Figure 1: ISIMA

Figure 2: Université de Clermont-Ferrand

Projet Pour l'ingénieur

Tests Unitaires

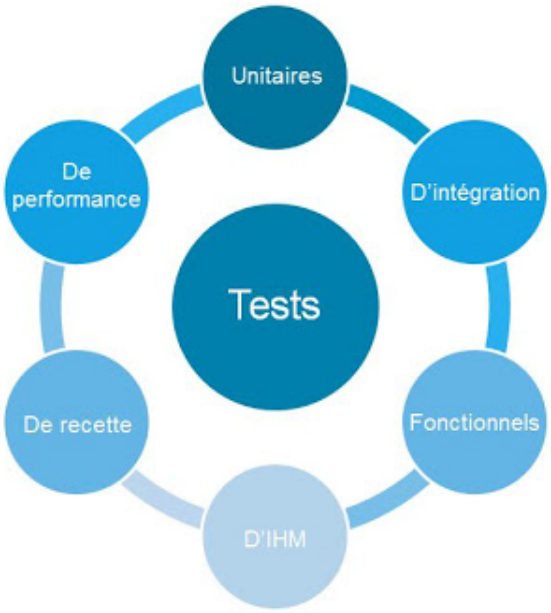

Figure 3: Tous les tests réalisables

CHASSAGNOL Rémi et CLIQUOT Théo
Professeur référent: Loic Yon
Travail à rendre pour le : 1/05/2021

Table des matières

Table des figures

1 Travail demandé

L'objectif de ce projet est de se familiariser avec la création d'une application web en prenant comme base la gestion de tests unitaires.
Il faudra concevoir et réaliser trois éléments pour cette application:

- Une interface web permettant la soumission d'un code (Java / C / C++ ou autres) et qui pourra afficher les résultats de tests unitaires sur ce code.
- Une partie serveur capable de lancer les tests unitaires
- Une base de données qui servira de «pont» entre la partie web et serveur.

Après 5 mois de travail sur ce projet, les 3 éléments principaux demandés ont été accomplis avec en plus quelques autres fonctionnalités ajoutées. Notamment le fait que n'importe quel langage pouvant être compilé par le serveur peut être testé et ceux sur autant d'exercices que l'on veut. Il suffit pour cela d'ajouter un nouveau test à la base de données. L'interface web a aussi été revue plusieurs fois afin de la rendre la plus ergonomique et simple à comprendre pour n'importe quel utilisateur.

2 Côté client : Html, CSS et JavaScript

2.1 Le contenu : l'html

Dans un premier temps on s'est surtout intéressé à la partie `HTML` qui était un tout nouveau langage pour nous (comme la plupart qui suivent). Sans commencer directement par la partie programmation. Il nous fallait déjà une idée des éléments HTML à utiliser et de leur disposition sur la page. Pour le squelette du site, nous nous sommes inspirés du site de **coding game** (voir 4). Le but étant de séparer les différentes parties de la page dans différents "panneaux" contenant la zone de texte pour le code et la zone d'affichage des résultats ainsi que des informations sur le site.

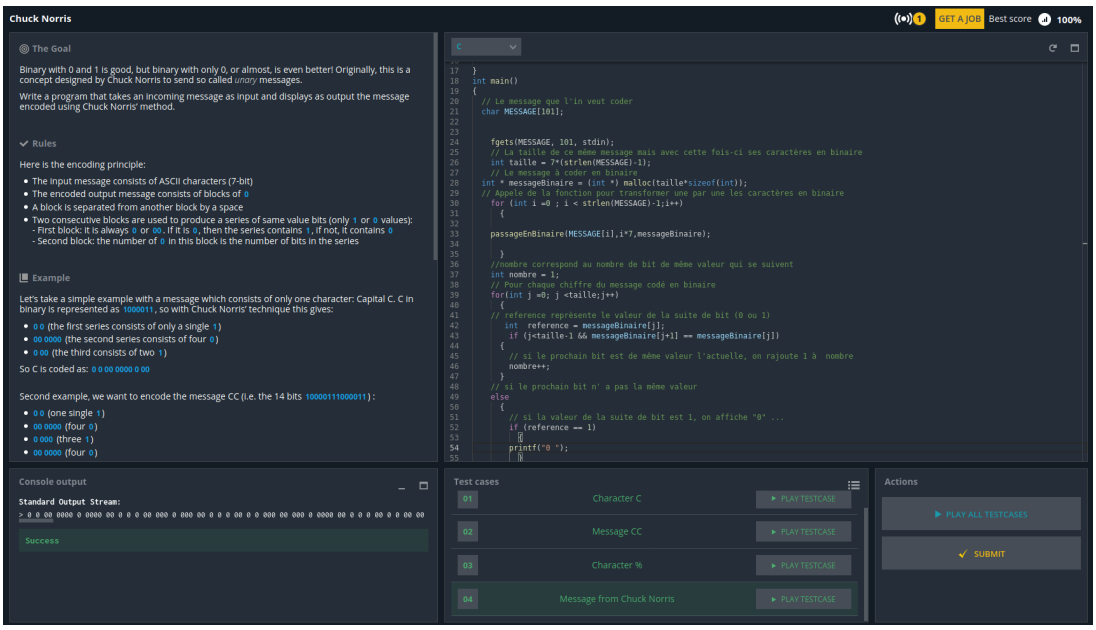

Figure 4: Le site de codingGame.

Le but serait donc d'avoir sur la partie gauche une zone pour entrer les informations et sur la droite tout ce qui concerne l'affichage des résultats ainsi que certaines informations.

La partie gauche contiendra donc forcément une balise `textArea` servant à rentrer notre code à tester et une balise `input` permettant à l'utilisateur de rentrer l'identifiant de l'exercice pour lequel il soumet une solution.

Dans la partie droite, nous souhaitions afficher

un paragraphe donnant de brèves indications sur le fonctionnement du site, ainsi qu'un bouton et une zone de texte permettant la soumission du code et l'affichage des résultats des tests. Pour cela le bouton doit appeler une fonction JavaScript que l'on verra plus tard. Une idée qui nous est venue un peu plus tard fut la mise en place d'un autre bouton afin d'obtenir des informations sur le test actuel. Ce bouton doit aussi appeler une fonction JavaScript qui agit de la même manière que le bouton pour envoyer le code.

Une fois une idée assez grossière de la composition de notre site, on s'est attelé à sa création. En travaillant sur la structure du site, nous nous sommes rendu compte que la partie gauche était beaucoup trop chargée en informations et texte et que cela nuisait à sa lisibilité. Nous avons donc décidé d'implémenter un système d'onglets en utilisant JavaScript pour faire de l'affichage dynamique. Cette nouvelle fonctionnalité nous a forcé à modifier notre page HTML ainsi que le CSS en cours de route mais le résultat obtenu était beaucoup plus satisfaisant. Cependant ce dernier sera expliqué plus loin. Le HTML n'est pas ce qui nous a pris le plus de temps car malgré un début compliqué dû à notre méconnaissance de ce langage, il est assez rapide à prendre en main et nous n'avions pas besoin de beaucoup de pratique pour réaliser notre objectif. Néanmoins, il nous à fallu plusieurs essais pour obtenir un résultat satisfaisant. De plus le code CSS et certaines parties du code JavaScript nous ont forcés à réécrire notre code. Au final, notre organisation en système d'onglet nous a permis de regrouper toutes les parties du site en une seule page HTML stockée dans le fichier `index.html`.

2.1.1 Explication du code

Le document `index.html` est composé des balises de base pour tout document HTML tel que le `<DOCTYPE>`, `<html>`, `<meta>`, `title`, `<head>` et `<body>`. On ajoute aussi dans la balise `header` cette ligne :

```
1       <link rel="stylesheet" href="./style.css" type="text/css"/>
2
```

Qui va permettre d'afficher la mise en forme `CSS` défini dans notre fichier `style.css` que nous détaillerons dans la partie dédiée.

