Systèmes à objets répartis

M1 Informatique, S8

Janvier 2026

Vincent Lannurien [1]
vincent.lannurien@univ-brest.fr

[1] Lab-STICC, CNRS UMR 6285, Université de Bretagne Occidentale, Brest

Introduction

http://info-demo-sor.univ-brest.fr:3000

Plan du cours

1. Organisation de l'UE
2. Cas d'étude applicatif
3. Architecture : modèle client/serveur
4. JavaScript, TypeScript et *runtime*
5. Serveur web
6. Client web
7. Authentification
8. Interactions
9. Déploiement de l'application
10. Débuggage et profilage
11. Auto-évaluation

Organisation de l’UE

Organisation de l'UE – Contenu

- Objectifs :
  - Développer une application suivant l'**architecture trois tiers**, s'appuyant sur des communications via **HTTP** et **WebSockets** ;
  - Comprendre les mécanismes de l'**authentification** (avec ou sans état) d'un client auprès d'un serveur ;
  - S'initier au **déploiement** d'une application répartie à l'aide d'un *reverse proxy*.
- Prérequis :
  - Côté client : HTML/CSS
- Méthode :
  - Un peu de cours…
  - Beaucoup de pratique !

Organisation de l'UE – Logistique

- Volume horaire :
  - CM : 3 séances de 2h (total : 6h)
  - TP : 8 séances de 2h (total : 16h)
- Contrôle des connaissances :
  - TP noté (2h sur cette partie du cours)
  - Examen final (2h en tout)

Organisation de l'UE – Technologies utilisées

- Côté serveur :
  - Runtime (Deno)
  - Langage (TypeScript)
  - Framework (Oak)
  - Base de données (SQLite)
- Côté client :
  - Serveur de développement (Vite)
  - Bundler (Rolldown)
  - Langage (TypeScript)
  - Framework (React + React Router)
- Communications :
  - Requêtes HTTP
  - WebSockets
  - JSON Web Tokens
- Déploiement :
  - Reverse proxy (nginx)

Cas d’étude applicatif

Présentation de l'application

Description

L'application est une **plateforme de sondages en ligne**.
Elle permet à des utilisateurs de **créer des sondages** et d'ajouter des **options de réponse**.
Les participants peuvent **voter** pour une ou plusieurs options selon des règles définies par le créateur du sondage.
L'application gère également l'**authentification** des utilisateurs et assure la **persistance** des données.

Présentation de l'application

Acteurs

Les acteurs de l'application sont les suivants :

- Utilisateur authentifié : peut créer des sondages, voter, et consulter les résultats ;
- Utilisateur invité : peut voter (si autorisé) et consulter les résultats (si autorisé) ;
- Administrateur : peut gérer les sondages et les utilisateurs.

Présentation de l'application

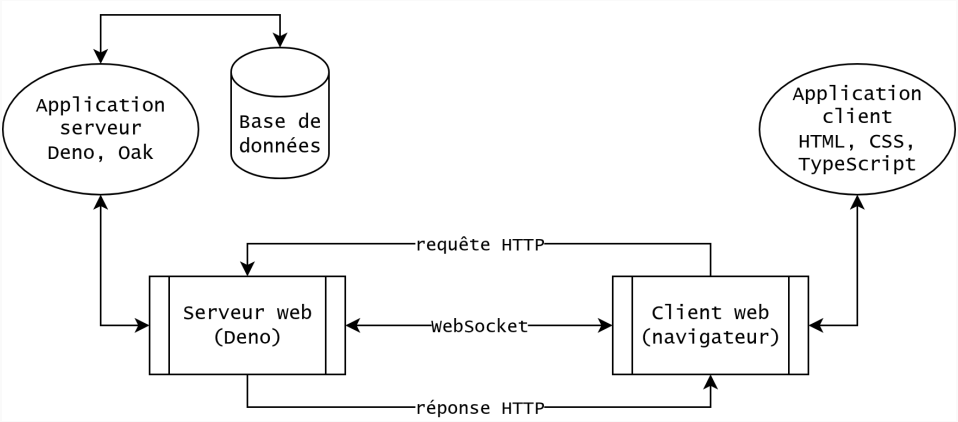
Fonctionnalités

Les principales fonctionnalités de l'application peuvent être résumées ainsi :

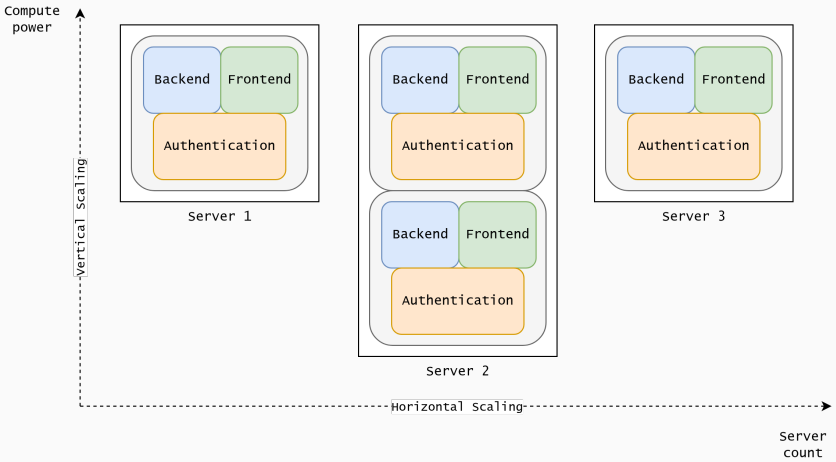
- Création de sondages avec : titre, description, date de création, date d'expiration, statut (actif/inactif) ;
- Ajout d'options à un sondage : texte descriptif ;
- Vote pour une option de sondage ;
- Consultation des résultats (nombre de votes par option) ;
- Gestion des utilisateurs (inscription, authentification).

Architecture : modèle client/serveur

Vue d'ensemble

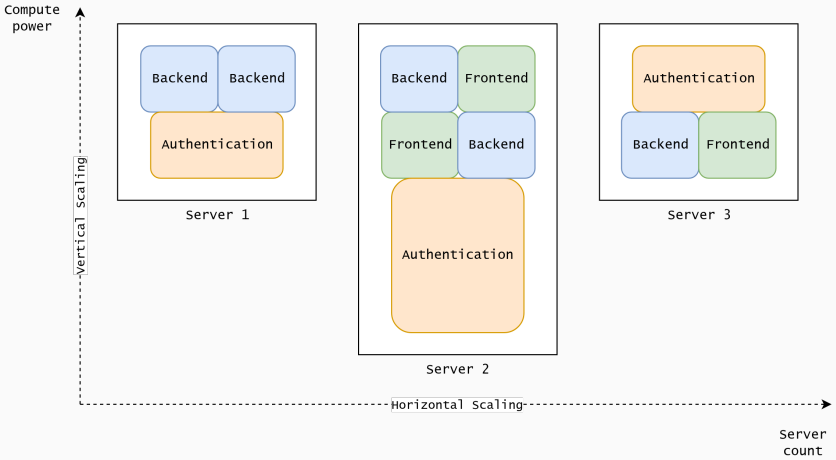


Monolithes ou microservices



1. [6] Vincent LANNURIEN et al. « Serverless Cloud Computing : State of the Art and Challenges ». *Serverless Computing : Principles and Paradigms*. 2023
2. [7] James LEWIS et Martin FOWLER. *Microservices*.

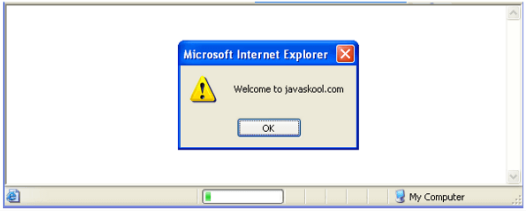
Monolithes ou microservices



1. [6] Vincent LANNURIEN et al. « Serverless Cloud Computing : State of the Art and Challenges ». *Serverless Computing : Principles and Paradigms*. 2023
2. [7] James LEWIS et Martin FOWLER. *Microservices*.

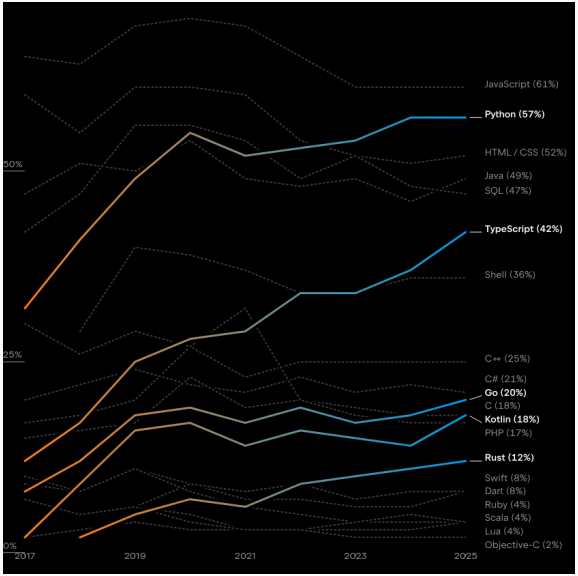
JavaScript, TypeScript et *runtime*

```
<html>
<head><title>Alert Page</title></head>
<Body>
    <script language="javascript">
        alert("Welcome to javaskool.com");
    </script>
</body>
<html>
```

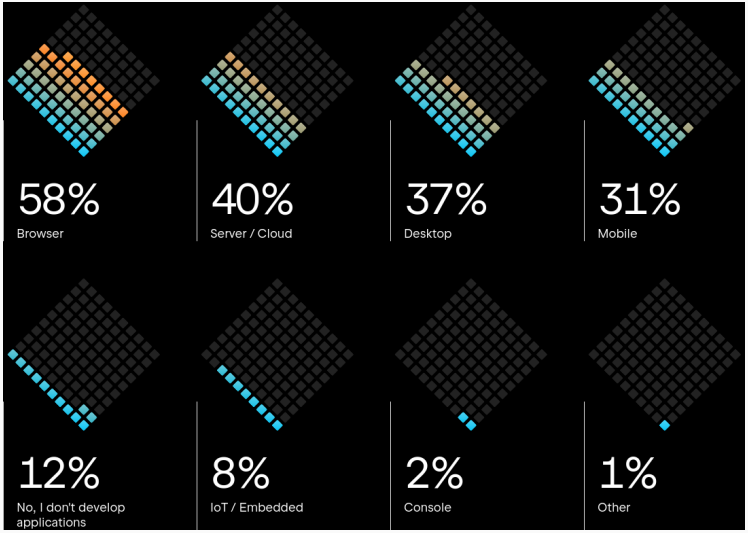


JavaScript

- 1995 : Brendan Eich développe en 10 jours un langage simple, interprété par le navigateur, pour dynamiser les pages web
- 1997 : première standardisation par ECMA (*European Computer Manufacturers Association*) sous le nom ECMAScript
- 2005 : AJAX popularise les applications web dynamiques
- 2008 : Google publie la machine virtuelle V8 pour l'exécution du code JavaScript dans Chrome
- 2009 : Node.js étend l'utilisation de JavaScript au côté serveur
- 2010 – 2014 : émergence des frameworks et d'un outillage moderne pour JavaScript (jQuery, AngularJS, etc.)
- 2015 : ECMAScript Edition 6 (ES6/ES2015) transforme le langage (modules, classes, *arrow functions*, *Promises*)



- *”Among the most dramatic rises in real-world usage over the last five years is TypeScript.”*



3. https://devecosystem-2025.jetbrains.com/tools-and-trends

TypeScript

- Conçu en 2012 par Anders Hejlsberg
- Open source, maintenu par Microsoft
- Ni interprété, ni compilé : langage **transpilé** vers JavaScript, *superset* de JS
- Exécutable dans n'importe quel *runtime* JS (navigateur, Node.js, etc.)
- Multi-paradigme : **fonctionnel**, générique, impératif, orienté objet

TypeScript

- **Typage statique** : détecter les erreurs et incohérences à la compilation (donc avant l'exécution)
- **Interopérabilité** : typage *progressif* (tout fichier JavaScript valide est un fichier TypeScript valide)

Adoption par l'industrie :

- Slack[4] : migration du frontend React vers TS pour sécuriser les appels d'API et réduire les erreurs au runtime
- Airbnb[5] : typage progressif du code JS pour réduire les coûts de maintenance
- Microsoft[6] : migration vers TS pour faciliter l'évolution de VS Code ($>$ 1M SLoC)
- Shopify[7] : migration frontend et backend pour fiabiliser les fonctions critiques

4. https://slack.engineering/typescript-at-slack/
5. https://www.youtube.com/watch?v=P-J9Eg7hJwE
6. https://www.git-tower.com/blog/developing-for-the-desktop-vscode
7. https://shopify.engineering/migrating-large-typescript-codebases-project-references

TypeScript

- Ressources :
  - *The TypeScript Handbook*[8]
  - *The Concise TypeScript Book*[9]
  - TypeScript TV[10] (liste des codes d'erreur)
  - TS Playground[11] (environnement d'exécution en ligne)

8. [8] Microsoft. *The TypeScript Handbook*. 2026
9. [9] Simone Poggiali. *The Concise TypeScript Book*. 2026
10. `https://typescript.tv/`
11. `https://www.typescriptlang.org/play/`

JavaScript, TypeScript et *runtime*

Typage statique et dynamique

TypeScript : typage du code JavaScript

Types de base disponibles dans TypeScript[12] :

- `boolean`
- `number`
- `string`
- Tableaux (dynamiques) – exemple : `string[]` ou `Array<string>`
- Tuples (taille fixe) – exemple : `[string, number]`
- `enum`
- `unknown` (type à préciser)
- `any` (à éviter : toute valeur autorisée)
- `void` (absence de valeur)
- `undefined` (valeur non définie)
- `null` (valeur vide)
- `never` (valeur impossible)

12. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

Exercice 1 : les pièges du typage dynamique

Fonctions et paramètres

- Écrire une fonction `addition(a, b)` qui additionne deux nombres.
- Tester la fonction avec les entrées suivantes :

```
addition(2, 3);      // attendu : 5
addition("2", 3);    // attendu : ?
addition(true, 3);   // attendu : ?
```

Questions

1. Que se passe-t-il pour les deux derniers cas ?
2. Quels problèmes cela pose-t-il poser dans une application réelle ?
3. Proposer une version TypeScript pour éviter ces erreurs *à la compilation*.

