

Rapport détaillé des bug et tests

Introduction :

Billed est une entreprise qui produit des solutions SaaS destinées aux équipes de ressources humaines. L'objectif ici est de tester et déboguer le parcours employé d'application Billed pour «Notes de frais » et de déboguer le parcours administrateur. Pour cela on utilise Kanban Notion pour résoudre les bug et effectuer l'ajout de tests.

[Bug report] – Login

Description

Dans le rapport de test "Login, si un administrateur remplit correctement les champs du Login, il devrait naviguer sur la page Dashboard", le test est passé au rouge.

To-do

Faire passer le test au vert en réparant la fonctionnalité.

On utilise chrome debugger pour mettre en œuvre des breakpoints, étape par étape et pas à pas dans le fichier JS Login.js du dossier et sous dossier /front/src/containers/ pour localiser le bug et ainsi modifier le code JS pour permettre à l'administrateur de se connecter :

```
//fix: login as admin
handleSubmitAdmin = (e) => {
  e.preventDefault();
  const user = {
    type: "Admin",
    email: e.target.querySelector(`input[data-testid="admin-email-input"]`).value,
    password: e.target.querySelector(`input[data-testid="admin-password-input"]`).value,
    status: "connected",
  };
};
```

Dans la fonction handleSubmitAdmin, les inputs utilisés pour le login admin sont ceux de l'employée (employee-email-input), je modifie donc le code JS pour utiliser les inputs admin avec « admin-email-input » et « admin-password-input »

Rapport de test avant la modification:

```
FAIL src/__tests__/Login.js
  ● Given that I am a user on login page › When I do fill fields in correct format and I click on admin button Login In › Then I should be identified as an HR admin in app
    TypeError: Cannot read properties of null (reading 'value')

  ● Given that I am a user on login page › When I do fill fields in correct format and I click on admin button Login In › Then I should be identified as an HR admin in app
    TypeError: Cannot read properties of null (reading 'value')

  ● Given that I am a user on login page › When I do fill fields in correct format and I click on admin button Login In › Then I should be identified as an HR admin in app
    expect(jest.fn()).toHaveBeenCalled()

    Expected number of calls: ≥ 1
    Received number of calls: 0

  ● Given that I am a user on login page › When I do fill fields in correct format and I click on admin button Login In › It should renders HR dashboard page
    expect(received).toBeTruthy()

    Received: null
```

Rapport de test après la modification :

On effectue les tests JEST coverage et unitaire sur Login.js

```
$ jest --coverage --noStackTrace --silent src/__tests__/Login.js
```

```
$ jest src/__tests__/Login.js
```

```
Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        3.016 s, estimated 4 s
Ran all test suites matching /src\\__tests__\\login.js/i.
```

[Bug report] – Bills

Description

Le test Bills / les notes de frais s'affichent par ordre décroissant est passé au rouge.

To-do

Faire passer le test au vert en réparant la fonctionnalité.

On utilise chrome debugger pour mettre en œuvre des breakpoints, étape par étape et pas à pas dans le fichier JS BillsUI.js du dossier et sous dossiers /front/src/views / pour localiser le bug et ainsi ajouter du code JS pour que les notes de frais s'affichent par ordre décroissant.

Je travaille sur `const rows= (data)`, je trie les data par ordre décroissant avec `sort()` en utilisant `Date()`. Cela retourne `true (1)` si `new Date(a.date)` est inférieur à `new Date(b.date)` sinon `return false(-1)`

```
20  },
21  //fix: trier les factures dans l'ordre décroissant
22  // fonction sort décroissante
23  // ex: 2012 - 2013 = -1
24  // 2012 - 2012 = 0
25  // 2013 - 2012 = 1
26  const rows = (data) => {
27    data.sort((a, b) => (new Date(a.date) < new Date(b.date) ? 1 : -1));
28    return data && data.length ? data.map((bill) => row(bill)).join("") : "";
29  };
30
```

