ANF Visualisation et Données

-

Architecture Web et Calculs

Caroline Ramond
Le 02/12/2016

A installer

- Node
- Python (2.7 de préférence)
- Librairie socketIO-client-2 de python
- git

Objectifs

- **Utiliser une interface web pour paramétrer et lancer des calculs Python.**
- **Visualiser les résultats / re-paramétrer le calcul en temps réel.**

- **Pourquoi une interface web ?**
  - Facilité de programmation
  - Portabilité
  - Architecture « nativement » adaptée au temps réel
  - Permet de décorréler le calcul et l'affichage (gain de performances)

- Exemple : http://localhost:3100/map

Plan de l'exposé

1. **Sites web "classiques »**
   Exemple de base : formulaire
   Javascript/DOM
   Autre exemple : annuaire

2. **Applications temps réel**
   AJAX
   Websockets
   Exemple : interface web - calculateur

3. **Mise en œuvre**
   Côté serveur : NodeJS
   Côté client : AngularJS
   Communication : SocketIO
   Calculateur Python

4. **TP**

Sites web "classiques"

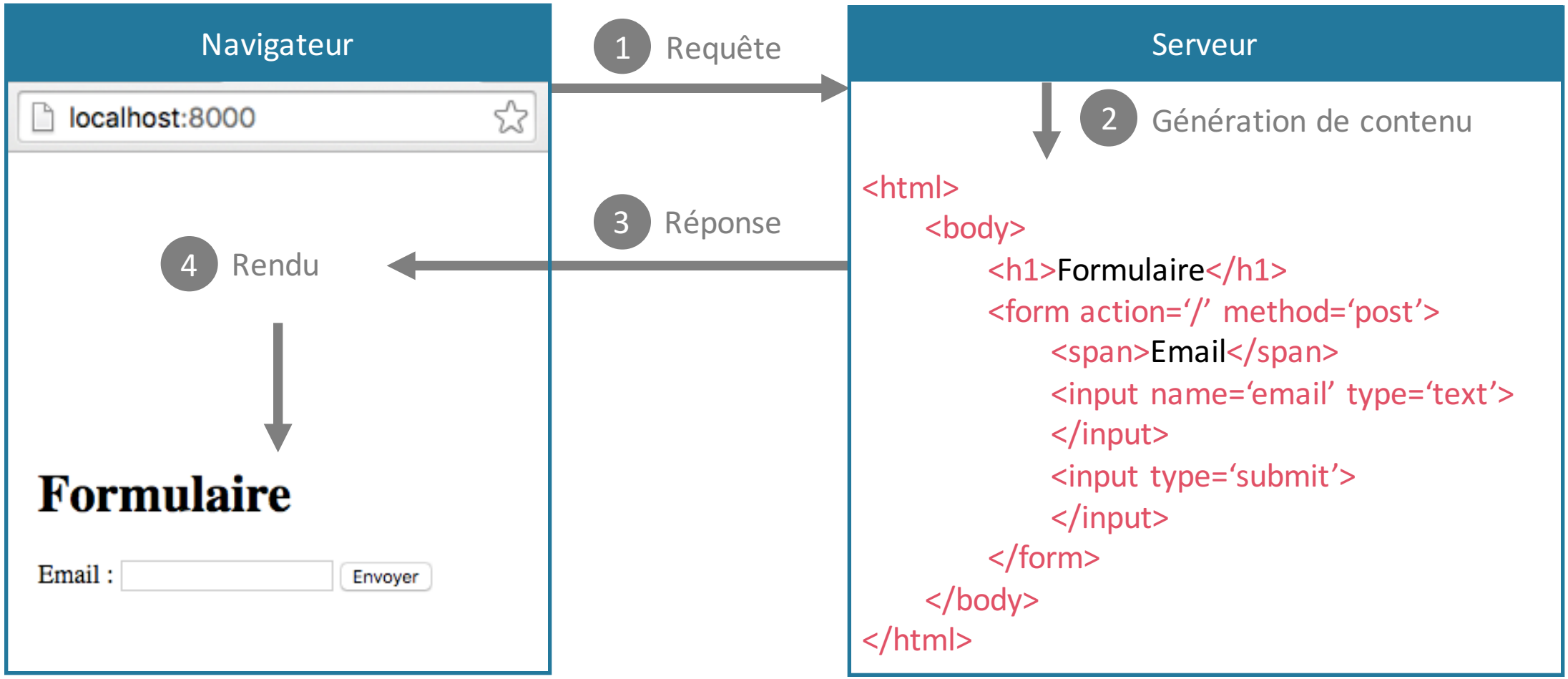
1.1. Exemple de base : Formulaire

1.2. Javascript/DOM

1.3. Autre exemple : Annuaire

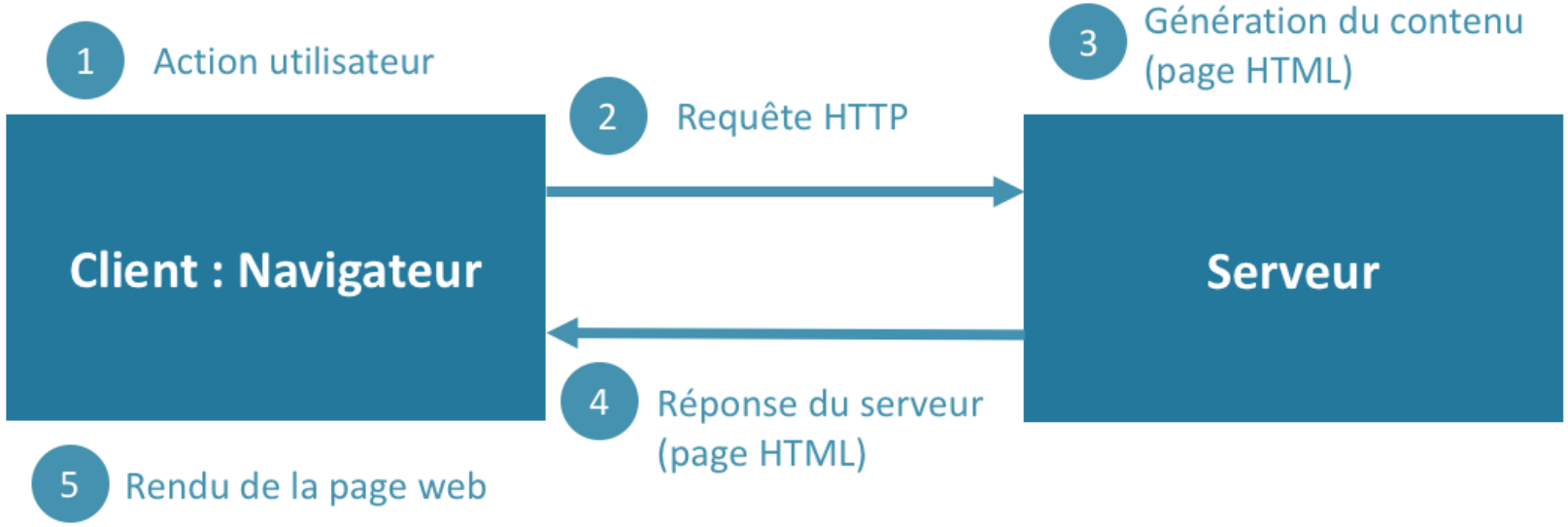
Sites web "classiques"

