Compte Rendu

1. Correction de l'Erreur de Parsing (`NumberFormatException`)

- **Problème Constaté :** Lors de l'exécution du solveur Cypher, une erreur `java.lang.NumberFormatException: For input string: "tCurrent(60"` était levée.
- **Analyse de la Cause :** L'erreur provenait de la méthode utilitaire `parseTimeFile` dans la classe `SolverCypher`. La sous-méthode `extractValue` n'était pas assez robuste pour extraire correctement la valeur numérique des lignes du fichier de configuration temporelle, en particulier pour le format `tCurrent(60)` . qui ne contient pas de virgule. Elle extrayait la chaîne `"tCurrent(60"` au lieu du nombre `60`.
- **Solution Implémentée :** La méthode `extractValue` a été entièrement réécrite pour utiliser une **expression régulière (regex)**. Cette nouvelle version identifie et extrait de manière fiable la séquence de chiffres, quel que soit le format de la ligne (`tCurrent(...)` ou `tLimit(...)`).

  Code Corrigé dans `SolverCypher.java` :

```
private String extractValue(String line) {
    // Le regex cherche les chiffres (\\d+) situés soit après une
parenthèse, soit après une virgule.
    java.util.regex.Pattern pattern =
java.util.regex.Pattern.compile("\\((\\d+)\\)|,\\s*(\\d+)\\)");
    java.util.regex.Matcher matcher = pattern.matcher(line);

    String value = null;
    while (matcher.find()) {
        if (matcher.group(1) != null) { // Pour le format "tCurrent(60)"
            value = matcher.group(1);
        } else if (matcher.group(2) != null) { // Pour le format
"tLimit('key', 1000)"
            value = matcher.group(2);
        }
    }
    if (value != null) return value;
    throw new IllegalArgumentException("Could not extract value from line:
" + line);
}
```

2. Correction de l'Erreur de Connexion (`ResultConsumedException`)

- **Problème Constaté :** Le solveur Cypher levait une exception `org.neo4j.driver.exceptions.ResultConsumedException`, indiquant que le programme tentait de lire un résultat de base de données alors que la connexion était déjà fermée.
- **Analyse de la Cause :** Le problème se situait dans la méthode `executeQuery` de la classe `Neo4jInterface`. Cette méthode utilisait un bloc `try-with-resources` qui fermait automatiquement la connexion à la base de données à la fin de la méthode. Cependant, la méthode retournait un objet `Result` dont les données n'avaient pas encore été lues. Au moment où le `SolverCypher` tentait de lire ce `Result`, sa connexion sous-jacente n'existait plus.

- **Solution Implémentée :** La méthode `executeQuery` a été modifiée pour "consommer" le résultat pendant que la connexion est encore active. Au lieu de retourner l'objet `Result`, elle retourne maintenant une `List<org.neo4j.driver.Record>`.

  Modification dans `Neo4jInterface.java` :

```
// Signature originale : public Result executeQuery(...)
// Signature corrigée :
public List<org.neo4j.driver.Record> executeQuery(String query, Map<String,
Object> params) {
    try (var driver = GraphDatabase.driver(...); var session =
driver.session()) {
        Result result = session.run(query, params);
        // On consomme le résultat et le stocke dans une liste avant que la
connexion ne se ferme.
        return result.list();
    }
}
```

La boucle dans `SolverCypher.java` a ensuite été adaptée pour parcourir cette `List<Record>`.

3. Correction de l'Affichage des Relations dans la Visualisation

- **Problème Constaté :** La visualisation du graphe avec `neovis.js` affichait la propriété `ctx` sur les arêtes (les relations), alors que l'objectif était d'afficher le **type** de la relation (ex: `used`, `wasGeneratedBy`).
- **Analyse de la Cause :** Après de multiples tests, il a été déterminé que la version `2.1.0` de `neovis.js` interprétait mal les configurations de style directes pour les légendes des arêtes. Les tentatives d'utiliser `caption: "<type>"` ou `label: "..."` causaient des conflits avec la fonctionnalité d'infobulle au survol.
- **Solution Implémentée :** La solution finale et la plus élégante a été d'utiliser la **configuration par fonction** de `neovis.js`, qui s'est avérée la plus fiable.

  Modification dans `index_pattern.html` : Le bloc statique `relationships` a été vidé de ses configurations de `label`. La boucle `forEach` qui appliquait les infobulles a été modifiée pour appliquer à la fois l'infobulle (`title`) et la légende (`label`) de manière dynamique.

```
// Ancien code dans la boucle :
// r[NeoVis.NEOVIS_ADVANCED_CONFIG] = { function:{ title:fullDetails } };

// Code final dans la boucle forEach des relations :
Object.values(config.relationships).forEach(r => {
    r[NeoVis.NEOVIS_ADVANCED_CONFIG] = {
        function: {
            // Cette fonction retourne dynamiquement le type de la relation
            label: rel => rel.type,
            // Cette fonction retourne les propriétés pour l'infobulle
            title: fullDetails
        }
    };
});
```

4. Correction de couleurs d’arêtes et des nœuds

Situation Initiale

Initialement, la librairie de visualisation `neovis.js` assigne automatiquement des **couleurs distinctes** aux différents types de nœuds (`Agent`, `Process`, `Artifact`) et d'arêtes (`used`, `wasGeneratedBy`, etc.). Bien que cette fonctionnalité permette de différencier les éléments, les couleurs choisies par défaut ne sont pas fixes et ne portent pas de signification sémantique pour notre projet.

