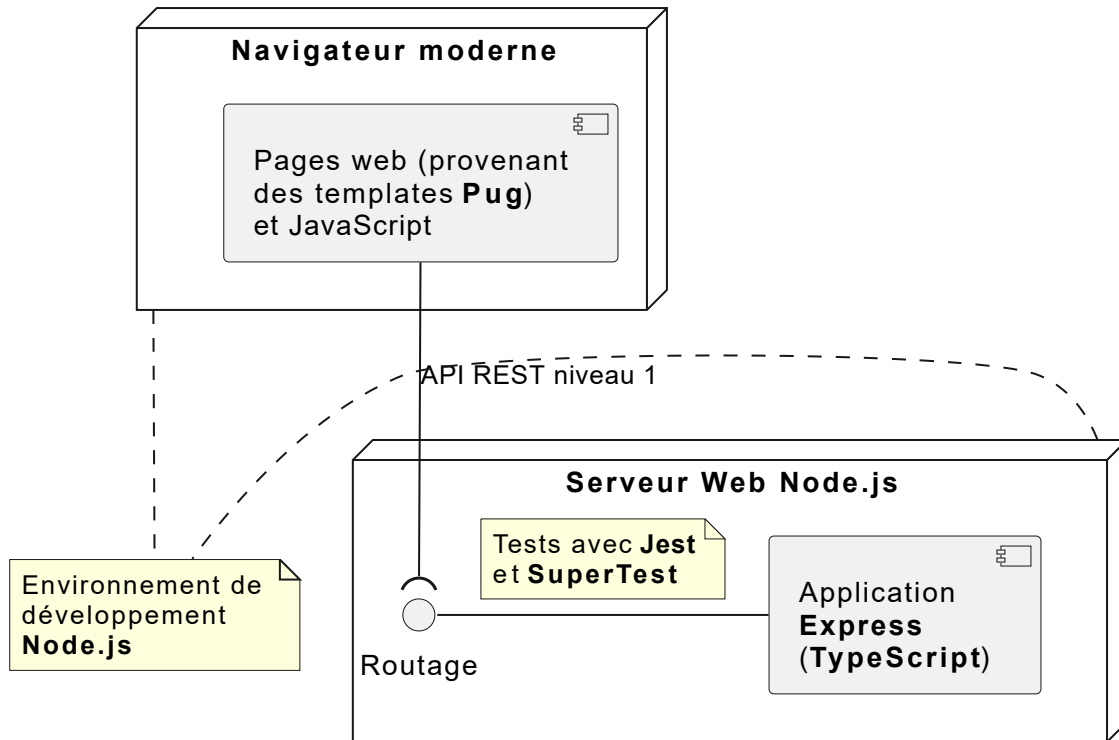
La date de remise du rapport et du code est **avant la séance 03 de laboratoire**. Notez que le calendrier des séances est différent pour chaque groupe-cours. Vérifiez avec votre auxiliaire d'enseignement (chargé.e de laboratoire).

Technologies vues dans cet exercice

Cet exercice permet de comprendre les bases des technologies suivantes:

- [Node.js](#)
 - [TypeScript](#)
 - [Express](#)
 - [Jest](#)
 - [SuperTest](#)
- [Pug](#) (anciennement Jade)

Le déploiement de la solution fonctionne comme le diagramme suivant:



Les bases de données et les cadriciels (Angular, React, etc.) sont des sujets traités dans d'autres cours des programmes de LOG et de GTI. Donc, vous ne pouvez pas utiliser ces technologies dans les laboratoires de LOG210.

Quant à la méthodologie de travail, ce tutoriel vous permettra aussi de savoir comment utiliser les outils suivants:

- [VisualStudio Code](#)
- [PlantUML](#) (pour les modèles UML)
- [GitHub Markdown](#) (pour la documentation)

Objectif de l'exercice

Cet exercice vous amènera à travers des étapes à ajouter une nouvelle fonctionnalité au **Jeu de dés**. Vous devez réaliser le cas d'utilisation *Redémarrer* qui va simplement redémarrer le jeu. Pour respecter le processus de génie logiciel enseigné dans LOG210, il faudra passer par les étapes suivantes:

- actualiser la documentation de la fonctionnalité (cas d'utilisation)
- actualiser des modèles de conception (diagrammes de séquence système et réalisations de cas d'utilisation);
- écrire des tests pour la fonctionnalité;
- écrire le code source;
- remettre (anglais *commit*) les changements.