Exercice 1 : les pièges du typage dynamique

```ts
function addition2(a: number, b: number): number {
  return a + b;
}

console.log(addition2(2, 3));
// L'argument de type 'string' n'est pas attribuable au paramètre
// de type 'number'.ts(2345)
console.log(addition2("2", 3));
// L'argument de type 'boolean' n'est pas attribuable au paramètre
// de type 'number'.ts(2345)
console.log(addition2(true, 3));
```

JavaScript : POJO (*Plain Old Java Object*)

```
const o = {
    "foo": "bar",
    "value": 42,
    other: true,
}

console.log(o.foo);
console.log(o["value"]);
console.log(o.other);
```

POJO et listes : *Spread operator*

```
const o1 = { "abc": 123 };
// Ajout d'une nouvelle valeur dans `o1`
o1["def"] = 456;
// Nouvel objet avec copie des valeurs de `o1`
const o2 = { ...o1, "ghi": 789 };


const l1 = [123];
// Ajout d'une nouvelle valeur à la fin de `l1`
l1.append(456);
// Nouvelle liste avec copies des valeurs de `l1`
const l2 = [...l1, 789];
```

TypeScript : typer les objets avec `Record`

```ts
const o: Record<string, string | number | boolean> = {
    "foo": "bar",
    "value": 42,
    other: true,
}

console.log(o.foo);
console.log(o["value"]);
console.log(o.other);
console.log(o.blabla); // undefined
```

Typage, interfaces et classes

```ts
// Définition
interface Bidule {
    truc: string;
    machin: number;
    toto?: boolean;
}

// Instanciation
const b: Bidule = {
    truc: "bidule",
    machin: 42,
};
```

```ts
// Définition
class Bidule {
    constructor(
        public truc: string,
        public machin: number,
        public toto?: boolean,
    ) {}
}

// Instanciation
const b: Bidule = new Bidule(
    "bidule", 42
);
```

Exercice 2 : interfaces et contrats

Typage des objets

- Écrire une fonction `afficherUtilisateur(u)` qui prend un objet représentant un utilisateur et affiche son nom et son âge.
- Tester la fonction avec les entrées suivantes :

```
afficherUtilisateur({ nom: "Alice", age: 25 });
afficherUtilisateur({ nom: "Bob" });
```

Questions

1. Que se passe-t-il pour le second cas ?
2. Comment TypeScript peut-il garantir la présence des propriétés attendues ?
3. Proposer une solution avec TypeScript.

Exercice 2 : interfaces et contrats

```ts
interface Utilisateur {
    nom: string;
    age: number;
}

function afficherUtilisateur(u: Utilisateur): void {
    console.log(`${u.nom} a ${u.age} ans`);
}

afficherUtilisateur({ nom: "Alice", age: 25 });
// L'argument de type '{ nom: string; }' n'est pas attribuable
// au paramètre de type 'Utilisateur'.
//  La propriété 'age' est absente du type '{ nom: string; }'
//  mais obligatoire dans le type 'Utilisateur'.ts(2345)
afficherUtilisateur({ nom: "Bob" });
```

Typage, interfaces et classes

- Types **effacés** à la compilation
  - Pas d'impact sur les performances (compromis : temps de compilation vs temps d'exécution)
  - Une erreur de typage n'empêche pas la compilation vers JavaScript
- Corollaire : il n'est pas possible de vérifier les types TypeScript **lors de l'exécution** du programme…
- … sauf pour les types primitifs de JavaScript !

Typage, interfaces et classes

```
// type union : soit number, soit string
const fn = (x: number | string) => {
    // typeof type guard
    if (typeof x === 'number') {
        return x + 1;
    }

    return -1;
};
// type union : soit string, soit null
const toUpperCase = (name: string | null) => {
    // truthiness narrowing
    if (name) {
        return name.toUpperCase();
    } else {
        return null;
    }
};
```

Exercice 3 : types au *runtime*

```ts
interface Animal {
  name: string;
}
interface Dog extends Animal {
  bark: () => void;
}
interface Cat extends Animal {
  meow: () => void;
}
const makeNoise = (animal: Animal) => {
  if (animal instanceof Dog) {
    // 'Dog' fait référence à un type mais s'utilise en tant que valeur ici.ts(2693)
    animal.bark();
  } else {
    animal.meow();
  }
};
```

Questions

1. Que signifie l'erreur levée par le compilateur ?
2. Comment peut-on vérifier la nature de l'objet `animal` au *runtime* ?

Exercice 3 : types au *runtime*

```ts
interface Dog {
  kind: "dog"; // Union discriminée
  bark: () => void;
}

interface Cat {
  kind: "cat"; // Union discriminée
  meow: () => void;
}

type Animal = Dog | Cat; // Type union

const makeNoise = (animal: Animal) => {
  if (animal.kind === "dog") animal.bark();
  else animal.meow();
};
```

Exercice 3 : types au *runtime*

```ts
class Dog {
  constructor(public name: string, public bark: () => void) {}
}

class Cat {
  constructor(public name: string, public meow: () => void) {}
}

type Animal = Dog | Cat; // Union type

const makeNoise = (animal: Animal) => {
  if (animal instanceof Dog) animal.bark();
  else animal.meow();
};
```

Exercice 3 : types au *runtime*

```ts
interface Dog {
  kind: "dog"; // Union discriminée
  bark: () => void;
}
interface Cat {
  kind: "cat"; // Union discriminée
  meow: () => void;
}
type Animal = Dog | Cat;
const makeNoise = (animal: Animal) => {
  if (animal.kind === "dog") animal.bark();
  else animal.meow();
};
const dog: Dog = {
  kind: "dog",
  bark: () => console.log("bark")
};
makeNoise(dog);
```

Type Guards : *compile time* et *run time*

- *Type narrowing* : *compile time*
- *Type assertion* : *run time*

```ts
interface Something { id: string };

export function isSomething(obj: unknown): obj is Something {
  return !!obj
    && typeof obj === "object"
    && "id" in obj && typeof obj.id === "string";
}

const bla = { id: "blabla" };
```

JavaScript, TypeScript et *runtime*

Généricité

Généricité

- `SomeContainer<T extends SomeData>`
  - `T` est un type paramètre contraint à hériter ou implémenter `SomeData`
  - Permet de réutiliser le même conteneur pour différents types de données
- Objectifs :
  - Définir des structures et fonctions abstraites sur un type variable
  - Maintenir les contraintes de type

```ts
interface SomeData {
  id: string;
}

interface SpecializedData extends SomeData {
  name: string;
  value: number;
}

interface SomeContainer<T extends SomeData> {
  data: T;
}
```

```ts
const c: SomeContainer<
  SpecializedData
> = {
  data: {
    id: "foo",
    name: "Foo Bar",
    value: 42,
  },
};
```

JavaScript, TypeScript et *runtime*

Gestion des erreurs

Gestion des erreurs : exceptions

```
const randInt = Math.floor(Math.random() * 3) + 1; // [1, 3]
const error =
  randInt === 1 ? new TypeError() :
  randInt === 2 ? new RangeError() :
  new EvalError();

try {
  throw error;
} catch {
  // handle error
}
```

Gestion des erreurs : exceptions

```
const randInt = Math.floor(Math.random() * 3) + 1; // [1, 3]
const error =
    randInt === 1 ? new TypeError() :
    randInt === 2 ? new RangeError() :
    new EvalError();

try {
  throw error;
} catch (e) {
  if (e instanceof TypeError) {
    // handle TypeError
  } else if (e instanceof RangeError) {
    // handle RangeError
  } else if (e instanceof EvalError) {
    // handle EvalError
  }
}
```

Gestion des erreurs : exceptions

```
try {
  const file = await Deno.open("example.txt", { read: true });

  for await (const chunk of Deno.iter(file)) {
      console.log(new TextDecoder().decode(chunk));
  }
} catch (e) {
  console.error("Error reading file:", e);
} finally {
  // Toujours fermer le fichier
  if (file) {
    file.close();
  }
}
```

JavaScript, TypeScript et *runtime*

Asynchronisme et parallélisme

Programmation asynchrone

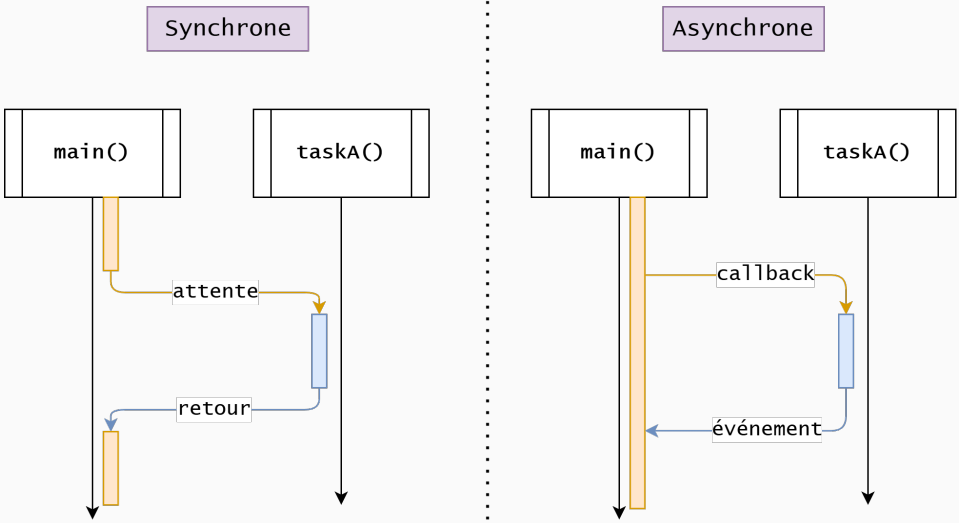
Comment manipuler des données…

- que l'on n'a pas encore ?
- que l'on n'aura peut-être jamais ?
- qui l'on obtient dans un temps variable ?

Par exemple…

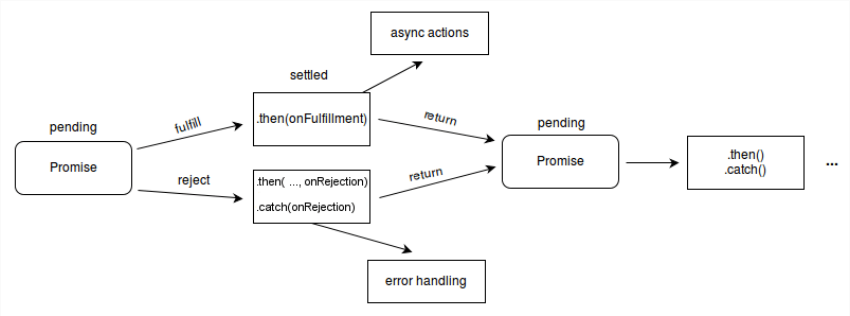
- le résultat d'une requête à une API ?
- le contenu d'un fichier volumineux sur le disque ?
- le résultat d'une requête à une base de données ?

