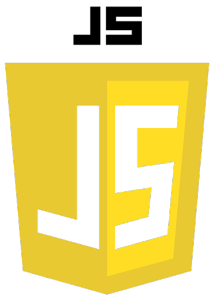
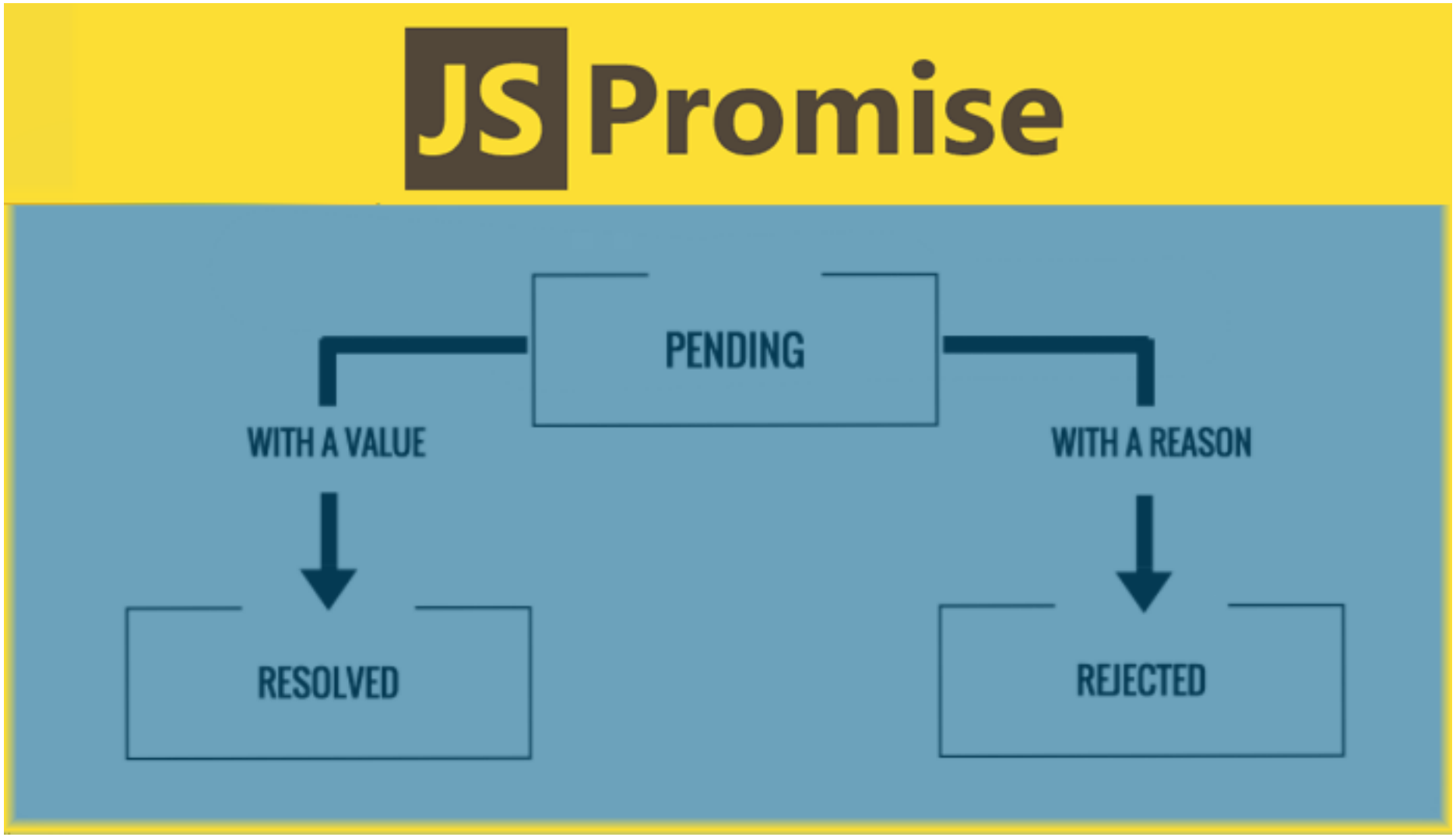


