Conception d'un jeu d'échecs

Pendant ce TP, nous allons créer une application web frontend permettant de jouer aux échecs. Son développement sera l'occasion de mettre en œuvre une méthodologie *Devops* avec des outils et frameworks tels que *git*, *jenkins*, *playwright*, ou encore *vue.js*.

Vous voulons amorcer un projet d'application web, uniquement *frontend* (il n'y aura pas dans un premier temps de partie *backend*) qui se base sur le framework **vue.js**.

Au début, cette application se contentera d'afficher le plateau d'un jeu d'échecs et de permettre d'y déplacer les pièces librement.

Pour gagner du temps, et étant donné que vous connaissez déjà **vue.js**, nous allons utiliser une IA agentique qui nous permettra de faire le gros du travail. Voici un exemple de *prompt* pour initialiser le projet :

> En utilisant le framework vue.js, initialise un projet d'application web (uniquement frontend) permettant d'afficher le plateeau d'un jeu d'échec avec toutes les pièces (noirs et blancs). Le joueur doit pouvoir déplacer les pièces librement, sans respecter les règles.

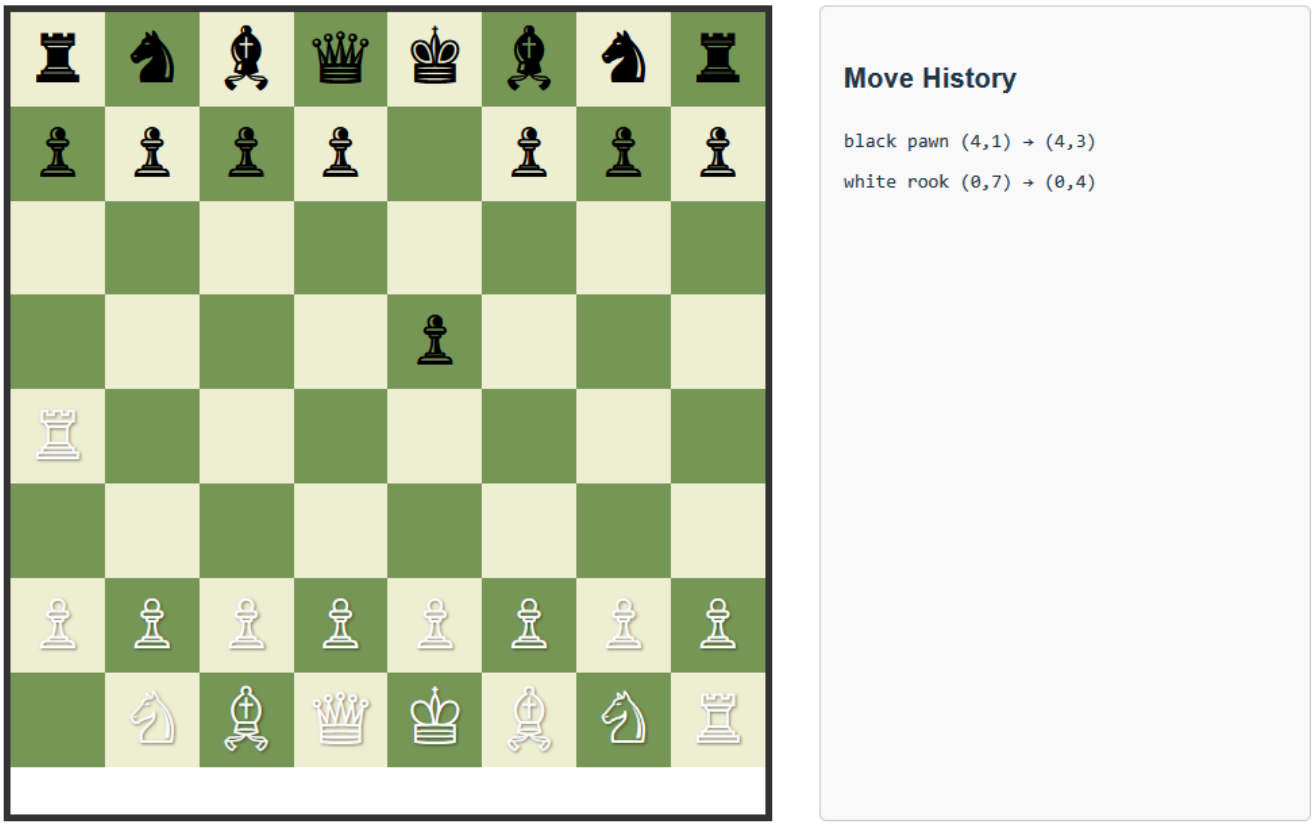
Selon le plan d'implémentation prévu par l'IA, on sera peut-être amené à préciser ce qui se doit se passer qand une pièce est déplacée sur un emplacement occupée par une autre pièce :

> Les pièces doivent pouvoir être déplacées n'importe où, y compris sur des cases déjà occupées, auquel cas la pièce remplace celle occupant l'emplacement.