Rapport de test

Rapport de test unitaire avant la modification :

```
FAIL src/__tests__/Bills.js
  ● Given I am connected as an employee > When I am on Bills Page > Then bills should be ordered from earliest to latest

    expect(received).toEqual(expected) // deep equality

    - Expected   - 1
    + Received   + 1

    Array [
      "2004-04-04",
    +  "2001-01-01",
      "2003-03-03",
      "2002-02-02",
    -  "2001-01-01",
    ]

    Test Suites: 1 failed, 1 total
    Tests:       1 failed, 1 passed, 2 total
    Snapshots:   0 total
    Time:        3.903 s
```

Rapport de test unitaire après la modification :

```
PASS src/__tests__/Bills.js
Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  0 total
Time:       2.304 s
Ran all test suites matching /src\\__tests__\\Bills.js/i.
```

[Bug Hunt] – Bills

Description

Je suis connecté en tant qu'employé, je saisis une note de frais avec un justificatif qui a une extension différente de jpg, jpeg ou png, j'envoie. J'arrive sur la page Bills, je clique sur l'icône "voir" pour consulter le justificatif : la modale s'ouvre, mais il n'y a pas d'image.

Si je me connecte à présent en tant qu'Admin, et que je clique sur le ticket correspondant, le nom du fichier affiché est null. De même, lorsque je clique sur l'icône "voir" pour consulter le justificatif : la modale s'ouvre, mais il n'y a pas d'image.

To-do

Comportements attendus :

- [] la modale doit afficher l'image.
- [] dans le dashboard, le formulaire correspondant au ticket doit afficher le nom du fichier.

Suggestion : empêcher la saisie d'un document qui a une extension différente de jpg, jpeg ou png au niveau du formulaire du fichier NewBill.js. Indice : cela se passe dans la méthode `handleChangeFile...`

On utilise chrome debugger pour mettre en œuvre des breakpoints, étape par étape et pas à pas dans le fichier JS NewBill.js du dossier et sous dossiers `/front/src/containers /` pour localiser le bug et puis on ajoute des nouvelles variables pour vérifier le format de l'image, on vérifie si le format est jpg jpeg ou png et si vrai on met `Imgformatvalide` à vrai sinon faux.

Si le format n'est pas valide on supprime le fichier de l'input, on ajoute la class is invalid pour dire à l'utilisateur que c'est invalide et on supprime la classe blue-border, puis on affiche un message d'erreur

Si c'est valide on supprime la classe is invalid on ajoute la classe border-blue. Si le format de l'image est valide et le formulaire on débute l'upload. On modifie aussi le HTML pour afficher un message d'alerte également pour indiquer que seulement les formats .jpg, jpeg et png sont acceptés :

```
// Nouvelle variable pour vérifier le format de l'image
const fileInput = this.document.querySelector('input[data-testid="file"]');
const fileAcceptedFormats = ["jpg", "jpeg", "png"];
const fileNameParts = fileName.split(".");
const fileExtension = fileNameParts[fileNameParts.length - 1].toLowerCase();
this.ImgFormatValide = false;
// Si il y a au moins 2 parties dans le nom du fichier continuer la verification
if (fileNameParts.length > 1) {
    // verifie si le format de l'image est jpg jpeg ou png, si vrai met ImgFormatValide à vrai sinon faux
    fileAcceptedFormats.indexOf(fileExtension) !== -1
        ? (this.ImgFormatValide = true)
        : (this.ImgFormatValide = false);
}
if (!this.ImgFormatValide) {
    // Si le format d'image n'est pas valide
    fileInput.value = ""; // ... supprimer le fichier de l'input
    fileInput.classList.add("is-invalid"); // ... ajouter la class is invalide pour dire à l'utilisateur que c'est invalide
    fileInput.classList.remove("blue-border"); // ... supprime la classe blue-border
    alert(
        "Le format de votre fichier n'est pas pris en charge." +
        "\n" +
        "Seuls les .jpg, .jpeg, .png sont acceptés."
    ); // message d'erreur pour l'utilisateur
} else {
    // Si le format de l'image est valide
    fileInput.classList.remove("is-invalid"); // ... supprimer la classe is valide
    fileInput.classList.add("blue-border"); // ... ajouter la classe border-blue
    const formData = new FormData();
    const email = JSON.parse(localStorage.getItem("user")).email;
    formData.append("file", file);
    formData.append("email", email);
    this.formData = formData; // alors cela peut etre utiliser dans d'autres méthodes
    this.fileName = fileName;
}
};
```

```
<label for="file" class="bold-label">Justificatif</label>
<input required type="file" accept=".jpg, .jpeg, .png" class="form-control blue-border" data-testid="file" />
<small class="form-text text-muted"> Les formats acceptés sont : .jpg, .jpeg, .png</small>
</div>
```

