



INTRODUCTION AUX API



m2iformation.fr



DÉFINITION D'UNE API

API signifie "**Application Programming Interface**". C'est un ensemble de règles et de protocoles qui permettent à différentes applications de communiquer entre elles.



m2iformation.fr



UTILITÉ DES API DANS LE DÉVELOPPEMENT WEB

Les **API** sont essentielles dans le développement web, car elles permettent d'échanger des **données** entre différentes parties d'une application ou entre différentes applications. Grâce aux API, les développeurs peuvent intégrer facilement des services et des fonctionnalités externes à leurs applications.



PRINCIPAUX TYPES D'API

- **REST (REpresentational State Transfer)** : Architectures d'API basées sur des principes de conception spécifiques pour créer des services web légers et faciles à maintenir.
- **GraphQL** : Langage de requêtes open-source et runtime créé par **Facebook** pour faciliter les communications avec les API et la manipulation des données.



LIBRAIRIES COMMUNES POUR INTERAGIR AVEC DES API

- **Axios** : Utilisation promesses pour gérer les requêtes HTTP
- **Fetch API** : API standard intégrée au navigateur pour effectuer des requêtes HTTP
- **jQuery AJAX** : Méthodes AJAX fournies par la librairie jQuery (à éviter pour les projets modernes)
- **SuperAgent** : Autre librairie basée sur les promesses pour effectuer des requêtes HTTP



LIBRAIRIES COMMUNES POUR INTERAGIR AVEC DES API



FETCH API



m2iformation.fr



PRÉSENTATION ET COMPATIBILITÉ

Fetch API est une interface web moderne pour effectuer des **requêtes HTTP**. C'est une alternative à l'objet **XMLHttpRequest**. Il est disponible dans la plupart des navigateurs modernes, y compris **Node.js**.



SYNTAXE

```
fetch(url, {
  method: "GET", // ou "POST", "PUT", "DELETE", etc.
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ /* données à envoyer */ })
})
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```



EXEMPLES D'UTILISATION

```
fetch("https://api.example.com/users")
  .then((response) => response.json())
  .then((users) => console.log(users))
  .catch((error) => console.error(error));
```



AXIOS



m2iformation.fr



PRÉSENTATION ET INSTALLATION

Axios est une bibliothèque qui facilite les **requêtes HTTP** en utilisant des **Promesses**. Elle fonctionne à la fois avec les navigateurs et **Node.js**. Pour l'installer, utilisez la commande suivante :

```
npm install axios
```



m2iformation.fr



SYNTAXE

```
import axios from "axios";

axios({
  method: "get", // ou "post", "put", "delete", etc.
  url: "https://api.example.com/users",
  headers: {
    "Content-Type": "application/json",
  },
  data: { /* données à envoyer */ },
})
  .then((response) => console.log(response.data))
  .catch((error) => console.error(error));
```





EXEMPLES D'UTILISATION

```
import axios from "axios";

axios.get("https://api.example.com/users")
  .then((response) => console.log(response.data))
  .catch((error) => console.error(error));
```



MÉTHODES HTTP



m2iformation.fr



GET

La méthode **GET** est utilisée pour **récupérer** des informations à partir d'une ressource spécifiée.

```
fetch("https://api.example.com/data", {  
  method: "GET",  
})  
.then(response => response.json())  
.then(data => console.log(data));
```



SYNTAXE

```
fetch("https://example.com/api/resource")
  .then((response) => response.json())
  .then((data) => console.log(data));
```

Note : Cette slide montre un exemple simple de chaînage de promesses pour traiter une requête HTTP asynchrone en TypeScript. Les étudiants doivent comprendre la nécessité de gérer les erreurs dans un scénario réel.



EXEMPLES D'UTILISATION EN TYPESCRIPT

```
async function getData(): Promise<void> {
  const response = await fetch("https://example.com/api/resource");
  const data = await response.json();
  console.log(data);
}
getData();
```





POST

La méthode **POST** est utilisée pour soumettre des données à une ressource spécifiée.

- Utilisation fréquente pour soumettre des formulaires ou créer de nouvelles ressources
- Les données envoyées sont cachées dans le corps de la requête
- POST n'est pas idempotent, envoyer la même requête plusieurs fois peut avoir des effets différents