Exemple de base : Formulaire



Sites web "classiques"

Architecture de base



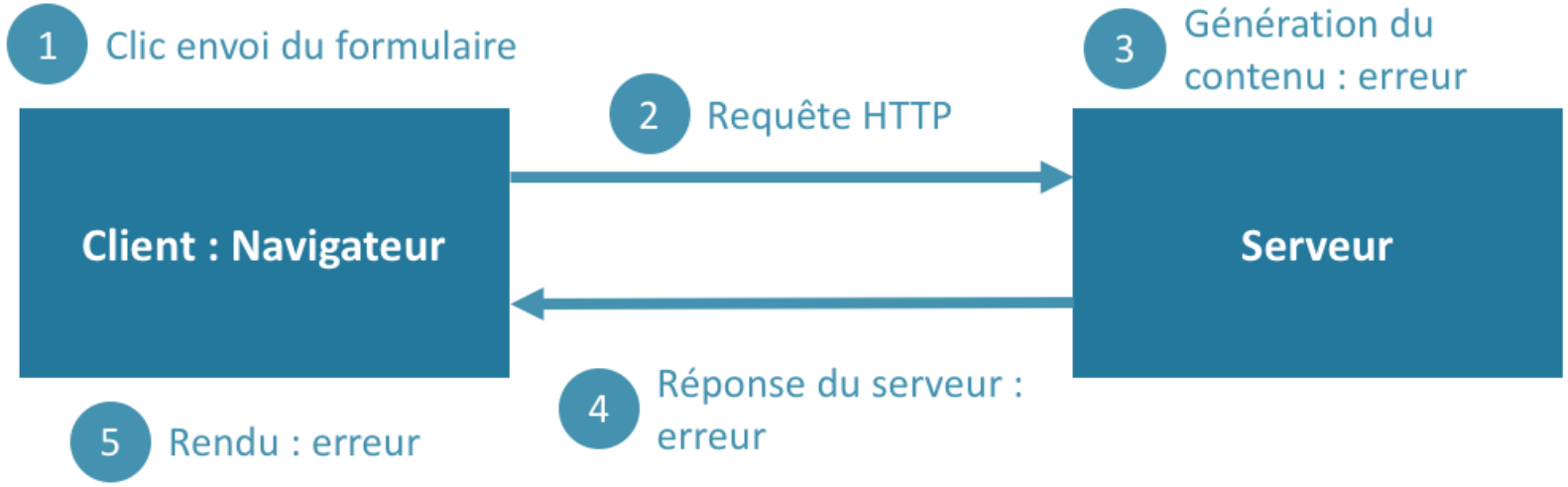
- L'intégralité des traitements se fait côté serveur.
- Contenu statique.

Sites web "classiques"

Inconvénients

Soumission d'un formulaire avec faute de frappe : localhost:8000/v1

- La faute n'est pas détectée au niveau du navigateur mais côté serveur
- L'utilisateur doit attendre la fin de la requête pour corriger son erreur
- Latence



Sites web "classiques"

1.1. Exemple de base : Formulaire

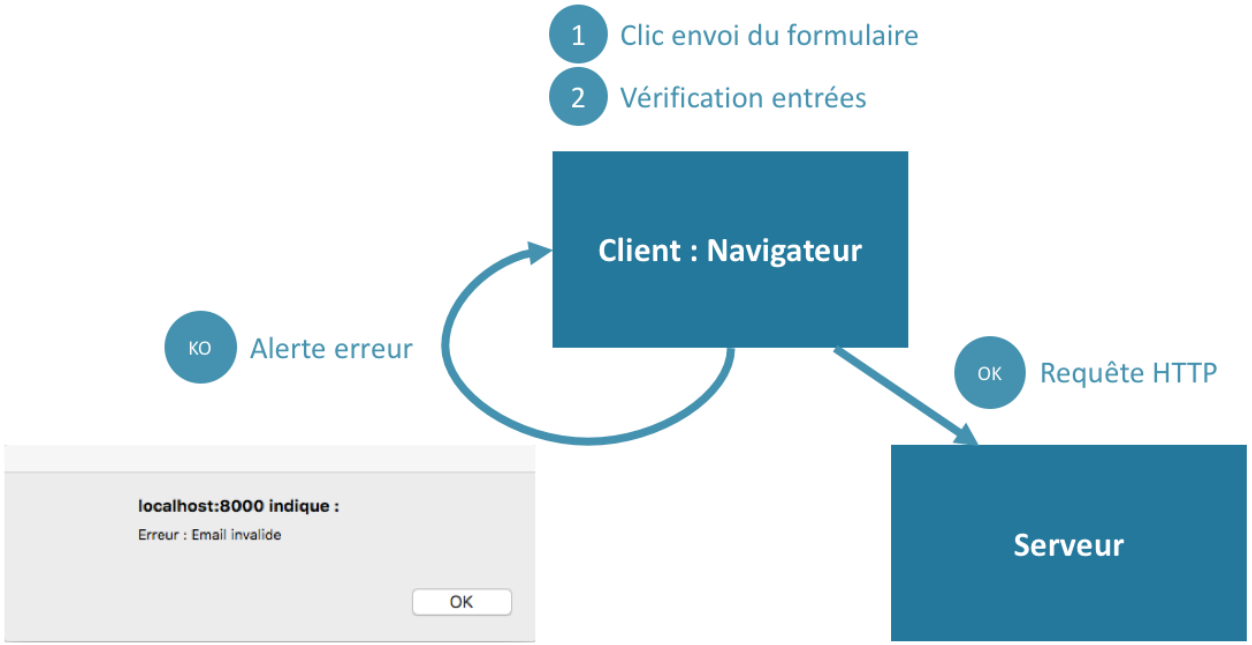
1.2. Javascript/DOM

1.3. Autre exemple : Annuaire

Sites web "classiques"

Javascript (1995)

- Langage de script interprété nativement par le navigateur.
- Les scripts sont envoyés par le serveur avec le contenu HTML.



