Programmation Web – Avancé

JavaScript & Node.js

Partie 23 : Promesses

Version 2020

Introduction aux promesses

Comment réaliser une partie de code asynchrone à notre époque ?

Programmation asynchrone : les callbacks

- Fonction asynchrone avec une callback en argument
- Callback exécutée quand fonction aynchrone terminée
- Plusieurs opérations asynchrones en série : “classic callback **pyramid of doom**” [79.]

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

Programmation asynchrone moderne : les promesses

- Promise = Objects
  - représentant un état intermédiaire d'une opération
  - code fournit s'exécute quand l'opération est finie ou que la promesse rate
- Fonctions modernes attachant les callbacks aux promesses retournées : **.then()**, **.catch()**, **.finally()**

Programmation asynchrone moderne : les promesses

- Utilisation de promesses [79.]

```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log("Got the final result: " + finalResult);
  })
  .catch(failureCallback);
```

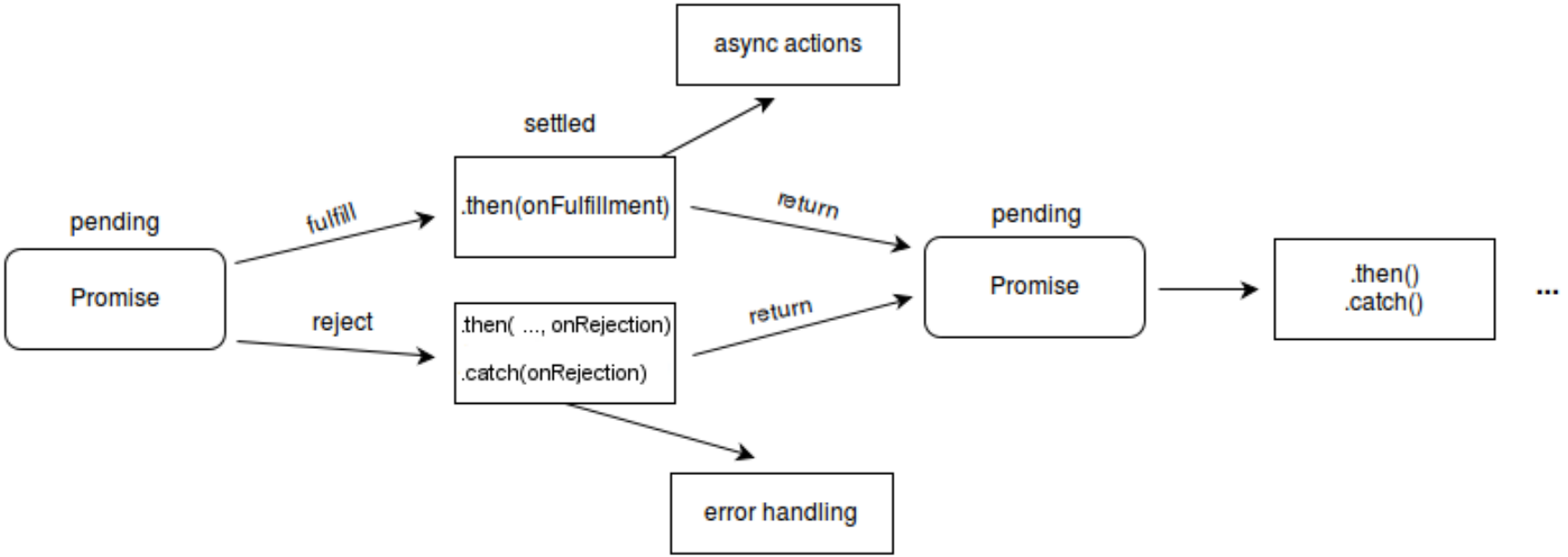
Programmation asynchrone moderne : les promesses

- Utilisation de promesses [79.]

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

  - Chaînage de promesses (actions en série) via le gestionnaire **.then** : seulement si return dans **callback**

Les états d'une promesse : pending, fulfilled, rejected



Promise [80.]

