

# Stage ASP.NET

## Développement d'applications Web en ASP.NET

Gilles NICOT – gilles.nicot@web-applications.fr

### Profil

- ✓ Développeur - Formateur .NET indépendant

### Pédagogie

- ✓ Lancement du site [www.tutorials-online.com](http://www.tutorials-online.com) en 1998 (communauté MS)
- ✓ Auteur des livres :
  - ❖ **Référence sur VB .NET** (4ème édition en 2005)
  - ❖ **Les Mathématiques au quotidien pour tous** en 2022 - [GigaMaths.fr](http://GigaMaths.fr)

### Service

- ✓ Consulting et développement .NET
- ✓ Développement d'applications Windows jusqu'en 2002
- ✓ Développement d'applications Web depuis 2002
- ✓ Création de [web-applications.fr](http://web-applications.fr) en 2007

### Conventions typographiques

- ❖ **Acronyme, technologie**
- ❖ **Fichier** ou **Dossier**
- ❖ *Expression*
- ❖ **[Touche]**
- ❖ [Lien](#), [rubrique d'aide](#), [Renvoi](#)
- ❖ **Commande**, **fenêtre** ou **boîte de dialogue**
- ❖ **Type**, **Membre**
- ❖ **Symbole**, **valeur dans une boîte de dialogue**

# Sommaire

<b>Introduction - Rappels</b>	<b>8</b>
<b>Roadmap .NET</b>	<b>8</b>
LINQ	8
Programmation asynchrone	9
Programmation par aspect (POA fr – POA en)	10
Core	10
NET 5+	10
<b>La notion d'architecture applicative</b>	<b>11</b>
Les règles de conception SOLID	11
La règle de SOC	11
Les modèles de conception (Design Patterns ou Patrons)	12
Les modèles d'architecture	12
<b>Modèles d'architectures applicative</b>	<b>13</b>
Problématiques dues au manque d'architecture	13
Le modèle MVC	14
<b>Historique du Web</b>	<b>15</b>
<b>HTML5</b>	<b>16</b>
Déclaration du DOCTYPE	16
Déclaration du jeu de caractères	16
Balises sémantiques	17
Formulaires	19
L'attribut type	19
<b>CSS3</b>	<b>20</b>
Rappels sur les sélecteurs	20
Calcul des valeurs des propriétés au rendu	21
La notion de Reset	21
La propriété display	22
Le modèle de boîte	23
La fusion des marges	23
Le positionnement	24
Le flottement	24
<b>Media Queries</b>	<b>25</b>
La notion de Responsive Web Design	26
Bootstrap	26

<b>JavaScript .....</b>	<b>28</b>
Qu'est-ce que JavaScript ? .....	28
Historique .....	28
JavaScript est dynamique .....	28
La notion de DOM .....	29
Le Hello World .....	30
Exécution .....	30
Appel d'une fonction avec un événement .....	30
Débogage .....	30
<b>Éléments du langage .....</b>	<b>31</b>
Syntaxe .....	31
Éléments de base .....	31
Gestion des erreurs .....	31
Littéraux .....	32
Symboles .....	32
Les variables .....	32
Activer le mode Strict .....	32
Les types et opérateur .....	33
<b>Fonctions .....</b>	<b>34</b>
<b>Objets prédéfinis .....</b>	<b>35</b>
Les nombres .....	35
Les chaînes de caractères .....	35
Les Booléens .....	36
Les dates .....	36
Les fonctions globales .....	36
<b>Objets et tableaux .....</b>	<b>37</b>
La primitive null .....	38
La primitive undefined .....	38
Définir des méthodes .....	38
Le mot réservé this .....	38
Tableaux .....	39
La sérialisation JSON .....	40
<b>DOM &amp; BOM .....</b>	<b>41</b>
DOM .....	41
Manipulation des éléments de la page .....	41
BOM .....	42
Gestion des événements .....	43
La méthode addEventListener .....	43