Résultat

Sites web "classiques"

DOM (1998)

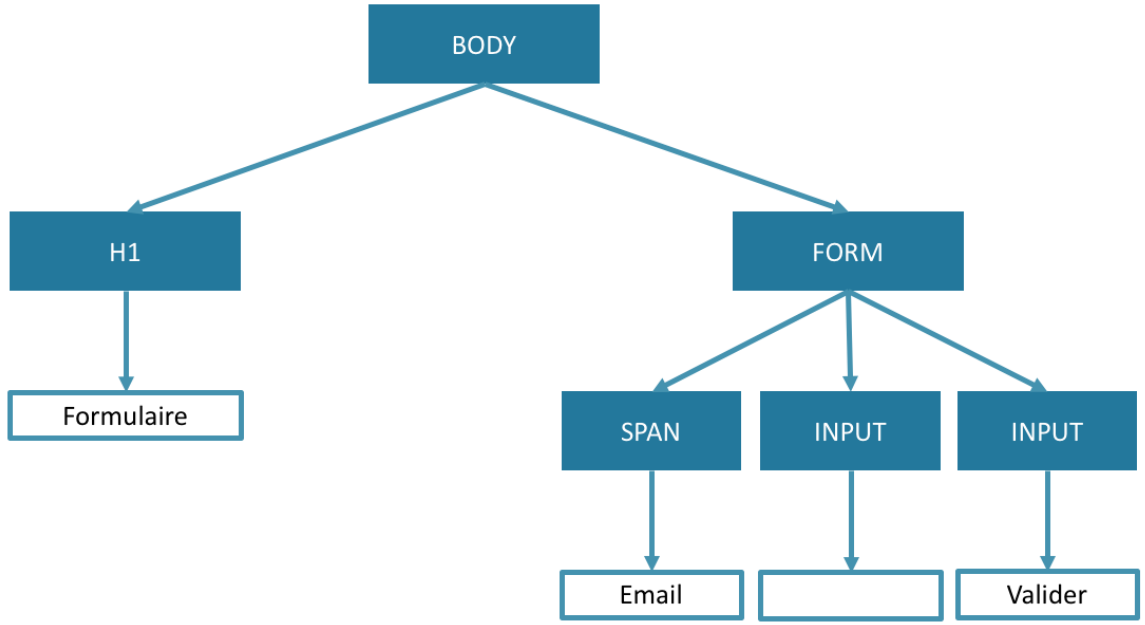
- DOM : Document Object Model

- Objet représentant le contenu d'une page web
  - Représentation du document sous forme d'arbre
  - Méthodes de parcours de l'arbre
  - Méthodes de recherche dans l'arbre
  - Méthodes d'ajout/retrait dans l'arbre
  - Détection d'évènements liés à la fenêtre, à la souris ou au clavier ('onresize', 'onload', 'onclick', 'onmousemove', 'onkeypress', etc)

Sites web "classiques"

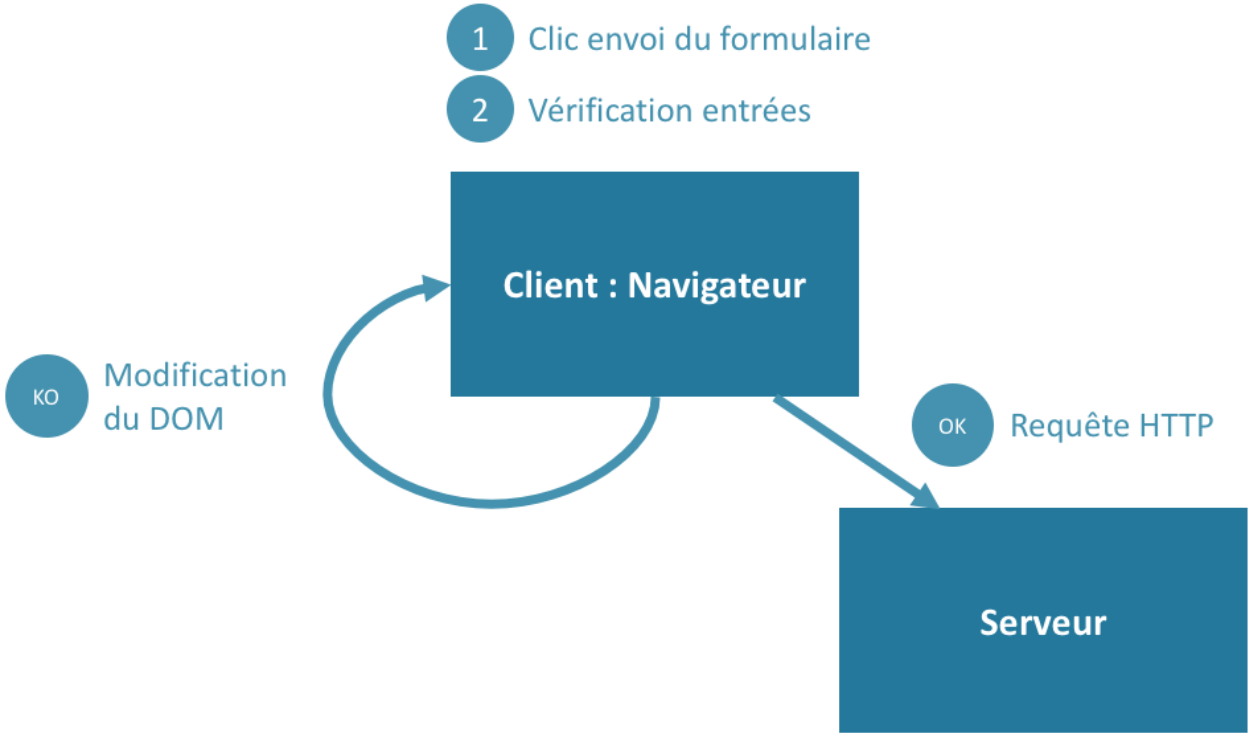
DOM (1998)

```
<html>
    <body>
        <h1>Formulaire</h1>
        <form action='/' method='post'>
            <span>Email</span>
            <input name='email' type='text'>
            </input>
            <input type='submit'>
            </input>
        </form>
    </body>
</html>
```



Sites web "classiques"

DOM (1998)



Résultat

Sites web "classiques"

1.1. Exemple de base : Formulaire

1.2. Javascript/DOM

1.3. Autre exemple : Annuaire

Sites web "classiques"