Il y aussi juste au dessus une ligne très similaire mais qui elle permet de charger `base.css`. Cette stylesheet réinitialise toutes les balises utilisées afin d'avoir le même résultat sur tout les navigateurs. En effet d'un navigateur web à l'autre, les mises en forme de base peuvent changer et si on ne les redéfinie pas dans le CSS il peut arriver que les paramètres par défaut entrainent des problèmes dans la mise en page.

On va ensuite avoir 2 balises `div` afin de séparer les parties gauche et droite de l'écran avec la première contentant de quoi entrer le code et la seconde le reste comme dit précédemment. Elles auront comme identifiant `left-panel` et `right-panel`. Dans la première on va ajouter un `TextArea` pour accueillir le code de l'utilisateur:

```
1       <textarea spellcheck="false" id="code-input" class="code-input"
2                                 name="code" cols="100" rows="40">Put your code here!</
        textarea>
3
```

Juste après, un autre `textarea` plus petit va permettre d'ajouter une fonctionnalité définit en JavaScript mit au point lorsque qu'on s'entraînait. Celui-ci indique juste le nombre de lignes, de mots et de caractères qu'on à écrit dans le `textarea`. On a décidé de le laisser.

Ensuite on a un autre `input` afin que l'utilisateur puisse entrer l'identifiant de son programme:

```
1       <input type="text" id="exercise-id" name="Exercice ID" required />
2
```

On a ainsi tous les éléments composant la partie gauche de l'écran.

En ce qui concerne la partie droite, elle a beaucoup été modifiée suite à la partie JavaScript, notamment par l'ajout d'onglet. On va juste s'intéresser à cette partie une fois les onglets implémentés car c'est elle qui est restée dans le résultat final.

Dans un premier temps on a une liste de balise `<a></a>` qui vont servir pour changer d'onglets, elles correspondent sur la page au bouton contrôlant les panels. Ces dernières vont à chaque fois qu'elles sont

activées, lancer une fonction JavaScript que nous détaillerons plus tard. Ensuite on va avoir nos onglets et leur contenu dans des `div`. En effet se ne sera pas notre JavaScript qui va générer les panels en fonction des clics, ce dernier va juste s'occuper de cacher ou faire apparaître le panel que l'on désire voir. Un exemple de panel:

```
1       <div id="panel0" class = "tabContent">
2       <h2>INSTRUCTIONS</h2>
3       <h3>Fonctionnement du site</h3>
4       <p class="paragraphe-instructions">
5       Ce site web est destiner à tester vos réponce aux différents exercices
6       donnés par vôtre professeur. <br /> Utilisez le bouton en haut à gauche de
7       la page pour faire défiler les différents menus qui seront afficher sur la
8       partie droite de vôtre écran.<br />
9       </p>
10      <h3>Lancement des tests</h3>
11      <p class="paragraphe-instructions">
12      <ol>
13      <li>Collez vôtre code réponce dans la zone de texte prévue à cet effet.</li>
14      <li>Entez l'identifiant du jeux de tests à utiliser.</li>
15      <li>Cliquez ensuite sur le bouton <strong>"Lancer les tests"</strong>.</li>
16      </ol>
17      </p>
18      </div>
19
```

Enfin on finit en appelant justement les différents scripts JavaScript dont on a besoin. Ces appels sont à la fin pour être sûr que tout nos éléments HTML soient définis, permettant ainsi d'éviter de potentiels erreurs si JavaScript tente d'interagir avec un élément qui n'a pas chargé. Cela permet aussi d'éviter que rien ne s'affiche si JavaScript ne charge pas correctement.

2.2 La mise en forme : le CSS

Le CSS a été écrit parallèlement au HTML, en effet lorsqu'on modifiait le format de notre page web, certain éléments ne semblaient pas à leur place, On a donc pendant longtemps tâtonné et modifié des éléments mineurs pour obtenir un résultat satisfaisant.

2.2.1 Explication du code

L'un des premiers objectif était la mise en forme des parties gauche et droite de la page, pour qu'elles soient notamment à la bonne position. Cela est rendu possible avec le CSS via les attributs `justify-content` et en fixant leur taille à 50% de la taille du document (`width`). Il vaut mieux préférer les pourcentages (positionnement relative) aux valeurs absolues comme px, cm ou encore mm (positionnement absolue) afin d'avoir une mise en forme ajustable en fonction de la taille de la fenêtre. Cela permet une meilleur compatibilité avec n'importe quel type d'écran ainsi que de donner à l'utilisateur la possibilité d'ajuster sans problèmes la taille de notre site. Le reste des attributs dans les sélecteurs pour notre partie gauche et droite sont surtout là pour la couleur, ici la couleur de fond (`background-color`) et avoir des espacements plus marqués (`margin`).

```
1       div.left-panel {
2         justify-content: left;
3         margin-right: 30px;
4         margin-left: 100px;
5         margin-bottom: 25px;
6         background-color: #ffffff;
7         box-shadow: 0 1px 6px 0 rgba(32, 33, 36, 0.28);
8         width: 50%;
9       }
10
11
12      div.right-panel {
13        justify-content: right;
14        margin-right: 100px;
15        margin-bottom: 25px;
16        background-color: #ffffff;
17        box-shadow: 0 1px 6px 0 rgba(32, 33, 36, 0.28);
18        width: 50%;
```

```
19      }
20
```

Il nous a aussi fallu mettre en forme les éléments de base tel que le `body`, les `header`... Afin d'avoir les plus gros éléments composants la page avec un format plus attrayant de celui de base. On a donc pour le body des attributs pour modifier ses marges et lui donner une couleur plus visible.

```
1   body {
2   background-color: #e7e9eb;
3   padding: 0 0 0 0;
4   margin: 0;
5   }
```

Le header suit la même logique à quelques variations près, notamment l'ajout d'une `box-shadow`

```
1   header {
2     position: relative;
3     width: 100%;
4     background-color: #1E2127;
5     text-align: center;
6     padding: 0 0 0 0;
7     height: 60px;
8     margin-bottom: 25px;
9     box-shadow: 0 1px 6px 0 rgba(32, 33, 36, 0.28);
10  }
```

On a aussi modifié les titres de premier et second niveaux, notamment en changeant la couleur de leur texte, la couleur de fond et leur police.

```
1   header h1 {
2     margin-top: 0;
3     color: #ffffff;
4     background-color: #1E2127;
5     padding: 5px 0 0 0;
6   }
7
8
9   h2 {
10    text-decoration: underline dotted;
11    margin-left: 10px;
12  }
```

Les autres éléments importants à modifier étaient les inputs, contenu en majorité dans le panel gauche. Comme pour les autres éléments HTML qu'on a vu avant, ce qu'on a surtout modifié c'est leur couleur, leur taille, ou leur position ainsi que leurs marges ou bordures. A noter que pour leur taille on utilise toujours `width` avec des pourcentages afin de garder un aspect dynamique. Il y a aussi l'apparition de l'attribut `max-width` afin d'éviter que notre `textarea` ne dépasse la taille de notre panel gauche à force de modifier la taille de la fenêtre.

```
1   div.code-input {
2     display: block;
3   }
4
5
6   textarea {
7     background-color: #282c34;
8     color: #bbc2cf;
9     margin-left: 5%;
10    max-width : 90%;
11    line-height: 1.5;
12    border-radius: 5px;
13    border: 1px solid #ccc;
14    box-shadow: 1px 1px 1px #999;
15  }
```

```
16
17
18  div.id-input {
19    display: table-cell;
20    margin-left:auto;
21    margin-right:auto;
22    width: 65%;
23  }
24
25
26  div.result-frame {
27    background-color: #557883;
28    margin: 10px;
29  }
```