Programmation asynchrone



Programmation asynchrone

- API : **Promises**
  - Traiter des opérations **asynchrones**
  - Une Promise *encapsule un résultat futur*
  - Elle est associée à des *callbacks* qui seront exécutés à l'obtention de ce résultat
  - Elle peut être *résolue* (réussite) ou *rejetée* (échec)



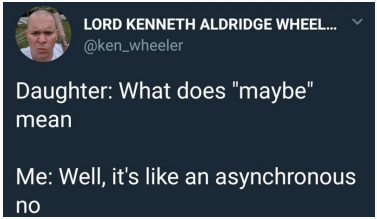
13. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Programmation asynchrone

```
function repeatMaybe(repeat: string): Promise<string> {
  return new Promise((resolve, reject) => {
    const maybe: number = randomIntegerBetween(0, 1);

    if (!maybe) {
        reject("Nope");
        return;
    }

    setTimeout(() => {
        resolve(repeat);
    }, 1000);
  });
}

const blabla = repeatMaybe("bla").then((result: string) => {
    console.log(result);
}).catch((error: string) => {
    console.log(error)
});
```

Programmation asynchrone

- API : `async`/`await`
  - Sucre syntaxique pour ”aplatir” du code asynchrone
  - `async` définit une fonction asynchrone
  - `await` met l’exécution du programme en pause dans l’attente d’un résultat




```
asyncFunction().then((res) => {
    console.log(res);
}).catch((err) => {
    console.log(err);
});
```

```
try {
    const res = await asyncFunction();
    console.log(res); // "bla"
} catch (err) {
    console.log(err); // "Nope"
}
```

Programmation asynchrone

```
function repeatMaybe(repeat: string): Promise<string> {
  return new Promise((resolve, reject) => {
    const maybe: number = randomIntegerBetween(0, 1);

    if (!maybe) {
        reject("Nope");
        return;
    }

    setTimeout(() => {
        resolve(repeat);
    }, 1000);
  });
}

try {
    const blabla = await repeatMaybe("bla");
    console.log(blabla); // "bla"
} catch (error) {
    console.log(error); // "Nope"
}
```

Programmation parallèle

- `Promise.all`
  - exécute toutes les Promises en **parallèle**
  - résout quand toutes les Promises ont **réussi**
  - rejette immédiatement si **une seule échoue**
  - retourne un tableau des résultats dans l'ordre initial des Promises
- `Promise.allSettled`
-
  - exécute toutes les Promises en **parallèle**
  - résout quand toutes les Promises sont terminées, **succès ou échec**
  - ne rejette **jamais**
  - retourne un tableau d'objets de type :
    - `status: 'fulfilled' | 'rejected', value | reason`

Exercice 4 : programmation asynchrone

Manipuler l'API Promise

- Écrire une fonction `readTextFileIfExists(path)` qui lit un fichier s'il existe et retourne son contenu si le fichier n'est pas vide :

```
readTextFileIfExists("data.txt")
  .then(text => console.log("File contents:", text))
  .catch(err => console.error("I/O error:", err.message));
```

Questions

1. Quelle est la signature de cette fonction en TypeScript ?
2. Comment utiliser cette fonction avec `await` ?
3. Comment utiliser cette fonction sur une liste de fichiers ?
4. Comment paralléliser la lecture des fichiers de la liste ?

Exercice 4 : programmation asynchrone

```
function readTextFileIfExists(path) {
  return new Promise((resolve, reject) => {
    if (!path) {
      reject(new Error("No file path provided"));
      return;
    }

    readFile(path, "utf8")
      .then(data => {
        if (data.length === 0) {
          reject(new Error("File is empty")); // explicit failure
        } else {
          resolve(data); // successful read
        }
      })
      .catch(err => reject(err)); // I/O error
  });
}
```

Exercice 4 : programmation asynchrone

```
try {
  const text = await readTextFileIfExists("data.txt");
  console.log("File contents:", text);
} catch (err) {
  console.error("I/O error:", err.message);
}
```

Exercice 4 : programmation asynchrone

```
const files = ["file1.txt", "file2.txt", "file3.txt"];

for (const file of files) {
  try {
    const text = await readTextFileIfExists(file);
    console.log("File contents:", text);
  } catch (err) {
    console.error("I/O error:", err.message);
    continue;
  }
}
```

🤔

Temps d'exécution total = ?

Exercice 4 : programmation asynchrone

```
const files = ["does_not_exist.txt", "file2.txt", "file3.txt"];

const contents = await Promise.all(
  files.map(readTextFileIfExists)
);
try {
    contents.forEach((text, i) => {
      console.log("File contents:", text);
    });
} catch (err) {
    console.error("I/O error:", err.message);
}
```

Temps d'exécution total = ?

Exercice 4 : programmation asynchrone

```
const files = ["does_not_exist.txt", "file2.txt", "file3.txt"];

const contents = await Promise.allSettled(
  files.map(readTextFileIfExists)
);
contents.forEach((result, i) => {
  if (result.status === "fulfilled") {
      console.log("File contents:", result.value);
  } else if (result.status === "rejected") {
      console.error("I/O error:", result.reason);
  }
});
```

🤔

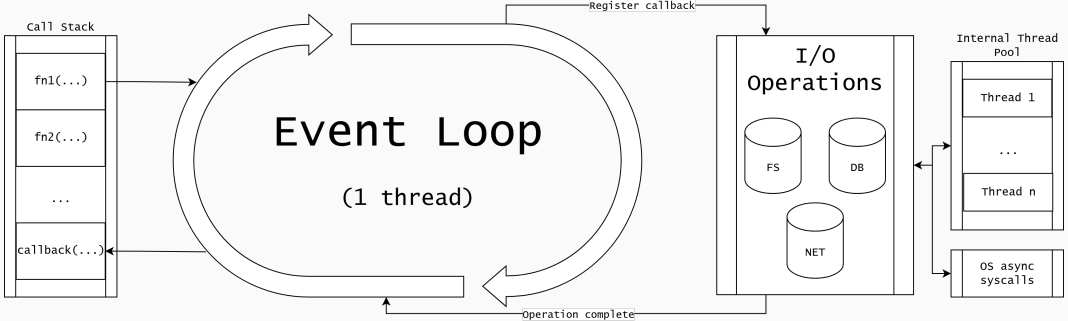
Temps d'exécution total = ?

JavaScript, TypeScript et *runtime*

Environnement d'exécution

Exécution : machine virtuelle

- JavaScript : langage interprété dans une VM
- Langage permet d'exprimer des instructions **asynchrones**
- Runtime **synchrone** : boucle d'événements (pas de coroutines, pas de threads [14])



14. Sauf *workers*, voir `https://developer.mozilla.org/en-US/docs/Glossary/Thread`

Runtime : Deno

- Runtime pour exécuter du code JavaScript en-dehors du navigateur
- Basé sur le moteur V8 (comme Node.js)
- Créé en 2018 par Ryan Dahl, développeur de Node.js [15]
- Points forts :
  - *Sandbox* : permissions explicites (lecture/écriture, accès au réseau, etc.)
  - Exécute du code TypeScript sans outillage supplémentaire
  - Inclut un formateur, un *linter*, une plateforme de test, un profileur

[15] https://www.youtube.com/watch?v=M3BM9TB-8yA

Sécurité, permissions

- *Supply-chain attack* : introduction de vulnérabilités dans un logiciel *via* compromission d'une ou plusieurs de ses dépendances
- Exemple : *Shai Hulud*, novembre 2025[16], 700 paquets compromis sur NPM
  - récupération de différents secrets présents sur la machine compromise […] ;
  - exfiltration des secrets ;
  - réplication par l'infection de paquets NPM ;
  - suppression de données utilisateurs ;
  - mise en place de persistance.

16. https://www.cert.ssi.gouv.fr/actualite/CERTFR-2025-ACT-051/

Gestion des dépendances

Outils et registres :

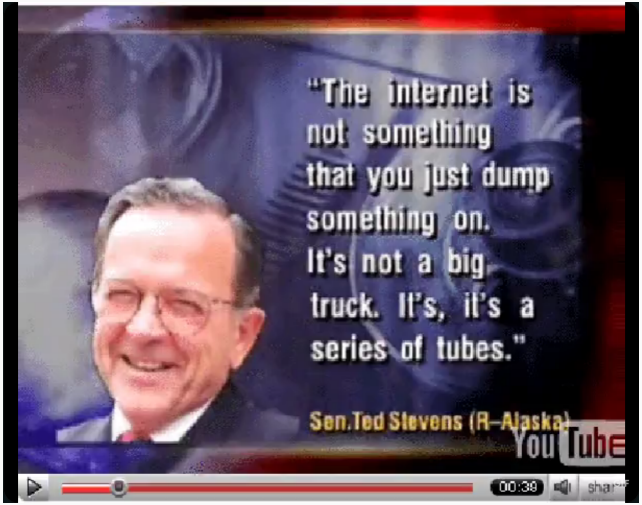
- `npmjs.org` : registre historique des paquets JavaScript
  - Gestionnaires de paquets : **npm** / **yarn**
- `jsr.org` : "nouveau" registre, orienté TypeScript
  - Gestionnaire de paquets : **jsr**, intégré dans Deno

Modules et imports :

- **CommonJS (CJS)** : compatibilité historique
  - Import : `const fs = require('fs');`
  - Export : `module.exports = { readFile };`
  - Chargement synchrone
- **ECMAScript Modules (ESM)** : standard moderne
  - Import : `import { readFile } from 'fs/promises';`
  - Export : `export const myFunc = () => {...};`
  - Chargement statique ou dynamique (asynchrone)

Serveur web

It's a series of tubes...



Résumé de la méthode en TP

1. La base de données retourne des **enregistrements** : ce sont des données arbitraires
   - Élément unique : `Record<string, SQLOutputValue> | undefined`
     - Peut être `undefined` si introuvable !
   - Liste : `Record<string, SQLOutputValue>[]`
     - Peut être vide !
2. On doit *vérifier le typage des données* pour que l'objet se conforme à son *interface base de données* (`PollRow`, …)
3. On peut alors *convertir l'enregistrement* en objet de notre *API* (`Poll`, …)
4. L'objet peut être retourné au client de manière *sûre*
   - *Compile time* : typage statique des données dans l'application client
   - *Run time* : pas d'erreur à l'exécution pour l'utilisateur

❶ Récupération des enregistrements en BDD

```
const pollId = ctx.params.pollId;
      const pollRow: Record<string, SQLOutputValue> | undefined
const pollRow = db.prepare(
  `SELECT id, title, description, user_id, created_at, expires_at, is_active
   FROM polls WHERE id = ?;`,
).get(pollId);

const pollOptionRows = db.prepare(
  `SELECT id, poll_id, text, vote_count FROM poll_options WHERE poll_id = ?;`,
).all(pollId);
```

❶ Récupération des enregistrements en BDD

```
const pollId = ctx.params.pollId;
      const pollRow: Record<string, SQLOutputValue> | undefined
const pollRow = db.prepare(
  `SELECT id, title, description, user_id, created_at, expires_at, is_active
   FROM polls WHERE id = ?;`,
).get(pollId);

const pollOptionRows = db.prepare(
  `SELECT id, poll_id, text, vote_count FROM poll_options WHERE poll_id = ?;`,
).all(pollId);
```

❷ Vérification des données (*type guard*)

```
if (!pollRow || !isPollRow(pollRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
      const pollOptionRows: Record<string, SQLOutputValue>[]
if (!pollOptionRows.every(isPollOptionRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
```

❶ Récupération des enregistrements en BDD

```
const pollId = ctx.params.pollId;
      const pollRow: Record<string, SQLOutputValue> | undefined
const pollRow = db.prepare(
  `SELECT id, title, description, user_id, created_at, expires_at, is_active
   FROM polls WHERE id = ?;`,
).get(pollId);

const pollOptionRows = db.prepare(
  `SELECT id, poll_id, text, vote_count FROM poll_options WHERE poll_id = ?;`,
).all(pollId);
```

❷ Vérification des données (*type guard*)

Données arbitraires

```
if (!pollRow || !isPollRow(pollRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
      const pollOptionRows: Record<string, SQLOutputValue>[]
if (!pollOptionRows.every(isPollOptionRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
```

❶ Récupération des enregistrements en BDD

```
const pollId = ctx.params.pollId;
      const pollRow: Record<string, SQLOutputValue> | undefined
const pollRow = db.prepare(
  `SELECT id, title, description, user_id, created_at, expires_at, is_active
   FROM polls WHERE id = ?;`,
).get(pollId);

const pollOptionRows = db.prepare(
  `SELECT id, poll_id, text, vote_count FROM poll_options WHERE poll_id = ?;`,
).all(pollId);
```

❷ Vérification des données (*type guard*)

Données arbitraires

```
if (!pollRow || !isPollRow(pollRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
      const pollOptionRows: Record<string, SQLOutputValue>[]
if (!pollOptionRows.every(isPollOptionRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
```

❸ Conversion des données et envoi au client

```
                                  const pollRow: PollRow
const poll = pollRowToApi(pollRow, pollOptionRows);

const responseBody: APIResponse<Poll> = {
  success: true,
  data: poll,
};

ctx.response.body = responseBody;
```

