Semestre P24

SR10

Compte Rendu des Tests

Etudiants :
Raphael Nguyen
Raphael Chauvier
Raphael Chauvier Chauvier

Tests de Modèle

Voici un résumé des tests effectués sur notre application, couvrant tous nos modèles pour s'assurer de leur bon fonctionnement. Nous avons donc pu testé les modèles suivants :

1. **AssociationFichiers**
2. **Candidature**
3. **Utilisateur**
4. **HistoriqueDemandes**
5. **FichePoste**
6. **OffreEmploi**
7. **Organisation**

Règles de Test

Nous avons appliqué des règles strictes pour garantir que chaque aspect des modèles fonctionne correctement :

1. **Tests de Création** : Vérifier la création des enregistrements avec des données valides et invalides.
    - Exemple : Création d'une association de fichiers avec des informations correctes.
2. **Tests de Lecture** : Assurer la lecture précise des enregistrements depuis la base de données.
    - Exemple : Lecture des informations d'une candidature spécifique.
3. **Tests de Mise à Jour** : Valider les mises à jour des enregistrements avec des données correctes.
    - Exemple : Mise à jour des informations d'un utilisateur.
4. **Tests de Suppression** : Vérifier la suppression des enregistrements sans affecter l'intégrité de la base de données.
    - Exemple : Suppression des candidatures et des fichiers associés.
5. **Tests de Validation** : Assurer que les validations sont correctement appliquées lors de la création ou de la mise à jour.
    - Exemple : Validation des informations de connexion d'un utilisateur.
6. **Tests de Gestion des Erreurs** : Simuler des erreurs pour vérifier leur gestion correcte et l'annulation des transactions si nécessaire.
    - Exemple : Gestion des erreurs lors de la suppression de fichiers associés.

Exemples de Cas de Test

AssociationFichiers

- **Création** : Tester la création d'associations de fichiers avec des données valides.
- **Lecture** : Vérifier la lecture des associations de fichiers.
- **Mise à Jour** : Tester les mises à jour des associations de fichiers.
- **Suppression** : Assurer la suppression des associations de fichiers et vérifier que les fichiers sont correctement supprimés du système de fichiers.
- **Gestion des Erreurs** : Simuler une erreur lors de la suppression de fichiers pour vérifier la gestion des erreurs.

Candidature

- **Création** : Tester la création de candidatures avec et sans fichiers associés.
- **Lecture** : Vérifier la lecture des candidatures.
- **Mise à Jour** : Tester les mises à jour des candidatures.
- **Suppression** : Assurer la suppression des candidatures et des fichiers associés.
- **Validation** : Vérifier si un candidat a déjà postulé pour une offre spécifique.
- **Gestion des Erreurs** : Tester la gestion des erreurs lors de la suppression des fichiers associés.

Utilisateur

- **Création** : Tester la création de comptes utilisateur.
- **Lecture** : Vérifier la lecture des informations utilisateur.
- **Mise à Jour** : Tester les mises à jour des informations utilisateur avec des champs valides.
- **Suppression** : Assurer la suppression des comptes utilisateur.
- **Validation** : Vérifier les informations de connexion utilisateur.
- **Gestion des Erreurs** : Tester la gestion des erreurs lors de la suppression d'un utilisateur avec des candidatures associées.

Etc.

Taux de Couverture

En suivant ces règles et en appliquant des tests unitaires avec Jest, nous avons atteint un taux de couverture de **100%** du code des modèles. Chaque fonction, scénario d'erreur et interaction avec la base de données ont été testés pour garantir la robustesse et la fiabilité de notre application.

Nous avons utilisé des mocks pour les dépendances externes (comme les modules de base de données et de fichiers) afin de simuler les interactions et de nous concentrer sur la logique métier de nos modèles.

Tests de Routage

Structure des Tests

Avant chaque test dans un **beforeAll()**, nous créons un agent avec la bibliothèque **request** : **request.agent(app)** qui nous permet d'envoyer de communiquer avec le serveur, c'est à dire d'envoyer des requêtes, et d'étudier les résultats.