La partie qui suit a été rajouté lorsqu'on a voulu implémenter la notion d'onglets dans notre page web. En effet il fallait mettre en forme et modifier le positionnement des boutons pour changer d'onglets. On a d'abord cherché à utiliser des balise `button` cependant, celle-ci n'étant pas adaptées elles étaient très compliquées à manipuler et à positionner et on a finalement décidé d'utiliser une liste de liens. Chacun des éléments de la liste possède un fond noir qui devient blanc lorsqu'on passe la souris dessus, c'est une méthode très utilisée dans beaucoup de sites qui rend l'utilisation des boutons simple et intuitive (si un utilisateur passe la souris sur un élément qui change aussitôt de couleur, il sait par habitude que c'est un élément avec lequel il peut interagir en cliquant dessus).

```
1   .tab {
2     overflow: hidden;
3     border: 1px solid rgb(0, 0, 0);
4     background-color: #5578FF;
5   }
6
7   .tab ul {
8     display: flex;
9     justify-content: center;
10    list-style-type: none;
11    text-align: center;
12    width; 100%;
13    margin: 0;
14    padding: 0;
15  }
16
17  .tab li {
18    display: table-cell;
19    background-color: black;
20    width: 33.5%;
21    margin: 0;
22    padding: 0;
23    outline: none;
24    cursor: pointer;
25    font-size: 17px;
26    font-family: Arial, Helvetica, sans-serif;
27  }
28
29  .tab li:hover {
30    background-color: #ffffff;
31  }
32
33  .tab li:active {
34    background-color: #ccc;
35  }
36
37  .tab li a {
38    color: white;
39    text-decoration: none;
40    margin: 0;
41  }
42
43  .tab li:hover a {
44    color: black;
45  }
```

De même il fallait cacher les panels non voulus avec l'attribut `display:none` et les mettre en forme.

```
1
2  .tabContent {
3    display: none;
4    padding: 10px 10px 10px 10px;
5    border: 1px solid #ccc;
6    width: 90%;
7    height: 90%;
8    margin-bottom: auto;
9    margin-top: 3%;
10   margin-left: auto;
11   margin-right: auto;
12   max-width : 90%;
13   max-height : 90%;
```

Il y a aussi d'autres éléments qu'on a modifié mais il sont très similaires à d'autres présents ici.

Pour finir voici 2 mises en formes différentes de notre site, la première est une des toutes premières versions et la seconde celle que l'on peut voir actuellement.

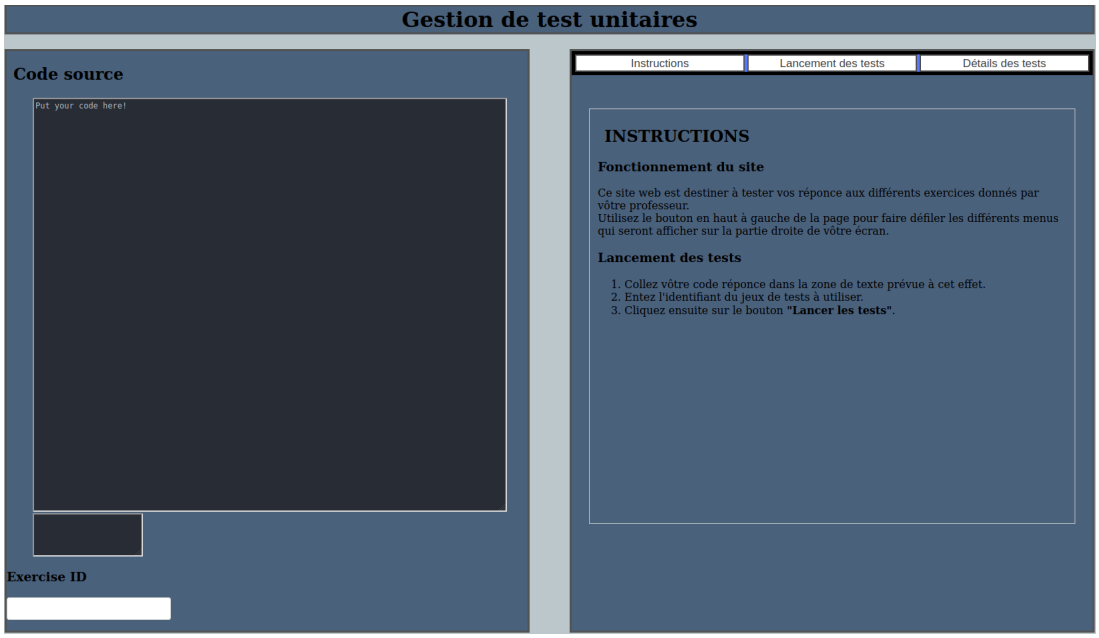

Figure 5: Un des tout premiers résultat satisfaisant

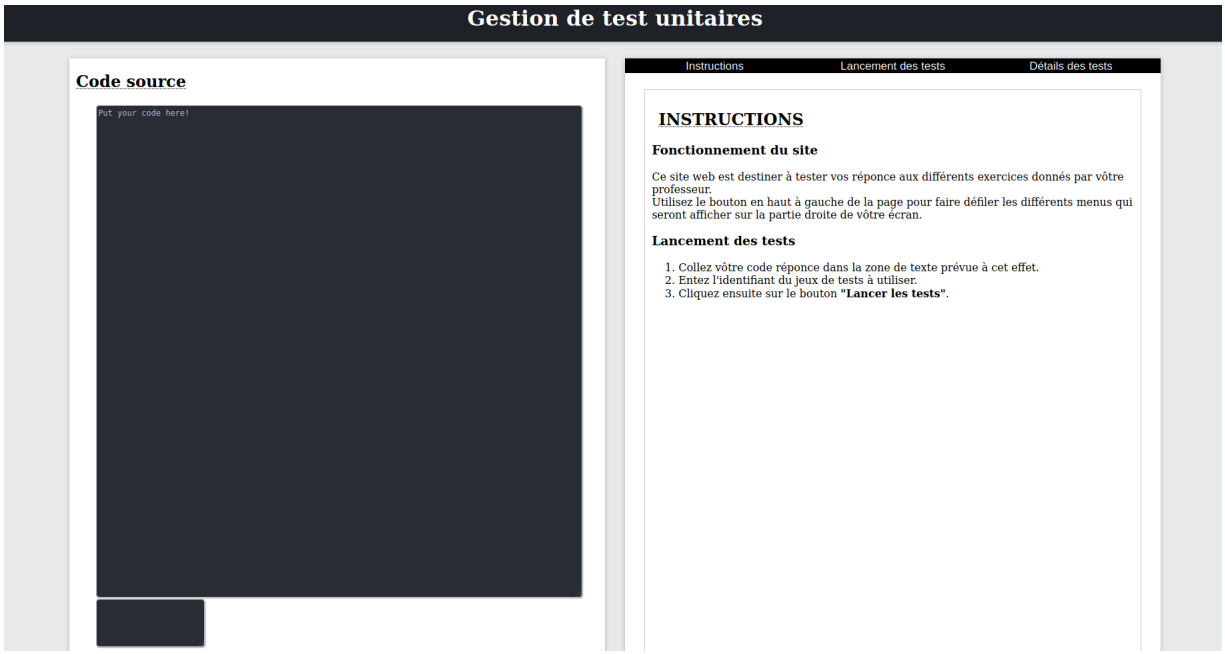

Figure 6: Le résultat final

2.3 Un peu de dynamisme côté client : JavaScript

Le JavaScript sert à deux choses dans notre page web:

- Apporter quelque fonctionnalités intéressantes que l'HTML ou le CSS ne peuvent pas faire. Ce qui concerne les onglets et un compteur de caractères, mots et lignes.
- La liaison entre notre côté client et le serveur, qui est une partie très importante car c'est via cette liaison qu'on va pouvoir communiquer avec la base de données et traiter le code des utilisateurs.