❶ Récupération des enregistrements en BDD

```
const pollId = ctx.params.pollId;
      const pollRow: Record<string, SQLOutputValue> | undefined
const pollRow = db.prepare(
  `SELECT id, title, description, user_id, created_at, expires_at, is_active
   FROM polls WHERE id = ?;`,
).get(pollId);

const pollOptionRows = db.prepare(
  `SELECT id, poll_id, text, vote_count FROM poll_options WHERE poll_id = ?;`,
).all(pollId);
```

```
if (!pollRow || !isPollRow(pollRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
      const pollOptionRows: Record<string, SQLOutputValue>[]
if (!pollOptionRows.every(isPollOptionRow)) {
  const responseBody: APIResponse<Poll> = {
    success: false,
    error: { code: APIErrorCode.NOT_FOUND, message: "Poll not found" },
  };

  ctx.response.status = 404;
  ctx.response.body = responseBody;
  return;
}
```

❷ Vérification des données (*type guard*)

Données arbitraires

❸ Conversion des données et envoi au client

```
                                  const pollRow: PollRow
const poll = pollRowToApi(pollRow, pollOptionRows);

const responseBody: APIResponse<Poll> = {
  success: true,
  data: poll,
};

ctx.response.body = responseBody;
```

Données conformes

Résumé de la méthode en TP

```ts
export interface PollRow {
  id: string;
  title: string;
  description: string | null;
  user_id: string | null;
  created_at: string;
  expires_at: string | null;
  is_active: number;
  [key: string]: SQLOutputValue;
}
```

```ts
export interface Poll {
  id: string;
  title: string;
  description?: string;
  options: PollOption[];
  userId?: string;
  createdAt: string;
  expiresAt?: string;
  isActive: boolean;
}
```

Résumé de la méthode en TP

```
export function isPollRow(
  obj: Record<string, SQLOutputValue>
): obj is PollRow {
  return !!obj &&
    typeof obj === "object" &&
    "id" in obj && typeof obj.id === "string" &&
    "title" in obj && typeof obj.title === "string" &&
    "description" in obj &&
    (typeof obj.description === "string" || obj.description === null) &&
    "user_id" in obj &&
    (typeof obj.user_id === "string" || obj.user_id === null) &&
    "created_at" in obj && typeof obj.created_at === "string" &&
    "expires_at" in obj &&
    (typeof obj.expires_at === "string" || obj.expires_at === null) &&
    "is_active" in obj && typeof obj.is_active === "number";
}
```

Résumé de la méthode en TP

```
export function pollRowToApi(
  row: PollRow,
  optionRows: PollOptionRow[],
): Poll {
  return {
    id: row.id,
    title: row.title,
    description: row.description ?? undefined,
    createdAt: row.created_at,
    expiresAt: row.expires_at ?? undefined,
    isActive: row.is_active === 1,
    options: optionRows.map(pollOptionRowToApi),
  };
}
```

Résumé de la méthode en TP

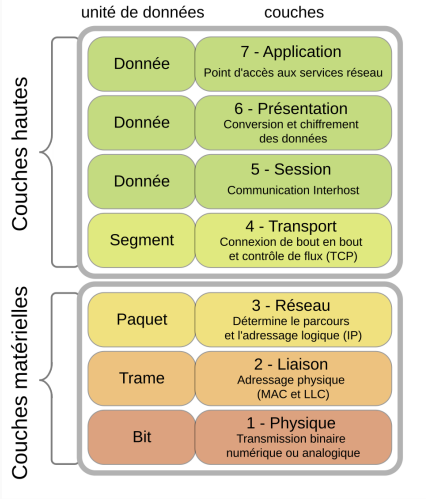
```
export function pollApiToRow(
  poll: Poll
): [PollRow, PollOptionRow[]] {
  return [
    {
      id: poll.id,
      title: poll.title,
      description: poll.description ?? null,
      user_id: poll.userId ?? null,
      created_at: poll.createdAt,
      expires_at: poll.expiresAt ?? null,
      is_active: Number(poll.isActive),
    },
    poll.options.map((pollOption) => pollOptionApiToRow(
      poll.id, pollOption
    )),
  ];
}
```

Serveur web

Protocole HTTP

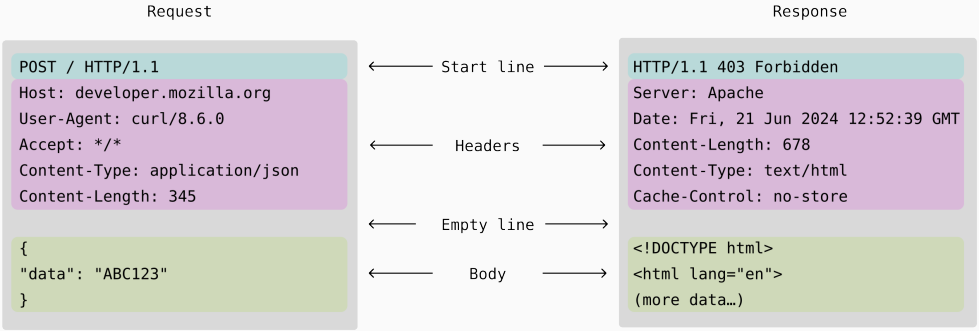
HTTP

- *Hypertext Transfer Protocol* : à l'origine, sert à partager des pages web reliées entre elles par des liens
- Inventé au CERN[17] par un chercheur britannique, Tim Berners-Lee, entre 1989 et 1991
- 1997 : standardisation de HTTP/1.1 :
  - Apparition de l'en-tête `Host`, permettant d'héberger plusieurs domaines sur la même adresse IP, donc la colocation de serveurs web



[17] Initialement *Conseil européen pour la recherche nucléaire*, plus grand centre de recherche au monde en physique des particules

HTTP : requêtes et réponses



HTTP : méthodes

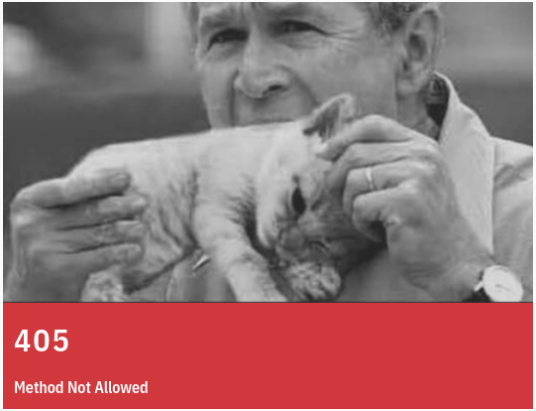
Méthodes principales pour les requêtes HTTP[18], aussi appelées *verbes* :

- `GET` : demande d'une ressource
- `POST` : envoi de données du *corps* de la requête vers une ressource
- `PUT` : remplacement d'une ressource par le *corps* de la requête
- `DELETE` : suppression d'une ressource

18. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods

HTTP : codes de réponse

- `100` à `199` : réponses informatives
  - `101 Switching Protocols`
- `200` à `299` : réponses de succès
  - `200 OK`
- `300` à `399` : réponses de redirection
  - `301 Moved Permanently`
- `400` à `499` : erreurs du client
  - `401 Unauthorized`
- `500` à `599` : erreurs du serveur
  - `500 Internal Server Error`



… et bien d'autres : `https://http.cat/`

HTTP : architecture REST

- *REpresentational State Transfer* [1] : défini par Roy Fielding en 2000, conçu pendant son doctorat (1996 - 1999)
- Modèle d'architecture logicielle pour applications réseau
- Manipuler des ressources distantes via des représentations textuelles
- Opérations uniformes, prédéfinies, et **sans état**

Méthodes HTTP

URI	GET	POST	PUT	PATCH	DELETE
Ressource collection, telle que `http://api.exemple.com/collection/`	*Récupère* les URI des ressources membres de la ressource collection dans le corps de la réponse.	*Crée* une ressource membre dans la ressource collection en utilisant les instructions du corps de la requête. L'URI de la ressource membre créée est *attribué automatiquement* et retourné dans le champ d'en-tête *Location* de la réponse.	*Remplace* toutes les représentations des ressources membres de la ressource collection par la représentation dans le corps de la requête, ou *crée* la ressource collection si elle n'existe pas.	*Met à jour* toutes les représentations des ressources membres de la ressource collection en utilisant les instructions du corps de la requête, ou *crée éventuellement* la ressource collection si elle n'existe pas.	*Supprime* toutes les représentations des ressources membres de la ressource collection.
Ressource membre, telle que `http://api.exemple.com/collection/item3`	*Récupère* une représentation de la ressource membre dans le corps de la réponse.	*Crée* une ressource membre dans la ressource membre en utilisant les instructions du corps de la requête. L'URI de la ressource membre créée est *attribué automatiquement* et retourné dans le champ d'en-tête *Location* de la réponse.	*Remplace* toutes les représentations de la ressource membre, ou *crée* la ressource membre si elle n'existe pas, par la représentation dans le corps de la requête.	*Met à jour* toutes les représentations de la ressource membre, ou *crée éventuellement* la ressource membre si elle n'existe pas, en utilisant les instructions du corps de la requête.	*Supprime* toutes les représentations de la ressource membre.

Exercice 5 : un serveur HTTP minimal avec TypeScript

Lire et écrire sur un socket TCP

Ci-dessous, la boucle principale pour ouvrir et écouter sur un socket TCP avec Deno. Dès la connexion d'un **client**, on passe le socket à la fonction `handleConn`.

```
// TCP socket
const listener = Deno.listen({ port: 8080 });
// Strings <-> Bytes
const decoder = new TextDecoder();
const encoder = new TextEncoder();
// Listen loop
for await (const conn of listener) {
  handleConn(conn);
}
```

Exercice 5 : un serveur HTTP minimal avec TypeScript

Lire et écrire sur un socket TCP

Ci-dessous, la fonction `handleConn` par laquelle chaque **requête** en provenance d'un **client** est traitée. Le serveur reçoit un **tableau d'octets**; on le convertit en une chaîne de caractères dans `rawRequest`.

```
async function handleConn(conn: Deno.Conn) {
  const buf = new Uint8Array(1024);
  const bytes = await conn.read(buf);
  if (bytes === null) {
    conn.close();
    return;
  }

  const rawRequest = decoder.decode(buf.subarray(0, bytes));
}
```

Exercice 5 : un serveur HTTP minimal avec TypeScript

```
curl -X POST http://localhost:8080 \
     -H "Content-Type: text/plain" \
     -d "hello world"
```

```
POST / HTTP/1.1
Host: localhost:8080
User-Agent: curl/8.14.1
Accept: */*
Content-Type: text/plain
Content-Length: 11

hello world
```

Questions

1. Quels sont les champs à inclure dans la réponse que l'on va retourner au client ?
2. Compléter la fonction `handleConn` pour signifier une réponse OK comportant du texte au format JSON.
3. Comment va-t-on gérer l'horodatage ?

Serveur web

Mise en œuvre

Framework pour serveur HTTP : Oak

- Inspiré par Express.js, initialement développé pour Deno
- Framework *middleware* : un logiciel qui s'intercale entre la requête d'un client et la réponse d'un serveur
- **Contexte** = **Requête** + **Réponse**
- Mécanisme principal : le **routage**
  - Pour un point d'entrée donné, appliquer telle fonction, retourner telle valeur
  - Requête : `GET http://blabla.com/item/123`
  - Méthode : `GET`
  - Route : `/item/:itemId`
  - Fonction : `getItem(id)`
  - Réponse : `{ id: 123, value: "..." }`

Framework pour serveur HTTP : Oak

```
// Configuration
const BASE_URL = "http://localhost:8000";
// Initialisation
const app = new Application();
const router = new Router();
// Middlewares
app.use(router.routes()); // Encore faut-il définir des routes ! ...
app.use(router.allowedMethods());
// Événements
app.addEventListener(
  "listen", () => console.log(`Server listening on ${BASE_URL}`),
);
app.addEventListener(
  "error", (e) => console.log(`Uncaught error: ${e.message}`),
);
// Boucle principale
if (import.meta.main) {
  await app.listen({ hostname: "localhost", port: 8000 });
}
// Export
export { app };
```

Routage avec Oak : paramètres de la requête

- L'utilisateur envoie une requête `GET`
- Adresse : `http://localhost:8000/polls/abcd`

```
router.get("/:pollId", (ctx) => {
  const pollId = ctx.params.pollId; // abcd
});
```

Routage avec Oak : paramètres de la requête

- L'utilisateur envoie une requête POST
- Corps de la requête : un objet JSON

```
{ "foo": "bar" }
```

- Adresse : `http://localhost:8000/polls`

```
router.post("/polls", async (ctx) => {
  const createPollRequest = await ctx.request.body.json();
  // { "foo": "bar" }
});
```

Routage avec Oak : réponse à la requête

```typescript
export interface APISuccess<T> {
  success: true;
  data: T;
  error?: never;
}

export interface APIFailure {
  success: false;
  data?: never;
  error: APIError;
}
```

Routage avec Oak : réponse à la requête

```
export enum APIErrorCode {
  NOT_FOUND = "NOT_FOUND",
  SERVER_ERROR = "SERVER_ERROR",
  TIMEOUT = "TIMEOUT",
  UNAUTHORIZED = "UNAUTHORIZED",
  VALIDATION_ERROR = "VALIDATION_ERROR",
}

export interface APIError {
  code: APIErrorCode;
  message: string;
}
```