LES ‘’PROMISES”

En JavaScript, une Promise est un objet représentant la fin d'une opération asynchrone. Une Promise peut être dans l'un des états suivants :

- pending (en attente) : l'opération asynchrone n'est pas encore terminée.
- fulfilled (résolue) : l'opération asynchrone s'est terminée avec succès et la Promise est considérée comme tenue.
- rejected (rejetée) : l'opération asynchrone a échoué et la Promise est considérée comme non tenue.

Les Promises sont souvent utilisées pour gérer les opérations asynchrones telles que les requêtes réseau ou les opérations de lecture/écriture de fichier.


JS Promise
PENDING
WITH A VALUE
WITH A REASON
RESOLVED
REJECTED

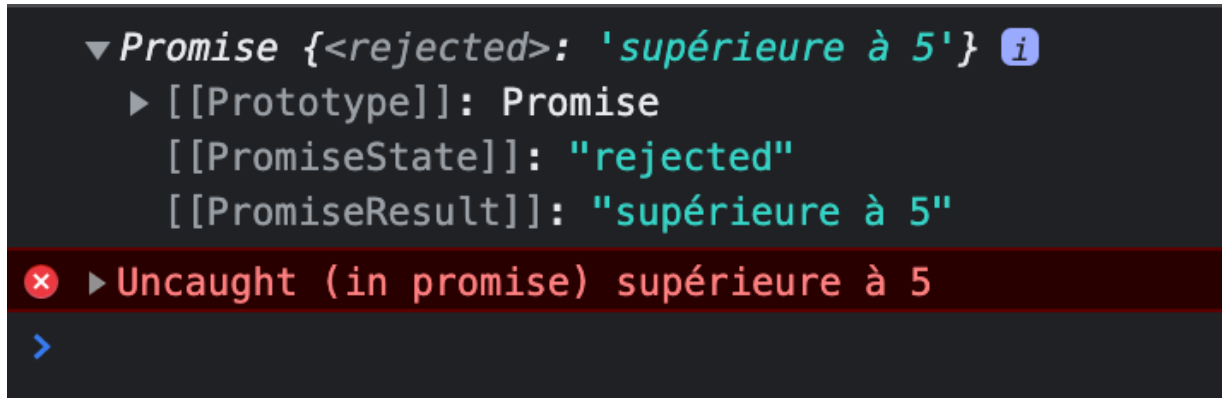
PROMESSE « RÉSOLUE» OU « REJETÉE »

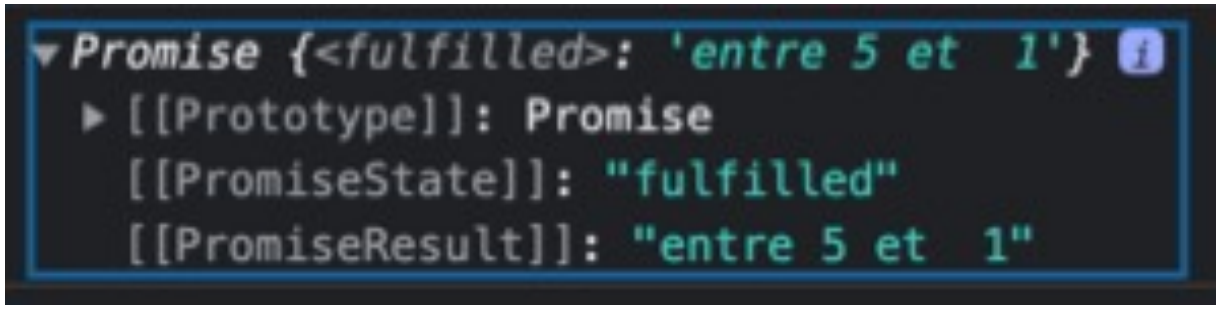
```
const promesse = new Promise((resolve,reject)=>{
        const aleatoire = Math.trunc((Math.random()*10)+1);
    if(aleatoire<=5)resolve("entre 5 et  1")
    else reject("supérieure à 5")
} )

console.log(promesse);
```