```
fetch('https://api.example.com/data', {
  method: 'POST',
  body: JSON.stringify({ name: 'John Doe', age: 30 }),
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```





SYNTAXE

```
fetch("https://example.com/api/resource", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ key1: 'value1', key2: 'value2' })
})
.then((response) => response.json())
.then((data) => console.log(data));
```





EXEMPLES D'UTILISATION EN TYPESCRIPT

```
async function postData(): Promise<void> {
  const response = await fetch("https://example.com/api/resource", {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ key1: 'value1', key2: 'value2' })
  });
  const data = await response.json();
  console.log(data);
}
postData();
```



PUT

La méthode **PUT** est utilisée pour mettre à jour des données sur une **ressource spécifiée**.



m2iformation.fr



SYNTAXE

```
fetch("https://example.com/api/resource/:id", {
  method: "PUT",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ key1: 'new_value1', key2: 'new_value2' })
})
.then((response) => response.json())
.then((data) => console.log(data));
```





EXEMPLES D'UTILISATION EN TYPESCRIPT

```
async function updateData(id: string): Promise<void> {
  const response = await fetch(`https://example.com/api/resource/${id}`, {
    method: "PUT",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ key1: 'new_value1', key2: 'new_value2' })
  });
  const data = await response.json();
  console.log(data);
}
updateData("1");
```





DELETE

La méthode **DELETE** est utilisée pour supprimer des données sur une **ressource spécifiée**.

```
fetch('https://api.example.com/resource/1', {
  method: 'DELETE',
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```





SYNTAXE

```
fetch("https://example.com/api/resource/:id", {  
  method: "DELETE"  
})  
.then((response) => response.json())  
.then((data) => console.log(data));
```



m2iformation.fr



EXEMPLES D'UTILISATION EN TYPESCRIPT

```
async function deleteData(id: string): Promise<void> {
  const response = await fetch(`https://example.com/api/resource/${id}`, {
    method: "DELETE"
  });
  const data = await response.json();
  console.log(data);
}
deleteData("1");
```



GESTION DES ERREURS ET DES CODES DE STATUT



CODES DE STATUT HTTP

Les codes de statut **HTTP** indiquent la réussite, l'échec ou la progression d'une requête à une **API**.

Classe de statut	Description
1xx	Information
2xx	Réussite (ex: 200 OK)
3xx	Redirection (ex: 301 Moved)
4xx	Erreur côté client (ex: 404 Not Found)
5xx	Erreur côté serveur (ex: 500 Internal Server Error)



SIGNIFICATION DES CODES HTTP (2XX, 3XX, 4XX, 5XX)

- **2xx** : Succès (ex: 200 OK)
- **3xx** : Redirection (ex: 301 Moved Permanently)
- **4xx** : Erreur côté client (ex: 404 Not Found)
- **5xx** : Erreur côté serveur (ex: 500 Internal Server Error)



EXEMPLES COURANTS (200, 201, 404, 500)

Code	Nom	Description
200	OK	Requête réussie
201	Created	Ressource créée avec succès
404	Not Found	Ressource demandée introuvable
500	Internal Server Error	Erreur interne lors du traitement



GESTION DES ERREURS EN TYPESCRIPT



m2iformation.fr



CATCHERROR

La fonction `catchError` est utilisée pour **traiter** les **erreurs** lors d'une requête.

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

const handleError = (error: any): any => {
  console.error(error);
  return throwError(error);
};

source$  

  .pipe(catchError(handleError))
  .subscribe(
    response => console.log(response),
    error => console.log("Une erreur s'est produite:", error)
  );
```



SYNTAXE

```
import { catchError } from 'rxjs/operators';

observable.pipe(catchError(error => /* gestion de l'erreur *())));

```

Note : Le "catchError" est un opérateur RxJS qui permet de gérer les erreurs de manière élégante. Il attrape l'erreur et retourne un nouvel Observable pour continuer le flux.