Enfin, afin de faciliter l'écriture de tests, c'est une bonne idée de demander à l'IA de regrouper une partie de la logique de l'application dans une classe *service* par exemple avec le *prompt* suivant :

> J'aimerais aussi que tu ajoutes une classe service qui permette de connaitre l'emplacement de chaque pièce ainsi que l'historique des déplacements

Les *prompts* donnés ci-dessus, ne sont que des indications, à vous de les adapter en fonction de votre environnement, sans perdre de vue l'objectif d'avoir un plateau sur lequel on peut déplacer librement les pièces d'un jeu d'échecs. A la fin, votre application soit se lancer avec la commande `npm run dev` et ressembler à la capture écran ci-dessous :



Vous noterez que l'IA a ajouté l'affichage de l'historique des déplacements, ce qui n'était pas demandé dans les objectifs, mais si c'est aussi votre cas, vous pouvez le laisser.

Vérifiez les fichiers créés par l'IA. Il doit en particulier exister des composants `.vue` pour l'UI et une classe service pour la logique de placement des pièces.

Placement du projet sous GIT

Le projet étant initialisé, placez le sous GIT avec les commandes adéquates, ou demandez à l'IA de le faire avec par exemple le *prompt* suivant :

> Place le projet sous GIT avec tous ses fichiers, et réalise le commit initial

Raccordez ensuite votre projet à un dépôt distant (*Gitlab*, *GitHub*, *bitbucket* ou autres) et faites un *push* du projet.

Ajout de tests unitaires

Testez votre application manuellement en déplaçant les pièces. Si les tests manuels passent, il est temps d'écrire des tests automatisés.

On commence en général par des tests unitaires qui sont des tests qui travaillent au niveau des méthodes élémentaires de votre projet. Si l'IA a bien fait son travail, la logique métier de notre application (qui se résumé pour l'instant au placement des pièces) est regroupée dans une classe *service*.

Ajoutez des tests pour tester les méthodes de la classe service :

> Ajoute au projet des tests qui vérifient que les méthodes de la classe ChessService fonctionnent correctement.

En principe, l'IA doit vous proposer d'utiliser la librairie *vitest* qui est bien adaptée aux applications web. Si ce n'est pas le cas, affinez votre prompt pour aiguiller l'IA vers cette librairie.

Cela doit se traduire par la création d'un nouveau fichier javascript qui doit tester chacune des méthodes de la classe *service* en vérifiant par exemple qu'un pièce déplacée a un nouvel emplacement, qu'elle remplace bien une autre pièce si l'emplacement était déjà occupé, et que l'historique est bien mis à jour.

Vérifiez que tous les tests passent avec la commande `npm run test`.

Si tout est ok, réalisez un nouveau *commit* de votre projet.

Ajout de tests UI

Les test unitaires ne disent rien sur le comportement IHM de votre application. Pour tester cet aspect, il faut automatiser son exécution dans un navigateur et que l'emplacement des composants clés à l'écran soit vérifié.

Plusieurs *framework* de tests permettent de faire cela. Nous allons utiliser ici *playwright*.

Demandez à l'IA d'ajouter ce type de tests sur le composant principal (celui qui affiche la plateau)

> Utilise Playwright pour créer des tests sur le composant ChessBoard.

Si l'IA travaille bien, les dépendances nécessaires doivent être ajoutées au projet et un nouveau fichier doit être créé dans un dossier `test`. Ce fichier contient des méthodes javascript qui testent l'emplacement initial des pièces sur le plateau, et les actions que l'utilisateur peut faire à partir de cette configuration initiale, donc pour l'instant le *drag and drop* des pièces.

Vérifiez que tous les tests passent avec la commande `npm run test:e2e` (à modifier en fonction du contenu du fichier `package.json`)

Ne soyez pas surpris de ne pas voir un navigateur qui s'ouvre pour les tests, les tests sont réalisés dans un environnement *headless*, dont sans affichage, mais c'est bien un moteur de navigateur web qui les exécutent.