Cas fulfilled (résolu) :

ou

Cas rejected (rejeté) :



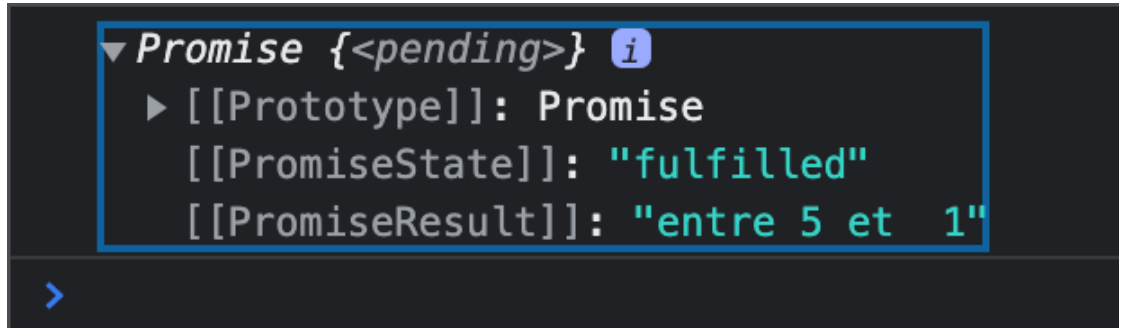


MISE EN EVIDENCE DE L'ETAT « PENDING »

On peut voir la promesse dans l'état « pending » pendant 3 seconde avant de passer dans l'état fulfilled ou rejected

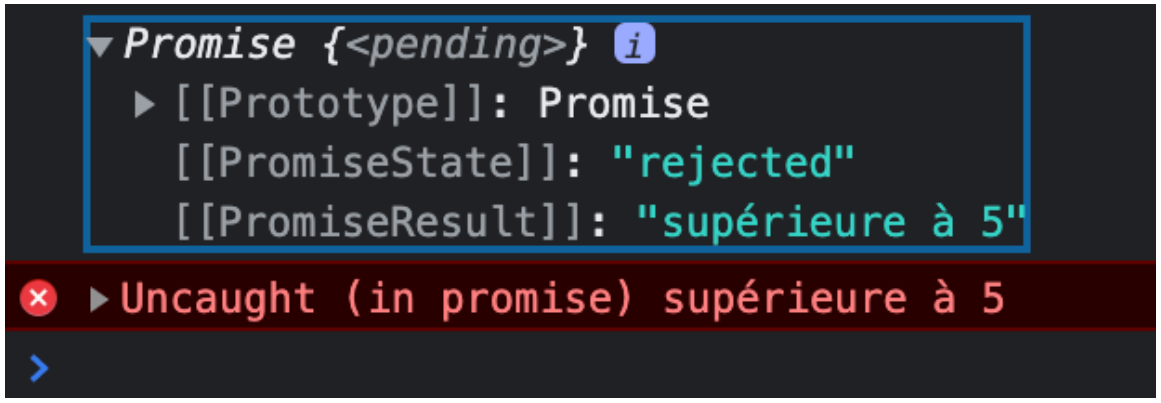
```
const promesse = new Promise((resolve,reject)=>{
    setTimeout(() => {
        const aleatoire = Math.trunc((Math.random()*10)+1);
    if(aleatoire<=5)resolve("entre 5 et  1")
    else reject("supérieure  5")
    } ,3000
    )

})
console.log( romesse);
```

Cas fulfilled :



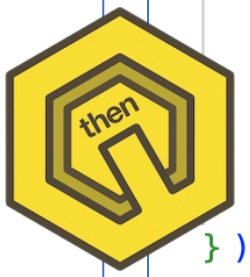
ou

Cas rejected :