EXEMPLES D'UTILISATION

```
import { catchError } from 'rxjs/operators';

fetchData() .pipe(
  catchError(error => {
    console.error('Erreur lors de la récupération des données:', error);
    return throwError(`Erreur: ${error.message}`);
  })
);
```



OPÉRATEURS TRY / CATCH

Les blocs `try` et `catch` permettent de gérer les **erreurs** de manière plus générale.

```
try {
    // Code à essayer
} catch (erreur) {
    // Code à exécuter en cas d'erreur
}
```



SYNTAXE

```
try {
    // Tentative d'exécution d'un code
} catch (error) {
    // Traitement de l'erreur
}
```



EXEMPLES D'UTILISATION

```
async function fetchData() {  
  try {  
    const response = await fetch('https://api.example.com/data');  
    const data = await response.json();  
    console.log('Données récupérées:', data);  
  } catch (error) {  
    console.error('Erreur lors de la récupération des données:', error);  
  }  
  
  fetchData();  
}
```



AUTHENTIFICATION ET AUTORISATION



m2iformation.fr



JWT (JSON WEB TOKENS)

Les **JSON Web Tokens (JWT)** sont un **standard ouvert** (RFC 7519) utilisé pour représenter des **revendications** entre les parties sous forme d'un **objet JSON**, utile pour l'**authentification** et l'**autorisation** de l'utilisateur.



PRÉSENTATION ET UTILITÉ

- Permet de partager des informations **sécurisées** entre les parties
- **Compact** et **portable**
- **Auto-déscriptif**



UTILISATION AVEC TYPESCRIPT

1. Installer la bibliothèque `**jsonwebtoken**` :

```
npm install jsonwebtoken
```

2. Importer la bibliothèque :

```
import jwt from 'jsonwebtoken';
```

3. Utilisation pour **générer** et **vérifier** les tokens.



OAUTH

OAuth est un protocole d'autorisation qui permet aux applications d'accéder aux **comptes des utilisateurs** sans partager leur mot de passe.

- Authentification tierse
- Échange de **jetons** pour accéder aux **ressources**



PRÉSENTATION ET UTILITÉ

- Utilisé par de grandes entreprises (Ex: **Google, Facebook, Twitter**)
- Permet l'**authentification** et l'**autorisation** des utilisateurs
- Protège les **informations sensibles** des utilisateurs



UTILISATION AVEC TYPESCRIPT

1. Choisir et installer une bibliothèque **OAuth** appropriée (Ex: passport-oauth2).
2. Configuration et utilisation pour l'**authentification** et l'**autorisation** de l'utilisateur.



UTILISATION DE TYPESCRIPT AVEC DES API



INTERFACES ET TYPES POUR LES DONNÉES DE L'API

Les **interfaces** et les **types** sont utilisés pour décrire la forme et la structure des données reçues ou envoyées par une API.

```
interface Utilisateur {
  id: number;
  nom: string;
  email?: string; // Propriété optionnelle
}

type Statut = "actif" | "inactif" | "suspendu";

function afficherUtilisateur(utilisateur: Utilisateur, statut: Statut) {
  console.log(`L'utilisateur ${utilisateur.nom} est ${statut}.`);
}
```



CRÉATION D'INTERFACES

```
interface User {  
    id: number;  
    name: string;  
    email: string;  
}
```

Propriété	Type	Description
id	number	Identifiant unique de l'utilisateur
name	string	Nom complet de l'utilisateur
email	string	Adresse e-mail de l'utilisateur





EXEMPLES D'UTILISATION AVEC DES DONNÉES D'API

```
function formatUser(user: User): string {
  return `${user.name} (${user.email})`;
}

fetch('https://api.example.com/users/1')
  .then(response => response.json())
  .then((user: User) => console.log(formatUser(user)));
```