Si tout est ok, et afin de nous placer dans un cas plus réel d'utilisation de *Git*, ajoutez une branche `dev` à votre projet (`git branch dev`) et placez-vous dessus (`git checkout dev`).

Faites un *commit* de votre projet sur la branche `dev`. Vérifiez que tous les tests passent (les manuels, les unitaires, et les UI) et si c'est le cas, fusionnez la branche dev avec la branche principale avec les commandes :

```
git checkout main
git merge dev
```

puis remettez-vous sur la branche `dev` avec un `git checkout dev`

L'idée est donc d'utiliser la branche `dev` pour les nouveaux développements, puis quand tout est ok, d'intégrer dans la branche principale les nouvelles modifications.

Pensez à effectuer à chaque fois les `push` nécessaires pour intégrer dans le dépôt distant les dernières modifications.

2. Automatisation avec Jenkins

Nous allons utiliser *Jenkins* pour pouvoir automatiser les phases de *build* et de test du projet en construction.

Installation de Jenkins

Jenkins est constitué d'une partie serveur qui a pour but de surveiller les nouveaux *commit* sur des projets, et d'une partie *agents* à qui le serveur délègue le soin de réaliser les actions que l'on souhaite automatiser. Nous allons ici installer l'ensemble des composants sur votre machine de travail. Sachez qu'en temps normal, agents et serveur peuvent tourner sur des machines différentes.

Pour installer *Jenkins* de façon simple nous utilisons ici une image *docker* customisé pour y intégrer certains *plugins*.

Même si la partie permettant de lancer *Jenkins* est totalement indépendante de notre projet d'échecs, pour des raisons de praticité, nous allons placer les fichiers *docker* dans l'espace du projet.

Créez à la racine du projet du jeu d'échecs, un dossier `jenkins` avec pour contenu deux fichiers :

Un fichier `Dockerfile` avec le contenu suivant :

```
FROM jenkins/jenkins:2.528.3-jdk21
USER root
RUN apt-get update && apt-get install -y lsb-release ca-certificates curl && \
    install -m 0755 -d /etc/apt/keyrings && \
    curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
&& \
    chmod a+r /etc/apt/keyrings/docker.asc && \
    echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] \
    https://download.docker.com/linux/debian $(. /etc/os-release && echo
\"$VERSION_CODENAME\") stable" \
    | tee /etc/apt/sources.list.d/docker.list > /dev/null && \
    apt-get update && apt-get install -y docker-ce-cli && \
    apt-get clean && rm -rf /var/lib/apt/lists/*
USER jenkins
RUN jenkins-plugin-cli --plugins "blueocean docker-workflow json-path-api"
```

et un fichier `docker-compose.yml` avec le contenu suivant :

```
services:
    docker:
        image: docker:dind
        privileged: true
        restart: always
        ports:
            - "2376:2376"
        volumes:
            - jenkins-docker-certs:/certs/client:rw
            - jenkins-data:/var/jenkins_home:rw
        environment:
            - DOCKER_TLS_CERTDIR=/certs
    jenkins:
        build: .
        restart: always
        ports:
            - "8080:8080"
            - "50000:50000"
        volumes:
            - jenkins-data:/var/jenkins_home:rw
            - jenkins-docker-certs:/certs/client:ro
        environment:
```

```
            - JAVA_OPTS="-Dhudson.model.DirectoryBrowserSupport.CSP="
            - DOCKER_HOST=tcp://docker:2376
            - DOCKER_CERT_PATH=/certs/client
            - DOCKER_TLS_VERIFY=1
volumes:
    jenkins-data:
    jenkins-docker-certs:
```

Lancez *Jenkins* avec la commande `docker compose up -d`

Pour l'éteindre, si besoin, vous utiliserez la commande `docker compose down`.

Procédez à la phase d'enregistrement dans votre serveur *Jenkins* personnel en connectant votre navigateur à l'adresse http://localhost:8080/. Cette phase est à faire une seule fois par serveur déployé.

Premier `Jenkinsfile`

Nous allons bientôt créer une automatisation Jenkins sur le serveur que nous venons de déployer. Les spécifications de cette automatisation seront définis dans un fichier de notre projet : `Jenkinsfile`. Ce fichier doit être placé à la racine du projet et être présent dans toutes les branches *Git*.