2.3.1 Premier cas : panel.js

L'un des premiers ajout assez simple fut le compteur de lignes, mots et caractères. Cette fonctionnalité n'était en aucun cas indispensable, cependant elle nous a permis de nous familiariser avec JavaScript avec un exemple concret (notamment la gestion des événements).

```
1  // Fonction pour indiquer le nbr de ligne,mot,lettre.
2  code.addEventListener('input', function (e) {
3      e.stopPropagation();
4      let compt_line = code.value.substr(0, code.selectionStart).split("\n").length;
5      let compt_word = compt_line + code.value.substr(0, code.selectionStart).split(" ").
       length - 1;
6      text_pos.value = "line :" + compt_line + "\nword :" + compt_word + "\nchar :" + code.
       value.length;
7  });
```

Cette fonction va être appelée chaque fois que l'utilisateur écrit dans le `textArea` contenant le code, pour cela on utilise un `addEventListener` qui prend en argument notre `event`. On va devoir exécuter dans cette dernière la fonction donnée en arguments. Cette fonction va regarder la string présente dans le `textArea` puis il nous suffit de compter le nombre de retours chariot (`\n`) pour avoir le nombre de lignes, le nombre de mots correspond de façon similaire au nombre d'espace plus le nombre de lignes, enfin pour le nombre de caractères il suffit juste de calculer la taille de la chaîne. Il faut ensuite afficher dans les éléments HTML voulus le résultat sous forme d'une petite phrase.

L'autre grande fonctionnalité dans notre fichier est la gestion des panels, qui se fait assez facilement mais qui donne un résultat bien plus propre pour la page web. Cependant il à fallut modifier aussi bien le HTML que le CSS pour inclure cette notion d'onglets comme expliqué dans les deux parties précédentes.

Notre script va juste s'occuper d'afficher le bon panel et de rendre les autres invisibles. Toute la partie texte se trouve dans le HTML et toute la partie forme dans le CSS.

```
// Fonction qui va afficher/cacher les onglets.
function changementPanel(panel)
{

   for (let g of tabContent) {
        g.style.display = "none";
    }

    document.getElementById(panel).style.display = "block";
}
document.getElementsByClassName("defaultOpen")[0].click();
```

Notre fonction va s'occuper de cacher tous les panels dans notre scène, puis d'afficher celui ayant la même classe que l'argument donné. Celle si sera appelée lorsqu'on appuie sur l'un des boutons de notre code HTML (du coup on a pas de `addEventListener`, c'est HTML qui va appeler cette fonction si il en a besoin). On à aussi dans notre document HTML un panel avec comme classe `defaultOpen` qui va nous permettre lorsque un utilisateur télécharge la page de déjà afficher un panel.

2.3.2 Second cas : script.js

Ce script gère tous ce qui à un lien avec la communication entre client et serveur. Et ceux en majorité à l'aide de la fonction `fetch` (sucre sytaxique pour faciliter l'utilisation de `XMLHttpRequest`). Pour ce qui concerne l'envoi de notre code (`recupCode`), il se divise en 3 requêtes vers notre serveur. La première va demander un identifiant afin qu'il puisse retrouver le résultat de son code sans problèmes à l'intérieur de la base de données (dans l'éventualité où plusieurs résultats sont mis en même temps dans la base). La seconde va poster notre code ainsi que son tout nouveau identifiant au serveur afin qu'il puisse la stocker dans la base de données et l'analyser par la suite. Et enfin la troisième et dernière requête va récupérer le résultat de notre code traité au bout d'un certain temps et l'afficher.

Pour simplifier les URL sur lesquelles on va envoyer nos requêtes, on utilise une variable globale `path` indiquant le chemin vers le dossier `tests_unitaires` contenant tout notre travail. En effet si on connait le chemin menant à ce dossier là, on peut appeler n'importe quel fichier de notre projet. Ce qui permet de ne modifier qu'une seule variable lorsqu'on change de serveur afin qu'il soit toujours fonctionnelle.

Requête pour obtenir un identifiant L'identifiant va être obtenu à l'aide d'une requête `HTTP GET`, en effet c'est le serveur qui génère l'identifiant et la récupéreration de données se fait bien avec une requête `GET` Pour cela on va appeler le code défini ci-dessous.

```
  // get a user id:
  async function  getId()
  {
    let code = document.querySelector('#code-input');
    // test id
    let id_exercice = document.querySelector('#exercise-id');

    return fetch(path+"php/get_id.php")
    .then((response) => response.json())
    .then(response => {
      user_id = response;
      console.log("GET ::: " + response);
    })
    .catch(error => {
      console.log("getIDError:" + error);
    })
  }
}
```

Cette fonction est la plus basique des trois. On va effectuer la requête sur un script `PHP` nommé `get_id` (qui sera expliqué comme tous les autres script PHP suivant dans leur partie dédiée (voir **??**)). La réponse fourni par ce script est au format `JSON`, on va donc appeler une première `promise` pour transformer notre réponse au format `JSON` en entier. Ensuite on va utiliser une autre `promise` pour stocker cet identifiant

dans une variable global de notre script. On en profite aussi pour afficher un message sommaire dans la console, ce qui est utile pour vérifier que les requêtes sont correctement effectuées.

Il faut bien noter que notre fonction contient le mot clé `async`. En effet on le verra par la suite mais il important que cette requête soit `await` car cela n'a aucun sens de commencer les requêtes suivantes sans posséder l'identifiant.

Requête pour envoyer notre code au serveur Cette fonction va utiliser la méthode `POST` afin d'envoyer nos informations (c'est à dire notre code ainsi qu'un identifiant pour savoir à quel tests il se rapporte) au serveur.

```
1
2       async function getCode()
3   {
4       return fetch(path+"php/get_code.php", {
5           method: 'post',
6
7       body: JSON.stringify({"user_id": user_id, "id_exercise": id_exercice.value, "code":
        code.value}),
8           headers: {
9               'Content-Type': 'application/json'}
10      })
11          .then(function (resp) {
12              console.log("REPONSE ::: ", resp)
13          })
14          .catch(function (err) {
15              console.log("ERROR ::: ", err)
16          })
17  }
```

La requête `POST` et un peu différente des `GET`, les informations à envoyer sont contenu dans le `body` de la requête. On à décidé dans notre cas d'envoyer nos données sous la forme `JSON`. Il faut aussi rajouter quelques informations à notre requête notamment le fait que ce soit un `POST` ou encore que les informations transmis sont sous le format `JSON` (défini dans le `headers`).

On va ensuite afficher la réponse obtenu mais seulement dans le but de voir l'état de notre requête, en effet vu que c'est un `POST` aucun retour n'est attendu normalement.

Le mot clé `async` et aussi très important dans cette fonction car on ne peut pas obtenir de réponse si notre code n'a même pas encore été reçu et traité par notre serveur.

Requête pour recevoir notre résultat au code envoyé La dernière requête appelée va être celle permettant de récupérer le résultat de notre code suite aux tests exécutés côté serveur. Pour cela on va utiliser une méthode `GET` (cependant il y a modification des données côté serveur, ce qui est contradictoire avec le fait que la requête `GET` doit respecter le principe d'idempotence et être sans effets de bord comme on va le voir ici [3.1.3], mais nous ne savions pas quel autre requête utiliser). Cette fonction va être très similaire à `getID`, à la seule différence notable que l'on va envoyer via l'URL l'identifiant de notre code (voir ci-dessous).

```
1   var urlResult = path+"php/get_result.php?user_id=" + user_id;
```

Les deux autres différences sont juste que notre réponse n'est pas sous le format `JSON` et il donc inutile de le transformer. De plus la réponse doit être affichée non pas dans la console mais dans l'élément `HTML` prévu à cette effet.

```
1   // get the result of our code
2   async function getTest()
3   {
4       var urlResult = path+"php/get_result.php?user_id=" + user_id;
5       await fetch(urlResult).then(data => {
6           console.log("GET ::: ", data);
7           data.text().then(function (text) {
8
9               document.querySelector('#result-output').innerText = text;
10          });
11      }).catch(error => {
```

```
            console.log(error);
            document.querySelector('#result-output').innerText = error;
        })
    }
```

Une fois cette requête et ses promise terminées, notre page devrait afficher le résultat du tests de notre code (dans le cas ou aucune erreur n'est apparu).

