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

https://info-demo-sor.univ-brest.fr

Plan du cours

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

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 pour 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)

Introduction

Contenu du cours

- État et gestion de l'état
  - Contrats
    - Interfaces
    - *Data Transfer Objects*
  - Persistance
    - Base de données
    - Transactions
- Protocoles
  - HTTP
    - *Status codes*
    - *CORS* (*Cross-Origin Resource Sharing*)
  - Authentification
    - *Tokens*
- Performances et profilage
  - Instrumentation (serveur)
  - Inspecteur (client)
  - Injection de trafic
- Déploiement
  - SSL, certificats
  - *Reverse proxy*

Technologies utilisées

- Côté serveur :
  - Runtime (Deno)
  - Langage (TypeScript)
  - Framework (Oak)
  - Base de données (SQLite)
- Côté client :
  - Bundler (Vite)
  - Langage (TypeScript)
  - Framework (React)
- Communications :
  - Requêtes HTTP
  - WebSockets

Cas d’usage

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

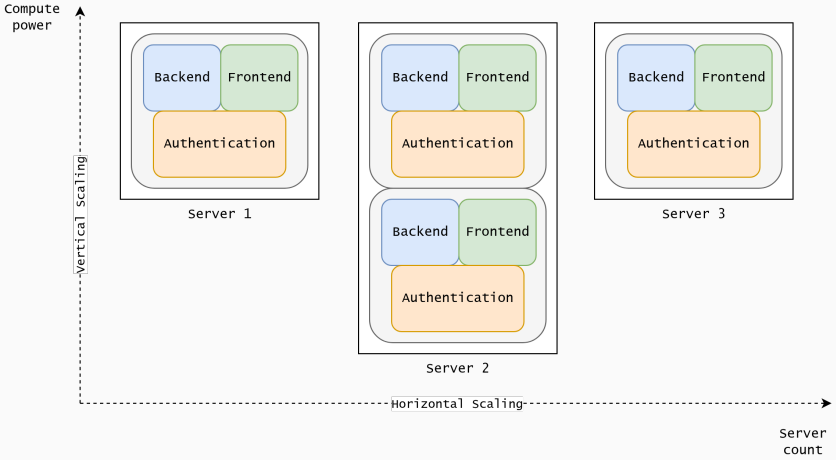
Les principales fonctionnalités de l'application peuvent être résumées ainsi :

- 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).