Vérifiez que vous êtes bien sur la branche `dev`, et créez un nouveau fichier appelé `Jenkinsfile` contenant le texte suivant :

```
pipeline {
    agent any
    stages {
        stage('step1') {
            steps {
                sh 'echo étape un'
            }
        }
        stage('step2') {
            steps {
                sh 'echo étape deux'
            }
        }
    }
}
```

Le pipeline ainsi défini servira juste à tester le bon fonctionnement de Jenkins. Nous le remplacerons par quelque chose de plus opération plus tard.

Faites un *commit* du projet, puis fusionnez la branche `dev` avec la branche principale afin que le fichier `Jenkinsfile` soit présent dans les deux branches.

Faîtes un push de toutes les modifications sur le dépôt distant.

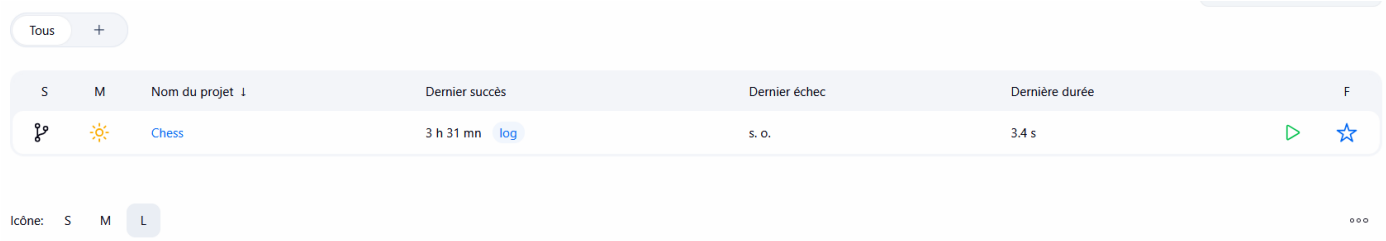
Création d'une automatisation Jenkins

Sur l'application web de Jenkins, cliquez sur le menu `Nouvel Item` pour créer votre première automatisation liée au projet. Donnez un nom à l'item (par exemple *Chess*) et dans le choix du type de l'item, sélectionnez `Pipeline Multibranches`.

Sur l'écran qui suit, vérifiez bien que vous êtes dans une configuration de build de type *by Jenkinsfile* et ajoutez une source pour que Jenkins puisse accéder à votre projet sous *Git* (choisissez le type *Git* pour la source). Renseignez l'URL https menant à votre dépôt Git distant, ainsi que les *credentials* nécessaire s'il s'agit d'un dépôt privé. Laissez les autres informations par défaut.

Cliquez sur le bouton `Save` pour finaliser la création.

En principe un nouveau projet doit apparaître sur la page d'accueil de Jenkins.



Si vous cliquez dessus, vous devez voir apparaître les deux automatisations créés par Jenkins, une pour la branche `dev`, l'autre pour la branche principale.

Lors de la première création, Jenkins lance automatiquement les deux automatisations. Vous pouvez ensuite les exécuter manuellement et séparément, en cliquant sur le symbole de *run* à droite.

En temps normal, le serveur Jenkins serait configuré pour que l'automatisation se lance également à chaque nouveau *commit* sur l'une des deux branches. Pour cela votre dépôt distant doit pouvoir entrer en contact avec le serveur ce qui n'est pas le cas actuellement car celui tourne en local. Ce n'est pas grave, nous lancerons manuellement l'automatisation dans le cadre du début ce TP, nous reviendrons la dessus un peu plus tard.

Cliquez sur un des automatisation (par exemple celle de la branche `dev`) et vérifiez les paramètres du *run* précédent. Vous pouvez pour cela cliquer sur le numéro du *run*, et accéder à la *Pipeline Overview* pour vérifier que le pipeline a bien exécuté ses deux étapes, avoir plus de détails en regardant le section *Pipeline Steps*, ou obtenir la trace complète d'exécution en passant par la sortie console *Console Output*.

Autre chose important à remarquer : Vous verrez une section *Replay* dans les items cliquables quand on observe les données d'un *run*. Cette section, vous permet de rejouer le *run*, mais en modifiant sa configuration.

Sans cette option, pour mettre au point le fichier `Jenkinsfile`, il faut le modifier dans son projet, puis faire un *commit*, puis faire un *push* du projet, afin que Jenkins puisse accéder à sa nouvelle version. Avec le bouton `Replay`, vous pouvez le faire directement sous Jenkins et relancer le `run`.

Essayez par exemple de relancer un *run*, en ajoutant une troisième étape qui affiche un nouveau message à l'écran. Vérifiez la bonne exécution de ce run. Avec votre deuxième *run*, vous pouvez observer que Jenkins conserve la trace de tous les *run*, avec un numéro pour chacun.