Les auxiliaires d'enseignement pourront vous aider si vous avez des questions.

Étapes


Vous pouvez cocher chaque étape dans la liste suivante:

0. préparer votre machine pour ce tutoriel;

- ☒ jeter un œil sur la [documentation du squelette](#)
- ☒ installer [node.js](#) sur votre machine
- ☒ installer [VSCode](#) sur votre machine
- ☒ installer l'[extension PlantUML dans VSCode](#)
- ☒ configurer l'extension PlantUML pour utiliser plantuml.com comme serveur en ajoutant/modifiant les "User settings" en VSCode:

```
"plantuml.server": "https://www.plantuml.com/plantuml",  
"plantuml.render": "PlantUMLServer",
```

(Si vous avez les droits d'administrateur Windows) installer [GraphViz](#) pour utiliser un serveur local de PlantUML plutôt que le serveur sur Internet et configurer `"plantuml.render": "Local"` selon la [documentation](#).

- ☒ installer [GitHub Desktop](#) sur votre machine
- ☒ [cloner](#) le dépôt de code de ce laboratoire sur votre machine
 -  Ne pas cloner le code dans un chemin qui contient des espaces ou des accents.
 - Facultatif : Lire cette [Présentation de GitHub dans Visual Studio Code](#) de Microsoft.
- ☒ faire un build du code

- Ouvrir le dossier du projet en VSCode **File > Open Folder...**
- Dans VSCode, ouvrir un terminal dans le menu **Terminal > New Terminal**

Sur Windows, utiliser [Node.js Command Prompt](#) si les étapes suivantes ne fonctionnent pas

- Dans le terminal, taper `npm install` pour installer les bibliothèques node du projet (une fois seulement).
- Dans le terminal, taper `npm run build` pour compiler le code source.

Normalement, on devrait voir les messages comme:

```
...  
[2:52:01 p.m.] Projects in this build:  
    * tsconfig.json  
  
[2:52:01 p.m.] Project 'tsconfig.json' is out of date because output  
file 'dist/app.js' does not exist  
  
[2:52:01 p.m.] Building project 'C:/Users/Moi/Documents/GitHub/log210-
```

```
enonce-lab0/tsconfig.json'
...
```

-  exécuter les tests
 - Dans le terminal, taper `npx jest --colors lab0.test.ts` pour exécuter les tests du lab 0.

Normalement, on devrait voir les messages comme:

```
...
Test Suites: 16 failed, 16 total
Tests:       45 failed, 1 passed, 46 total
Snapshots:   0 total
Time:        9.908 s
Ran all test suites matching /lab0.test.ts/i.
```

Certains tests valident le contenu des fichiers rajoutés pour ce laboratoire (la documentation, la nouvelle fonctionnalité) et d'autres valident les fonctionnalités de base du squelette (dont certains vous aurez à modifier). Puisque vous venez de commencer le lab 0, c'est normal d'avoir beaucoup de tests "failed". Le but est de faire passer tous les tests au fur et à mesure que vous apprenez des aspects technologiques du laboratoire.

La rétroaction de ce laboratoire ne viendra pas d'un auxiliaire d'enseignement, car c'est un travail individuel. Ce sont des tests automatiques qui vont diagnostiquer les problèmes pour vous. **Si vous avez une difficulté que vous n'arrivez pas à résoudre, vous devez poser des questions aux auxiliaires d'enseignement.**

- Pour voir la liste des tests pour le lab0, taper `npx jest --listTests lab0.test.ts`
- Pour exécuter un test individuel, p. ex. `identification-lab0.test.ts` -- très utile pour ne pas avoir trop d'informations avec tous les tests -- taper `npx jest --colors identification-lab0.test.ts` (il n'est pas nécessaire de spécifier tout le chemin du fichier de test):

```
$ npx jest --colors identification-lab0.test.ts
FAIL test/squelette/identification-lab0.test.ts
  README identification
    × devrait trouver votre nom (2 ms)
    × devrait trouver votre courriel
    × devrait trouver Votre code moodle
    × devrait trouver votre compte github

  ● README identification › devrait trouver votre nom

    expect(received).toBeFalsy()

    Received: true

    11 | describe('README identification', () => {
```

```

12 |   it('devrait trouver votre nom', () => {
> 13 |     expect(content.includes("Entrer votre
nom")).toBeFalsy();
    |
    | ^
14 |   });
15 |
16 |   it('devrait trouver votre courriel', () => {

    at Object.<anonymous> (test/squelette/identification-
lab0.test.ts:13:50)