Routage avec Oak : réponse à la requête

- Et finalement, le type de **toute** réponse de l'API… 🥁🥁🥁

```
export type APIResponse<T> = APISuccess<T> | APIFailure;
```

Routage avec Oak : réponse à la requête

- Et finalement, le type de **toute** réponse de l'API… 🥁🥁🥁

```
export type APIResponse<T> = APISuccess<T> | APIFailure;
```

Routage avec Oak : réponse à la requête

- Si l'entité demandée est trouvée :

```
const responseBody: APIResponse<Poll> = {
  success: true,
  data: poll,
};

// Oak passe implicitement le code HTTP OK dans la réponse
ctx.response.body = responseBody;
```

Routage avec Oak : réponse à la requête

- Si l'entité demandée est introuvable :

```
const responseBody: APIResponse<Poll> = {
  success: false,
  error: {
    code: APIErrorCode.NOT_FOUND,
    message: "Poll not found",
  },
};

ctx.response.status = 404;
ctx.response.body = responseBody;
```

Routage avec Oak : réponse à la requête

- Le serveur rencontre une erreur :

```
const responseBody: APIResponse<Poll> = {
  success: false,
  error: {
    code: APIErrorCode.SERVER_ERROR,
    message: "Failed to fetch poll",
  },
};

ctx.response.status = 500;
ctx.response.body = responseBody;
```

Routage avec Oak : contexte

Quelques propriétés importantes de l'objet `ctx: Context`[19] :

- `request: Request` – An object which contains information about the current request.
- `response: Response` – An object which contains information about the response that will be sent when the middleware finishes processing.
- `state: S` – The object to pass state to front-end views. This can be typed by supplying the generic state argument when creating a new app.
  - *On each request/response cycle, the context's state is cloned from the application state. This means changes to the context's .state will be dropped when the request drops, but "defaults" can be applied to the application's state. Changes to the application's state though won't be reflected until the next request in the context's state.*

19. https://jsr.io/@oak/oak/doc/context/~/Context

Middleware

- Code applicatif qui se positionne **entre la requête et la réponse**
- **Intercepte** les données pour leur appliquer des modifications ou effectuer des effets de bord
- En pratique :
  - Une **fonction** qui prend en paramètre le **contexte** (`ctx`) et la **prochaine fonction** (`next`)
  - Invoquée dans une **chaîne** de traitement, ordonnée, pour chaque requête
  - Capable d'observer, modifier, court-circuiter ou ignorer le cycle de vie requête-réponse
  - Appelée par le framework (Oak) lors de la réception d'une requête

Middleware

```
interface HelloWorldState {
  hello?: string;
  world?: string;
}
async function hello(ctx: Context<HelloWorldState>, next: Next) {
  ctx.state.hello = "Hello";
  await next();
}
async function world(ctx: Context<HelloWorldState>, next: Next) {
  ctx.state.world = "world";
  await next();
}
const router = new Router<HelloWorldState>();
router.get("/hello", hello, world, (ctx) => {
  ctx.response.body = `${ctx.state.hello}, ${ctx.state.world}!`;
});
const app = new Application<HelloWorldState>();
```

Exercice 6 : gestion des erreurs du serveur

Un unique point de sortie pour les réponses d'erreur

On veut éviter de dupliquer du code pour retourner des erreurs au client dans les fonctions de nos routes. On va donc utiliser un **middleware** qui sera chargé de remplir la réponse en cas d'erreur.

```
export async function errorMiddleware(ctx: Context, next: Next) { }
```

Questions

1. Sur quel mécanisme s'appuyer pour intercepter les erreurs levées dans les routes de l'application ?
2. Comment différencier une erreur *prévue* d'une erreur *inattendue* ?

Serveur web

Gestion de l’état

Base de données

- 2000 : création de SQLite par D. Richard Hipp
- Objectifs initiaux : base de données relationnelle **légère** et **sans serveur** pour être **embarquée** dans les applications
- Caractéristiques :
  - *In-memory* : les opérations sont réalisées dans la RAM (pas sur le disque)
  - *Single-file* : pas de processus serveur, pas de runtime, pas de dépendances, etc.
- Simple à déployer, simple à maintenir
- Compétitive en matière de fonctionnalités (transactions, concurrence des accès en lecture/écriture, extensible, etc.)

```
sqlite3 database.db < schema.sql
```

Base de données : rappels sur SQL

```sql
CREATE TABLE things (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    maybe TEXT,
    active INTEGER NOT NULL
);

CREATE TABLE thing_options (
    id TEXT PRIMARY KEY,
    thing_id TEXT NOT NULL,
    value TEXT NOT NULL,
    FOREIGN KEY (thing_id) REFERENCES things(id) ON DELETE CASCADE
);
```

Base de données : transactions

- **Interdépendance** (sémantique et/ou fonctionnelle) entre deux requêtes **distinctes**
- Les deux doivent réussir ou échouer **ensemble** pour éviter une **incohérence dans l'état** de l'application

```
// Enregistrer le vote
const voteId = crypto.randomUUID();
const now = new Date().toISOString();
db.prepare(
  "INSERT INTO votes " +
  "(id, poll_id, option_id, user_id, created_at) " +
  "VALUES (?, ?, ?, ?, ?);",
).run(voteId, pollId, optionId, userId || null, now);

// Incrémenter le compteur
db.prepare(
  "UPDATE poll_options SET vote_count = vote_count + 1 WHERE id = ?;",
).run(optionId);
```

Base de données : transactions

- Mécanisme : **transactions**
- Unités de travail qui garantissent l'**atomicité** des opérations

```
try {
  db.prepare("BEGIN").run();     // Début de transaction

  db.prepare("...").run();       // Requête 1
  db.prepare("...").run();       // Requête 2

  db.prepare("COMMIT").run();    // Transaction validée
} catch {
  db.prepare("ROLLBACK").run(); // Transaction annulée
}
```

Exercice 7 : gestion des erreurs du serveur

Besoin d'atomicité

1. Quelles opérations doivent être **atomiques** dans notre application de sondage ?
2. Pourquoi ? C'est-à-dire : quels problèmes peuvent survenir si les sous-opérations, disjointes, échouent ?

Serveur web

Architecture

Organisation du dépôt

Exercice 8 : architecture du serveur

Organisation des fichiers de l'application

Oak ne privilégie pas de paradigme précis, la responsabilité incombe aux développeurs d'organiser le code de leur application. À ce stade, les composants principaux sont :

- Routes
- Middleware
- Interfaces
- Base de données

Questions

1. Doit-on privilégier une organisation type `module/module.routes.ts` ? `routes/module.ts` ? Donner des avantages et des inconvénients.
2. Pour les deux cas, montrer les directives `export` et `import` associées.

Serveur web

Fiabilité et tests

Tests

```
import { assertEquals, assert } from "@std/assert";

const BASE_URL = `http://localhost:8000`;

Deno.test({
  name: "polls management API",
  async fn() {
    const listRes = await fetch(`${BASE_URL}/polls`);
    assertEquals(listRes.status, 200);

    const listBody = await listRes.json();
    assert(listBody.success);
    assert(listBody.data.length >= 1);
    },
});
```

Exercice 9 : un premier jeu de tests

Création d'un jeu de tests unitaires

On souhaite éviter toute **régression** dans le comportement du serveur. Il faut donc tester **unitairement** les fonctionnalités de base de l'API.

Questions

1. Écrire un test pour vérifier le fonctionnement nominal de :
   - la fonction de vérification de l'interface d'un enregistrement issu de la base de données (`PollRow`);
   - la fonction de conversion de cet enregistrement en objet de l'API (`Poll`);
2. Écrire un test pour vérifier le fonctionnement nominal de la route `/polls`.

Client web

Client web

Requêtes HTTP

HTTP, encore

- Avant 2014 : API `XMLHttpRequest`
  - Basée sur l'utilisation de *callbacks*
- Depuis 2014 : API `fetch`
  - Basée sur l'utilisation de *Promises*
  - 2014 : *Web Hypertext Application Technology Working Group (WHATWG) Fetch Standard*[20]
  - 2015 : implantation dans Chrome et Firefox

Standard Fetch

The Fetch standard defines requests, responses, and the process that binds them : fetching.

20. `https://fetch.spec.whatwg.org/`

Requêtes HTTP : XMLHttpRequest

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com/data");
xhr.addEventListener("load", () => {
  try {
    if (xhr.status >= 200 && xhr.status < 300) {
      const data = JSON.parse(xhr.responseText);
      console.log("Response:", data);
    } else {
      throw new Error(`HTTP ${xhr.status}`);
    }
  } catch (error) {
    console.error("Error:", error);
  }
});
xhr.addEventListener("error", () => {
  console.error("Network Error");
});
xhr.send();
```

Requêtes HTTP : `fetch`

```
fetch("https://api.example.com/data")
  .then(response => {
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    return response.json();
  })
  .then(data => console.log("Response:", data))
  .catch(error => console.error("Error:", error));
```

Requêtes HTTP : `fetch` (avec `await`)

```
try {
  const response = await fetch("https://api.example.com/data");
  if (!response.ok) throw new Error(`HTTP ${response.status}`);
  const data = await response.json();
  console.log("Response:", data);
} catch (error) {
  console.error("Error:", error);
}
```

Client web

IHM et DOM

Un client web minimal avec JavaScript

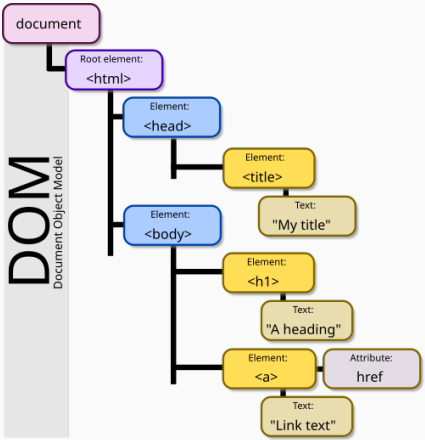
```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Title</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
  <body>
    <main class="content"></main>
  </body>
</html>
```

Un client web minimal avec JavaScript

```
// script.js
document.addEventListener("DOMContentLoaded", () => {
  fetch("http://localhost:8000")
    .then(response => {
      if (!response.ok) {
        throw new Error(`HTTP error ${response.status}`);
      }
      return response.text();
    })
    .then(data => {
      const main = document.querySelector("main.content");
      main.innerHTML = data;
    })
    .catch(error => {
      console.error("Fetch failed:", error);
    });
});
```

Qu’est-ce qu’une page web ?

- *Document Object Model* : représentation de la structure d’un document (une page web) en mémoire
- Forme : arbre logique (les branches relient des nœuds, les nœuds contiennent des objets)
- Les nœuds peuvent être associés à des gestionnaires d’événements (*Event Listeners*) qui s’exécutent lors d’un changement dans le DOM



21. https://dom.spec.whatwg.org/
22. https://fr.wikipedia.org/wiki/Document_Object_Model

IHM : paradigme

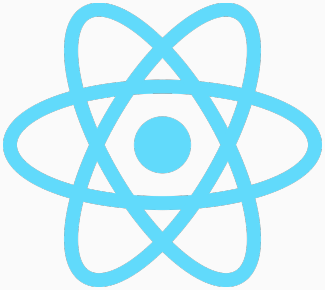
- Paradigme impératif : "s'il se passe quelque chose, modifier l'état local de telle manière" (commander des changements)
  - L'état est **mutable**, le flot de contrôle décide des changements dans l'interface
  - Un événement déclenche une modification de l'interface
- Paradigme réactif : "dès qu'il se passe quelque chose, recalculer l'état pour redessiner la page" (décrire un résultat)
  - L'état est **immuable**, l'interface est une fonction de l'état
  - UI $= f(\text{state})$
  - Un événement définit une transition d'état
    - Le prochain DOM est calculé à partir de l'événement
    - React le compare (*diff*) avec l'arbre courant
    - React applique le delta des modifications au DOM

Client web

React

Framework : React

- Initialement *ReactJS*, créé par Jordan Walke (Facebook) en 2013
- Besoin : un framework efficace pour développer des IHM interactives pour des applications intensives en données (Facebook Ads)
- Fondamentaux :
  - Architecture à base de composants : ”conteneurs” réutilisables, chacun maintient son propre état local
  - Déclaratif plutôt qu’impératif : les développeurs définissent l’apparence de l’interface pour chaque état possible, et React gère la mise à jour du DOM automatiquement
  - Document abstrait : le *DOM virtuel* permet de représenter les mises à jour efficacement en mémoire
  - Flot de données unidirectionnel : les données circulent du parent vers les composants enfants via les *props*



- De nombreux outils sont utilisés pour produire une application client :
  - **Vite** : outils de développement
    - inclut le compilateur TypeScript (`tsc` et `esbuild`)
    - fournit un serveur de développement avec rechargement à chaud des modules (*Hot Module Reload*)
  - **Rolldown** : *bundler*
    - prend des modules JavaScript et les assemble en un seul module
    - optimise le résultat pour la production (*tree-shaking*)

React : vue d'ensemble

- Deux concepts pour développer avec React :
  - **Composants** : gèrent la *vue* (l'interface)
    - prennent des *props* (propriétés) en entrée, en lecture seule
    - maintiennent un *state* (état) dont les changements provoquent un re-rendu
  - ***Hooks*** : gèrent le *modèle* (la logique) derrière la vue
- Composants et *hooks* sont des **fonctions** !
- Un composant retourne une sous-partie de l'arbre du DOM au format JSX[23]
- Un *hook* retourne des valeurs arbitraires (fonctions, tuples, etc.)

23. https://www.typescriptlang.org/docs/handbook/jsx.html

React : exemple (composant)