Exemple : Annuaire

https://www.math.u-psud.fr/-Les-membres-

- Contenu « statique » : bandeau titre, logos, menus
- Pourtant, rechargement et rendu de l'**intégralité** de la page à chaque filtrage

- Transmission de l'intégralité de l'annuaire puis appel serveur pour filtrage à chaque clic
- Le navigateur peut effectuer le filtrage lui-même une fois les données récupérées

Sites web "classiques"

Récapitulatif : Inconvénients

- Dans un site web « classique », toutes les prises de décision se font au niveau du serveur.
  - Contenu statique
  - Beaucoup de redirections
  - Utilisateur bloqué (appels synchrones)

- Dans un site web « classique », le serveur renvoie des pages HTML complètes.
  - Rendu intégral à chaque appel serveur
  - Redondance des données

→ **Latence incompatible avec le temps réel**

Applications temps réel

2.1. AJAX

2.2. Websockets

2.3. Exemple : interface web - calculateur

Applications temps réel

Architecture AJAX

Asynchronous Javascript And XML

- Pas de rechargement de la page : toutes les données d'affichage sont transmises une et une seule fois
- Intelligence de l'affichage ramenée au navigateur : affichage dynamique (Javascript)
- Appels serveurs limités à l'alimentation en données : plus légers (XML) et asynchrones

- Applications plus réactives
- Asynchronisme : compatibilité temps réel
- « Separation of concerns » : serveur = données, client = affichage

Applications temps réel

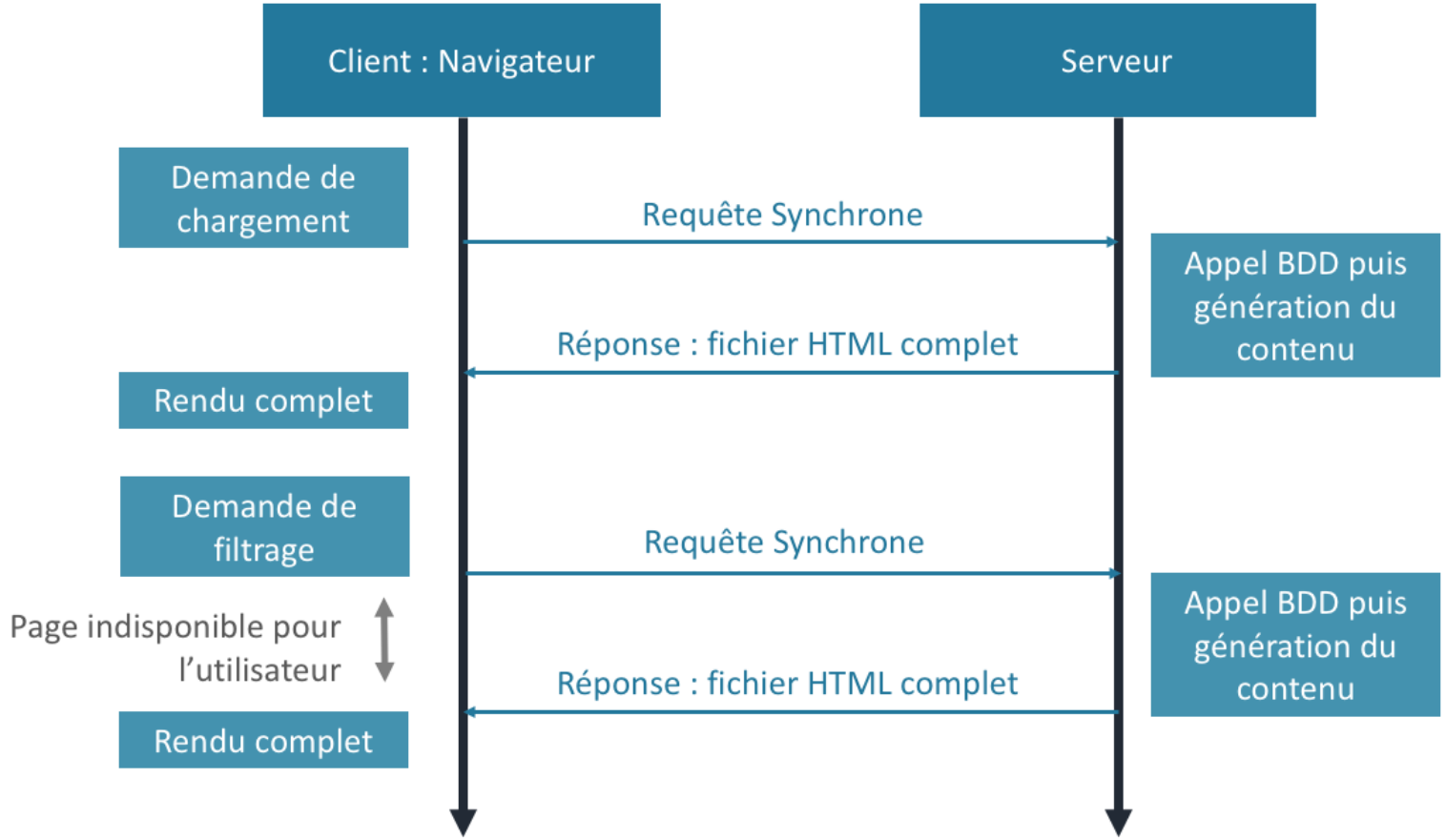
Architecture AJAX

L'architecture AJAX repose sur :

- **XMLHttpRequest** : dialogue asynchrone avec le serveur
- **XML** (Extensible Markup Language) : Format de données retourné par le serveur (analogue au HTML, sans informations de mise en page)
- **JSON** (JavaScript Object Notation): autre format de données retourné par le serveur, adapté à la syntaxe Javascript
- **Javascript**, **DOM** : modifications dynamiques du contenu
- **CSS** : mise en forme du contenu