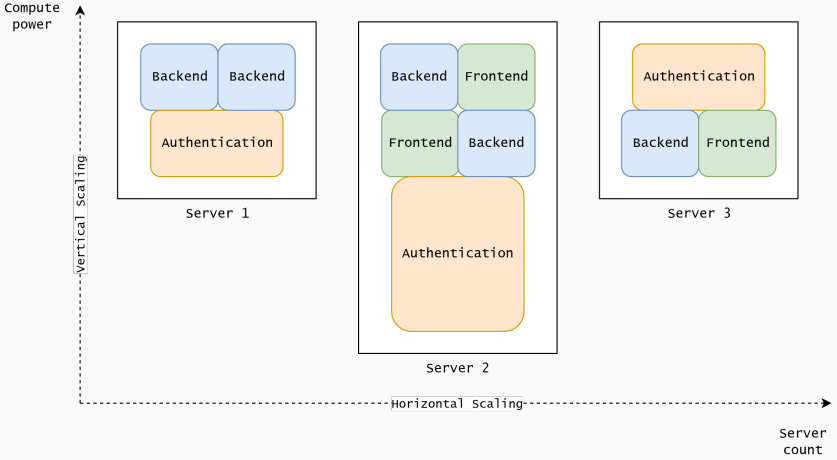
Architecture : modèle client/serveur

Vue d’ensemble

Monolithes ou microservices



Monolithes ou microservices



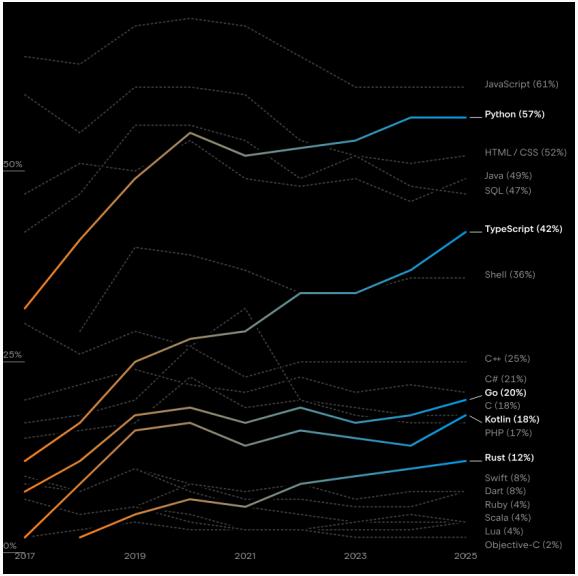
JavaScript, TypeScript et runtime

JavaScript

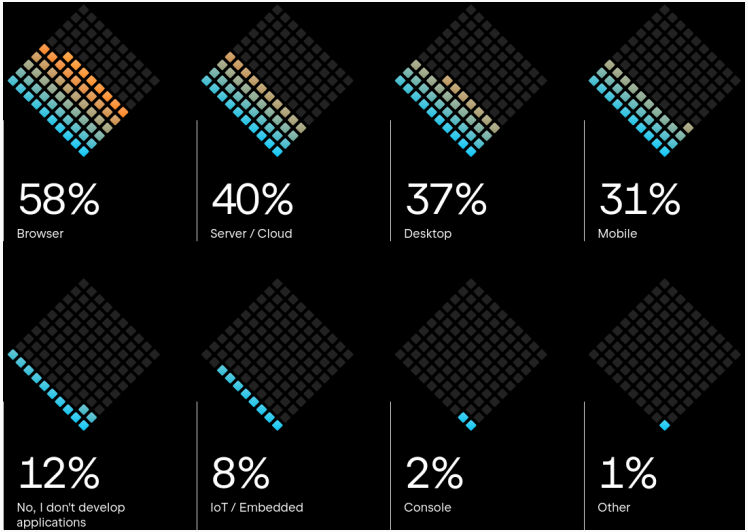
```
1 <html>
2 <head><title>Alert Page</title></head>
3 <Body>
4     <script language="javascript">
5         alert("Welcome to javaskool.com");
6     </script>
7 </body>
8 <html>
```

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.”*



. 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[1] : migration du frontend React vers TS pour sécuriser les appels d'API et réduire les erreurs au runtime
- Airbnb[2] : typage progressif du code JS pour réduire les coûts de maintenance
- Microsoft[3] : migration vers TS pour faciliter l'évolution de VS Code ($>$ 1M SLoC)
- Shopify[4] : migration frontend et backend pour fiabiliser les fonctions critiques

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

TypeScript

- Ressources :
  - *The TypeScript Handbook*[5]
  - *The Concise TypeScript Book*[6]
  - TypeScript TV[7] (liste des codes d'erreur)
  - TS Playground[8] (environnement d'exécution en ligne)

5. [2] Microsoft. *The TypeScript Handbook*. 2026
6. [3] Simone Poggiali. *The Concise TypeScript Book*. 2026
7. `https://typescript.tv/`
8. `https://www.typescriptlang.org/play/`

TypeScript : typage du code JavaScript

Types de base disponibles dans TypeScript[9] :

- `boolean`
- `number`
- `string`
- Tableaux (dynamiques) – exemple : `string[]` ou `Array<string>`
- Tuples (taille fixe) – exemple : `let x: [string, number]`
- `enum`
- `unknown` (type à préciser en le vérifiant à l'exécution)
- `any` (à éviter : toute valeur autorisée)
- `void` (absence de valeur)
- `null` (valeur vide) et `undefined` (valeur non définie)

9. 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);
```

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"; // Tagged union
  bark: () => void;
}

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

type Animal = Dog | Cat; // Union type

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();
};
```

Généricité

- `SomeContainer<T extends SomeData>`

Exercice 3 : types au *runtime*

```ts
interface Dog {
  kind: "dog"; // Tagged union
  bark: () => void;
}
interface Cat {
  kind: "cat"; // Tagged union
  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);
```

Gestion des erreurs : exceptions

- Exceptions
- `try/catch`

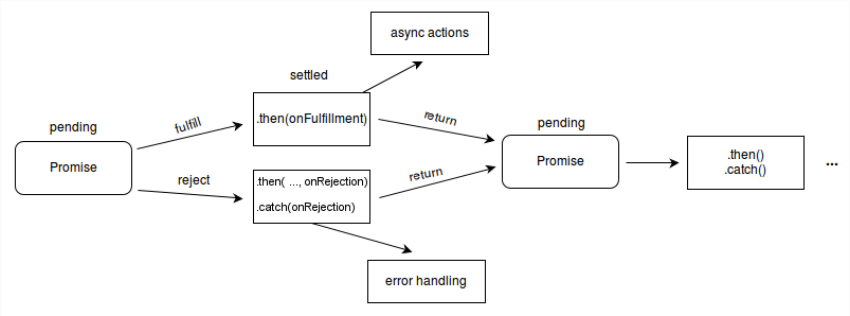
Programmation asynchrone

TODO : Exemple (requête API) avec figure
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 ?