```

- README identification > devrait trouver votre courriel

```
expect(received).toBeFalsy()
```

```
Received: true
```

```

15 |
16 |   it('devrait trouver votre courriel', () => {
> 17 |     expect(content.includes("Entrer votre
courriel")).toBeFalsy();
    |
    | ^
18 |   });
19 |
20 |   it('devrait trouver Votre code moodle', () => {

    at Object.<anonymous> (test/squelette/identification-
lab0.test.ts:17:55)

```

- README identification > devrait trouver Votre code moodle

```
expect(received).toBeFalsy()
```

```
Received: true
```

```

19 |
20 |   it('devrait trouver Votre code moodle', () => {
> 21 |     expect(content.includes("Entrer votre code moodle obtenu
à partir de Signets")).toBeFalsy();
    |
    | ^
22 |   });
23 |
24 |   it("devrait trouver votre compte github", () => {

    at Object.<anonymous> (test/squelette/identification-
lab0.test.ts:21:85)

```

- README identification > devrait trouver votre compte github

```
expect(received).toBeFalsy()
```

```
Received: true
```

```

23 |
24 |   it("devrait trouver votre compte github", () => {
> 25 |     expect(content.includes("Entrer l'identifiant de votre
    |     compte github")).toBeFalsy();
    |
^
26 |   });
27 | });
28 |

    at Object.<anonymous> (test/squelette/identification-
lab0.test.ts:25:77)

Test Suites: 1 failed, 1 total
Tests:       4 failed, 4 total
Snapshots:   0 total
Time:        1.34 s, estimated 2 s

```

Une croix (rouge), p. ex. ✖ **devrait trouver votre nom** signifie qu'un test échoue, tandis qu'un test réussi s'affiche avec une coche ✓ (verte).

- ☒ exécuter le serveur sur localhost
 - Dans le terminal, **npm start**
- ☒ exécuter l'application avec un navigateur moderne à l'URL <http://localhost:3000>
 - ☒ démarrer une partie avec un joueur
 - ☒ essayer le bouton pour jouer
 - ☒ regarder la page Classement
 - ☒ revenir à la page d'accueil pour voir le jeu
- ☒ regarder dans VSCode la structure des pages statiques dans **views/** ainsi que la barre de navigation (**views/includes/navbar.pug**) qui est **include** dans les pages statiques (**views/index.pug, views/stats.pug**).
- ☒ regarder comment dans **src/app.ts** un gabarit Pug **views/index.pug** est utilisé:

```

// Route pour jouer (index)
router.get('/', (req, res, next) => {
  res.render('index',
    // passer objet au gabarit (template) Pug
    {
      title: `${titreBase}`,
      user: user,
      joueurs: JSON.parse(jeuRoutes.controleurJeu.joueurs)
    });
});

```

- La **barre de navigation** `views/includes/navbar.pug` provient de Bootstrap. Elle est personnalisée selon les valeurs dans l'objet (`user`) passé grâce à la fonctionnalité de **mixins de Pug**. Cet objet est initialisé plus haut dans `src/app.ts`:

```
// Extrait de src/app.ts
user = { nom: 'Pierre Trudeau', hasPrivileges: true, isAnonymous: false
};
```

Cet objet sera un mécanisme de gérer les vues. Par exemple, si `objet.isAnonymous` est vrai, `navbar.pug` n'affiche pas de lien, car l'utilisateur n'est pas connu et donc doit se connecter pour avoir plus d'accès.


Dans cet exercice, vous n'avez pas à gérer les connexions, mais sachez que le squelette supporte cette dimension. Ça sera utile pour votre projet en équipe.