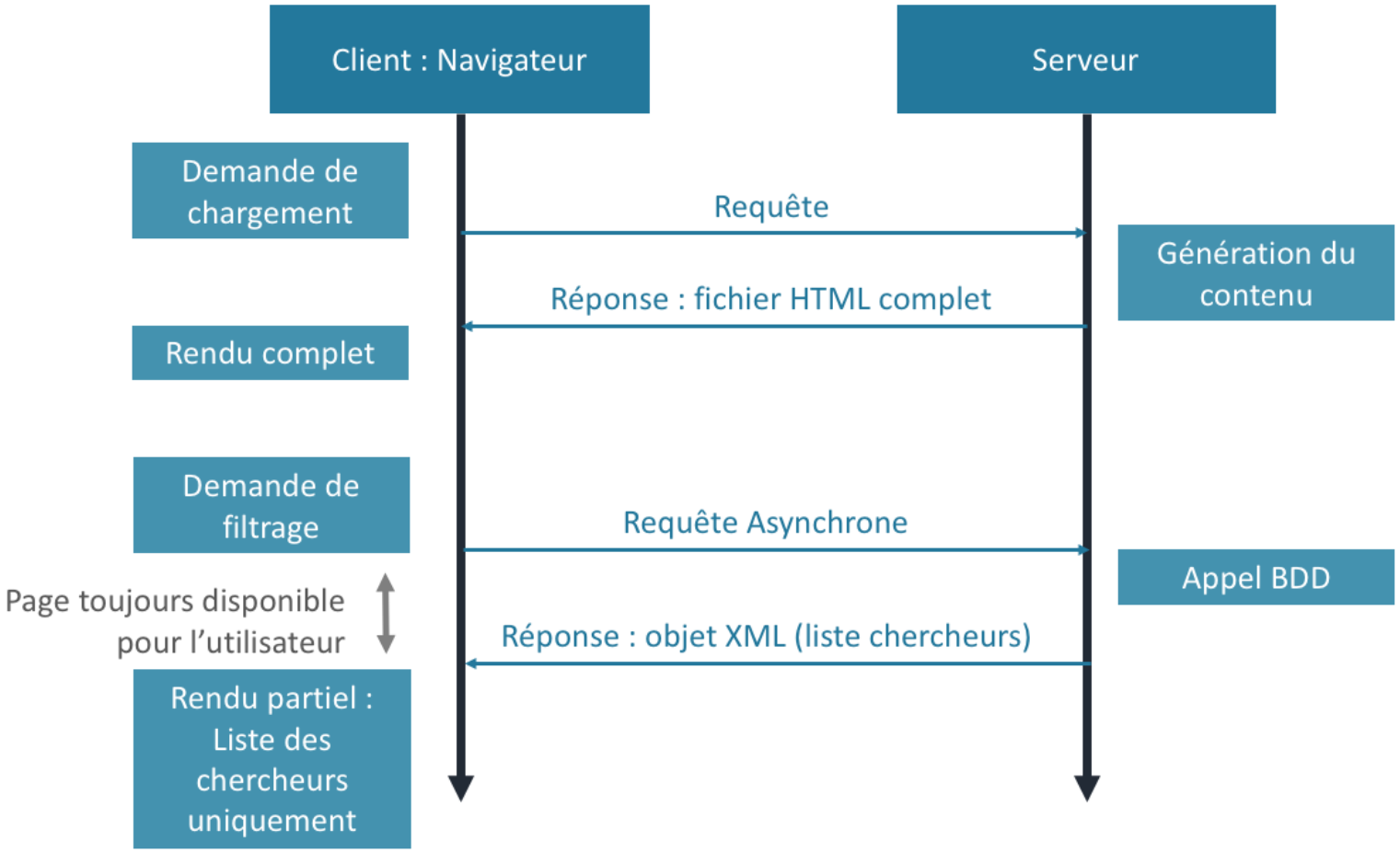
Applications temps réel

Exemple : Annuaire version 1



Applications temps réel

Exemple : Annuaire version 2



Applications temps réel

2.1. AJAX

2.2. Websockets

2.3. Exemple : interface web - calculateur

Applications temps réel

Websockets

Communication « classique » : Requêtes client vers serveur

- Ouverture de connexion au moment de la requête
- Fermeture de connexion une fois la requête résolue.
- Le serveur ne peut pas « recontacter » le client une fois la connexion fermée.

Communication via websockets : Communication client-serveur bilatérale

- Connexion ouverte tant que client et serveur sont disponibles
- Communication bilatérale