La promesse représente une valeur qui peut être disponible plus tard, de manière asynchrone, et peut être gérée en utilisant les mécanismes de promesse tels que "then" et "catch"

```
const promesse = new Promise((resolve,reject)=>{
    setTimeout(() => {
        const aleatoire = Math.trunc((Math.random()*10)+1);
    if(aleatoire<=5)resolve("entre 5 et  1")
    else reject("supérieure à 5")
    } ,3000
    )

} )
.then((text)=>{console.log(text);} )
.catch((text)=>{console.log(text);} )

console.log("Hello world")
```




```
const promesse = new Promise((resolve,reject)=>{
    setTimeout(() => {
        const aleatoire = Math.trunc((Math.random()*10)+1);
    if(aleatoire<=5)resolve("entre 5 et  1")
    else reject("supérieure à 5")
    } ,3000
    )

} )
.then((text)=>{console.log(text);} )
.catch((text)=>{console.log(text);} )
console.log(promesse);
console.log("Hello world")
```

```
Hello world
supérieure à 5
>
```

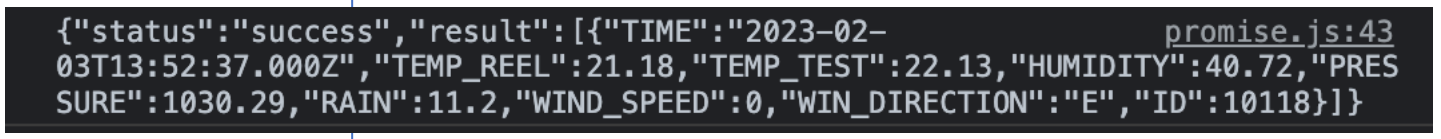
```
▶ Promise {<pending>}
Hello world
entre 5 et  1
>
```

PROMESSE SUR REQUÊTE HTTP :

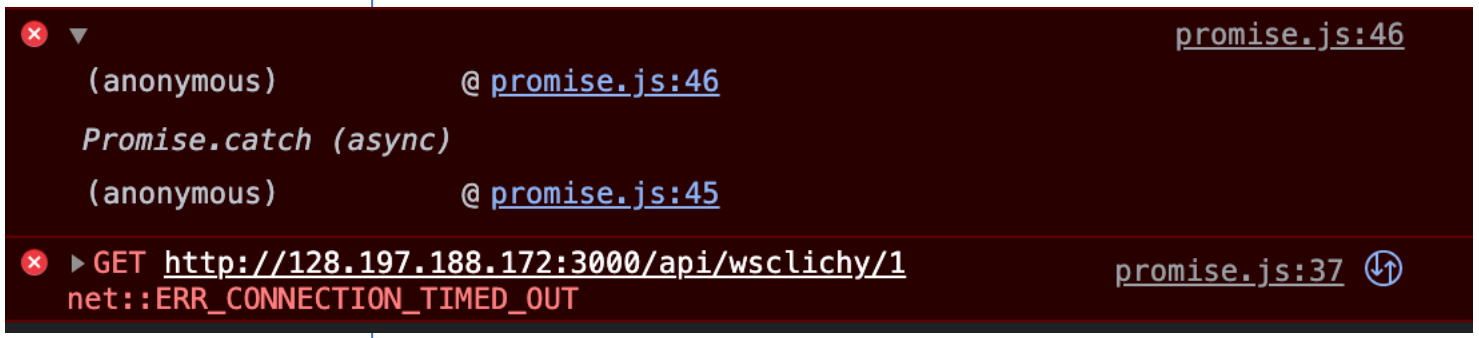
```
function makeRequest(url) {
    return new Promise(function(resolve, reject) {
      var xhr = new XMLHttpRequest();
      xhr.open("GET", url, true);
      xhr.onload = function() {
        if (xhr.status === 200) {
          resolve(xhr.responseText);
        } else {
          reject(xhr.statusText);
        }
      };
      xhr.onerror = function() {
        reject(xhr.statusText);
      };
      xhr.send();
    });
  }

  makeRequest("http://138.197.188.172:3000/api/wsclichy/1")
    .then(function(response) {
      console.log(response);
    })
    .catch(function(error) {
      console.error(error);
    });
```

xhr.open : méthode de l'objet XMLHttpRequest qui prend 3 argument

RESOLVE :



REJECT :



xhr.onload : est un écouteur d'événement de l'objet XMLHttpRequest qui se déclenche lorsque la requête HTTP a été complétée avec succès.