<b>Techniques avancées .....</b>	<b>44</b>
Object vs Array .....	44
L'objet event .....	44
Les méthodes call et apply .....	45
Créer un constructeur d'objet.....	46
La notion de prototype .....	47
Recommandations .....	48
<b>jQuery .....</b>	<b>49</b>
Fonctionnalités principales.....	49
Hello JQuery .....	50
La méthode ready .....	50
Gestion des événements.....	51
<b>Les sélecteurs .....</b>	<b>52</b>
Sélecteurs d'éléments courants .....	52
Sélecteurs par relation .....	52
Sélecteurs d'attributs .....	52
Pseudo-sélecteurs .....	53
Gestion de la sélection .....	54
Récupérer l'objet DOM d'une sélection.....	54
<b>Manipuler le DOM .....</b>	<b>55</b>
Manipuler les éléments .....	55
Manipuler les attributs .....	56
<b>AJAX.....</b>	<b>57</b>
Principe.....	57
<b>Le Framework AJAX jQuery .....</b>	<b>58</b>
La méthode load() .....	58
La méthode get() .....	59
La méthode post() .....	59
La méthode ajax() .....	59
Les promesses .....	60
<b>L'API Fetch.....</b>	<b>61</b>
<b>Aperçu des API's WEB standard.....</b>	<b>62</b>

<b>Application MVC</b>	<b>64</b>
<b>Introduction</b>	<b>64</b>
La notion de Pipeline	64
Organisation de l'application	65
<b>Système d'injection de dépendances (DI)</b>	<b>66</b>
<b>Configuration des routes</b>	<b>67</b>
<b>Web API REST</b>	<b>68</b>
<b>Syntaxe Razor</b>	<b>69</b>
Html et Tag Helpers	69
<b>Vues partagées</b>	<b>70</b>
<b>Validations des données</b>	<b>71</b>
Validation unobtrusive	72
Protections contre les attaques	73
<b>Authentification interne par Identity</b>	<b>74</b>
<b>Persistance des données de session</b>	<b>75</b>
<b>Journalisation</b>	<b>76</b>
<b>Filtrage et gestion des exceptions</b>	<b>76</b>
<b>Préparation du déploiement</b>	<b>77</b>
Gestion des configurations du projet	77
Configuration des pages d'erreurs	77
Publication	77
<b>Entity Framework Core</b>	<b>78</b>
Introduction	78
Fonctionnalités	79
<b>Définition du modèle</b>	<b>79</b>
<b>Générer la base</b>	<b>80</b>
Alimenter la base à sa création	80
<b>Conventions</b>	<b>81</b>
<b>Configuration des tables</b>	<b>82</b>
<b>Configuration des propriétés</b>	<b>83</b>
Nom de colonne	83
Type de colonnes	83
Clé	83
Existence	83
Type string	84
Colonne de type TimeStamp	84
Type complexe	84

<b>Configuration des relations .....</b>	<b>85</b>
Relations 1-N .....	85
Relations 1-1 .....	86
Relations N-M .....	86
<b>Configuration des hiérarchies.....</b>	<b>87</b>
TPH (Table Par Hierarchie) .....	87
L'héritage TPT (Table Par Type) .....	87
<b>Requêtage .....</b>	<b>88</b>
Principe du SQL dynamique.....	88
LINQ To Entities .....	89
La méthode Find .....	89
La propriété Local .....	89
Chargement des propriétés de navigation .....	90
<b>Le pattern async/await .....</b>	<b>91</b>
<b>Validations locales.....</b>	<b>92</b>
Validation d'une propriété.....	92
Validation d'une entité.....	93
Gérer les erreurs de validation .....	94
<b>Mises à jour .....</b>	<b>95</b>
La classe EntityEntry .....	95
La méthode SaveChanges .....	96
<b>Contrôler la création avec des associations .....</b>	<b>97</b>
<b>Migrations .....</b>	<b>98</b>
<b>Application Blazor .....</b>	<b>100</b>
<b>Introduction.....</b>	<b>100</b>
<b>Organisation de l'application .....</b>	<b>101</b>
Blazor WebAssembly .....	101
Blazor Serveur.....	102
Système de routage.....	103
<b>La notion de composant .....</b>	<b>104</b>
Paramétrage .....	104
Injection de dépendances .....	104
Isolation CSS .....	104
<b>Liaisons de données .....</b>	<b>105</b>
<b>Interopérabilité JavaScript .....</b>	<b>106</b>
<b>Intégration à une application MVC .....</b>	<b>108</b>
Intégration du modèle d'authentification Identity.....	108

# Développement d'applications ASP .NET

## Présentation

### Objectifs

Ce cours vous permettra de découvrir comment concevoir une application Web ASP.NET, aussi bien côté client (HTML5, Javascript, jQuery, Bootstrap et Blazor) que côté serveur (MVC, REST et Entity Framework Core), avec une gestion d'utilisateurs (authentification). Chaque point sera expliqué puis mis en œuvre sous forme d'ateliers pratiques réalisés sous Visual Studio en C#.