Applications temps réel

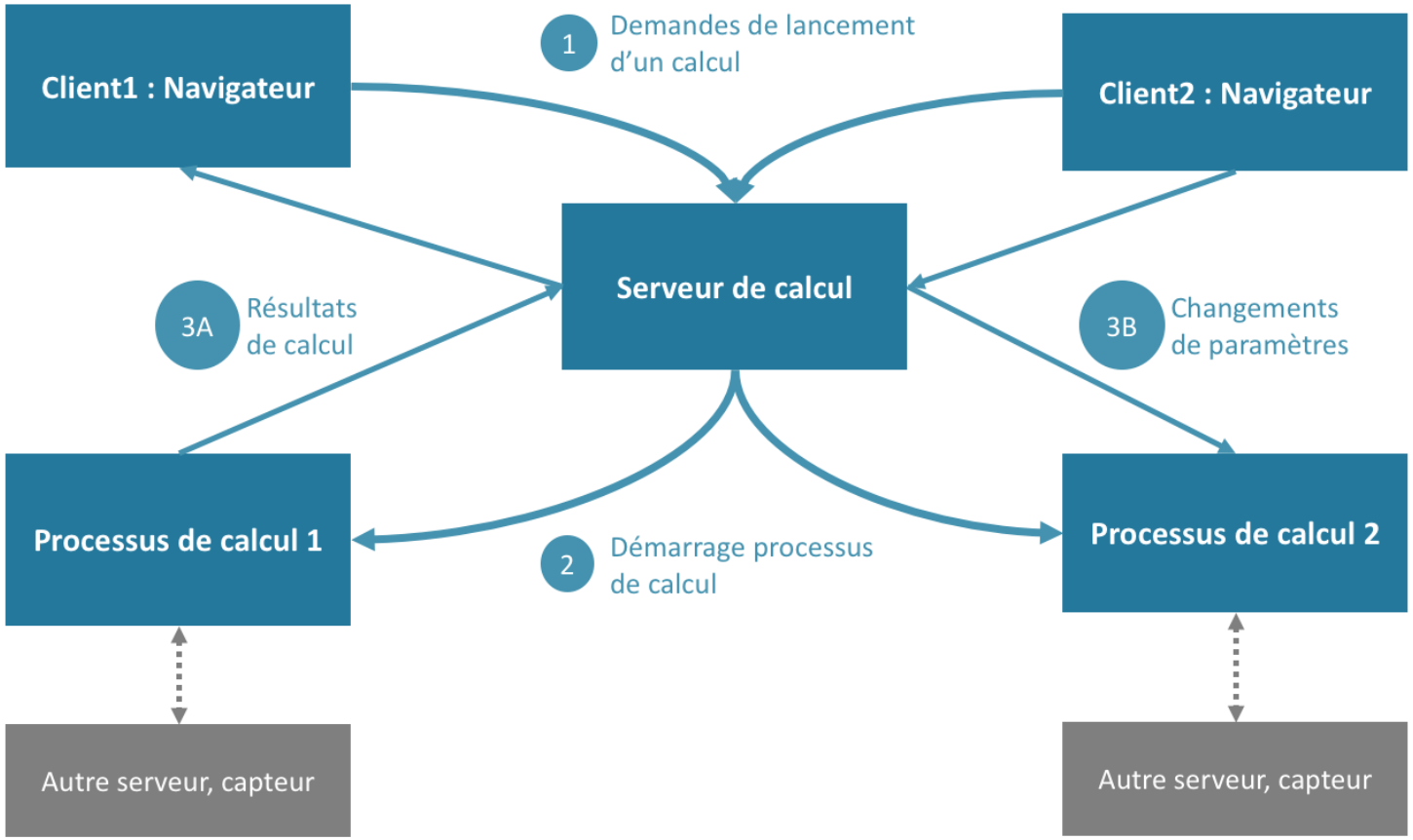
2.1. AJAX

2.2. Websockets

2.3. Exemple : interface web - calculateur

Applications temps réel

Interface web-calculateur



Mise en oeuvre

3.1. Côté serveur : NodeJS

3.2. Côté client : AngularJS

3.3. Communication : SocketIO

3.4. Calculateur Python

Côté serveur : NodeJS

Environnement de développement côté serveur.

Langage : Javascript (seul environnement permettant d'interpréter du javascript en dehors d'un navigateur)

Avantages :

- Même langage côté client et serveur
- Asynchronisme : adapté aux applications temps réel

Côté serveur : NodeJS

Tutoriel : Création d'une application nodeJS

- S'assurer que node et npm ont bien été installés :
  ```
  node --version
  npm --version
  ```

- Initialiser l'application (créé un fichier package.json pour lister les propriétés et dépendances) :
  ```
  npm init
  ```

  Préciser au moins le champ "entry point" : server.js

- Installer les dépendances :
  ```
  npm install --save express
  npm install --save nodemon
  ```

Côté serveur : NodeJS

Tutoriel : Création d'une application nodeJS

Module Express : Infrastructure permettant de générer des serveurs web facilement
Nodemon : Utilitaire, daemon.

- Configurer le script de lancement de l'application : éditer package.json

```
{
    "name": "test",
    ...
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
        ”start": ”nodemon server.js"
    }
}
```

Côté serveur : NodeJS

Tutoriel : Création d'une application nodeJS

Créer le fichier server.js. Dans ce fichier :

- Invoquer les dépendances :

```
var express = require('express');
```

- Créer le serveur :

```
var server = express();
```

- Lancer le serveur (écoute) :

```
server.listen(8000, function () {
        console.log('Server now listening on port 8000');
});
```

- Tester : depuis la console, lancer

```
npm start
```

Côté serveur : NodeJS

Tutoriel : Création d'une application nodeJS

- Créer un fichier 'index.html' dans le même répertoire :

```
<html>
    <body>
        <h1>Hello World</h1>
    </body>
</html>
```

- Envoyer le fichier index.html en réponse à une requête GET :

  Dans le fichier server.js, avant la ligne 'server.listen…', ajouter

```
server.get('/', function (request, response) {
    res.sendfile('index.html');
});
```

- Tester : ouvrir un navigateur et rdv à l'adresse 'localhost:8000'

Côté serveur : NodeJS

Tutoriel : Création d'une application nodeJS

- Couper le daemon dans votre dossier

- Récupérer le code sur github :
  git clone https://github.com/CarolineRamond/ANF.git [nom_dossier]
  cd [nom_dossier]
  git checkout step0

- Installer les dépendances :
  npm install

- Lancer le daemon dans le nouveau dossier :
  npm start

Mise en oeuvre

3.1. Côté serveur : NodeJS

3.2. Côté client : AngularJS

3.3. Communication : SocketIO

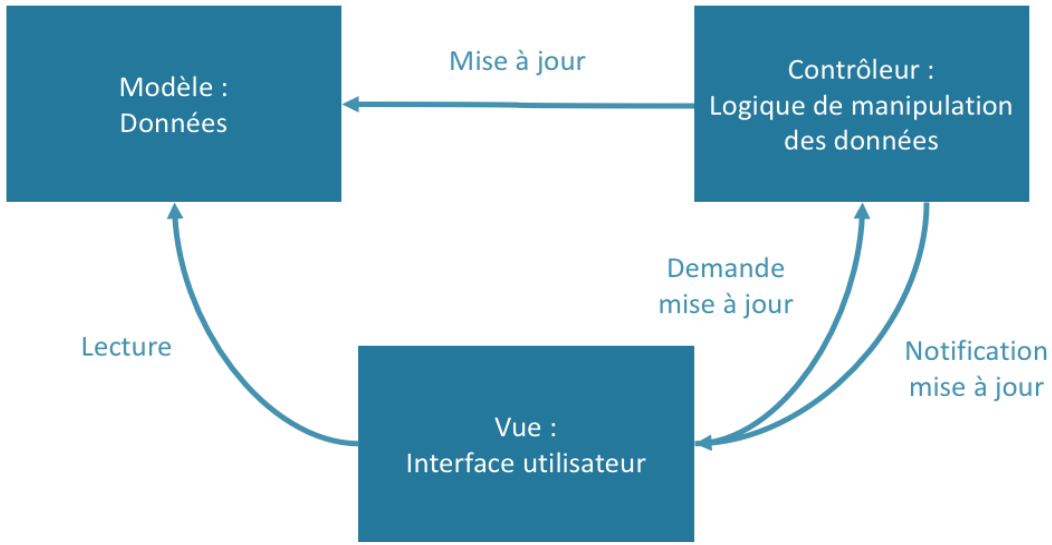
3.4. Calculateur Python

Côté client : AngularJS

Framework Javascript destiné à la partie client.

Fonctionne selon le patron de conception MVC (Modèle-Vue-Contrôleur) :

- Modèle : les données
- Vue : ce que l'utilisateur voit
- Contrôleur : les actions sur les données



Vue : « coquille vide » destinée à accueillir les données du modèle, manipulées grâce au contrôleur.

Côté client : AngularJS

Concepts clé :

- **"Data binding" :** lien entre la vue et les paramètres du contrôleur.
  Le contrôleur "expose" les données nécessaires à la vue.
  Si la donnée change dans le contrôleur, elle change dans la vue et vice-versa.

- **Directives :** permettent la modification effective du DOM (le contrôleur **ne modifie pas directement le DOM**) à partir des données exposées par le contrôleur.
  De nombreuses directives sont disponibles dans la librairie (ng-..), de nouvelles peuvent être développées.