[Bug Hunt] – Dashboard

Description

Je suis connecté en tant qu'administrateur RH, je déplie une liste de tickets (par exemple : statut "validé"), je sélectionne un ticket, puis je déplie une seconde liste (par exemple : statut "refusé"), je ne peux plus sélectionner un ticket de la première liste.

To-do

Comportement attendu : pouvoir déplier plusieurs listes, et consulter les tickets de chacune des deux listes.

Pas besoin d'ajouter de tests

On utilise chrome debugger pour mettre en œuvre des breakpoints, étape par étape et pas à pas dans le fichier JS Dashboard.js du dossier et sous dossiers /front/src/containers / pour localiser le bug et ainsi ajouter du code JS manquant « `$(`#status-bills-container${this.index}`)` » pour qu'il soit possible de déplier la liste des tickets en même temps.

```
handleShowTickets(e, bills, index) {
  if (this.counter === undefined || this.index !== index) this.counter = 0;
  if (this.index === undefined || this.index !== index) this.index = index;
  if (this.counter % 2 === 0) {
    $(`#arrow-icon${this.index}`).css({ transform: "rotate(0deg)" });

    $(`#status-bills-container${this.index}`).html(
      cards(filteredBills(bills, getStatus(this.index)))
    );
    this.counter++;
  } else {
    $(`#arrow-icon${this.index}`).css({ transform: "rotate(90deg)" });
    $(`#status-bills-container${this.index}`).html("");
    this.counter++;
  }
}

bills.forEach((bill) => {
  // fix les erreurs d'expansion de liste dans l'admin
  $(`#status-bills-container${this.index} #open-bill${bill.id}`).click((e) =>
    this.handleEditTicket(e, bill, bills)
  );
});
```

[Bug Hunt] – Fix title display bug

Description

Quand le nom de la dépense est vide on ne peut quand même soumettre la new bill.

To-do

J'ai mis en place un input required typ= « text » pour rendre obligatoire l'écriture dans le champ.

```
</div>
<div class="col-half">
  <label for="expense-name" class="bold-label">Nom de la dépense</label>
  <input required type="text" class="form-control blue-border" data-testid="expense-name" placeholder="Vol Paris Londres" />
</div>
```

[Ajout de tests unitaires et d'intégration]

Le rapport de couverture de branche de Jest indique que les fichiers suivants ne sont pas couverts (cf. copie d'écran) :

Respecter la structure des tests unitaires en place : Given / When / Then avec le résultat attendu. Un exemple est donné dans le squelette du test `tests/Bills.js`

- [] composant `container/Bills` :
 - [] couvrir un maximum de "statements" c'est simple, il faut qu'après avoir ajouté tes tests unitaires et d'intégration [le rapport de couverture du fichier container/Bills](#) soit vert. Cela devrait permettre d'obtenir un taux de couverture aux alentours de 80% dans la colonne "statements".

Après l'ajout des 13 tests pour `Bills.js`, j'obtiens un taux de couverture de 83,33% dans la colonne statements pour `Bills.js` :

All files containers

85.05% Statements 182/214 71.64% Branches 48/67 64.81% Functions 35/54 87.06% Lines 175/201

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File ▲	Statements ▾	Branches ▾
Bills.js	<div><div></div></div> 83.33%	20/24