### Participants

Cette formation s'adresse aux développeurs connaissant les Web Forms, ou à des chefs de projets souhaitant avoir une vue d'ensemble de la création d'applications Web ASP.NET.

### Prérequis

Bonnes connaissances des langages C#, HTML et SQL. Expérience requise.

### Utilisation du support

Le support a été rédigé avec un souci de clarté. Lorsque c'est utile, un point peut être illustré avec un schéma, une image-écran ou un extrait de code.

### Utilisation des exemples

Les exemples/ateliers proposés dans ce stage ont été conçus avec un objectif didactique/pédagogique et non pas avec celui d'être réutilisables telles quelles dans des applications réelles.

# Introduction - Rappels

## Roadmap .NET

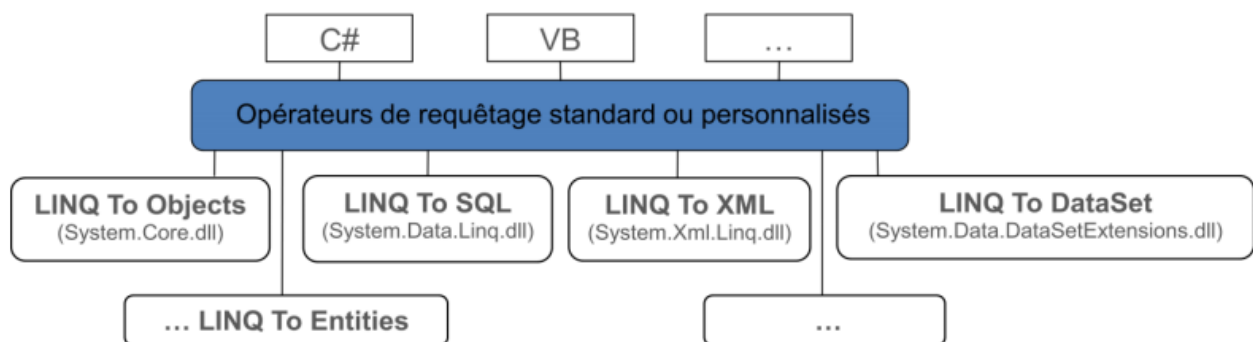
### LINQ

#### Problématique

Le paradigme POO s'est généralisé en // des structures de données stockées avec des différents formats (données relationnelles, XML, fichiers csv, ...).

#### Principe

LINQ permet d'unifier les manipulations de ces différents types de sources de données avec une syntaxe unifiée constitué d'un jeu d'opérateurs (méthodes d'extension) intégré aux langages. Dans le cas de données relationnelles, on parle **MRO** (Mapping Relationnel Objet).



Pour ce faire, les langages ont été enrichis avec :

- ❖ L'Inférence de type (var) pour manipuler les types anonymes de manière typée.
- ❖ Les expressions Lambda.
- ❖ Les méthodes d'extension qui permettent d'ajouter des opérations à une classe existante sans modifier cette classe.



## Programmation asynchrone

Initialement intégrée au Framework .NET sous forme de callbacks **IAsyncResult** puis d'événement **Completed**, la programmation asynchrone a été simplifiée par les **Tasks**, notamment en termes de synchronisation et décharges de données entre threads. Le pattern **async/await** intégré à la version 4.5 du Framework rend cette technique nettement plus simple, avec une syntaxe similaire à celle d'un code synchrone.

Ce pattern consiste à :

1. préfixer les méthodes à appeler de manière asynchrone, dont le résultat peut être **void**, **Task** ou **Task<T>** (où **<T>** représente le type du résultat), avec le mot réservé **await**. Par convention, ces méthodes sont suffixées par **Async** (**OperationAsync** par exemple).
2. ajouter le mot réservé **async** devant la définition de la méthode contenant l'appel asynchrone.