Côté client : AngularJS

Vue	Contrôleur

<section ng-controller='MyCtrl'>
 	
<h1 ng-bind='myTitle'></h1>
 	
<input type='text' ng-model='myVar'></input>
 	
<button ng-click='sayHello'>Hello</button>
 	
</section>
	myApp.controller('MyCtrl', function ($scope) {
	
	$scope.myTitle = 'Hello World';
	
	$scope.$watch('myVar', function () {
	console.log($scope.myVar);
	});
	
	$scope.sayHello = function () {
	alert('Hello World');
	};
	});

Côté client : AngularJS

Tutoriel : Création d'une application angularJS

- Inclure la librairie angularJS dans le fichier client/index.html :

```
<html>
    <header>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.5.8/angular.js">
      </script>
    </header>
    <body>
        <h1>Hello World</h1>
    </body>
</html>
```

- Dans le dossier 'client', créer un fichier app.js :

```
touch app.js
```

Côté client : AngularJS

Tutoriel : Création d'une application angularJS

Dans le fichier app.js :

- Créer l'application 'myApp'.

```
var app = angular.module('myApp', []);
```

- Lui ajouter un contrôleur nommé 'myCtrl'. Ce contrôleur nécessite le service '$scope'.
  Il expose la variable 'myTitle', la fonction 'sayHello', et surveille la variable 'myVar'.

```
app.controller('myCtrl', ['$scope', function ($scope) {
        $scope.myTitle = 'Hello World';
        $scope.$watch('myVar', function () {
            console.log($scope.myVar);
        });
        $scope.sayHello = function () {
            alert('Hello World');
        };
}]);
```

Côté client : AngularJS

Tutoriel : Création d'une application angularJS

- Relier l'application et le contrôleur à la vue dans le fichier client/index.html, et charger le fichier app.js :

```
<html>
    <header>
      …
    </header>
    <body ng-app='myApp'>
      <section ng-controller='myCtrl'>
          <h1>Hello World</h1>
      </section>
    </body>

    <script type='text/javascript' src='app.js'></script>
</html>
```

Côté client : AngularJS

Tutoriel : Création d'une application angularJS

- Compléter le fichier client/index.html :

```
<section ng-controller='MyCtrl'>
    <h1 ng-bind='myTitle'></h1>
    <input type='text' ng-model='myVar'></input>
    <button ng-click='sayHello()'>Hello</button>
</section>
```

- Tester : ouvrir un navigateur à l'adresse localhost:8000

Mise en oeuvre

3.1. Côté serveur : NodeJS

3.2. Côté client : AngularJS

3.3. Communication : SocketIO

3.4. Calculateur Python

Communication : SocketIO

Tutoriel

SocketIO : protocole de websockets, disponible côté client et côté serveur.

Avant de poursuivre : git checkout step1

1. **Création du serveur de sockets :**

   - Installer le module socketIO pour le serveur :
     npm install --save socket.io

   - Dans le fichier server.js, créer le serveur de sockets :
     var io = require('socket.io').listen(8080);

Communication : SocketIO

Tutoriel

- Détection des connexions/déconnexions :

```
io.on('connection', function (socket) {

    console.log('Connexion détectée');

    socket.on('disconnect', function() {
        console.log('Déconnexion détectée');
    });

});
```

L'objet **socket** représente l'interface de dialogue entre le serveur et le client nouvellement connecté.

Communication : SocketIO

Tutoriel

2. **Connexion au serveur :**

- Charger la librarie socketIO pour le client : dans index.html,

```
<header>
    …
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
</header>
```

- Se connecter au serveur de sockets : dans app.js,

```
var socket = io.connect('http://localhost:8080');
```

- Tester : ouvrir/fermer plusieurs fenêtres de navigateur à l'adresse localhost:8000, puis vérifier la console serveur.

Communication : SocketIO

Tutoriel

Avant de poursuivre : git checkout step2

3. Emission de messages client -> serveur:

- Côté client : modifier la fonction $scope.sayHello() :

```
$scope.sayHello = function () {
    var obj = { message: 'Hello', from: $scope.myVar };
    socket.emit('hello', obj);
};
```

‘hello’ : nom de l’événement transmis
Obj : objet transmis (objet JSON pouvant contenir chaînes de caractères, tableaux, tableaux binaires, etc)

- Côté serveur :

```
socket.on('hello', function (data) {
    console.log('Message reçu : ' + data.message +
            ' from ' + data.from);
});
```

Communication : SocketIO

Tutoriel

4. **Emission de messages serveur -> client :**

- Côté serveur :

```
socket.on('hello', function (data) {
    console.log('Message reçu : ' + data.message +
            ' from ' + data.from);
    socket.emit('welcome', data.from);
});
```

- Côté client :

```
socket.on('welcome', function (data) {
    console.log('Welcome ' + data);
});
```

Communication : SocketIO

Tutoriel

4. **Emission de messages serveur -> tous les clients :**

- Côté serveur :

```
socket.on('hello', function (data) {
    console.log('Message reçu : ' + data.message +
            ' from ' + data.from);
    // socket.emit('welcome', data.from);
    io.emit('welcome', data.from);
});
```

Communication : SocketIO

Tutoriel

5. Broadcast (émission de message vers tous les clients sauf l'émetteur) :

- Côté serveur :

```
socket.on('hello', function (data) {
    console.log('Message reçu : ' + data.message +
            ' from ' + data.from);
    // socket.emit('welcome', data.from);
    // io.emit('welcome', data.from);
    socket.broadcast.emit('welcome', data.from);
});
```

Mise en oeuvre

3.1. Côté serveur : NodeJS

3.2. Côté client : AngularJS

3.3. Communication : SocketIO

3.4. Calculateur Python

Calculateur Python

Avant de poursuivre : `git checkout step4`

→ Squelette de calcul disponible dans le fichier calcul.py

Calculateur Python

Sockets

Librairie socketIO-client-2 de Python :

- Importer la librairie :
  ```
  from socketIO_client import SocketIO
  ```
- Se connecter au serveur :
  ```
  socket = SocketIO('localhost', 8080)
  ```
- Emettre des messages :
  ```
  socket.emit([nom_message], [contenu_message])
  socket.wait(seconds=0.1)
  ```
- Recevoir des messages :
  ```
  socket.on([nom_message], [callback])
  ```

Calculateur Python

Gestion des processus

Lancement d'un processus (depuis le serveur) :

- Importer la librairie child_process :

```
var child_process = require('child_process');
```

- Lancement d'un processus :

```
var ps = child_process.exec([commande],
    callback(error, stdout, stderr){ .. } );
```