Une fois le test terminé (**afterAll()**), nous supprimons cet agent en supprimant sa référence (**agent = null;**)

Nos tests de routing ont pour but de tester les fonctionnalités de nos routes, et en particulier de savoir si le cycle complet d'interaction avec une route se déroule comme prévu.

Pour ce faire nous aurons besoin des fonctionnalités suivantes :

1. Définir des **états de session personnalisés** pour guider le test sans avoir à saisir de véritables informations de la base de données.
2. Effectuer un **mocking des fonctions du modèle de données** afin d'éviter d'affecter la base pendant les tests.

Simulation de Session

Pour les tests de routage, nous avons eu besoin d'émuler nos sessions. Pour ce faire, nous avons implémenté des **routes spéciales** définies sur le **routeur "testrouter"**.

Par exemple, la route **/test/mock-recruteur** remplit les informations de sessions avec un email arbitraire et un statut de recruteur (affilié à une organisation arbitraire), tandis que **/test/mock-candidat** crée une session qui sera authentifiée avec succès comme candidat. Cela

permet de donner illusion à notre système d'authentification par session qu'il rencontre un type d'utilisateur particulier, et donc **d'étudier son comportement dans ces cas de figure**.

```
router.get( path: '/mock-recruteur', handlers: (req
    req.session.userEmail = 'test@email.com';
    req.session.userType = 'recruteur';
    req.session.userAffiliation = '123456789';
    res.send( body: 'ok');
});
```

Définition des routes de tests dans /routes/testrouter.js

Ces routes sont activées uniquement en mode **test**, via la commande **npm test**. Le processus d'activation repose sur une variable d'environnement "NODE_ENV" qui est définie à la valeur 'test' lors de l'exécution de jest. La définition de cette variable est effectuée dans **.jest/setEnvVars.js** `process.env.NODE_ENV = 'test';`, que nous avons renseigné dans la config de jest à **package.json**. Cette structure nous a été suggérée sur Stack Overflow pour gérer nos express sessions dans nos tests.

```
if (process.env.NODE_ENV === 'test') {
  router.use('/test', testRouter);
}
```

Dans routes/index.js on ajoute les routes de test de manipulation de session seulement si on est dans l'environnement test.

Simulation de BDD

Afin d'éviter de modifier la base de données dans certains tests, nous avons également simulé les requêtes envoyées à la base de données à l'aide de **jest.mock**

Cela a été implémenté comme pour les tests de modèle, à la différence que cette fois nous avons rendus valide **tous les tests de base de données qui n'étaient pas liés à ce que nous testions dans la route**.

Typiquement il s'agit d'insérer pour de faux la fausse offre d'emploi lors du test de la route **/jobs/add_offer** afin de **tester le comportement de la route** lorsque l'ajout de l'offre est un succès.

Tests Orientés Requête

Par manque de temps dû à la complexité du projet, et tel qu'il nous a été expliqué dans les consignes, nous avons préféré implémenter un ensemble de tests qui vérifient des aspects différents de nos routes, plutôt que d'implémenter à la chaîne une multitude de tests identiques pour des tâches différentes.

Nos tests de routage permettent ainsi de vérifier :

Les tests ci-dessous se retrouvent dans le fichier tests/routing/jobs.test.js

1 : Si les URLs de mocking de session ("/test/reset-session", …) fonctionnent bien, cela à des fins de débogage de nos scripts JEST.

```
it('should grant access to mock-recruteur', async () : Promise<void>  => {
    const res = await agent.get('/test/mock-recruteur');
    expect(res.statusCode).toBe( expected: 200);
});
```

2 : Si un utilisateur non connecté peut correctement accéder aux pages non secure, ici nous testons la page /login.

```
it('unregistered users should not be redirected from the login page', async ()
    const res = await agent.get('/login');
    expect(res.statusCode).toBe( expected: 200);
});
```

3 : Si un utilisateur non connecté est correctement redirigé vers la page de connexion si l'URL le permet.

```
it('unregistered users should be redirected to the login page', async () 
    const res = await agent.get('/jobs/add_job');
    expect(res.statusCode).toBe( expected: 302);
    expect(res.headers.location).toBe( expected: '/login');
});
```

4-6 : Si les middlewares de sécurité des routes fonctionnent correctement. (donc **tester nos middlewares**)

Afin de nous assurer que la sécurisation des routes soit correctement implémentée, malgré d'éventuels bugs qui pourraient survenir au cours du développement, nous avons implémenté des tests spécifiquement centrés sur la sécurité des pages dans nos tests de routing.