Rassemblement des ces trois fonctions Pour avoir le résultat escompté lorsqu'on clique sur le bouton, il suffit donc d'appeler à la suite ces trois fonctions (en n'oubliant pas d'enlever le fait qu'elles sont asynchrones de base).

```
async function recupCode() {
    if(document.getElementById('exercise-id').value != "")
    {
        await getId();
        await getCode();
        setTimeout(getTest,2000);
    }
}
```

get_test ne peut se lancer que lorsque notre requête dans get_code est fini, c'est à dire lorsque le serveur reçoit nos informations. Cependant on ne peut pas être sûr qu'elle soit traitée et analysée dans la foulée, on à donc préféré attendre deux seconde afin de bien laisser le temps au serveur de traiter notre code.

Voici comment se déroule la communication entre JavaScript et php lorsque l'utilisateur appui sur le bouton pour lancer les tests:

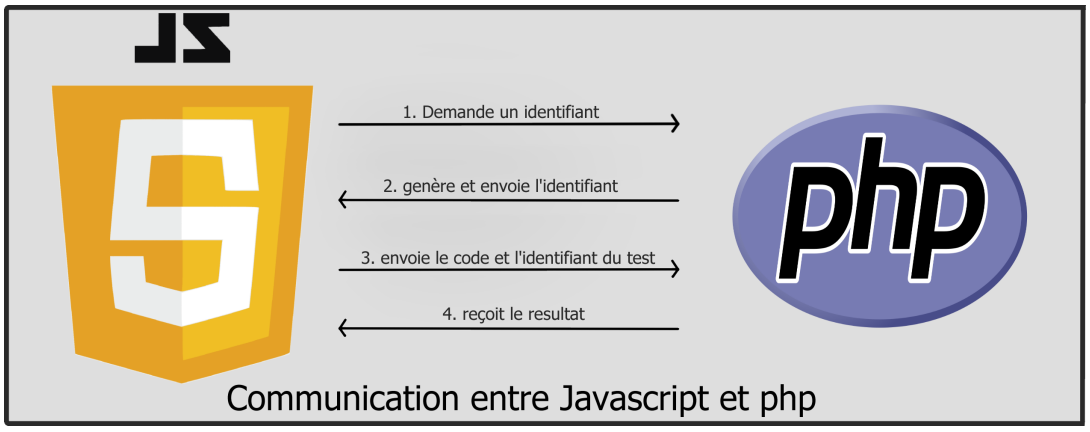

Figure 7: Communication entre JavaScript et php

2.3.3 Une autre requête : obtenir des informations

On a aussi un bouton sur notre page pour avoir plus d'informations et de détails sur les tests. Cette fonction utilise le même principe que getTest.

```
function displayTestsDetails() {
    var urlInfo = path+"php/get_info.php?id_exercise=" + id_exercice.value;
    fetch(urlInfo).then(response => response.text()).then(data => {
        console.log(data);
        document.querySelector('#test-details').innerText = data;
    }).catch(error => {
        console.log(error);
        document.querySelector('#test-details').innerText = error;
    })
}
```

3 Côté serveur : PHP python et SQL

3.1 Le lien entre les deux, le PHP

Dans notre projet, php va servir seulement pour recevoir les requêtes effectuées par le côté client grâce au JavaScript avec `fetch` et aux différentes actions à mener en fonction de ces requêtes. Pour cela on a divisé les tâches en 4 grandes parties, chacune traitée dans un script différents.

Pour pouvoir envoyer des requêtes en PHP à la base de données on utilise l'extension `PDO`.

Le PHP était aussi un langage totalement inconnu pour nous deux et ce projet nous à permit d'en avoir un avant goût.

3.1.1 Obtention d'un ID (script get_id)

Notre client va envoyer son code au serveur qui va donc être stocké dans la base de données, cependant comme décrit plus loin il va falloir se rappeler quel est son code pour pouvoir lui renvoyer la réponse adéquate. Pour cela on va lui donner une ID (un entier) compris entre 1 et 10000 *unique* (dans le cas ou le site devrait avoir à interagir avec plus de 10000 utilisateurs, on peut augmenter ce nombre ou même utiliser une suite de caractères alphanumériques). Pour cela la méthode utilisée consiste à obtenir de façon aléatoire un entier, et ensuite on regarde si il n'existe pas une ID équivalente dans la base de données, si il n'y en a pas alors notre ID est bien *unique* et il nous suffit alors de `echo` de notre ID (ce qui va être echo est ce qui va être reçu par fetch en réponse). Dans l'autre cas on recommence l'étape jusqu'à ce que l'ID soit bonne. Dans le cas où tous les ID possible sont utilisés (ce qui est très peu probable mais possible). Notre boucle va être infini, cependant on réalise la requête à chaque tour, ainsi notre script python peut entre temps libérer une ID, la rendant disponible. Une fois notre ID pour le client trouvée on prépare la base de données à recevoir un code avec cette ID, ainsi, si une requête est effectuée dans la foulée par un autre utilisateur il n'aura pas la possibilité de prendre la même ID.

```php
<?php
  #header("Access-Control-Allow-Origin: *");
  #header("Content-Type:application/json");

  function isIN($newID, $ids) {
    while($id = $ids->fetch()) {
      if ($newID == $id['idProgramme']) {
        return true;
      }
    }
    return false;
  }

  # Getting current used id list
  $myPDO = new PDO("sqlite:../db/sql/tests_unitaires.db");
  $newID;

  # Generating new id
  do {
  $ids = $myPDO->query("SELECT id_programme FROM programme_utilisateur;");
    $newID = random_int(0, 10000);
  } while (isIN($newID, $ids));

  # Adding newID to db:

  $request = $myPDO->prepare("INSERT INTO programme_utilisateur(id_programme,
    code, test) VALUES (?, NULL, NULL)");
  $request->execute(array($newID));
  echo $newID;
?>
```

3.1.2 Stocker le code dans la base de donnée (script get_code)

Une fois l'ID obtenu pour le code et une place réservée dans la base de données. Il reste encore à stocker le code ainsi que l'identifiant du test. On va donc devoir dans un premier temps récupérer ces deux variables et pour cela un problème s'est posé: en effet normalement toute information fournie via une requête HTTP `POST` peut être récupérée en PHP à l'aide de la variable spéciale `$_POST`. Cependant,

l'information transmise sous le format JSON n'est pas placée dans cette variable, il faut donc la décoder pour pouvoir l'utiliser. Une solution à ce problème fut d'utiliser ce bout de code ci-dessous.

```php
<?php

  header("Access-Control-Allow-Origin: *");
  header("Content-Type:application/json");

  $personalPost = json_decode(file_get_contents('php://input'),true);

  $userID = $personalPost['user_id'];
  $exerciseID = $personalPost['id_exercise'];
  $code = $personalPost['code'];


```

Ensuite on réalise une simple requête UPDATE pour ajouter les variables code et exerciseID dans le tuple de la table programme_utilisateur.

```php
  $myPDO = new PDO('sqlite:../db/sql/tests_unitaires.db');
  $request = $myPDO->prepare("UPDATE programme_utilisateur SET code = ?, test
    = ? WHERE id_programme = ?;");

  $request->execute(array($code,$exerciseID,$userID));
?>
```

3.1.3 Obtenir les résultats (script get_result)

Ce script va aussi s'exécuter en deux temps. Lorsqu'on va appeler ce script il va tout d'abord nous renvoyer le résultat du test stocké dans la base de données, plus précisément dans la table retour (qui sera remplie via le script python que nous détaillerons plus tard). Pour cela on va de la même manière qu'avant faire une requête SQL avec PDO en précisant que l'on veut le tuple avec notre identifiant fourni par get_id. Pour récupérer cette identifiant il suffit juste d'utiliser la variable spéciale $_GET de PHP, qui consiste en un tableau contenant chacune des informations passées à la requête HTTP de tel sorte que la clé soit le nom de la variable et le résultat le contenu de cette case.