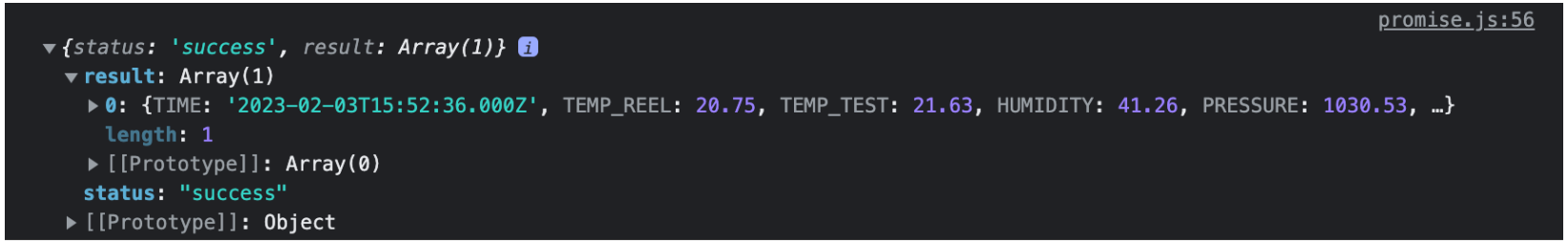
REQUÊTE HTTP EN UTILISANT FETCH:

```
fetch('http://138.197.188.172:3000/api/wsclichy/1')
.then(response => {
  if (!response.ok) {
    throw new Error(response.statusText);
  }
  return response.json();
} )
.then(data => console.log(data))
.catch(error => console.error(error));
```

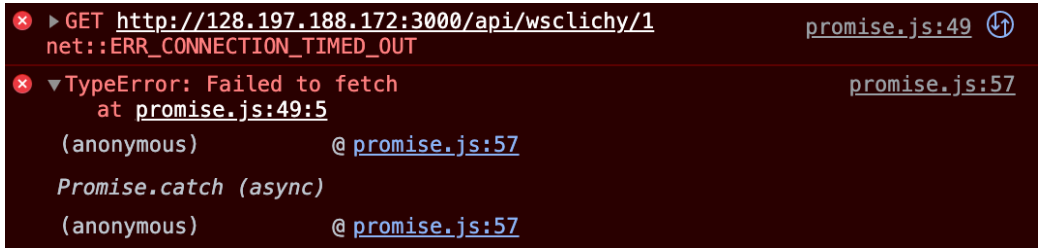
.json() parse le json (convertit les objets json en objet javascript)

La propriété ok de l'objet de réponse renvoyé par la méthode fetch indique si la requête HTTP a réussi ou non.

RESOLVE :



REJECT :