```tsx
import { useCounter } from "./hooks/useCounter";
// pages/Counter.tsx
function Counter({ initialValue = 0 }: { initialValue?: number }) {
  const { count, increment, decrement } = useCounter(initialValue);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
    </div>
  );
}

export default Counter;
```

React : exemple (*hook*)

```ts
import { useState } from "react";
// hooks/useCounter.ts
export function useCounter(initialValue: number = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(prev => prev + 1);
  const decrement = () => setCount(prev => prev - 1);
  return { count, increment, decrement };
}
```

React : exemple (instanciation)

```
export default function App() {
  return <Counter initialValue={0} /> // prend un `number`
}
```

- *Props* du composant : la valeur initiale du compteur `initialValue`
- `count` est une **variable d'état**, `setCount` est une fonction d'écriture de l'état
- Cliquer sur les boutons + et − modifie **l'état** du composant et provoque son re-rendu

React : cycle de vie d'un composant

- *Mount* (montage)
  - Phase de **création et d'insertion** d'un composant dans le DOM
  - Initialise son état à partir des ses *props*
  - Est *rendu* (retourne le JSX à afficher)
- *Update* (mise-à-jour)
  - Phase de **re-rendu** d'un composant
  - Déclenchée par un changement dans les *props* ou dans l'état du composant
  - Exécute les effets dont les **dépendances** comportent un changement
- *Unmount* (démontage)
  - Phase de **suppression** d'un composant dans le DOM
  - Déclenchée par un changement de condition, un changement de route, etc.
  - Exécute les fonctions de nettoyage des *effets*

React : *hook* useState

Modèle mental

La mémoire d'un composant, qui provoque les mises à jour de l'interface.

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Incrémenter
      </button>
    </div>
  );
}
```

React : *hook* useEffect

Modèle mental

Réagir aux changements dans l'environnement.

```
function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // Fonction de nettoyage
  }, []); // Pas de dépendances : s'exécute une fois, au premier rendu

  return <div>Secondes : {seconds}</div>;
}
```

React : *hook* useRef

Modèle mental

Une boîte qui survit aux rendus, sans affecter l'interface.

```
function Tracker() {
  const [renderCount, setRenderCount] = useState(0);
  const clickCountRef = useRef(0); // Valeur mutable persistante

  // Mise-à-jour sans re-rendu
  const handleClick = () => clickCountRef.current += 1;
  // Force le re-rendu
  const handleRefresh = () => setRenderCount(prev => prev + 1);

  return (
    <div>
      <p>Button cliqué (compteur): {clickCountRef.current} fois</p>
      <p>Button cliqué (rendu): {renderCount} fois</p>
      <button onClick={handleClick}>Incrémenter</button>
      <button onClick={handleRefresh}>Rafraîchir</button>
    </div>
  );
}
```

React : routage

- **React Router**[24] : gère la **navigation** au sein d'une application React
- Choisir le composant à rendre en fonction de l'**URL** :
  - Routes : associer des *composants* à des *chemins* ;
  - `Link` : naviguer sans recharger la page ;
  - `useParams` : accéder aux parties dynamiques de l'URL ;
  - `useNavigate` : navigation par programmation.

24. https://reactrouter.com/

React : routage

```
import React from "react";
import { useParams, useNavigate } from "react-router";

function Home() {
  return <h2>Accueil</h2>;
}

function User() {
  const { id } = useParams(); // paramètre de l'URL
  const navigate = useNavigate();

  return (
    <div>
      <h2>Utilisateur {id}</h2>
      <button onClick={() => navigate("/")}>Retour</button>
    </div>
  );
}
```

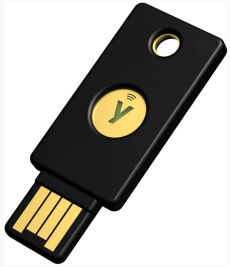
React : routage

```
import { BrowserRouter, Routes, Route, Link } from "react-router";

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Accueil</Link> |
        <Link to={`/user/${id}`}>Utilisateur</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/user/:id" element={<User />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Authentification

Concepts généraux

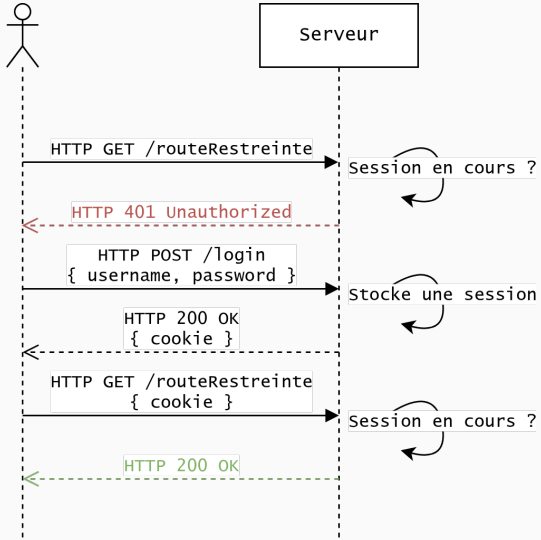
- Mécanisme de **contrôle d'accès**
- S'assurer de la légitimité d'une entité (humain ou système)
- Pour autoriser ses requêtes vers différentes ressources



Username:
admin
Password:
123
Login

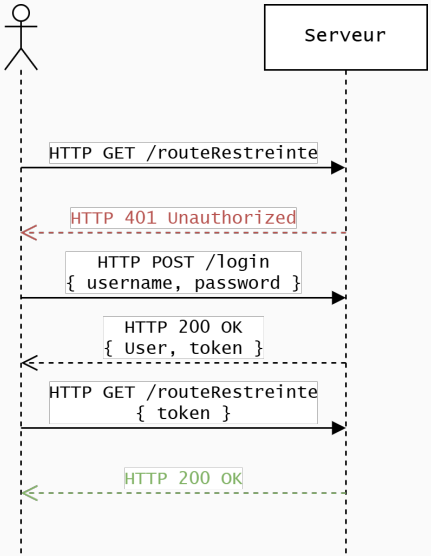
Stateful : sessions et cookies

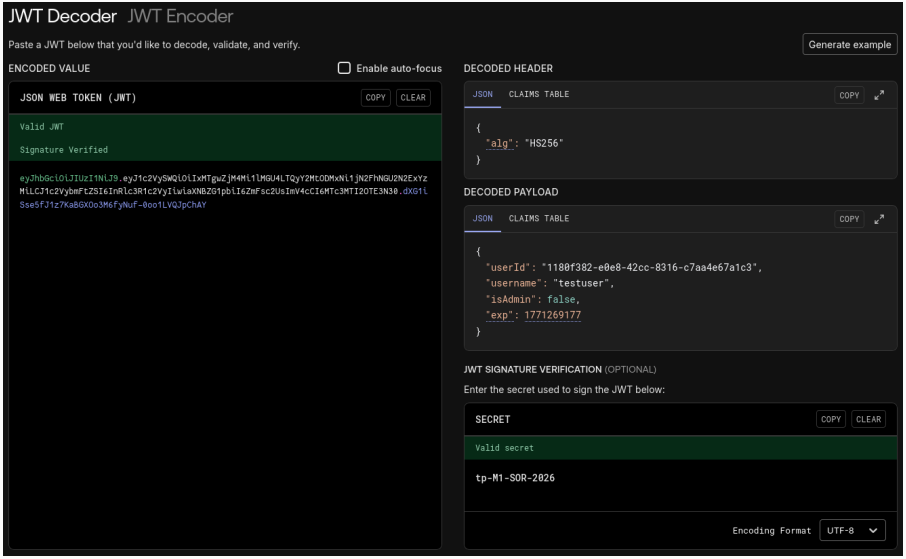
1. Le client envoie une requête de connexion au serveur
2. Si le client est autorisé…
3. Le serveur retourne un identifiant de session dans un *cookie*
4. Le client peut demander une ressource protégée en envoyant sa requête et son cookie
5. Si la session en cours est valide…
6. Le serveur lui retourne la ressource demandée



Stateless : JWT (jetons)

1. Le client envoie une requête de connexion au serveur
2. Si le client est autorisé…
3. Le serveur retourne un *jeton* avec une période de validité
4. Le client peut demander une ressource protégée en envoyant sa requête et son jeton
5. Si le jeton est valide…
6. Le serveur lui retourne la ressource demandée



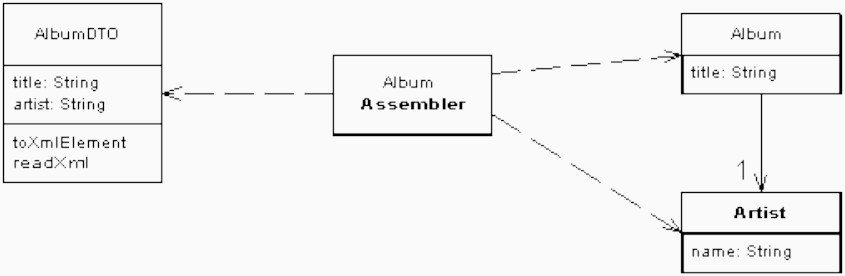


25. https://www.jwt.io/

Interactions

Requêtes/Réponses : *Data Transfer Objects*

- Un objet, sérialisable, pour franchir la frontière entre les processus [26]
- Formalise les points de communications dans une application à plusieurs tiers



26. [2] Martin Fowler. *Patterns of enterprise application architecture*. 2012

Requêtes/Réponses : *Data Transfer Objects*

- Mise en œuvre en TypeScript :
  - Interface *ad hoc*
  - Type intersection
  - Type `Omit`

```ts
type User = {
  name: string;
  password: string;
};

type UserWithoutPassword = Omit<
  User, "password"
>;

const user: UserWithoutEmail = {
  name: "Alice"
};
```

```ts
type UserWithoutPassword = {
  name: string;
};

type User = UserWithoutPassword & {
  password: string
};

const alice = { name: "Alice" };
const user: User = {
  ...alice,
  password: "123"
};
```

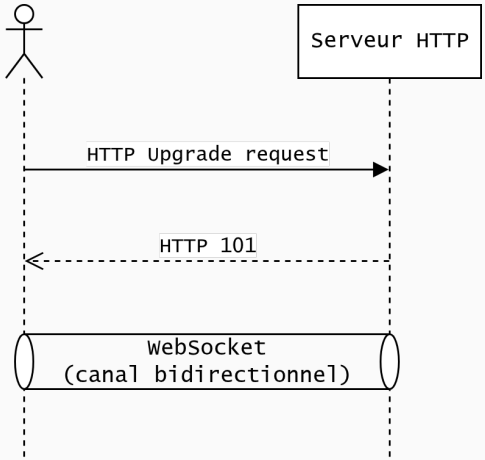
Protocole bidirectionnel : WebSocket

Client :

```
const ws = new WebSocket(
  `${WS_URL}/votes/${pollId}`
);
```

Serveur :

```
if (!ctx.isUpgradable) {
  throw new APIException(
    APIErrorCode.BAD_REQUEST,
    400,
    "WebSocket required"
  );
}

const ws: WebSocket = ctx.upgrade();
```



Déploiement de l’application

Nom d’hôte, port, adresse

https://foo.bar.com[:443]/baz.html

Nom d'hôte, port, adresse

- `https` : protocole
- `foo.bar` : nom d'hôte (*hostname*)
- `com` : *TLD* (*Top-Level Domain*)
- `443` : port (HTTP : 80, HTTPS : 443)
- `baz.html` : nom de fichier

Oak : configuration dynamique côté serveur

```ts
export const PROTOCOL = Deno.env.get("PROTOCOL") || "http";
export const HOSTNAME = Deno.env.get("HOSTNAME") || "localhost";
export const PORT = Number(Deno.env.get("PORT")) || 8000;
export const BASE_URL = `${PROTOCOL}://${HOSTNAME}:${PORT}`;
```

Oak : configuration dynamique côté serveur

