

# Programmation Web – Avancé

JavaScript & Node.js

Les promesses



Attribution –  
Partage dans les  
Mêmes Conditions  
4.0 International  
(CC BY-SA 4.0)

*Presentation template  
by [SlidesCarnival](#)*



# 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 asynchrone 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

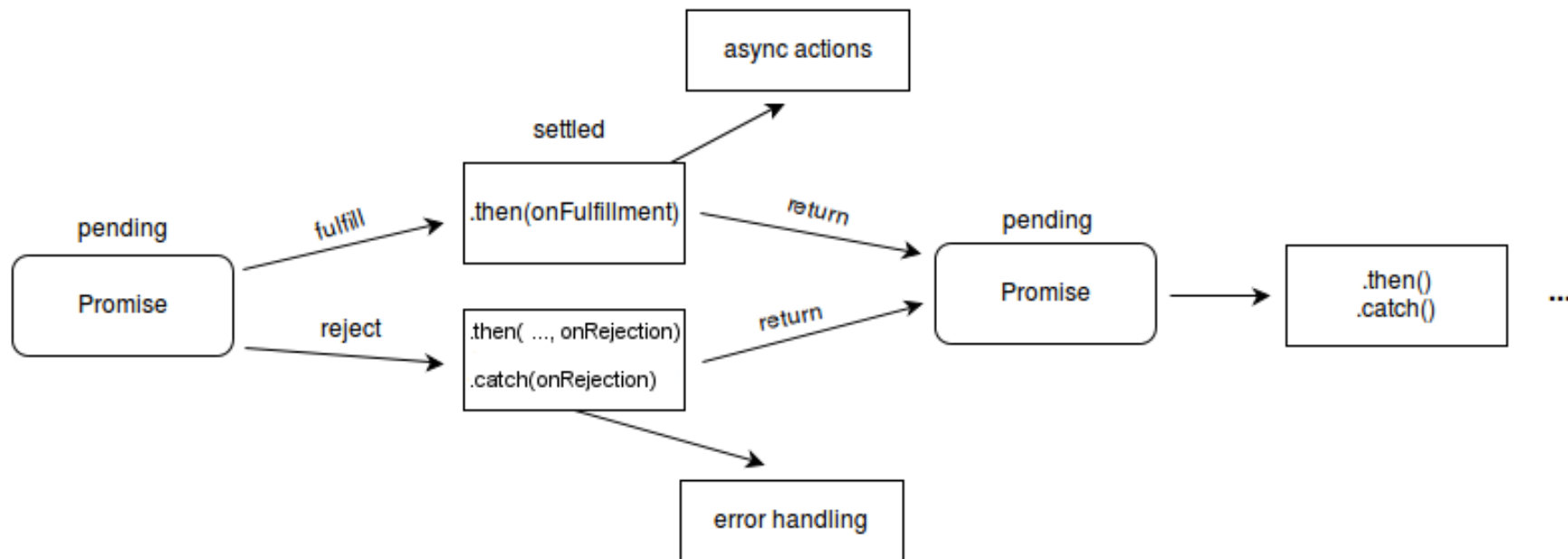
### ● Utilisation de promesses [\[79.\]](#)

```
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

```
new Promise(executor)
```

```
state: "pending"  
result: undefined
```

*resolve(value)*

*reject(error)*

```
state: "fulfilled"  
result: value
```

```
state: "rejected"  
result: error
```

**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	<u>LoginPage.js:35</u>
onLogin:end	<u>LoginPage.js:38</u>
asyncLogin:end of fetch() user:	<u>LoginPage.js:79</u>
▶ {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	<u>LoginPage.js:35</u>
onLogin:end	<u>LoginPage.js:38</u>
asyncLogin2:end of fetch() user:	<u>LoginPage.js:102</u>
▶ {email: "teacher@vinci.be", password: "Teacher"}	







## Les promesses

- ◎ DEMO : Création d'une RESTfull API pour une pizzeria : Part 6 – Sécurisation : hachage des passwords et protection contre attaques XSS
  - Version **async/await** de création d'une promesse : **/model/users.js** pour **login** asynchrone (comparaison de pwd async)
  - Est-ce un souci que **register** n'est pas asynchrone alors que **addOne** l'est ?