- la technologie Bootstrap est chargée dans un en-tête (`views/includes/head.pug`) qui, lui, est inclus dans chaque page statique. Sachez que les versions de Bootstrap ont évolué assez rapidement et il faut faire attention à la version si vous trouvez un exemple intéressant que vous voulez réutiliser. La documentation de Bootstrap est excellente, donc vérifiez toujours avec ça.
- les pages statiques définies avec Pug permettent de créer facilement une interface humain-machine, mais il doit y avoir des routes définies dans `src/app.ts` pour que les fichiers `.pug` soient bien rendus au navigateur.
- repérer dans le fichier `src/app.ts` la définition des *route handlers* pour les pages statiques, p. ex. `/` vers `views/index.pug` et `/stats` vers `views/stats.pug`. Cela est important pour étendre le squelette dans le cadre du projet en équipe. Il y a aussi des pages statiques pour aider avec la gestion des connexions (`views/signin.pug` et `views/signout.pug`) qui sont intégrées également dans le squelette (routes et barre de navigation).
- noter que la syntaxe des fichiers Pug est sensible à l'indentation (comme en Python). Si vous trouvez un exemple de code Pug à intégrer dans votre projet, l'éditeur VSCode va (par défaut) utiliser une indentation de 4 espaces, tandis que beaucoup d'exemples sur Internet utilisent 2 espaces.

1. Actualiser la documentation de la fonctionnalité

Note: il est fortement recommandé de faire un commit et push du code **sur la branche master** (puisque le travail est individuel) au moins à la fin de chaque étape à partir de maintenant. Les auxiliaires d'enseignement auront accès à votre dépôt de code source et pourraient vous aider (surtout à distance) si votre code est synchronisé souvent avec le dépôt. Rappel: [Présentation de GitHub dans Visual Studio Code](#).

La documentation des fonctionnalités se trouve dans le fichier `docs/Squelette.md`. Dans cette étape, vous devez:

-  dans le fichier `docs/Squelette.md` qui sert de documentation, ajouter les informations pour vous identifier dans la section **Identification de l'étudiant**.

Vérifier le travail avec le test, `npx jest --colors identification-lab0.test.ts`

- ☒ dans le fichier `docs/Squelette.md`, ajouter le cas d'utilisation *Redémarrer* (texte) juste après le texte du cas d'utilisation *Jouer aux dés*:

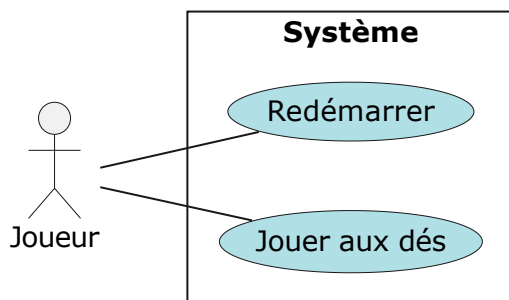
```
#### Redémarrer
```

1. Le Joueur demande à redémarrer l'application.
2. Le Système termine tous les jeux en cours et redémarre l'application.

- ☒ ajouter le cas d'utilisation au diagramme UML (PlantUML)
 - installer l'extension `PlantUML` dans VSCode
 - visionner [cette vidéo](#) (*activer les sous-titres en français*) pour savoir comment créer et modifier un diagramme PlantUML
 - modifier `docs/modeles/dcu.puml` pour inclure un nouveau cas d'utilisation (*Redémarrer*) as `R #powderblue` et le lien avec l'acteur `J -- R` comme dans l'exemple partiel suivant:

```
...
rectangle "Système" {
  (Jouer aux dés) as JP #powderblue
  (Redémarrer) as R #powderblue
  J -- JP
  J -- R
}
...
```

Vous devriez voir un diagramme comme ceci:



► Facultatif : faire en sorte que la documentation soit correctement liée aux fichiers `.puml`

Question: Comment faire en sorte que `docs/Squelette.md` affiche la version modifiée du fichier `docs/modeles/dcu.puml` après un *push* des fichiers vers GitHub? *Réponse:* Il faut modifier le markdown suivant dans `docs/Squelette.md`:

```
### Diagramme de cas d'utilisation
```



```
![[Diagramme de cas d'utilisation](http://www.plantuml.com/plantuml/proxy?cache=no&fmt=svg&src=https://raw.githubusercontent.com/profcfuhrmanets/log210-jeu-de-des-node-express-ts/master/docs/modeles/dcu.puml)
```

La partie de l'URL `src=https://raw.githubusercontent.com/profcfuhrmanets/log210-jeu-de-des-node-express-ts/master/docs/modeles/dcu.puml` doit pointer sur le fichier dans *votre* dépôt plutôt que sur celui duquel ce projet a été copié.

Pour obtenir l'URL "raw" d'un fichier sur GitHub, naviguez vers le fichier, cliquez sur le bouton **Raw**, puis copiez l'URL du navigateur. Finalement, cela doit être fait une seule fois pour chaque diagramme `.puml` et les changements vont suivre automatiquement dans la documentation. Cependant, il peut prendre quelques minutes avant que le cache du navigateur se rafraîchisse. Pour en savoir plus, lire [la question sur StackOverflow](#).