Ensuite une fois qu'on a echo le résultat (c'est à dire que l'utilisateur va recevoir la réponse à son test) il nous suffira de supprimer celui-ci de la base de données, afin de libérer la place maintenant inutile et éviter par la suite de renvoyer ce code si une réponse avec le même identifiant arrive.

```php
<?php
  ini_set('display_errors', 1);
  ini_set('display_startup_errors', 1);
  error_reporting(E_ALL);

  $userID = $_GET['user_id'];
  $myPDO = new PDO('sqlite:../db/sql/tests_unitaires.db');

  $request = $myPDO->prepare("SELECT resultat FROM retour WHERE id_utilisateur = :id");
  $request->execute(['id' => $userID]);

  while($res = $request->fetch())
  {
      echo str_replace("\\n","\n",$res["resultat"]);
  }

  $request = $myPDO->prepare("DELETE FROM retour WHERE id_utilisateur = :userID;");
  $result = $request->execute(['userID' => $userID]);
?>
```

3.1.4 Information sur le test (script get_info)

On a aussi intégré un moyen pour l'utilisateur d'avoir directement sur le site un résumé rapide de ce qui est attendu pour le test (ce résumé doit être écrit préalablement dans la base de données dans la variable `info_test` présent dans la table `test`). Ainsi il aura juste à marquer le bon identifiant du texte dans le `textarea` prévue à cette effet et cliquer sur le bouton pour avoir ce résumé.

Pour cela le code php est très similaire à ce qui a été vu précédemment. On récupère dans la variable spéciale `$_GET` de PHP notre `id_exercise` contenu dans le `textarea`, puis on effectue une requête à la base de données pour récupérer la variable contenant le résumé vu plus haut, et enfin on le retourne avec un echo.

3.2 La base de données

Pour la gestion de la base de données, nous avons décidé d'utiliser `SQLite` car c'est un outil simple d'utilisation que nous avions vue dans le cours de base de données enseigné à la fac. Le projet ne nécessitait pas de réflexion très aboutie sur la construction du système d'information, cependant, nous avons tout de même décidé de créer une base de données assez complexe (pour nôtre niveau) pour deux principales raisons. D'une part pour simplifier de potentiels ajouts de fonctionnalités plus avancées au site et d'autre part, pour approfondir et mettre en pratique ce que nous avions appris durant nôtre quatrième semestre.

3.2.1 Conception de la base

La construction de la base s'est faite en plusieurs étapes, dès le départ, il semblait évident qu'il y avait trois principales informations à stocker. Tout d'abord, il y a les **tests**, le but du site étant de faire de la gestion de tests unitaires, cela nécessite un moyen de stocker ces tests, ainsi que toutes les informations les concernant comme le langage, la commande de compilation ou encore de potentielles informations les décrivant. Les deux autres informations à stocker vont de paire puisqu'il s'agit des **programmes à tester** (envoyés via une requête HTTP depuis un navigateur) et des **résultats aux tests**. A la fin de cette première étape, nous avons séparé ces trois informations dans trois différentes tables `tests`, `programme_utilisateur` et `retour`.

Après la création des trois tables décrites ci-dessus, une autre question s'est posée à nous: *quels informations devons nous exactement stocker dans chacune des tables ?* Pour la table `programme_utilisateur`, il nous faut un identifiant que l'on a appelé `id_programme` qui est la **clé primaire** de la table de type `TEXT` (nous avons convenu que cela était moins restrictif qu'un simple entier). Ensuite, nous avons deux autres attributs, `code` de type `TEXT` qui correspond au code envoyé par l'utilisateur du site et un attribut `test` qui est une **clé étrangère** correspondant à l'identifiant du test demandé par l'utilisateur (lui aussi de type `TEXT`).

```
1   CREATE TABLE programme_utilisateur
2   (
3     id_programme TEXT,
4     code TEXT,
5     test TEXT,
6     PRIMARY KEY(id_programme),
7     FOREIGN KEY(test) REFERENCES tests(id_test)
8   );
9
```

La table `retour` est identifiée par l'attribut `idRetour` (`TEXT`), elle possède un attribut `resultat` (correspondant au résultats des tests) et une clé étrangère `idUtilisateur` correspondant à l'identifiant de la table décrite précédemment.

```
1   CREATE TABLE retour
2   (
3     id_retour TEXT,
4     resultat TEXT,
5     id_utilisateur TEXT,
6     PRIMARY KEY(id_retour),
7     FOREIGN KEY(id_utilisateur) REFERENCES Utilisateur(id_utilisateur)
8   );
9
```

Pour la création des attributs de `tests`, nous avons du commencer par étudier l'organisation de la partie serveur du site pour comprendre comment les programmes sont stockés et compilés. Nous en avons conclus qu'il était important de stocker les commandes de compilation et d'exécution dans les attributs `command_compilation` et `command_execution` qui seront utilisés par **python**. A noter que si un programme est écrit dans un langage interprété comme *python* ou encore *haskell* (qui peut être compilé et interprété), le champ `command_compilation` sera vide et `command_execution` contiendra la commande pour exécuter le programme (par exemple `python3 prog.py` pour python ou `runghc prog.hs` pour haskell). Il faut aussi stocker un chemin vers le répertoire qui contient le fichier source du tests (par exemple, `../data/cMaths` qui contient le fichier `main.c`) dans l'attribut `path_to_file` pour les suppressions de fichiers temporaires comme les fichier de code sources qui contiennent le code de l'utilisateur du site ou les fichiers binaires. Cette table possède aussi trois autres attributs, `id_test` qui est la clé primaire de la table, `info_test` qui contient une description du test passé (et qui peut être `NULL`) et `langage` qui correspond à l'extension de fichier du langage (**py** pour python par exemple).

```
1   CREATE TABLE tests
2   (
3     id_test TEXT,
4     langage TEXT,
5     command_compilation TEXT,
6     command_execution TEXT,
7     nom_compilation TEXT,
8     nom_execution TEXT,
9     path_to_file TEXT,
10    info_test TEXT,
11    PRIMARY KEY(id_test)
12  );
13
```

Cette dernière table a été réécrite plusieurs fois durant le développement du script **python**. En effet, nous avons commencé **SQL** bien avant **python** et nous ne connaissions pas encore le module `subprocess` utilisé par le script. De plus, certaines fonctionnalités ont été apportées durant le développement. Au départ, nous avions pensé le site pour gérer des tests unitaires uniquement pour des programmes écrits en **C**, cependant, nous avons jugé plus intéressant et moins restrictif que de le rentre compatible avec tous les langages. Ce choix nous a obligé à stocker la commande d'exécution dans la table en plus de la commande de compilation, pour pouvoir travailler avec des langages exécutés par des machines virtuelles comme **Java**. Il aurait été facile d'utiliser la notations **shebang** pour manipuler des programme écrits dans des langages comme python de la même façon que des fichier binaires générés par un compilateur(`./prog`), mais cela n'aurait pas été possible avec les langages comme **Java**. Dans le cas de **Java**, la solution aurait été de passer par un script **shell**, mais cela aurait rendu l'utilisation du site beaucoup plus complexe pour la personne qui fourni les tests.

3.3 Python

Pour compiler et exécuter les programmes des utilisateurs du site puis mettre à jours la table `retour` de la base de données, nous avons décidé d'utiliser un script python avec les modules `sqlite3` et `subprocess`. Le module `sqlite3` permet au programme de se connecter à la base de données **SQLite** de façon similaire à **php** et le module `subprocess` permet d'exécuter des commandes shell et de récupérer leur sortie au format textuel.

3.3.1 Intéraction avec SQLite

Comme expliqué précédemment, le module `sqlite3` permet de se connecter à la base de données et d'exécuter des requêtes **sql** de façon similaire à php. On utilise la fonction `connect()` qui prend en paramètre le chemin vers le fichier **db** (le fichier contenant la base de données générée par SQLite) sous forme de chaîne de caractères et retourne un objet qui représente la base de données. Pour créer la connexion dans le script, on utilise une fonction nommée `create_connection()`:

```
1           """ Return a connection to the db corresponding to the path given as
2           parameter """
```

```
3           connection = None
4           try:
5               connection = sqlite3.connect("./sql/tests_unitaires.db")
6               #print("connection to SQLite DB successful")
7           except Error as err:
8               print(f"Connection to DB failed: {err}")
9
10          return connection
11
```

d

Cette fonction retourne la connections vers la base, et en cas de problème, affiche le message d'erreur généré par la fonction `connect()`.