```
# fichier .env à la racine du projet
PROTOCOL=https
HOSTNAME=api.coucou.localhost
PORT=443
```

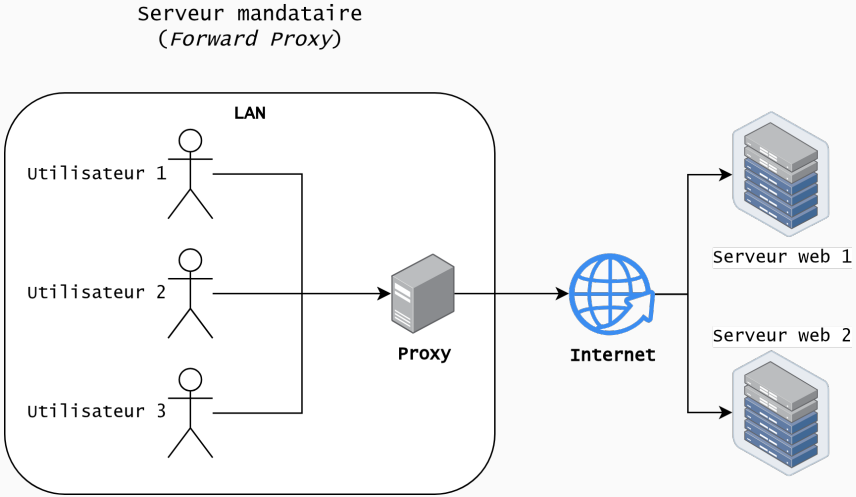
Vite : configuration dynamique côté client

```ts
// TypeScript triple-slash directive
// include type declarations from the package vite/client
/// <reference types="vite/client" />

const apiProtocol = import.meta.env.VITE_API_PROTOCOL || "http";
const wsProtocol = import.meta.env.VITE_WS_PROTOCOL || "ws";
const serverHost = import.meta.env.VITE_SERVER_HOST || "localhost";
const serverPort = import.meta.env.VITE_SERVER_PORT || "8000";

export const API_URL = `${apiProtocol}://${serverHost}:${serverPort}`;
export const WS_URL = `${wsProtocol}://${serverHost}:${serverPort}`;
```

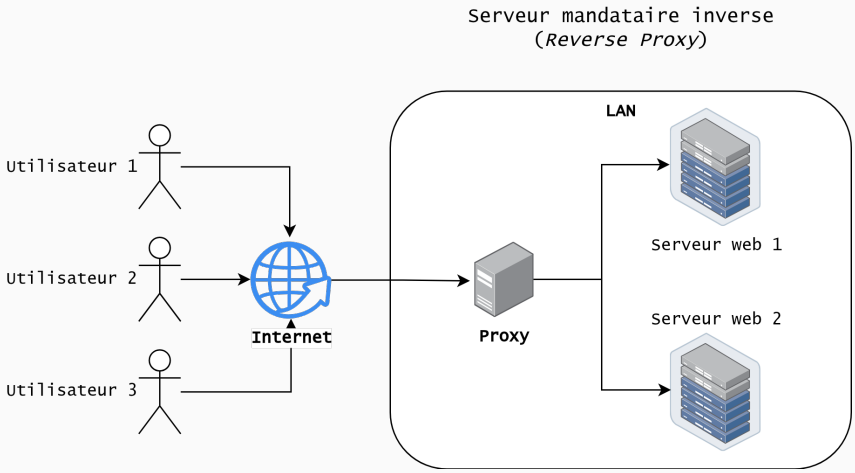
Vite : configuration dynamique côté client

```
# fichier .env à la racine du projet
VITE_API_PROTOCOL=https
VITE_WS_PROTOCOL=wss
VITE_SERVER_HOST=api.coucou.localhost
VITE_SERVER_PORT=443
```

Serveur mandataire – *Forward proxy*



Serveur mandataire inverse – *Reverse proxy*



Reverse proxy

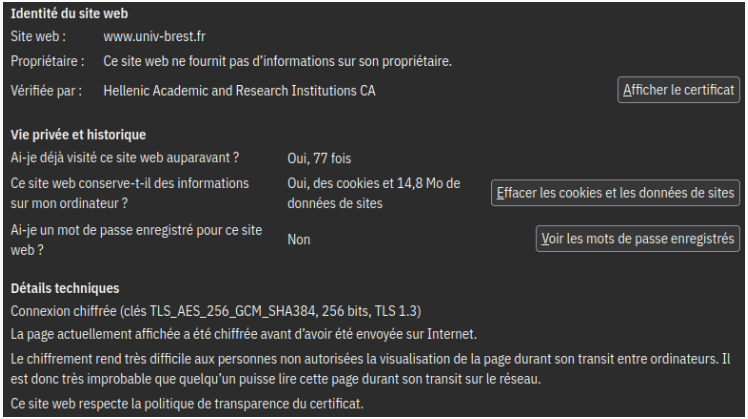
- **nginx** : créé par Igor Sysoev en 2004
- Premier serveur web au monde (plus de 33% des sites web)


```
http {
    server {
        listen 80;
        server_name api.coucou.localhost;

        location / {
            proxy_pass http://127.0.0.1:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

HTTPS : *Hypertext Transfer Protocol Secure*

À quoi ça sert ?

Vérifier l'**identité** d'un site web, grâce à un **certificat** d'authentification émis par une **autorité** de confiance, afin de **chiffrer** les communications



`mkcert` : créer un certificat

```
mkcert \
  -key-file univ-brest.fr.pem \
  -cert-file univ-brest.fr.pem \
  *.univ-brest.fr
```

Ah bon ?

Est-ce que ce certificat fait foi ?

Warning: Potential Security Risk Ahead

Firefox detected a potential security threat and did not continue to 192.168.1.5. If you visit this site, attackers could try to steal information like your passwords, emails, or credit card details.

Learn more…

Go Back (Recommended) Advanced…

192.168.1.5:40443 uses an invalid security certificate.

The certificate is not trusted because it is self-signed.

Error code: MOZILLA_PKIX_ERROR_SELF_SIGNED_CERT

View Certificate

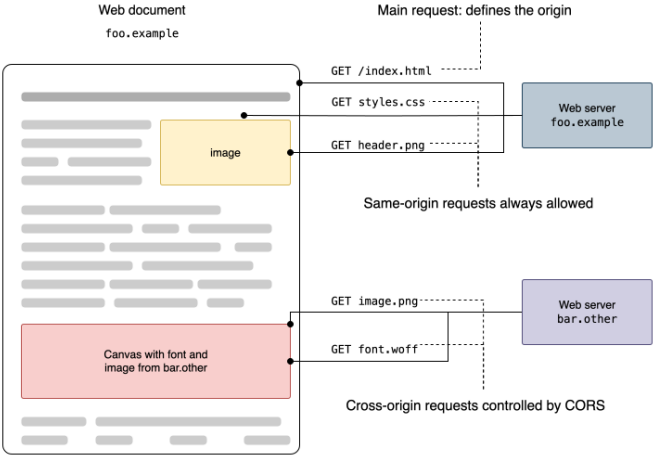
Go Back (Recommended) Accept the Risk and Continue

Reverse proxy avec SSL

```
http {
    server {
        listen 80;
        server_name api.coucou.localhost;
        return 301 https://$host$request_uri;
    }

    server {
        listen 443 ssl http2;
        server_name api.coucou.localhost;

        ssl_certificate     ./certs/api-coucou.pem;
        ssl_certificate_key ./certs/api-coucou-key.pem;

        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_prefer_server_ciphers on;

        location / {
            proxy_pass http://127.0.0.1:8000;
            ...
        }
    }
}
```

Cross-Origin Resource Sharing (CORS)

- `fetch` (ou `XMLHttpRequest`) : récupérer des ressources distantes lors de l'exécution des scripts de la page (au *runtime*)
- Pour des raisons de sécurité, les navigateurs imposent une politique stricte : *Same-Origin*
  - Origine = Protocole + Nom d'hôte + Port
- **Problème** : que se passe-t-il si le serveur n'est pas à la même adresse (origine) que le client ?
- Par défaut : la connexion est **refusée**



27. https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

Cross-Origin Resource Sharing (CORS)

- **CORS** : mécanisme basé sur les en-têtes HTTP
- Permet à un serveur de déterminer les **origines** autorisées pour les clients qui s'y connectent

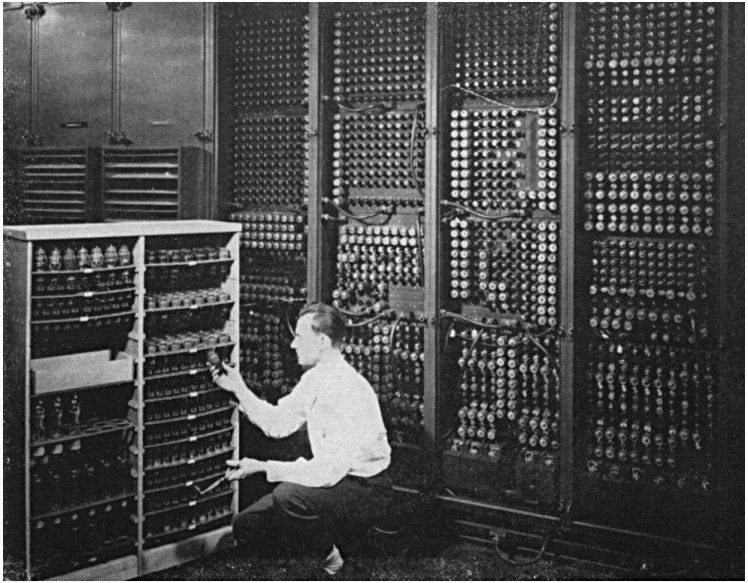
```
Access-Control-Allow-Origin: https://app.coucou.localhost
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Authorization, Content-Type
```

Configurer CORS au niveau du serveur

Configurer le *middleware* CORS :

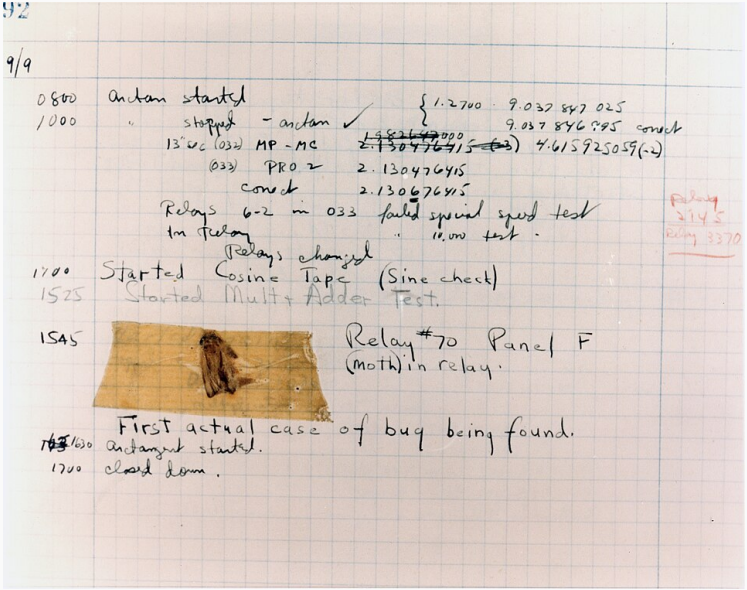
```
app.use(oakCors({
  origin: "https://app.coucou.localhost",
  credentials: true,
  methods: ["GET", "POST", "PUT", "DELETE", "OPTIONS"],
  allowedHeaders: ["Authorization", "Content-Type"],
}));
```

Débuggage et profilage



"U.S. Army Photo", from M. Weik, "The ENIAC Story" A technician changes a tube.

The operators affixed the moth to the computer log, with the entry : "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program".