► Facultatif : faire en sorte que la documentation soit correctement liée aux fichiers `.svg`

Question: Comment faire en sorte que `Squelette.md` affiche la version modifiée du fichier `docs/modeles/dcu.puml` après un *push* des fichiers vers GitHub? *Réponse:* Il faut exporter les diagrammes puml à l'aide du menu contextuel "Export current file diagrams" et modifier le markdown suivant dans `docs/Squelette.md`:

```
### Diagramme de cas d'utilisation

![[Diagramme de cas d'utilisation]
(docs/modeles/dcu/Diagramme%20de%20cas%20d'utilisation.svg)
```

L'avantage de cette méthode est que votre documentation locale est immédiatement à jour.

Vérifier le travail avec le test, `npx jest --colors dcu-puml-lab0.test.ts`:

```
PASS test/modeles/dcu-puml-lab0.test.ts
  docs/modeles/dcu.puml
    ✓ devrait contenir (Redémarrer) as R #powderblue (1 ms)
    ✓ devrait contenir J -- R (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.375 s
Ran all test suites matching /dcu-puml-lab0.test.ts/i.
```

Repérer et valider le(s) test(s) individuellement à la fin de chaque étape.

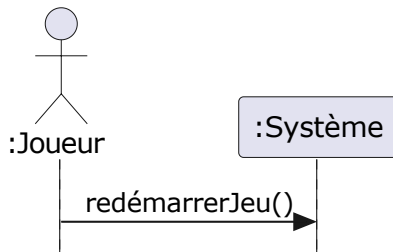
2. Actualiser des modèles de conception (diagrammes de séquence système et réalisations de cas d'utilisation)

Les modèles de conception guident l'implémentation. Vous aurez à revenir à cette section durant ce tutoriel.

- ☒ ajouter un nouveau DSS `docs/modeles/dss-redemarrerJeu.puml` pour le cas d'utilisation (PlantUML)

Faire un diagramme en PlantUML qui ressemble à ceci:

DSS pour le scénario *Redémarrer*



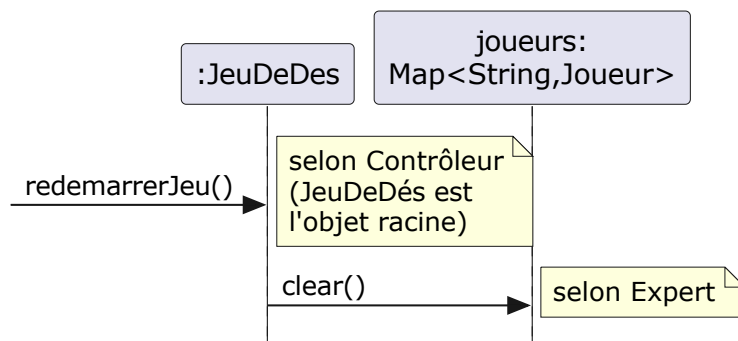
- ☒ prendre connaissance de la postcondition du contrat d'opération :

Toutes les instances de Joueur en cours ont été supprimées.

- ☒ ajouter une nouvelle réalisation de cas d'utilisation (RDCU) `docs/modeles/rdcu-redemarrerJeu.puml` pour l'opération système `redemarrerJeu()` (PlantUML)

Faire un diagramme en PlantUML qui ressemble à ceci:

RDCU pour redemarrerJeu



Valider avec les tests `test/modeles/rdcu-redemarrerJeu-puml-lab0.test.ts` et `test/modeles/dss-redemarrer-puml-lab0.test.ts`.

À partir de maintenant, consultez le code existant pour vous aider à compléter les étapes

3. Écrire des tests pour la fonctionnalité

- ☐ ajouter de nouveaux cas de test pour Redémarrer (Jest/SuperTest)
 - ouvrir le fichier `test/routes/jeuRouter-redemarrerJeu-lab0.test.ts`
 - y créer une suite de tests nommée `'GET /api/v1/jeu/redemarrerJeu'` avec `describe`
 - Tous les tests reliés au cas d'utilisation y seront contenus
 - y utiliser `beforeAll` pour créer deux joueurs avant l'exécution des tests

Cette méthode s'exécutera avant tous les tests, pour satisfaire la précondition du cas d'utilisation (un joueur doit exister)

- y créer un test afin de tester le scénario principal (succès) avec `it` et le nommer adéquatement

Le test doit appeler la route `GET /api/v1/jeu/redemarrerJeu`, qui correspond à l'opération système `redemarrerJeu` du DSS. Puisque l'opération n'a aucun retour, le test doit uniquement valider le succès de l'opération, c'est-à-dire, que le code HTTP (`status`) est 200 et que la réponse est du `JSON`.