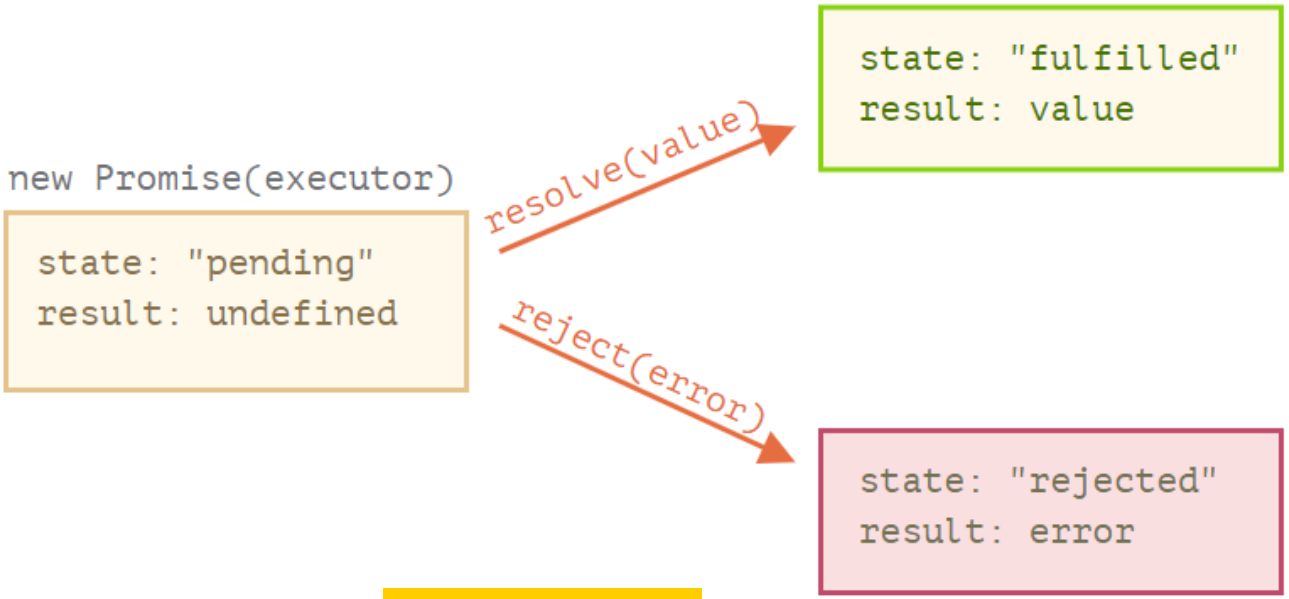
Chaînage des promesses

- Actions en série via la méthode **.then(gestionnaire)** [81.]
- Appel du **gestionnaire** si la promesse est résolué avec la valeur retournée en argument

Chaînage des promesses

- Si le gestionnaire retourne [81.] :
  - **Une valeur** : la promesse retournée par **then** est résolue avec la valeur
  - **Pas de valeur** : la promesse retournée par **then** est résolue avec **undefined**
  - **Une autre promesse “pending”** : la promesse retournée par **then** est résolue/rejetée à la suite de la résolution/rejet de la promesse retournée par le **gestionnaire**
  - ...

Création d'une promesse



Les bases des Promises [83.]

Création d'une promesse

- Création d'une promesse [84.] :
  new Promise(executor)
- **executor** : fonction définissant une tâche asynchrone et ayant généralement 2 arguments :
  - **resolve** : fonction appelée quand la tâche asynchrone est terminée avec succès et retourne le résultat en tant que valeur
  - **reject** : fonction appelée quand la tâche échoue et retourne l'erreur

Création d'une promesse

```
const asyncLogin = (user) => {
  return new Promise((resolve, reject) => {
      fetch("/api/users/login", {// method & headers to be added
      body: JSON.stringify(user), // body data type must match "Content-Type" header
    })
      .then((response) => {
        if (!response.ok)
          throw new Error("Error code : "+response.status+" : "+response.statusText);
        return response.json();
      })
      .then((data) => {
          console.log("asyncLogin:end of fetch()", "user:", user);
          resolve(data);
      })
      .catch((err) => reject(err));
  });
};
```

Utilisation de la promesse

```
console.log("onLogin:before async call");
asyncLogin(user).then(onUserLogin).catch(onError);
// re-render the navbar for the authenticated user and redirect to the user list
console.log("onLogin:end");
```

```
onLogin:before async call                    LoginPage.js:35
onLogin:end                                  LoginPage.js:38
asyncLogin:end of fetch() user:              LoginPage.js:79
▶ {email: "teacher@vinci.be", password: "Teacher"}
```

async / await

- Simplification de la syntaxe des promesses
- **async** et **await** [85.] :
  - Fonction « taggée » par **async** renvoie automatiquement une promesse
  - Utilisation de **await** seulement au sein d'une fonction **async**
  - **await** : attente de la résolution de la fonction pour chaîner des tâches asynchrone
  - Bloc **try** / **catch** pour gérer les erreur

Création d'une promesse

```
const asyncLogin2 = async (user) => {
  try {
    let response = await fetch("/api/users/login", {
      method: "POST", // *GET, POST, PUT, DELETE, etc.
      body: JSON.stringify(user), // body data type must match "Content-Type" header
      headers: {
        "Content-Type": "application/json",
      },
    });
    if (!response.ok)
      throw new Error("Error code : "+response.status+" : "+response.statusText);
    let data = await response.json();
    console.log("asyncLogin2:end of fetch()", "user:", user);
    return data;
  } catch (err) { return err;}
};
```

Utilisation de la promesse

```
console.log("onLogin:before async call");
asyncLogin2(user).then(onUserLogin).catch(onError);
// re-render the navbar for the authenticated user and redirect to the user list
console.log("onLogin:end");
```

```
onLogin:before async call                    LoginPage.js:35
onLogin:end                                  LoginPage.js:38
asyncLogin2:end of fetch() user:             LoginPage.js:102
▶ {email: "teacher@vinci.be", password: "Teacher"}
```

Les promesses

- **DEMO : Création et utilisation d'une promesse**
  - Version classique :
    Promesse faisant l'appel au login de l'API et renvoyant l'utilisateur en cas de résolution : affichage de message dans la console pour tracer les opérations asynchrones
  - Version **async**/**await**