Modifications Apportées

Pour améliorer la lisibilité et créer une légende visuelle cohérente, le code de la configuration de `neovis.js` (dans le fichier `index_pattern.html`) a été modifié pour que **nous puissions choisir nous-mêmes les couleurs**.

1. Couleurs des Nœuds : Une propriété `color` a été ajoutée à la configuration de chaque type de nœud dans le bloc `labels`.

- **Code de la solution :**

```
   labels: {
    "Agent" :    { "label": "name", "color": "#FFC300" }, // Jaune
    "Artifact" : { "label": "name", "shape": "circle", "color": "#C70039"
}, // Rouge
    "Process" :  { "label": "name", "color": "#3375FF"  }  // Bleu
},
```

2. Couleurs des Arêtes (Relations) : Une "palette de couleurs" (`colorMap`) a été définie. Ensuite, une fonction a été ajoutée pour lire le type de chaque relation et lui assigner la couleur correspondante de notre palette.

- **Code de la solution :**

```
// Création de notre palette de couleurs
const colorMap = {
    "wasGeneratedBy":  "#C70039",
    "wasControlledBy": "#3357FF",
    "used":            "#28B463",
    "wasTriggeredBy":  "#F9C74F",
    "wasDerivedFrom":  "#9B59B6"
};

// Application dynamique de la couleur
Object.keys(config.relationships).forEach(type => {
    config.relationships[type][NeoVis.NEOVIS_ADVANCED_CONFIG] = {
        function: {
            label: rel => rel.type,
            title: fullDetails,
            // On assigne la couleur depuis notre palette
            color: rel => colorMap[rel.type] || '#848484'
        }
    };
});
```

Résultat Final

Grâce à ces modifications, la visualisation n'utilise plus les couleurs par défaut de la librairie. Le graphe affiche maintenant un schéma de couleurs personnalisé et cohérent, où chaque type de nœud et de relation est immédiatement identifiable par une couleur que nous avons nous-mêmes choisie.

5. Sécurisation des Identifiants de la Base de Données

- **Problème Constaté :** Une faille de sécurité majeure a été identifiée : la méthode `buildVizHtmlFile` écrivait les identifiants de connexion à la base de données (URI, utilisateur et mot de passe) en clair directement dans le fichier `index.html` généré. Ce fichier, potentiellement stocké sur le disque ou dans le cache du navigateur, exposait des informations sensibles.
- **Analyse de la Cause :** Ce comportement était dû à la conception initiale de l'intégration avec `neovis.js`. La librairie, étant exécutée côté client (dans le `WebView`), avait besoin de ces identifiants pour établir sa propre connexion à Neo4j. Le mécanisme de remplacement de placeholders (`%DB_URL%`, etc.) était la méthode directe pour les lui fournir.
- **Solution Implémentée :** L'architecture de la visualisation a été refactorisée pour ne plus jamais écrire les identifiants dans un fichier. La nouvelle approche consiste à passer ces informations de manière sécurisée, en mémoire, de Java vers Javascript.
    1. **Modification de `index_pattern.html` :** La fonction Javascript `draw()` a été modifiée pour accepter les identifiants comme paramètres (`draw(serverUrl, serverUser, serverPassword)`). L'attribut `onload` a été retiré du `<body>`, empêchant tout démarrage automatique.
    2. **Modification de `Neo4jInterface.java` :** La méthode `buildVizHtmlFile` a été simplifiée pour ne plus remplacer que le placeholder de la requête Cypher (`%CYPHER_QUERY%`), ignorant ceux des identifiants. Des *getters* (`getUri()`, `getUser()`, `getPassword()`) ont été ajoutés pour permettre un accès contrôlé à ces informations depuis Java.
    3. **Modification de `ScreenController.java` :** La méthode `initGraphVizScreen` orchestre désormais le processus de manière sécurisée.
        - Elle charge d'abord le fichier `index.html` (qui est maintenant un simple squelette sans identifiants).
        - Elle attache un "écouteur" (`Listener`) au `WebView`, qui attend que la page soit complètement chargée.
        - Une fois la page prête, et seulement à ce moment, elle exécute un script qui appelle la fonction `draw()` en lui passant les identifiants (récupérés depuis l'objet `Neo4jInterface`) comme arguments.