- y créer un autre test pour valider la postcondition du contrat d'opération.

Le test doit vérifier qu'il n'y a plus de joueurs.

- ☒ vérifier que les tests ne passent pas (Jest/SuperTest) `npx jest --colors jeuRouter-redemarrerJeu-lab0.test.ts` va indiquer `n failed`

Cela est normal, car nous avons écrit plusieurs tests avant d'avoir écrit les fonctionnalités, selon la pratique de *Développement piloté par les tests*.

4. Écrire la fonctionnalité

- [x] ajouter l'opération système `redemarrerJeu` dans le contrôleur GRASP `src/core/jeuDeDes.ts` (TypeScript)

Cette méthode correspond à l'opération système (unique) définie dans le diagramme de séquence système (DSS).

- [x] coder l'opération `redemarrerJeu` selon la RDCU (TypeScript)

Pour la logique du code, consulter le diagramme de séquence (RDCU) créé à l'étape précédente.

- [x] ajouter une nouvelle route, dans `src/routes/jeuRouter.ts` (Express)
 - ajouter la fonction `redemarrerJeu`, juste avant la fonction `init()`

Elle doit :

- avoir les mêmes paramètres que les autres fonctions.
- appeler l'opération système créée précédemment
- afficher à l'utilisateur (`flash`) que l'application redémarre
- retourner à l'utilisateur l'état du système, le code HTTP 200 (Ok) et le `message` :
`'Success'`

- ajouter, à la fin de la fonction `init()`, une route `GET /redemarrerJeu` pour lier l'URI à `redemarrerJeu`

Il s'agit d'une *définition de route* dans *Express*. Lorsqu'il y aura une requête HTTP `GET` avec `api/v1/jeu/redemarrerJeu`, la fonction `redemarrerJeu` dans la même classe sera appelée (*callback*). Cette fonction est aussi appelée un *route handler* en anglais.

- [x] faire un build (Node.js)


`npm run build` devrait passer sans erreurs. Si vous avez des erreurs, essayer de lire et de comprendre pourquoi. Si vous êtes bloqués pendant plus de 5 minutes, demandez de l'aide à un auxiliaire d'enseignement.

- [x] vérifier que les tests pour la nouvelle fonctionnalité redémarrerJeu passent: `npx jest --colors jeuRouter-redemarrerJeu-lab0.test.ts`. (Node.js)
- [x] vérifier que TOUS les tests des fonctionnalités de base passent (Node.js)

`npx jest --colors --coverage --testPathIgnorePatterns=lab0.test.ts` devrait indiquer que tous les tests passent.

- [x] ajouter le bouton dans `views/index.pug` (PugJS.org)

Facultatif : pour une explication de PUG (anciennement Jade) avec Express, il y a [cette vidéo](#).

Dans `views/index.pug` après le texte ici, ajouter la ligne `button#redemarrer Redémarrer` ( attention à l'indentation):

```
ul.entries
  each joueur in joueurs
    li.joueur
      -var nom = joueur.nom;
      strong(id=nom) #{nom}
      = ' '
      | tentatives: #{joueur.lancers}, réussites: #
{joueur.lancersGagnes}
      = ' '
      button.lancer(id=nom) Lancer dés
      = ' '
      button.terminer(id=nom) Terminer
    else
      li
        em Pas de joueurs encore.

button#redemarrer Redémarrer
```

- [x] ajouter le JavaScript pour le bouton afin d'invoquer le nouveau service

Dans `public/lib/main.js` on trouve le code pour les boutons. Après la logique pour traiter le clic sur le bouton *Démarrer* (`demarrer.addEventListener("click", function(){...});`), ajouter une nouvelle logique pour le bouton *Redémarrer* qui fait un `GET` sur `/api/v1/jeu/redemarrerJeu`:

```
document.getElementById("redemarrer").addEventListener("click", function ()
{
  fetch("/api/v1/jeu/redemarrerJeu")
  .then(function()
  {
    location.reload();
```

```
});
});
```

Refaire le build et relancer le serveur dans le terminal. Recharger la page web et vérifier que le bouton fonctionne comme il le faut en créant une nouvelle partie pour un joueur et ensuite cliquant sur *Redémarrer*.