28. https://en.wikipedia.org/wiki/Grace_Hopper

What could go wrong…

- Niveau client : connexion instable
- Niveau serveur : machine indisponible
- Niveau base de données : données corrompues
- Interactions API / BDD : transactions échouées
- Infrastructure : dépendance compromise
- Sécurité : clef privée publiée sur GitHub :-)
- Déploiement : certificats TLS expirés
- Gestion de l'état : cache périmé

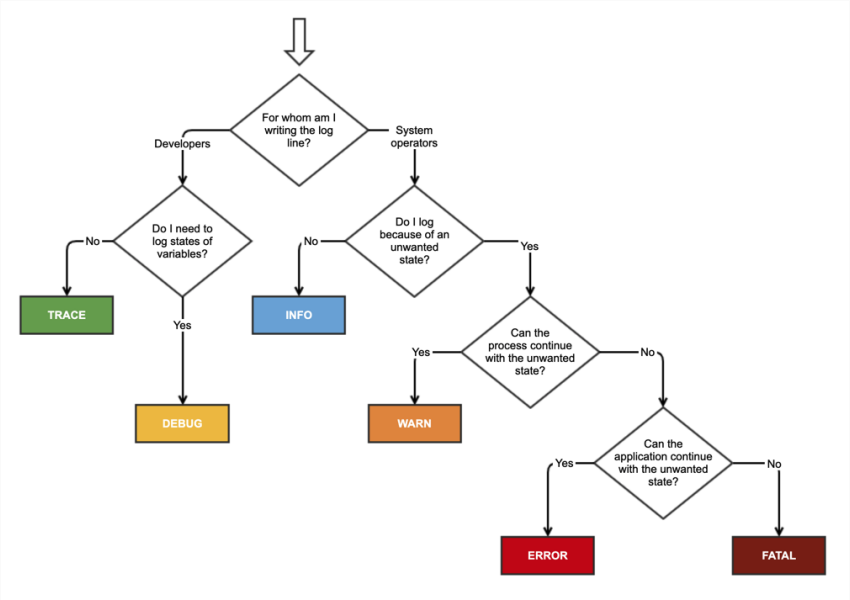
29. https://xkcd.com/1700/

Débuggage et profilage

Outillage

Logging

- *Log level*
  1. Trace : `console.trace`
  2. Debug : `console.debug`
  3. Info : `console.info`
  4. Warning : `console.warn`
  5. Error : `console.error`



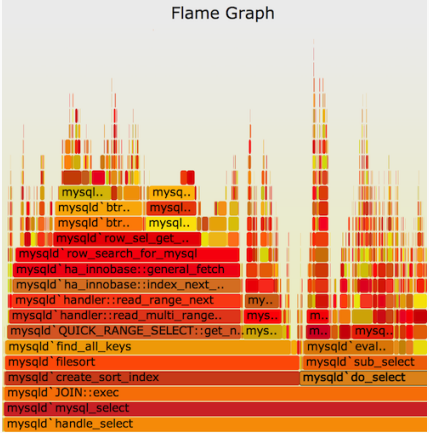
https://stackoverflow.com/a/64806781 (Taco Jan Osinga)

Débuggage et profilage

Profilage des performances

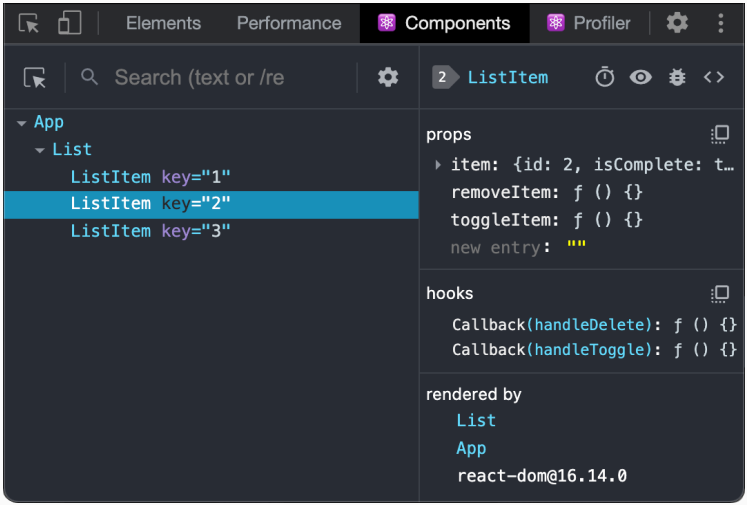
Côté serveur

- `deno run --v8-flags="--prof" main.ts`
  - Crée un fichier .log dans le répertoire courant
  - *"V8 has built-in sample-based profiling. Profiling is turned off by default, but can be enabled via the `--prof` command-line option. The sampler records stacks of both JavaScript and C/C++ code."*
- Lecture du fichier avec cpupro [30]
  - Visualisation : *flame graph* [31]



30. https://discoveryjs.github.io/cpupro/
31. [3] Brendan Gregg. « The Flame Graph ». *Communications of the ACM* (2016)

Côté client : *React Developer Tools*



https://react.dev/learn/react-developer-tools

Auto-évaluation

Examens : sujets abordés

- Pour le contrôle continu :
  - Développement d'une ou plusieurs fonctionnalité(s) supplémentaire(s)
  - Modifications côté serveur et côté client
  - Validées par un scénario de test et/ou des tests unitaires
- Pour l'examen final :
  1. Modélisation et typage (TypeScript)
  2. Programmation asynchrone (Promise)
  3. Développement côté serveur (Oak)
  4. Développement côté client (React)
  5. Communications (WebSocket) et authentification (JWT)
  6. Déploiement (nginx)

Auto-évaluation

Modélisation et typage (TypeScript)

Test de connaissances

Question

1. À quoi sert la dernière ligne de cette définition ?
2. Comment appelle-t-on une telle propriété ?

```
export interface PollOptionRow {
  id: string;
  poll_id: string;
  text: string;
  vote_count: number;
  [key: string]: SQLOutputValue;
}
```

Test de connaissances

Question

1. Comment appelle-t-on une fonction telle que définie ci-dessous ?
2. Dans quelles situations a-t-on besoin d'une telle fonction ?

```
export function isPollOptionRow(
  obj: Record<string, SQLOutputValue>,
): obj is PollOptionRow;
```

Test de connaissances

Question

1. Comment appelle-t-on le type donné ci-dessous ?
2. Comment s'en sert-on et pourquoi ?

```
export type RegisterRequest = Omit<
  User, "id" | "isAdmin" | "createdAt"
> & { password: string };
```

Auto-évaluation

Programmation asynchrone (Promise)

Test de connaissances

Question

1. Quel est le type de retour d'une fonction qui retourne une valeur après un délai (en millisecondes) ?
2. Implanter une fonction qui résout ou rejette selon que l'entrée est respectivement positive ou négative.

```
function delayedEcho(value: number, ms: number): Promise<number> {
  // À compléter
}

// Test succès
delayedEcho(10, 300)
  .then() // ... À compléter
  .catch() // ... À compléter
// Test erreur
delayedEcho(-5, 300) // ... À compléter
  .then() // ... À compléter
  .catch() // ... À compléter
```

Test de connaissances

Question

1. Quelles sont les différences entre deux fonctions qui retournent une Promise lorsque l'une est déclarée `async` et l'autre non ?
2. Utiliser `await` pour consommer la Promise dans `echoDouble` et retourner le double de la valeur initiale.

```
function delayedEcho(value: number, ms: number): Promise<number>;

async function echoDouble(value: number): Promise<number> {
  // À compléter
}

// Test
echoDouble(10).then(result => {
  console.log(result); // 20
});
```

Test de connaissances

Question

1. Quelle est la différence entre **concurrence** et **parallélisme** ?
2. Compléter la fonction ci-dessous pour faire la somme du résultat de deux appels à `delayedEcho` en 500 ms (plutôt que 1000 ms).

```
function delayedEcho(value: number, ms: number): Promise<number>;

async function sumParallel(a: number, b: number): Promise<number> {
  // À compléter (appels à delayedEcho avec son paramètre ms = 500)
}

// Test
sumParallel(10, 20).then(result => {
  console.log(result); // retourne `30` en ~500 ms
});
```

Auto-évaluation

Développement côté serveur (Oak)

Test de connaissances

Question

1. Quelles sont les méthodes HTTP utilisées dans le cadre d'une API REST ?
2. Écrire un serveur Oak qui répond à `GET /hello/:name` avec `Hello, {name}`.

```
import { Application, Router } from "@oak/oak";
const router = new Router();
const app = new Application();
// À compléter
```

Test de connaissances

Question

1. Quel code d'erreur HTTP doit-on retourner au client lorsqu'une ressource est introuvable ?
2. Reprendre le code de la question précédente. Retourner une erreur lorsque le paramètre `name` vaut "Toto".

```
import { Application, Router } from "@oak/oak";
const router = new Router();
const app = new Application();
// À compléter
```

Test de connaissances

Question

1. Qu'est-ce qu'un *middleware* ? Quels sont ses paramètres ? Comment chaîne-t-on des *middlewares* ?
2. Ajouter un middleware à la route créée précédemment. Il doit logguer la méthode et l'URL de la requête émise par le client avant de traiter la requête.

```
export async function logMiddleware(ctx, next); // À compléter
```

Auto-évaluation

Développement côté client (React)

Test de connaissances

Question

1. Qu'est-ce que l'**état** d'un composant ?
2. Utiliser le *hook* `useState` pour stocker un compteur dans l'état du composant, et incrémenter le compteur au clic.

```
import { useState } from "react";
export default function Counter() {
  // À compléter : déclarer un état count initialisé à 0
  return (
    <div>
      <p>Compteur : {/* afficher count */}</p>
      <button onClick={/* incrémenter count */}>
        Incrémenter
      </button>
    </div>
  );
}
```

Test de connaissances

Question

1. Qu'est-ce qu'un **effet de bord** ?
2. Utiliser le *hook* `useEffect` et l'API `fetch` pour récupérer des données depuis le serveur (`http://localhost:8000/hello/Alice`).

```tsx
import { useEffect, useState } from "react";
type APIResponse = { message: string; };
export default function FetchMessage() {
  const [data, setData] = useState<APIResponse | null>(null);
  useEffect(() => {
    // À compléter : fetch + setData
  }, []);

  return (
    <div>
      {data ? <p>{data.message}</p> : <p>Chargement...</p>}
    </div>
  );
}
```

Test de connaissances

Question

1. Dans l'API `fetch`, quel moyen est mis à notre disposition pour traiter une erreur lors de la récupération de données distantes ?
2. Complétez le composant précédent pour traiter une requête vers `http://localhost:8000/hello/Toto` (qui retourne une erreur, donc).

```
export default function FetchMessage() {
  const [data, setData] = useState<APIResponse | null>(null);
  // À compléter : ajouter un état `error`
  useEffect(() => {
    // À compléter : fetch, if -> throw, catch -> setError
  }, []);

  if (/* erreur présente */) {
    return <p>Error: {/* message */}</p>;
  }
}
```

Auto-évaluation

Communications (WebSocket) et authentification (JWT)

Test de connaissances

Question

1. Qu'est-ce que signifie *full-duplex* dans le contexte d'un canal de communications ?
2. Écrire une route Oak qui ouvre un WebSocket avec le client, lui dit "Bonjour" et "Au revoir", et lui renvoie ses propres messages.

```
router.get("/ws", (ctx) => {
  const ws: WebSocket = ...; // À compléter
  ws.onopen = () => { ... }; // À compléter
  ws.onmessage = () => { ... }; // À compléter
  ws.onclose = () => { ... }; // À compléter
});
```

Test de connaissances

Question

1. Quels sont les trois champs d'un *JSON Web Token* ? Quelle dynamique existe-t-il entre eux ?
2. Compléter la fonction `verifyJWT` qui vérifie la validité d'un JWT et retourne un objet `AuthPayload` en cas de succès.

```
interface AuthPayload {
  username: string;
  exp: number;
}
export async function verifyJWT(token: string): Promise<AuthPayload | null> {
  // À compléter...
}
```

Test de connaissances

Question

1. Une fois le jeton récupéré côté client, comment le fait-on valoir auprès du serveur pour authentifier nos requêtes ?
2. Compléter la fonction suivante, qui prend en entrée un jeton et une requête existante pour lui ajouter les en-têtes nécessaires à l'authentification :

```
const authFetch = async (token: string, input: RequestInfo) => {
  const res = await fetch(
    ... // À compléter
  );
};
```

Auto-évaluation

Déploiement (TLS et nginx)

Test de connaissances

Question

1. Expliquer le code ci-dessous.
2. Les fichiers `.pem` ont été générés localement par `mkcert`. Que va-t-il se passer lors de la connexion d'un client ? Pourquoi ?

```
const listener = Deno.listenTls({
  port: 443,
  hostname: "coucou.localhost",
  cert: await Deno.readTextFile("coucou.localhost.pem"),
  key: await Deno.readTextFile("coucou.localhost-key.pem"),
});
```

Bibliographie

Références i

[1] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[2] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2012.

[3] Brendan Gregg. « The Flame Graph ». *Communications of the ACM* 59.6 (mai 2016), p. 48-57. ISSN : 0001-0782, 1557-7317. DOI : 10.1145/2909476.

[4] Alexis King. *Parse, Don't Validate*. Nov. 2019. URL : https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/ (visité le 27/01/2026).

[5] Martin Kleppmann. *Designing Data-Intensive Applications : The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. 1st. Sebastopol, CA : O'Reilly Media, 2017, p. xix + 590. ISBN : 978-1449373320.

[6] Vincent Lannurien et al. « Serverless Cloud Computing : State of the Art and Challenges ». *Serverless Computing : Principles and Paradigms*. Sous la dir. de Rajalakshmi Krishnamurthi et al. Cham : Springer International Publishing, 2023, p. 275-316. ISBN : 978-3-031-26633-1. DOI : 10.1007/978-3-031-26633-1_11.

[7] James Lewis et Martin Fowler. *Microservices*. URL : https://martinfowler.com/articles/microservices.html (visité le 19/01/2026).

[8] Microsoft. *The TypeScript Handbook*. 2026. URL : https://www.typescriptlang.org/docs/handbook/intro.html (visité le 19/01/2026).

[9] Simone Poggiali. *The Concise TypeScript Book*. 2026. URL : https://gibbok.github.io/typescript-book/book/the-concise-typescript-book/ (visité le 19/01/2026).

[10] Yehonathan Sharvit. *Data-Oriented Programming : Reduce Software Complexity*. Shelter Island, NY : Manning Publications, 2022, p. 325. ISBN : 978-1617298578.