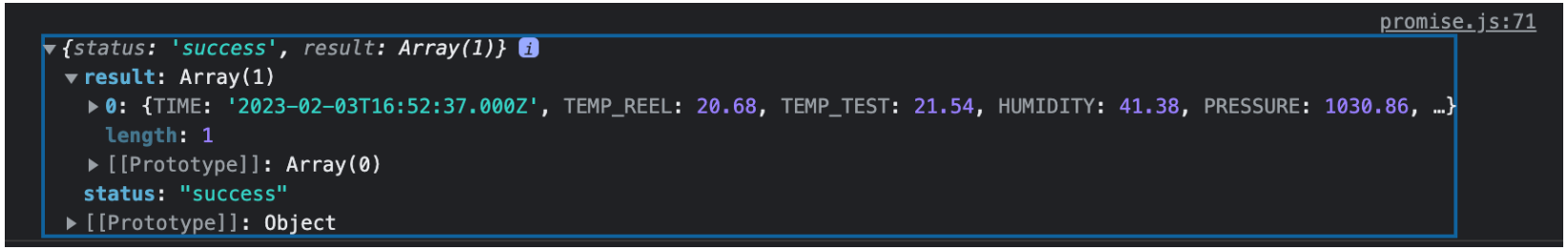
REQUÊTE HTTP EN UTILISANT AXIOS:

```
axios.get('http://138.197.188.172:3000/api/wsclichy/1')
  .then(response => {
    console.log(response.data);
  } )
  .catch(error => {
    console.error(error);
  } );
```

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Plus simple encore que fetch, le parsing du json est intégré

RESOLVE :



REJECT :

ENCHAINEMENT DES PROMESSES:

```
fetch('http://138.197.188.172:3000/api/wsclichy/1')
.then(response => {
  if (!response.ok) {
    throw new Error(response.statusText);
  }
  return response;
} )
.then(res => res.json())
.then(data => data.result[0])
.then(time => time.TIME)
.then(datafinal => console.log(datafinal))
.catch(error => console.error(error));
```

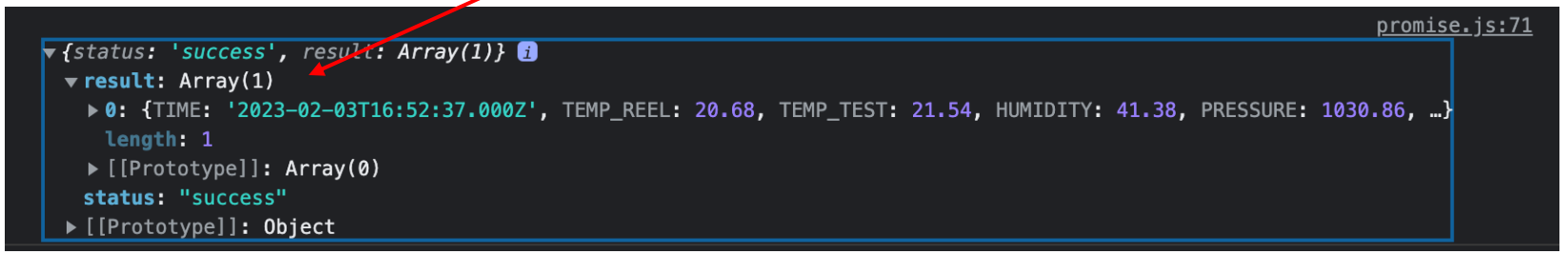
On renvoit une promesse avec l'objet javascript

On renvoit une promesse avec l'objet 0 du tableau

On renvoit une promesse avec la clé TIME

On renvoit une promesse qui affiche le temps TIME

```
2023-02-03T18:12:37.000Z
```



ENCHAINEMENT DES PROMESSES AVEC ASYNC/AWAIT:

async et await sont des fonctionnalités de JavaScript qui permettent de gérer les opérations asynchrones de manière plus simple et plus lisible.

async est utilisé pour déclarer une fonction asynchrone. Une fonction asynchrone peut être utilisée pour effectuer des opérations asynchrones, telles que les requêtes réseau, sans bloquer le reste du code.

await est utilisé pour attendre la résolution d'une promesse. Lorsqu'une expression est précédée par await, l'exécution du code s'arrête jusqu'à ce que la promesse soit résolue.

```
async function getData() {
    try {
      const response = await fetch('http://138.197.188.172:3000/api/wsclichy/1');
      const data = await(response.json());
      const tab = await(data.result[0]);
      const time = await(tab.TIME);

      console.log(time);
    }  catch (error) {
      console.error(error);
    }
  }

  getData();
```

```
2023-02-03T19:12:37.000Z
```