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)



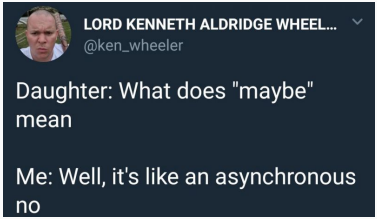
. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

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`
- `Promise.allSettled`

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

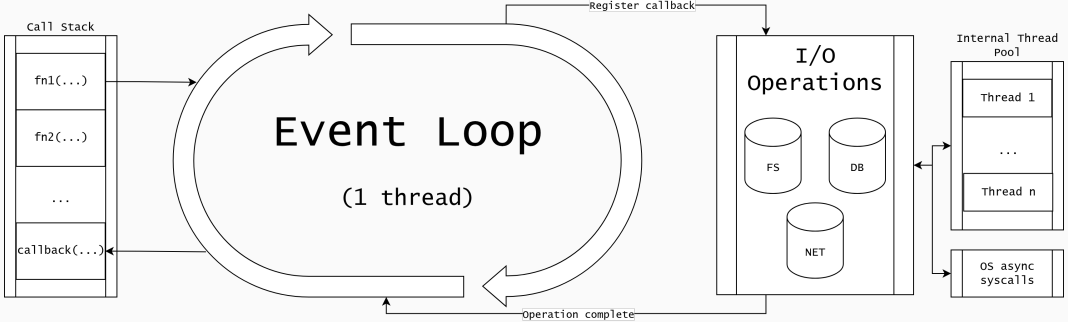
```
const files = ["does_not_exist.txt", "file2.txt", "file3.txt"];

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 = ?

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 [10])



10. 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 [11]
- 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

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

Sécurité, permissions

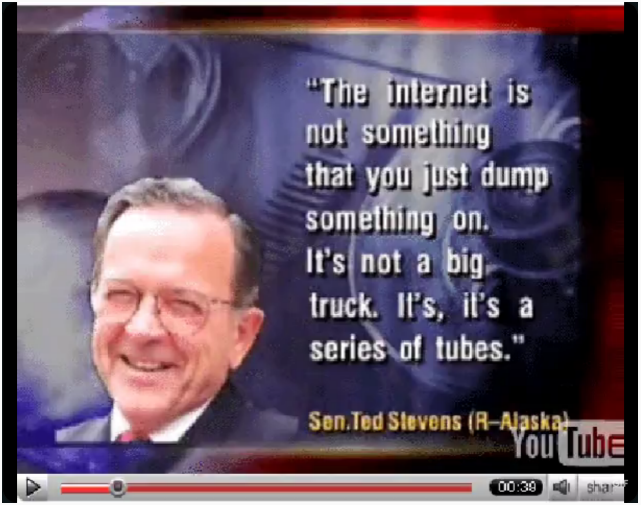
- *Supply-chain attack* : introduction de vulnérabilités dans un logiciel *via* compromission d'une ou plusieurs de ses dépendances
- Exemple : *Shai Hulud*, novembre 2025[12], 700 paquets compromis sur NPM
  - récupération de différents secrets présents sur la machine compromise […] ;
  - exfiltration des secrets ;
  - réplication par l'infection de paquets NPM ;
  - suppression de données utilisateurs ;
  - mise en place de persistance.

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

Gestion des dépendances

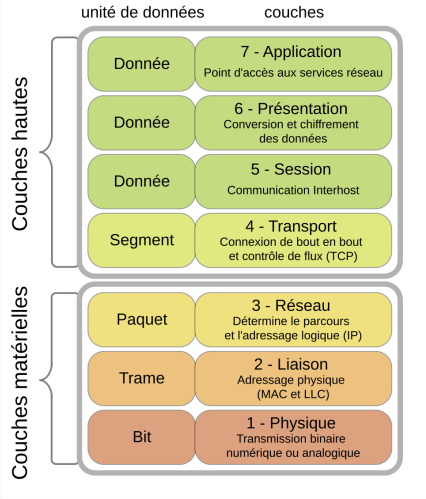
Outils (npm, etc.)
Modules (ESM vs CommonJS) et imports

Serveur web


"The internet is not something that you just dump something on. It's not a big truck. It's, it's a series of tubes."
Sen.Ted Stevens (R–Alaska)
YouTube
00:39

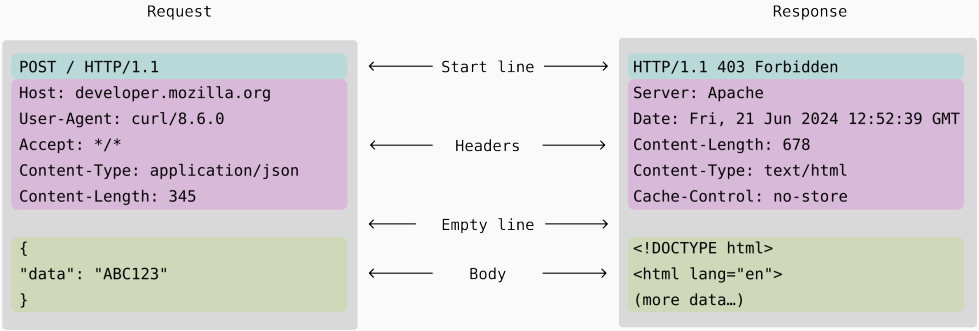
HTTP

- *Hypertext Transfer Protocol* : à l'origine, sert à partager des pages web reliées entre elles par des liens
- Inventé au CERN[13] 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



[13] 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[14], 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

14. 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`

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"  \
        --data "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. TODO : new Date().toUTCString();

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`
  - Route : `/item/:itemId`
  - Fonction : `getItem(id)`
  - Réponse : `{ id: 123, value: "..." }`

JSON

Middleware

Middleware : CORS

TODO : à mettre côté client ?

Middleware : Gestion des erreurs

Base de données

- SQLite
- Interactions avec SQLite : bibliothèque
- Conversion des **enregistrements** vers des **objets**

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