PROMESSES ET ASYNC / AWAIT

Les **promesses** et **Async / Await** sont utilisés pour gérer les opérations asynchrones en TypeScript, comme les **appels API**.

```
function getApiData(): Promise<any> {
  return new Promise((resolve, reject) => {
    // Simule un appel API
    setTimeout(() => {
      resolve({ data: "Data from API" });
    }, 1000);
  });
}

async function main() {
  try {
    const data = await getApiData();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
```





SYNTAXE ET UTILISATION

- Promise: gestion d'opérations **asynchrones** avec .then () et .catch ()
- async / await: syntaxe simplifiée pour gérer les **promesses**

```
// Exemple avec Promise
const promesse = new Promise((resolve, reject) => {
    // Code asynchrone
    if /* condition */ {
        resolve("Résultat");
    } else {
        reject("Erreur");
    }
});

promesse.then(result => {
    console.log("Succès : ", result);
}).catch(error => {
    console.log("Erreur : ", error);
});
```



EXEMPLES D'UTILISATION AVEC DES APPELS D'API

```
// Utilisation de Promesses
fetch('https://api.example.com/users/1')
  .then(response => response.json())
  .then((user: User) => console.log(formatUser(user)))
  .catch(error => console.error(error));

// Utilisation de Async / Await
async function getUser(userId: number): Promise<User> {
  const response = await fetch(`https://api.example.com/users/${userId}`);
  const user: User = await response.json();
  return user;
}

getUser(1).then(user => console.log(formatUser(user)));
```



UTILISATION DE TYPESCRIPT AVEC DES API



INTERFACES ET TYPES POUR LES DONNÉES DE L'API

Les **interfaces** et les **types** permettent de définir la structure des données reçues d'une **API**, offrant ainsi un meilleur typage et des erreurs de compilation en cas de mauvaise utilisation.

```
interface Person {
  name: string;
  age: number;
}

function displayPerson(person: Person) {
  console.log(`Name: ${person.name}, Age: ${person.age}`);
}
```



CRÉATION D'INTERFACES

```
interface User {  
    id: number;  
    name: string;  
    email: string;  
}  
  
interface Post {  
    id: number;  
    userId: number;  
    title: string;  
    body: string;  
}
```



EXEMPLES D'UTILISATION AVEC DES DONNÉES D'API

```
// Avec Fetch API
fetch("https://jsonplaceholder.typicode.com/users/1")
  .then((response) => response.json())
  .then((user: User) => console.log(user.name));

// Avec Axios
import axios from "axios";

axios.get<User>("https://jsonplaceholder.typicode.com/users/1")
  .then((response) => console.log(response.data.name));
```



PROMESSES ET ASYNC / AWAIT

Les **promesses** et **Async / Await** facilitent la gestion des opérations **asynchrones**, telles que les appels d'API, en fournissant une syntaxe plus claire et plus lisible.

```
function fetchApiData(): Promise<data> {
  return new Promise((resolve, reject) => {
    // Code pour récupérer les données de l'API
    // Si succès, resolve(data)
    // Si erreur, reject(error)
  });
}

async function processData() {
  try {
    const data = await fetchApiData();
    console.log("Data reçue:", data);
  } catch (error) {
    console.error("Erreur lors de la récupération des données:", error);
  }
}
```





SYNTAXE ET UTILISATION

```
// Promesses
myApiCall()
  .then((data) => handleData(data))
  .catch((error) => handleError(error));

// Async / Await
async function fetchData() {
  try {
    const data = await myApiCall();
    handleData(data);
  } catch (error) {
    handleError(error);
  }
}
```



EXEMPLES D'UTILISATION AVEC DES APPELS D'API

```
// Promesses avec Fetch API
fetch("https://jsonplaceholder.typicode.com/users")
  .then((response) => response.json())
  .then((users: User[]) => displayUsers(users))
  .catch((error) => console.error("Error fetching users:", error));

// Async / Await avec Axios
async function fetchUsersAsync() {
  try {
    const response = await axios.get<User[]>("https://jsonplaceholder.typicode.com/users");
    displayUsers(response.data);
  } catch (error) {
    console.error("Error fetching users:", error);
  }
}
```



m2iformation.fr