Automatisation du *build* du projet

Passons maintenant à une vraie automatisation, qui va construire le projet. Pour construire un projet *vue.js*, il faut d'abord installer les dépendances avec la commande `npm install`, puis lancer le *build* avec la commande `npm run build`. Modifiez la fichier `Jenkinsfile` (ou utiliser le *replay*) pour modifier le pipeline afin qu'il devienne :

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
                sh 'npm run build'
            }
        }
    }
}
```

Lancez un nouveau *run*.

Le *run* échoue car Jenkins ne peut pas accéder à la commande `npm`. En effet, il s'agit d'une commande de l'environnement `node.js` qui n'est pas installé dans le `Dockerfile` ayant permis de déployer Jenkins. Nous pourrions le faire, mais une meilleure idée est de dire à Jenkins d'utiliser *docker* pour compiler le projet, en partant d'une image dans laquelle `node.js` est déjà installé. Tant que nous y sommes, nous prendrons une image dans laquelle non seulement `node.js` est installé, mais aussi `playwright qui nous sera utile pour automatiser les tests UI plus tard.

Dans le fichier `Jenkinsfile` c'est le paramètre `agent` qui détermine le type d'agent qui va réaliser l'automatisation. On peut préciser un agent différent pour chaque étape du pipeline. Voici sa syntaxe quand on veut utiliser un agent docker avec en paramètre l'image à utiliser :

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent { docker {
              image 'mcr.microsoft.com/playwright:v1.57.0-noble'
              args '--network=host'
            } }
            steps {
                sh 'npm install'
                sh 'npm run build'
            }
        }
    }
}
```

Lancez un nouveau *run* avec cette configuration, cette fois-ci cela devrait fonctionner.

Automatisation des tests du projet

Ajoutez au pipeline, deux nouvelles étapes, une pour les tests unitaires, l'autre pour les tests UI. Pour rappel, on lance les tests unitaires avec la commande `npm run test` et les tests d'intégration avec la commande `npm run test:e2e`.

Vérifiez que cela fonctionne.

Pour que l'on puisse accéder au résultats des tests de façon plus confortable sous Jenkins, on peut demander aux tests de produire un rapport au format HTML qui pourra être consulté.

Modifiez le fichier de votre projet `package.json` pour ajouter les paramètres supplémentaires aux lancements des tests.

Pour que *vitest* produise un rapport HTML, on doit le lancer comme ceci : `vitest --reporter=html run`

Pour que *playwright* fasse de même, on doit le lancer comme ceci : `playwright test --reporter=html`

Exécutez à nouveau vos tests manuellement dans votre projet, et constatez qu'effectivement des rapports html ont été crées. L'un se trouve dans un nouveau dossier `html`, et l'autre dans un nouveau dossier `playwright-report`. Les deux s'appellent `index.html`.

Pour pouvoir accéder à ces fichiers sous Jenkins, il faut indiquer dans l'automatisation. Cela se fait généralement dans une section `post { }` qui s'exécute dans tous les cas après la section `step { }` , que ce soit en cas d'échec ou de réussite.

Voici un exemple de section `post` qui publie le fichier `index.html` situé dans le dossier `html` du projet :

```
post {
   always {
       publishHTML([
          allowMissing: true,
          alwaysLinkToLastBuild: false,
          icon: '', keepAll: true,
          reportDir: 'html',
          reportFiles: 'index.html',
          reportName: 'VitestReport',
          reportTitles: '',
          useWrapperFileDirectly: true
          ])
     }
}
```

Placez cette syntaxe à la suite de vos `step` de tests, exécutez à nouveau un *run*. Vous devez voir apparaître dans les résultats, deux sections qui vous donnent accès au rapports de tests :

- Status
- Changes
- Console Output
- Edit Build Information
- Supprimer le build "#18"
- Timings
- Git Build Data
- VitestReport
- playwrightReport
- Open Blue Ocean
- Pipeline Overview
- Restart from Stage
- Replay
- Pipeline Steps
- Workspaces
- Previous Build
- Next Build

#18 (2 janv. 2026, 08:54:18)

Branch indexing

This run spent:

- 6.7 s waiting;
- 51 s build duration;
- 57 s total from scheduled to completion.

Revision: 31515a80a07322acccd053269ab8d8fcf21e8b69
Repository: https://gitlab.com/coursisis/chess.git

- dev

Aucun changement.