5. Afficher le classement sur nouvelle page

Il existe un lien dans la barre de navigation «Classement» pour la page `/stats`. Cependant, cette page n'affiche pas la colonne Ratio, car l'information n'est pas encore calculée.

- [x] Modifier le *route handler* dans `src/app.ts` et le gabarit `view/stats.pug` pour que le ratio se calcule et s'affiche. La classe `src/core/Joueur.ts` ne contient pas de propriété `ratio`, mais on peut la calculer dans le *route handler*. Il faut passer un nouveau tableau de joueurs, mais les objets doivent contenir une propriété `ratio` qui est le nombre de succès divisé par le nombre de tentatives. [Astuce sur stackoverflow](#).

```
const joueurs: Array<Joueur> = JSON.parse(jeuRoutes.controleurJeu.joueurs);
const joueursAvecRatio = /* à compléter en ajoutant joueur.ratio */;
res.render('stats',
  // passer objet au gabarit (template) Pug
  {
    title: `${titreBase}`,
    user: user,
    // créer nouveau tableau de joueurs qui est trié par ratio
    joueurs: joueursAvecRatio /* à modifier */
```

Pour afficher le ratio, il faut modifier le gabarit `views/stats.pug` (la ligne à modifier est en commentaire):

```
each joueur in joueurs
  tr
    td #{joueur.nom}
    td(style="text-align: right") #{joueur.lancers}
    td(style="text-align: right") #{joueur.lancersGagnes}
    //- td(style="text-align: right; font-family: monospace") #
    {joueur.ratio.toFixed(8)}
```

- [x] Trier le tableau de `joueursAvecRatio` pour que le classement s'affiche en ordre décroissant par ratio. [Astuce sur stackoverflow](#).

Puisqu'il s'agit simplement d'une nouvelle vue sur les informations déjà présentes dans le système, on ne doit pas faire une RDCU. C'est-à-dire qu'on ne modifie pas *la logique d'affaires* ou l'état des objets du domaine.

6. Documenter les classes logicielles

- [x] Générer un diagramme de classes pour la solution avec le script `npm run uml-classes-puml` qui utilise l'outil `tplant`. Par défaut, le diagramme `App.puml` est placé dans le répertoire `docs/modeles`. Chaque fois que vous modifiez votre code source, c'est une bonne idée d'actualiser ce diagramme avec ce script.
- [x] Visualiser le fichier généré `App.puml` dans VS Code avec l'extension de PlantUML. Comprendre le design du jeu de dés.
- [x] Intégrer ce diagramme dans votre rapport `Squelette.md` dans une section nommée **Diagramme de classes logicielles**.

7. Pratiquer ce qui a été appris

- [x] Modifier le cas d'utilisation *Jouer aux dés* pour que le joueur lance **trois** dés plutôt que deux et la condition pour gagner soit que le *total soit inférieur ou égal à 10*. Il faut passer par toutes les étapes, y compris modifier le MDD, les contrats, les RDCU, les tests et le code. Cependant, cette fois-ci vous devez vous débrouiller, en vous référant à des étapes plus haut.

⚠ Certains tests de base du squelette lancés par `npx jest --colors --coverage --testPathIgnorePatterns=lab0.test.ts` ne seront plus bons, car ils valident les fonctionnalités du jeu avec **deux** dés. Il faut les modifier pour la nouvelle fonctionnalité avec **trois** dés. Certains de ces tests sont plus difficiles à faire passer, notamment ceux dans `test/core/jeuDeDes.test.ts` qui valident les valeurs retournées par la méthode `brasser()`. En fait, la **probabilité d'avoir certaines valeurs (ex. 3 et 18) avec trois dés** est faible (ex. $1/216 \approx 0.5\%$). Il faut donc faire plus d'essais (jusqu'à 2000?) pour obtenir toutes les valeurs dans le test. Faire autant d'essais dans un test n'est pas idéal puisque ça prend du temps. Il serait plus efficient si on utilisait des éléments de remplacement (un dé *temporaire* qui retourne toujours une même valeur), mais cette façon de faire (parfois appelée des **mocks**) est hors du cadre du cours.