Extrait de la logique clé dans `ScreenController.java` :

```
// ... après avoir chargé l'URL dans le webEngine ...
webEngine.getLoadWorker().stateProperty().addListener(
    (obs, oldState, newState) -> {
        if (newState == Worker.State.SUCCEEDED) {
            // On appelle la fonction Javascript "draw" en passant les
identifiants en mémoire
            String script = String.format("draw('%s', '%s', '%s');",
```

```
                                            neo.getUri(), neo.getUser(),
neo.getPassword());
            webEngine.executeScript(script);
        }
    }
);
webEngine.load(url);
```

Résultat Final : Grâce à cette modification, le mot de passe et les autres identifiants ne sont plus jamais écrits sur le disque. Ils ne transitent qu'en mémoire entre les composants de l'application au moment de l'affichage, ce qui corrige la faille de sécurité.

6. Phase de Tests et Validation

Après la phase d'implémentation, une phase de tests a été menée pour garantir la robustesse et la fiabilité de l'application, en particulier des composants nouvellement créés.

6.1. Correction et Validation des Parsers (Traducteur.Parser)

Avant de tester les solveurs, il était impératif de s'assurer que les composants en amont étaient fiables.

- **Problème Constaté :** La classe `Parser` initiale était fragile. Sa logique était basée sur des positions de caractères fixes, la rendant susceptible d'échouer au moindre changement de format du fichier d'entrée Prolog. De plus, sa classe de test (`testParser`) n'utilisait pas les bonnes pratiques, rendant les tests non fiables.
- **Solution Implémentée :**
    1. **Refactorisation du `Parser` :** La classe a été entièrement réécrite pour utiliser des **expressions régulières (`regex`)**. Cette approche garantit une extraction d'informations fiable et robuste, indépendamment de la longueur des noms ou des espacements. Des `Set` ont été utilisés en interne pour gérer automatiquement la déduplication des résultats.
    2. **Modernisation de la Classe de Test :** La classe `testParser` a été mise à jour vers **JUnit 5**. Elle utilise maintenant un mécanisme de création et de suppression de fichiers temporaires (`@BeforeEach`, `@AfterEach`), garantissant que chaque test est parfaitement **isolé et indépendant**, ce qui est une pratique essentielle pour des tests de qualité.

Extrait de la classe `testParser` corrigée :

```
class testParser {
    private File tempFile;
    private Parser parser;

    @BeforeEach // Exécuté avant chaque test
    void setUp() throws IOException {
        tempFile = File.createTempFile("test parser", ".pl");
        parser = new Parser(tempFile);
    }

    @AfterEach // Exécuté après chaque test
    void tearDown() {
```

```
        if (tempFile != null) tempFile.delete();
    }
    // ...
}
```

6.2. Tests Unitaires du `SolverCypher`

L'objectif principal était de valider la logique du `SolverCypher` en l'isolant du reste de l'application. Pour cela, une approche de test unitaire avancée a été choisie, utilisant un **framework de "mocking" (Mockito)**.

- **Méthodologie :**
    1. **"Mocker" les dépendances :** Au lieu de se connecter à une vraie base de données, l'objet `Neo4jInterface` est "mocké" (simulé). Cela nous permet de contrôler précisément ce que la base de données est censée retourner pour un test donné.
    2. **Préparation du Scénario (`when...thenReturn`) :** Pour chaque test, nous configurons le mock pour qu'il retourne une liste de résultats spécifique (soit une liste vide pour un cas conforme, soit une liste contenant des enregistrements `Record` mockés pour un cas de violation).
    3. **Exécution et Vérification :** Le solveur est ensuite exécuté, et nous vérifions que le nombre et le type d'objets `Issue` créés correspondent exactement à ce que nous attendions.
- **Couverture des Tests :** La suite de tests `SolverCypherTest` a été conçue pour obtenir une couverture maximale en testant de multiples scénarios pour chaque principe du RGPD :
    - **Cas Conformes :** Vérification qu'aucune violation n'est levée lorsque les règles sont respectées (ex: une donnée effacée à temps, un consentement valide).
    - **Cas Non-Conformes :** Vérification qu'une violation est bien détectée lorsque les règles ne sont pas respectées (ex: un consentement manquant, un délai de conservation dépassé).
    - **Tests des Méthodes Utilitaires :** Des tests spécifiques ont été écrits pour valider le bon fonctionnement des méthodes internes critiques comme `parseTimeFile` et `extractValue`, en utilisant la réflexion Java pour y accéder.
- **Extrait de la classe `SolverCypherTest` avec Mockito :**

```
class SolverCypherTest {
    private Neo4jInterface neo; // C'est un mock
    private SolverCypher solver;
    @BeforeEach
    void setUp() {
        neo = mock(Neo4jInterface.class); // On crée le mock
        solver = new SolverCypher(neo);
    }
   @Test
    @DisplayName("✕ non-conforme – 1 résultat → 1 issue")
    void nonCompliant() throws IOException {
        // Arrange : on simule le retour de la base de données
        Record rec = mockRecord(Map.of("S", "Alice", "TE", 50));
        when(neo.executeQuery(anyString(),
anyMap())).thenReturn(List.of(rec));

        // Act
        solver.solve(List.of("rightAccess"), "",
timeFile.toString());
        // Assert
      assertEquals(1, solver.getIssues().size());  }}
```