```
// Persist 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);

// Increment tally
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();     // Start transaction

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

  db.prepare("COMMIT").run();    // Commit transaction
} catch {
  db.prepare("ROLLBACK").run(); // Rollback on any error
}
```

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);
    },
});
```

Client web

HTTP, encore

- API `fetch`

Un client web minimal avec TypeScript

```
const foo = "bar";
```

Framework : React

- Hooks
  - `useParams`
  - `useState`
  - `useRef`
  - `useEffect`
- Functional Components
- React Router

Bundling

- Bundler
- Vite

React : architecture

- Service Layer Pattern

Bundler

Authentification

Concepts généraux

Stateful : Sessions

Stateless : JWT

Interactions

REST

Data Transfer Objects

WebSockets

Déploiement de l’application

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

Reverse proxy

```
http {
    server {
        listen 80;
        server_name 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;
        }
    }
}
```

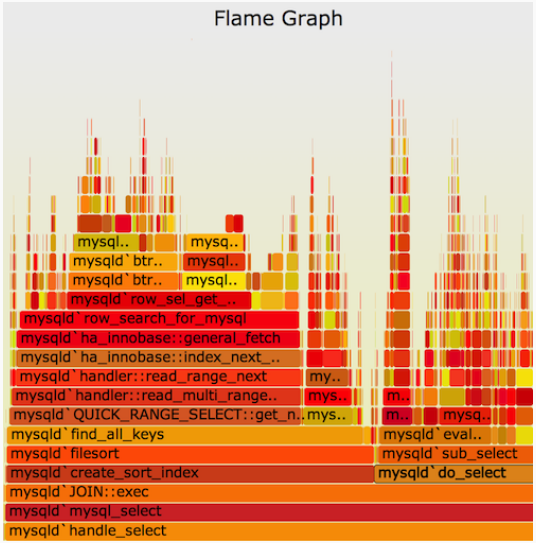
Certificats SSL

- Protocole (TLSv1.2, TLSv1.3)
- Algorithme de chiffrement : AES 128/256, etc.

Débuggage et profilage

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 [15]
  - Visualisation : *flame graph* [16]



[15] https://discoveryjs.github.io/cpupro/

[16] [1] Brendan Gregg. « The Flame Graph ». *Communications of the ACM* (2016)

Côté client

Bibliographie

Références i

[1] 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. (Visité le 16/01/2026).

[2] Microsoft. *The TypeScript Handbook*. 2026.

[3] Simone Poggiali. *The Concise TypeScript Book*. 2026.