8. Générer le rapport en format PDF

- [x] Assurez-vous de générer une version PDF de votre fichier `docs/Squelette.md` nommé `docs/lab0.pdf`

Un menu contextuel devrait vous permettre de réaliser cette tâche dans Visual Studio Code si vous avez installé les bonnes extensions. Il se peut que vous ayez à installer d'autres modules (un *exporter* pour Chromium). Il se peut que le fichier soit créé dans un répertoire `out/docs/` aussi avec le nom `Squelette.pdf`.

9. Apprendre à faire face aux parasites et aux mollassons dans une équipe

Il peut arriver qu'une équipe soit composée des personnes qui travaillent beaucoup moins que les autres. Pour vous sensibiliser aux problèmes typiques et vous outiller à agir rapidement en cas de difficultés, il y a un texte à lire et à intérioriser.

- [x] Lire le texte **Faire face aux parasites et aux mollassons dans une équipe**
- [x] Écrire une réponse dans le fichier `docs/experience-parasites-mollassons.md` du dépôt suivant les directives dans ce fichier.

10. Vérifier la correction automatique

Cet exercice sera noté quasi automatiquement lorsque vous transférez votre code dans GitHub Classrooms. Les tests associés à ce projet permettent de vérifier que la majorité des modifications que vous deviez réaliser ont été faites. Il y a deux volets de la correction automatique avec les tests:

- Documentation et fonctionnalités rajoutées: `npx jest --colors lab0.test.ts` (exécuter tous les tests ayant `lab0.test.ts` dans le nom)
- Fonctionnalités de base du squelette: `npx jest --colors --coverage --testPathIgnorePatterns=lab0.test.ts` (exécuter tous les tests **n'ayant pas** `lab0.test.ts` dans le nom et mesurer la couverture)

Assurez-vous qu'aucun test n'est en échec et que la couverture de test (**branches**) est de 100 % (la couverture donne quelques points bonis) pour la commande `npx jest --colors --coverage`.

Important: La seule rétroaction pour ce laboratoire est à travers les commandes ci-dessus. Vous n'aurez pas de rétroaction individuelle après la date de remise, alors si vous avez des tests qui ne passent pas ou des questions concernant les technologies, **c'est votre responsabilité de demander de l'aide de l'auxiliaire de laboratoire avant la remise.**

11. Remettre (anglais *commit*) tous les changements sur la branche **master**

- ☐ À l'aide de Zoom ou de tout autre outil d'enregistrement vidéo, enregistrez une démonstration du fonctionnement de l'interface utilisateur avec au moins 2 joueurs. Sauvegarder le résultat dans le fichier **demo.mp4** et placer ce fichier dans le même répertoire que README.md.
- ☐ Faire une remise de la solution du projet (GitHub) incluant votre réponse dans `docs/experience-parasites-mollassons.md`

Félicitations! Vous avez réussi les défis technologiques nécessaires pour être performant dans les laboratoires de LOG210! Ce tutoriel vous sera sûrement utile pendant le développement du projet itératif à suivre, car il y a des [liens pour la documentation des technologies différentes](#).

Si vous avez terminé rapidement grâce à votre expérience, pensez à aider vos coéquipiers qui pourraient toujours avoir des questions. Mais ne faites pas le travail à leur place, car le but est que toute l'équipe soit performante sur le plan technologique. Cherchez à augmenter le facteur de bus (voir les notes de cours pour l'explication) de l'équipe! En plus, le mentorat est une caractéristique importante du leadership.

Calcul de la note

Le calcul de la note du laboratoire se fait à partir des résultats des tests automatiques roulés avec la commande `npx jest --coverage` (au début du projet et à la fin du projet) et une évaluation faite par l'auxiliaire d'enseignement:

variable	explication
\$e\$	10 points si vous avez complété un texte d'au moins 300 mots dans <code>docs/experience-parasites-mollassons.md</code> (sinon 0 point)
\$b\$	nombre de nouveaux boutons fonctionnels (max 1) ^[1]
\$c\$	10 points si la page de classement fonctionne correctement (sinon 0 point) ^[1]

variable	explication
\$C\$	10 points (bonis) multipliés par la couverture (%) de test de toutes les branches ^[1]
\$v\$	nombre de tests valides ("passed")
\$d\$	nombre de tests déjà valides au début du projet
\$t\$	nombre total de tests

[^1]: évaluation faite par l'auxiliaire d'enseignement

$$\text{Note} = \frac{e + b + c + C + v - d}{10 + 6 + 10 + t - d} 100$$