La fonction de callback est exécutée une fois le processus terminé

- Envoi de signal au processus :

```
ps.kill('SIGSTOP');
```

SIGSTOP : pause,
SIGCONT : resume,
SIGTERM : arrêt (conditions normales)
SIGKILL : arrêt (« radical »)

TP

- Lancer un processus de calcul depuis l'interface web
  Pour vérifier le lancement : ps aux | grep python
  Corrigé : git checkout step5

- Afficher les données du calculateur dans l'interface web (afficher dans la console ou utiliser ng-bind)
  Pour convertir un buffer en tableau : new Float64Array(buff)
  Pour forcer le rafraîchissement de la vue : $scope.$apply()
  Corrigé : git checkout step6

- Associer un et un seul processus à chaque interface web connectée. L'interface ne doit recevoir que les données de son processus de calcul.
  Utiliser les identifiants de sockets : socket.id
  Utiliser l'émission de message ciblée : io.to(socketId).emit(…)
  Corrigé : git checkout step7

TP

- Terminer le processus de calcul lié à l'interface (à la demande et/ou à la fermeture de la page)
  Corrigé : git checkout step8

- Changer la longueur du tableau renvoyé par le calculateur via l'interface
  Corrigé : git checkout step9

TP

Namespaces

Pour distinguer les connexions des calculateurs de celles des interfaces.

- Côté serveur :

```
var clientio = io.of('/client');
var calcio = io.of('/calc');

clientio.on('connection', function (socket) { .. })
```

- Côté client (interface) :

```
var socket = io.connect('http://localhost:8080/client');
```

TP

Namespaces

Pour distinguer les connexions des calculateurs de celles des interfaces.

- Côté client (calcul) :

```
from socketIO_client import SocketIO, BaseNamespace

self.socketIO = SocketIO('localhost', 8080, BaseNamespace, params={"clientId" : self.clientId })
self.namespace = self.socketIO.define(BaseNamespace, '/calc')

self.namespace.emit(...)
self.socketIO.wait(seconds=1)
```

Corrigé : git checkout step10

TP

ThreeJS

- ThreeJS : bibliothèque de visualisation 3D pour Javascript.

- Fonctionnement : attacher une fenêtre de visualisation à un élément du DOM

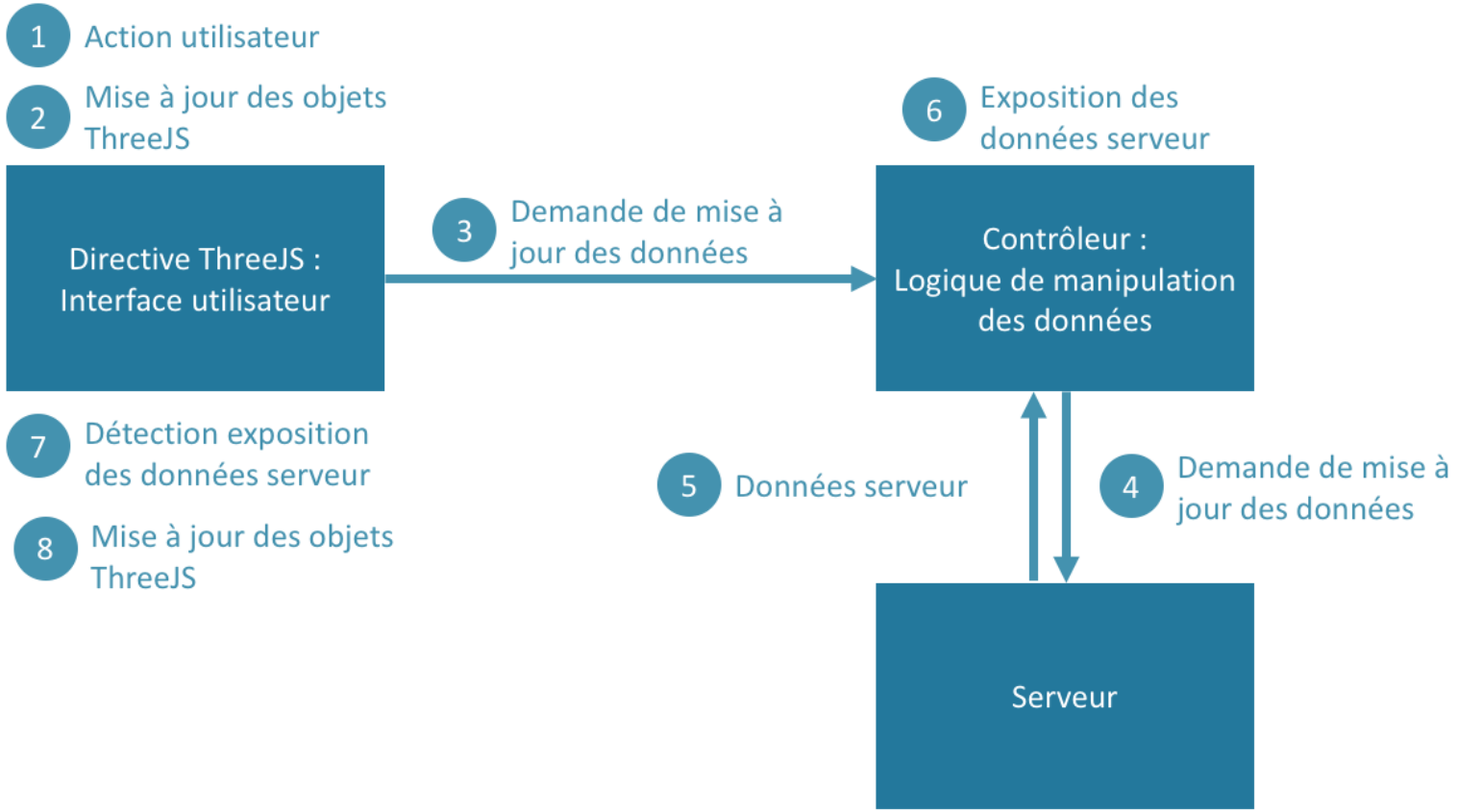
→ Interface avec Angular : directive.

- Interface 3D "classique" : scène, acteurs, costumes, renderer, etc.

Résultat : git checkout step11

TP

ThreeJS


1 Action utilisateur
2 Mise à jour des objets ThreeJS
Directive ThreeJS : Interface utilisateur
3 Demande de mise à jour des données
6 Exposition des données serveur
Contrôleur : Logique de manipulation des données
7 Détection exposition des données serveur
8 Mise à jour des objets ThreeJS
5 Données serveur
4 Demande de mise à jour des données
Serveur