Ci-dessous un exemple de tests pour `Bills.js` que j'ai ajouté pour obtenir un taux de couverture > 80% :

```
// test sur le bouton newBill: clique sur le bouton puis ouverture de la bill
describe("Given I am connected as Employee and I am on Bill page, there are a newBill button", () => {
  describe("When clicking on newBill button", () => {
    test("Then, bill form should open", () => {
      Object.defineProperty(window, "localStorage", { value: localStorageMock });
      window.localStorage.setItem(
        "user",
        JSON.stringify({
          type: "Employee",
        })
      );
      // construit l'interface user
      const html = BillsUI({ data: [] });
      document.body.innerHTML = html;
      const onNavigate = (pathname) => {
        document.body.innerHTML = ROUTES({ pathname });
      };
      // init store
      const store = null;
      const bill = new Bills({
        document,
        onNavigate,
        store,
        localStorage: window.localStorage,
      });
      // Mock handleClickNewBill
      const handleClickNewBill = jest.fn(() => bill.handleClickNewBill());
      // Obtenir eye button dans DOM, ajout event et userEvent
      screen.getByTestId("btn-new-bill").addEventListener("click", handleClickNewBill);
      userEvent.click(screen.getByTestId("btn-new-bill"));
      // screen doit montrer envoyer une note de frais
      expect(handleClickNewBill).toHaveBeenCalled();
      expect(screen.getByText("Envoyer une note de frais")).toBeTruthy();
    });
  });
});
```


Chaque test est structuré de la manière suivante : un nom de composant, une condition ou expression testée et écrite de manière à qu'elle soit compréhensible par un product owner, et une description du comportement attendu. Chaque test est commenté dans le code pour en comprendre le fonctionnement rapidement.

- [] composant views/Bills : Le taux de couverture est à 100% néanmoins si tu regardes le premier test il manque la mention "expect". Ajoute cette mention pour que le test vérifie bien ce que l'on attend de lui.

Je modifie et optimise le code pour le premier test où il manque la mention expect :

```
describe("Given I am connected as an Employee", () => {
  // Test pour BillsUI.js
  // Quand je suis sur la page bills il y a une icone bill sur le cote vertical, doit etre surligné
  describe("When I am on Bills page, there are a bill icon in vertical layout", () => {
    test("Then, the bill icon in vertical layout should be highlighted", () => {
      Object.defineProperty(window, "localStorage", { value: localStorageMock });
      window.localStorage.setItem(
        "user",
        JSON.stringify({
          type: "Employee",
        })
      );
      const html = BillsUI({ data: [] });
      document.body.innerHTML = html;
      expect(document.querySelector("#layout-icon1").classList.contains("active-icon"));
    });
  });
});
```

Le taux de couverture est bien à 100% pour src/views :

All files views

100% Statements 33/33 100% Branches 32/32 100% Functions 15/15 100% Lines 29/29

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements
Actions.js	100%
BillsUI.js	100%
DashboardFormUI.js	100%
DashboardUI.js	100%
ErrorPage.js	100%
LoadingPage.js	100%
LoginUI.js	100%
NewBillUI.js	100%
VerticalLayout.js	100%

- [] ajouter un test d'intégration GET Bills. Tu peux t'inspirer celui qui est fait (signalé en commentaires) pour Dashboard.

On ajoute un test d'intégration GET Bills pour les erreurs 404 et 500 :

```
// Test d'intégration GET 404 et 500
describe("Given I am a user connected as Employee", () => {
  describe("When I navigate to Bill", () => {
    test("Then, fetches bills from mock API GET", async () => {
      const getSpy = jest.spyOn(store, "get");
      // obtenir les bills et le nouveau bill
      const bills = await store.get();
      // getspy doit etre appelé au une fois
      expect(getSpy).toHaveBeenCalledTimes(1);
      // le nombre de bills doit etre de 4
      expect(bills.data.length).toBe(4);
    });
    test("Then, fetches bills from an API and fails with 404 message error", async () => {
      store.get.mockImplementationOnce(() => Promise.reject(new Error("Erreur 404")));
      // création interface user avec erreur de code
      const html = BillsUI({ error: "Erreur 404" });
      document.body.innerHTML = html;
      const message = await screen.getByText(/Erreur 404/);
      // attendre le message erreur 404
      expect(message).toBeTruthy();
    });
    test("Then, fetches messages from an API and fails with 500 message error", async () => {
      store.get.mockImplementationOnce(() => Promise.reject(new Error("Erreur 500")));
      // création interface user avec erreur de code
      const html = BillsUI({ error: "Erreur 500" });
      document.body.innerHTML = html;
      // attendre le message erreur 500
      const message = await screen.getByText(/Erreur 500/);
      expect(message).toBeTruthy();
    });
  });
});
```

[] composant container/NewBill :

- [] ajouter un test d'intégration POST new bill. Tu peux t'inspirer celui qui est fait (signalé en commentaires) pour Dashboard.