Une fois que la connections vers la base de données est créée, il faut générer un curseur pour pouvoir exécuter des requêtes. Cela se fait à l'aide de la méthode `cursor()` de l'objet `connection`.

```
1       cur = connection.cursor()
2
```

c

L'exécution de requêtes **sql** se fait alors de la façon suivante:

```
1       request_result = cur.execute(sql_select).fetchall()
2
```

s

La méthode `execute()` prend en paramètre une chaîne de caractères contenant la requête et l'exécute. Pour récupérer les données, on peut ensuite utiliser les méthodes `fetchall()` et `fetchone()`. Dans l'exemple ci-dessus, on utilise la méthode `fetchall()` qui retourne un objet itérable qui contient les informations extraites. La mise à jours de la base se fait de façon similaire, en exécutant une requête de type `INSERT` ou `UPDATE`. A noter que pour que les modifications apportées à la base soient sauvegardées, il faut utiliser la méthode `commit()` de l'objet connections et il faut utiliser la méthode `close()` pour fermer la connections.

3.3.2 Les fonctions intermédiaires

Comme expliqué dans la partie précédente, nous savons maintenant comment interagir avec la base de données **SQLite**. Maintenant, il faut pour chaque entrées de la table `programme_utilisateur`, effectuer les tests et mettre à jours la table `retour` de la base. Pour cela nous avons utilisé une boucle infinie, dans laquelle on vérifie si il y a des informations des la table `programme_utilisateur`.

Avant de détailler le ce qui ce passe dans la boucle, commençons par décrire les différentes fonctions intermédiaires utilisées dans le script.

Tout d'abord, nous avons une fonction qui nous permet de récupérer toutes les informations sur le test demandé par l'utilisateur du site:

```
12          if id_exercice != None:
13              ids = cur.execute("""SELECT id_test FROM tests;""").fetchall()
14
15              l = [i[0] for i in ids]
16              if id_exercice in l:
17                  sql_test = f"""SELECT langage, command_compilation,
18                      command_execution, path_to_file, nom_compilation,
19                      nom_execution FROM tests WHERE id_test='{id_exercice}';"""
20                  return cur.execute(sql_test).fetchone()
21          return None
22
```

d

Le fonctionnement est le suivant: on commence par sélectionner tous les identifiants disponibles dans la table `tests` à l'aide d'une requête SQL et on les met dans une liste. Si l'identifiant demandé n'est pas valide, il ne se trouve pas dans la liste, dans ce cas on met à jour la table retour avec un message d'erreur pour faire comprendre à l'utilisateur du site qu'il s'est trompé lorsqu'il à écrit l'identifiant du test et on retourne `None`. Sinon, on récupère toutes les informations utiles comme les commandes de compilation et d'exécution et on les retourne sous la forme d'un tuple.

Ensuite, on a une fonction `make_test()` permettant de compiler et d'exécuter les programmes pour passer les tests.

```
23                      command_compilation, command_execution, n_comp, n_exec):
24
25          generateUserFile(path_file, code, language, n_comp)
26
27          # compilation / execution
28          retourCompilation =
29              subprocess.run(f"cd ../data/{path_file} && {command_compilation}",
30                             stdout=subprocess.PIPE, stderr=subprocess.PIPE,
31                             shell=True, timeout=1, text=True)
32          retourExecution =
33              subprocess.run(f"cd ../data/{path_file} && {command_execution}",
34                             stdout=subprocess.PIPE, stderr=subprocess.PIPE,
35                             shell=True, timeout=1, text=True)
36
37          # delete all tmp file.
38          if n_comp != "":
39              subprocess.run(
40                  f"rm ~/serveur/www/tests_unitaires/data/{path_file}{n_comp}.{language}",
41                  shell=True)
42          if n_exec != "":
43              subprocess.run(
44                  f"rm ~/serveur/www/tests_unitaires/data/{path_file}{n_exec}", shell=True)
45          return retourCompilation, retourExecution
46
```

d

Cette dernière fait appel à la fonction `generateUserFile` qui permet de copier le code envoyé par l'utilisateur dans un fichier dans le même répertoire que les fichiers de code source des tests avant de pourvoir compiler et exécuter.

```
47          with open(f"../data/{path_file}{n_comp}.{language}", "w", encoding="utf8") as
        code_file:
48              code_file.write(code)
49
```

d

Une fois que le fichier contenant le code source de l'utilisateur a été créé, on fait appel aux fonctions de `subprocess` pour compiler et exécuter les programmes, puis récupérer le résultat des commandes shell dans des variables, qui seront retournées à la fin de l'exécution. A noter que l'on ne retourne pas uniquement le résultat de l'exécution du code mais aussi les erreurs générées par les différentes commandes. Cela permet de faire comprendre à l'utilisateur que son code ne compile pas ou ne s'exécute pas et qu'il a certainement mal copié son code dans la zone de texte prévue à cet effet sur le site. À la fin, la fonction va supprimer les fichiers temporaires dont le fichier de code source créé au début de l'exécution de la fonction mais aussi les fichier exécutables pour les langages compilés. Il est très important de ne pas oublier de supprimer le fichier exécutable, en effet, le fichier de code source lui est toujours modifié, on pourrait donc ne pas le supprimé. Cependant, si le code ne compile pas, l'ancien fichier binaire n'est pas modifié (dans le cas ou il n'a pas été supprimé), donc le binaire exécuté sera celui de l'utilisateur précédent. Supprimer les fichiers temporaires permet aussi d'éviter que ces dernier ne prennent trop de place sur le serveur. De plus, par sécurité, il est fortement déconseillé de laisser des fichiers temporaires avec des droits d'exécution sur un serveur, surtout quand on ne connait pas leurs contenu.

Nous avons rencontré plusieurs problèmes avec `subprocess`, notamment au niveau de l'exécution des commandes pour compiler les programmes en **C**. En effet, nous utilisons le compilateur **gcc** qui ne supporte pas le fait de pouvoir compiler depuis un autre répertoire, ce qui nous a obligé à utiliser la commande `cd../data` $path_file pour faire en sorte que `subprocess` se place dans le bon rpertoire rep `"../data/ path_file"`.

Enfin, on a la fonction `update_db()` qui utilise les fonctions du module `sqlite3` pour exécuter des requêtes SQL et mettre à jours la base de données.

```
50          """ Update the table 'retour' of the db with id_result, result and id_user,
51          which are string given as parameter. """
52          cur.execute("INSERT INTO retour(id_retour, resultat, id_utilisateur) values
        (?,?,?)",
53                      (id_result, result, id_user))
```

```
54          connection.commit()
55
```

d

Les autres fonctions son secondaires, ce sont des assesseurs ou des fonctions intermédiaires permettant d'améliorer la lisibilité du code et le formatage des résultats. Par exemple, la fonctions `update_db_result()` permet de mettre le résultats des tests dans la base quand il n'y a pas eu d'erreur au niveau de l'identifiant du test. Elle utilise le retour de la fonction `parse_result()` qui retourne la chaine de caractères finale à afficher.

3.3.3 La boucle infinie

Maintenant que l'on connait toutes les fonctions intermédiaires importantes, passons à la boucle infinie. Tout d'abord, on itère sur toutes les entrées de la table qui contient les programmes à compiler à l'aide d'une boucle `for`. Pour chaque entrée, on va mettre les informations retournées par la fonction `withdrawInformation()` dans une variable `information`. Si l'identifiant du test était valide, cette variable ne contient pas `None` et on peut compiler. Dans le cas contraire, on se contente de supprimer l'entrée de la table `programme_utilisateur` et de passer à la suivante.