Au cours de ces tests, nous présentons différents états de sessions qui peuvent être construits par une diversité d'utilisateurs via le processus de login (sessions candidat, sessions recruteur, …) et nous vérifions que les pages répondent bien positivement ou négativement en fonction.

```
it('candidates should have receive a 500 error code denying illegal access', async ()
    await agent.get('/test/mock-candidat');
    const res = await agent.get('/jobs/add_job');
    expect(res.statusCode).toBe( expected: 403);
});
```

Test de sécurité implémenté dans tests/routing/jobs.test.js associé à la route /jobs/add_job qui doit refuser les utilisateurs qui ne sont pas recruteurs.

```
it('registered recruitors should have access to this page', async () : Promise<void>  => {
    await agent.get('/test/mock-recruteur');
    const res = await agent.get('/jobs/add_job');
    expect(res.statusCode).toBe( expected: 200);
});
```

Au contraire, un recruteur affilié doit avoir accès à la page.

Les tests ci-dessous se retrouvent dans le fichier tests/routing/postes.test.js

7-8 : Si une offre d'emploi peut être ajoutée avec succès par un recruteur, **et également de tester si nos fonctions comme isUserLegitimate s'exécutent là où c'est prévu.**

```
it('should add an offer successfully with valid inputs', async () : Promise<void>  => {
    FichePoste.isUserLegitimate.mockResolvedValue( value: true);
    OffreEmploi.create.mockResolvedValue( value: {});

    const res = await agent.post('/jobs/add_offer')
        .send({
            dateValidité: "2030-12-12", // Future date
            listePieces: 'Some required documents',
            nombrePieces: 2,
            idFiche: '123'
        });

    expect(res.statusCode).toBe( expected: 302);
    expect(res.header.location).toBe( expected: '/jobs/add_offer');
    expect(FichePoste.isUserLegitimate).toHaveBeenCalledWith( expected: '123', '123456789');
    expect(OffreEmploi.create).toHaveBeenCalled();
});
```

Notez que l'ajout de l'offre d'emploi a été émulé avec un **jest.mock**, tout comme le processus de vérification de légitimité.

9 : Si la page refuse correctement un utilisateur invalide **en postulant que notre fonction isUserLegitimate** fonctionne correctement. (fonctionnement analogue à la fonction précédente)

10 : Si notre vérification de la validité de données d'input conduit bien à un refus avec des données invalides.

```
it('should return validation errors with invalid inputs', async () : Promise<void>  => {
    const res = await agent.post('/jobs/add_offer')
        .send({
            dateValidité: 'invalid-date',
            listePieces: '',
            nombrePieces: 'not-an-integer',
            idFiche: ''
        });

    expect(res.statusCode).toBe( expected: 302);
    expect(res.header.location).toBe( expected: '/jobs/add_offer');
    expect(FichePoste.isUserLegitimate).not.toHaveBeenCalled();
    expect(OffreEmploi.create).not.toHaveBeenCalled();
});
```

Extension des tests

Grâce aux tests que nous avons implémentés, nous remarquons qu'il serait possible d'automatiser la généralisation de ces tests, par exemple en implémentant un registre de pages équipé de tel ou tel middleware et qui seraient toutes soumises simultanément au même test.

Il serait également possible dans une moindre mesure d'automatiser davantage les tests sur la sécurité des pages. En revanche, nous pensons également qu'une bonne modularité de nos routes nous permettrait de tester uniquement certaines fonctionnalités comme l'authentification sur une page donnée, puisqu'elle sera implémentée sur toutes les autres. C'est par exemple le cas de la procédure d'authentification (il suffit de tester /jobs/add_job pour savoir que ça fonctionne partout).