On ajoute un test d'intégration POST Bills pour les erreurs 404 et 500, il s'agit d'un mock de l'appel API POST et des erreurs 404 et 500. Tous les tests sont au vert et les mocks sont dans le dossier __mocks__ et importés dans les fichiers de test du composant testé :

```
//test intégration POST
describe("When I navigate to Dashboard employee", () => {
  test("Add a bill from mock API POST", async () => {
    const postSpy = jest.spyOn(store, "post");
    const bill = {
      id: "47qAXb6fIm2zOKkLzMr0",
      vat: "80",
      imageUrl:
        "https://test.storage.tld/v0/b/billable-677b6.a..f-1.jpg?alt=media&token=c1640e12-a24b-4b11-ae52-529112e9602a",
      status: "accepted",
      type: "Hôtel et logement",
      commentAdmin: "ok",
      commentary: "séminaire billed",
      name: "encore",
      fileName: "preview-facture-free-201801-pdf-1.jpg",
      date: "2004-04-04",
      amount: 400,
      email: "a@a",
      pct: 20,
    };
    const bills = await store.post(bill);
    expect(postSpy).toHaveBeenCalledTimes(1);
    expect(bills.data.length).toBe(5);
  });
  test("Add bills from an API and fails with 404 message error", async () => {
    store.post.mockImplementationOnce(() => Promise.reject(new Error("Erreur 404")));
    const html = BillsUI({ error: "Erreur 404" });
    document.body.innerHTML = html;
    const message = await screen.getByText(/Erreur 404/);
    expect(message).toBeTruthy();
  });
  test("Add bill from an API and fails with 500 message error", async () => {
    store.post.mockImplementationOnce(() => Promise.reject(new Error("Erreur 500")));
    const html = BillsUI({ error: "Erreur 500" });
    document.body.innerHTML = html;
    const message = await screen.getByText(/Erreur 500/);
    expect(message).toBeTruthy();
  });
});
```

- [] composant container/NewBill :
 - [] couvrir un maximum de "statements" : c'est simple, il faut que le rapport de couverture du fichier container/NewBill soit vert (accessible à [cette adresse](#) quand tu auras lancé le serveur). Cela devrait permettre d'obtenir un taux de couverture aux alentours de 80% dans la colonne "statements".

NewBill.js		96.15%	50/52
------------	---	--------	-------

On effectue 7 tests pour NewBill, ci-dessous un exemple de test unitaire :

```
//Quand je suis sur la nouvelle facture et que l'utilisateur accepte un format de fichier
//Alors, le nom du fichier doit etre correctement affiché dans l'input et imgformatvalide doit etre vrai.
describe("When I am on NewBill page, and a user upload a accepted format file", () => {
  test("Then, the file name should be correctly displayed into the input and ImgFormatValide should be true", () => {
    // localStorage - Employe

    window.localStorage.setItem(
      "user",
      JSON.stringify({
        type: "Employee",
      })
    );
    // construit l'interface utilisateur
    const html = NewBillUI();
    document.body.innerHTML = html;
    // Init onNavigate
    const onNavigate = (pathname) => {
      document.body.innerHTML = ROUTES({ pathname });
    };
    const store = null;
    // Init newBill
    const newBill = new NewBill({
      document,
      onNavigate,
      store,
      localStorage,
    });
    const handleChangeFile = jest.fn(() => newBill.handleChangeFile);
    const file = screen.getByTestId("file");

    window.alert = jest.fn();

    file.addEventListener("change", handleChangeFile);
    fireEvent.change(file, {
      target: {
        files: [new File(["file.png"], "file.png", { type: "image/png" })],
      },
    });
    // affichage message alerte erreur
    jest.spyOn(window, "alert");
    expect(alert).not.toHaveBeenCalled();

    expect(handleChangeFile).toHaveBeenCalled();
    expect(file.files[0].name).toBe("file.png");
    expect(newBill.fileName).toBe("file.png");
    expect(newBill.imgFormatValide).toBe(true);
    expect(newBill.formData).not.toBe(null);
  });
});
```

Chaque test est structuré de la manière suivante : un nom de composant, une condition ou expression testée et écrite de manière à qu'elle soit compréhensible par un product owner, et une description du comportement attendu. Chaque test est commenté dans le code pour en comprendre le fonctionnement rapidement.