Pour compiler les programmes et récupérer le résultats des tests, on fait appel à la fonction `make_test()`. Comme expliqué précédemment, cette dernière prend en paramètre le code source envoyé par l'utilisateur et les informations sur le test et retourne le résultat des tests dans un tuple. Ensuite, on met à jours la base de données avec la fonction `update_db_result()`, qui, comme expliqué précédemment, va permettre de formater le résultat avant de le mettre dans base. Enfin, on supprime l'entrée de la table `programme_utilisateur` et on passe à la suivante et ainsi de suite jusqu'à ce qu'il n'y ait plus de programme à tester. A noter qu'après chaque tours de boucle, on sauvegarde les modifications apportées à la base de donnée à l'aide de la méthode `commit()` de l'objet `connection` décrit en début de cette partie.

```
1       while True:
2           count_rows = cur.execute(
3               "SELECT * FROM programme_utilisateur;").fetchall()
4           if len(count_rows) > 0:
5               sql_select = """SELECT * FROM programme_utilisateur;"""
6               for id_user, code, id_exercice in cur.execute(sql_select).fetchall():
7                   print(id_user, code, id_exercice)
8
9                   information = withdrawInformation(id_exercice)
10                  if information != None:
11
12                      result = make_test(code, id_user, id_exercice,
13                                         path_file(information), language(information),
14                                         command_compilation(information),
15                                         command_execution(information),
16                                         name_comp(information), name_exec(information))
17
18                      update_db_result("r"+id_user, result, id_user)
19
20                      sql_supr = f"""DELETE FROM programme_utilisateur
21                      WHERE id_programme='{id_user}';"""
22                      cur.execute(sql_supr)
23
24                  elif code != None:
25                      update_db("r"+id_user, "Error : Pas la bonne ID", id_user)
26                      sql_supr = f"""DELETE FROM programme_utilisateur
27                      WHERE id_programme='{id_user}';"""
28                      cur.execute(sql_supr)
29
30                  connection.commit()
31
```

Ce script python est fait pour s'exécuter en arrière plan sur le serveur, il est nécessaire que celui-ci soit actif pour que le site fonctionne, il faudra donc le lancer avant le déploiement du site. Il faudra aussi le relancer après chaque mise à jour de la base de données si l'on souhaite rajouter des tests. Dans le cas contraire les modification apportées à la base pourraient ne pas être prises en compte par python ou même générer une erreur lors de l'exécution.

4 Les limites du projet

Avant de conclure, on va s'intéresser aux limites du projet, car si le site fonctionne et que les objectifs de départ ont été atteint, il reste encore des choses à améliorer.

Tout d'abord, la récupération des résultats avec JavaScript fonctionne, cependant, les tests que l'on a fait on été effectués dans des conditions parfaites, il n'y avait jamais plus de deux personnes connectés au site. Pour que le site puisse supporter plus de connections, il faudrait changer le fait que JavaScript attende 1 seconde avant de demander les résultats et de s'arrêter. En effet, si le code n'a pas été compilé et que les tests n'ont pas encore été effectués par python avant la fin du `sleep(1000)`, l'utilisateur ne verra jamais les résultats. Pour remédier à ça, il faudrait ajouter la possibilité de demander les résultats plusieurs fois, en regardant si ils sont disponibles tout les X temps (10 secondes par exemple) jusqu'à ce qu'on obtienne les résultats ou une erreur. Une autre solution serait de passer par une **Push API** qui permet de recevoir des notifications du serveur. C'est notifications sont asynchrones à l'application ce qui permet d'envoyer des messages sur l'état de l'exécution des scripts sur le serveur ou encore sur l'état de la base de données.

Une autre amélioration possible pourrait porter sur la création d'une meilleur interface coté serveur, notamment sur l'ajout de nouveaux tests. Par exemple, on aurait pu ajouter le fait de pouvoir passer par un fichier `csv` pour mettre des informations dans la table `tests`.

On pourrait aussi rendre le site moins fragile, en faisant plus de gestion d'erreur. Par exemple, si un utilisateur se déconnecte avant d'avoir effectué la requêtes permettant de recevoir ses résultats, ces derniers ne sont jamais supprimés de la base de données puisque la suppressions est faite dans le script `get_result` qui n'a pas été appelé dans ce cas. On pourrait donc avoir un autre script qui supprime toutes les entrées trop anciennes de la bases.

Enfin, concernant l'affichage coté client, on pourrait remplacer la `textarea` par quelque chose de plus évolué et plus puissant. Par exemple, on pourrait avoir un petit éditeur similaire à celui utilisable sur le site `caseine` avec de la coloration syntaxique.

Le site n'est pas encore parfait et il pourrait être amélioré pour être plus agréable à utiliser mais aussi moins perméable aux potentiels erreurs. On aurait aussi pu rendre le code plus propre et mieux organisé pour faciliter les différentes mises à jour ou ajouts de fonctionnalités. Cependant, le fait d'être capable de penser à ce genre de détails durant le développement aurait nécessité plus d'expérience de notre part.

5 Conclusion

Durant les 5 mois que nous avons passé sur ce projet, nous avons découvert les bases du développement web, autant sur la partie client (avec HTML, CSS et JavaScript) que sur la partie serveur (PHP et python). De plus, cela nous a permis d'approfondir des notions que nous avions déjà abordé (comme SQL par exemple) et de les mettre en pratique pour construire quelque chose de concret. Nous avons aussi appris différents concepts comme le fonctionnement des requêtes HTTP et nous avons vu comment combiner plusieurs langages pour créer une application. Au final, le site n'est composé que d'une très petite quantité de ligne de codes, cependant, la majeur partie du temps passé sur le projet n'a pas directement concerné sa création mais plus la découverte de nouveaux outils. Par exemple, nous avons mis beaucoup de temps à comprendre comment faire fonctionner `fetch` et `PHP` pour pouvoir échanger des données entre le client et le serveur. Nous avons aussi rencontré plusieurs problèmes au niveau du matériel, en effet, au départ, nous souhaitions tester le site sur `perso.isima` mais nous avons rencontré divers problèmes qui nous on poussé à l'installation d'un serveur web local. Pour conclure, notre ressentie final sur ce projet et que, malgré tous ce qu'on à enduré pour réussir, nous sommes heureux du résultat et du chemin parcouru, qui était très enrichissant et nous à permit de nous améliorer dans de nombreux domaines.

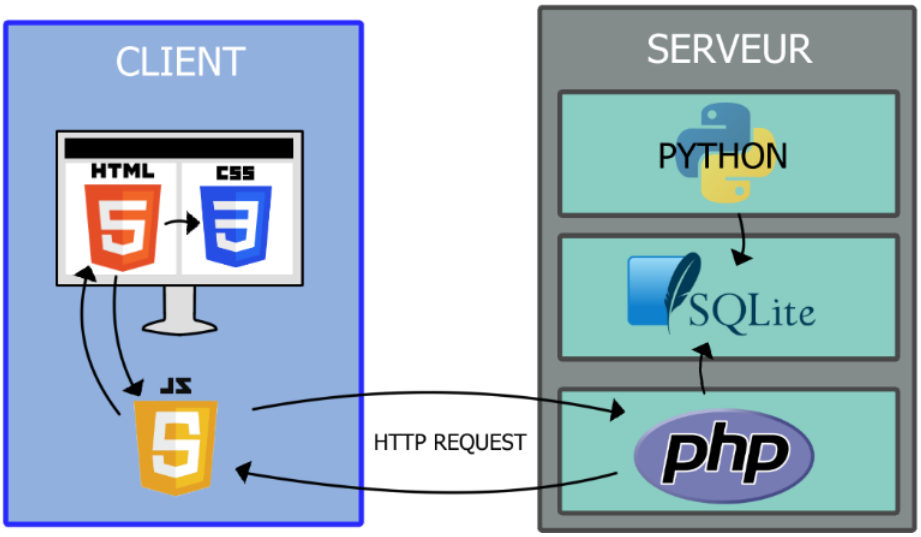

Figure 8: Résumé du fonctionnement global du site