Rapport de tests avant :

On constate que les tests pour Bills.js et NewBill.js sont KO à 0% :

PASS src/__tests__/Dashboard.js					
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	66.81	78.89	61.76	67.73	
constants	100	100	100	100	
routes.js	100	100	100	100	
usersTest.js	0	0	0	0	
containers	60.3	65.45	50.94	61.41	
Bills.js	48.28	50	25	50	14,20,24-27,36-56
Dashboard.js	89.29	80.49	79.17	95.83	142-145
Login.js	58.54	0	36.36	58.54	29,48-57,63-72,78-92
Logout.js	100	100	100	100	
NewBill.js	0	0	0	0	6-73
views	100	100	100	100	
Actions.js	100	100	100	100	
BillsUI.js	100	100	100	100	
DashboardFormUI.js	100	100	100	100	
DashboardUI.js	100	100	100	100	
ErrorMessage.js	100	100	100	100	
LoadingPage.js	100	100	100	100	
LoginUI.js	100	100	100	100	
NewBillUI.js	100	100	100	100	
VerticalLayout.js	100	100	100	100	
Test Suites: 2 failed, 9 passed, 11 total					
Tests: 3 failed, 39 passed, 42 total					
Snapshots: 0 total					
Time: 5.223 s					

Rapport de tests après :

On obtient bien un taux de couverture pour Bills.js et NewBill.js supérieur à 80% :

```
$ npm run test

> test
> jest --coverage --noStackTrace --silent
```

PASS	src/__tests__/ErrorPage.js
PASS	src/__tests__/LoadingPage.js
PASS	src/__tests__/VerticalLayout.js
PASS	src/__tests__/routes.js
PASS	src/__tests__/Actions.js
PASS	src/__tests__/DashboardFormUI.js
PASS	src/__tests__/Login.js
PASS	src/__tests__/NewBill.js
PASS	src/__tests__/Logout.js
PASS	src/__tests__/Dashboard.js
PASS	src/__tests__/Bills.js

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	87.5	81.73	72.86	89.12	
constants	100	100	100	100	
routes.js	100	100	100	100	
usersTest.js	0	0	0	0	
containers	85.05	71.64	64.81	87.06	
Bills.js	83.33	33.33	71.43	81.82	38-63
Dashboard.js	83.33	82.22	64	88.61	18-19,89,156,166-176,185-188
Login.js	73.17	0	45.45	73.17	27,47,58-70,76-92
Logout.js	100	100	100	100	
NewBill.js	96.15	75	77.78	96.15	108-110
views	100	100	100	100	
Actions.js	100	100	100	100	
BillsUI.js	100	100	100	100	
DashboardFormUI.js	100	100	100	100	
DashboardUI.js	100	100	100	100	
ErrorPage.js	100	100	100	100	
LoadingPage.js	100	100	100	100	
LoginUI.js	100	100	100	100	
NewBillUI.js	100	100	100	100	
VerticalLayout.js	100	100	100	100	

```
Test Suites: 11 passed, 11 total
Tests: 58 passed, 58 total
Snapshots: 0 total
Time: 9.534 s
```

Conclusion :

J'ai présenté et expliqué ici les différentes solutions apportées aux bugs, ainsi que les rapports de test unitaires et d'intégration et de couverture JEST actualisés. J'ai également expliqué comment j'ai ajouté les tests unitaires et d'intégration en prenant un exemple donné pour chacun.

Ce projet m'a permis d'écrire des tests unitaires et d'intégration avec Javascript, de déboguer une application web avec Chrome Debugger et de rédiger un plan de test End-To-End manuel.