**Exemple :**

```
private async void AppelOperationAsync () {
    int nbrChars = await OperationAsync();
    Console.WriteLine = $"Lecture de {nbrChars} caractères";
}

async Task<int> OperationAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("http://docs.microsoft.com");

    // await attend le résultat de la Mt Async
    string contenuPage = await getStringTask;

    return contenuPage.Length;
}
```

## Programmation par aspect (POA fr – POA en)

Séparer la programmation des aspects techniques (logs, erreurs, etc.) de la logique métier, notamment avec une approche déclarative (les attributs en .NET).

[https://fr.wikipedia.org/wiki/Programmation\\_orientée\\_aspect](https://fr.wikipedia.org/wiki/Programmation_orientée_aspect)

Un attribut s'implémente avec une classe. Il peut donc accepter des arguments correspondant aux surcharges de ses constructeurs.

Quelques attributs courant en ASP .NET : [HttpPost], [Route], [ActionName], [ValidateAntiForgeryToken], [Authorize], [AllowAnonymous], ...

## Core

En 2016, Microsoft lance la plateforme, pour optimiser le hosting Azure et rendre les applications réellement multiplateformes, notamment avec l'avènement de conteneurs d'applications tels que **Docker**. Pour ce faire, le Framework .NET est remplacé par un ensemble allégé du Framework nommé **Framework Standard**, commun à toutes les plateformes et tous les types d'applications.

Les fonctionnalités spécifiques sont ajoutées sous forme de packages Nuget. Le tout peut être compilé de manière totalement indépendante d'un framework ou d'un environnement.

## NET 5+

NET 5 fait l'unification du Framework .NET et de Core et préparer l'unification du développement d'interfaces multiplateformes avec **NET MAUI**, une évolution des **Xamarin.Forms** intégrant **Blazor**.

.NET – A unified platform



<https://dotnet.microsoft.com/platform/support/policy/dotnet-core>

# La notion d'architecture applicative

Avec l'avènement de la POO, il s'est donc vite avéré indispensable de savoir organiser le code des applications de façon à ce qu'ils soient maintenables, testables et évolutifs, en suivant des modèles de conception connus et adaptés aux solutions développées.

## Les règles de conception SOLID

Ces problématiques ayant été clairement identifiées, des guides de conception générales liées à la POO ont tout d'abord été proposées sous l'acronyme **SOLID**.

[https://fr.wikipedia.org/wiki/SOLID\\_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))

- ❶ **Responsabilité unique** (Single responsibility) : une classe (ou une méthode) doit avoir une et une seule responsabilité (code simple et lisible).
- ❷ **Ouvert/fermé** : une entité applicative (classe ou module) doit être ouverte à l'extension, mais fermée à la modification.
- ❸ **Substitution de Liskov** : une instance de type T peut être remplacée par une instance S dérivée de T, sans modification du comportement.
- ❹ **Ségrégation des Interfaces** : préférer plusieurs interfaces spécifiques plutôt qu'une seule interface générale.
- ❺ **Inversion des Dépendances** : dépendre d'abstractions plutôt que d'implémentations.

Toutes ces directives visent à assouplir le code de façon à ce qu'il puisse être modifié et étendu sans impacter l'existant. Plus concrètement, il s'agit de réduire au strict minimum les dépendances des différentes parties de l'application entre elles. On parle ainsi de découplage. A l'inverse, un code mal organisé est dit "fortement couplé" et devient de plus en plus difficile à maintenir en raison d'un nombre d'effets de bords croissant et non contrôlé introduits involontairement au fil de ses évolutions.

## La règle de SOC

Il est également très fréquent de désigner l'objectif de découplage avec l'acronyme **SOC (Separation of concerns)**, qui peut se résumer à la séparation des différentes parties de l'application.

[https://en.wikipedia.org/wiki/Separation\\_of\\_concerns](https://en.wikipedia.org/wiki/Separation_of_concerns)

**Remarque :** **HTML5** illustre bien ce principe en séparant le contenu (HTML), la présentation (CSS) et le comportement (JavaScript).

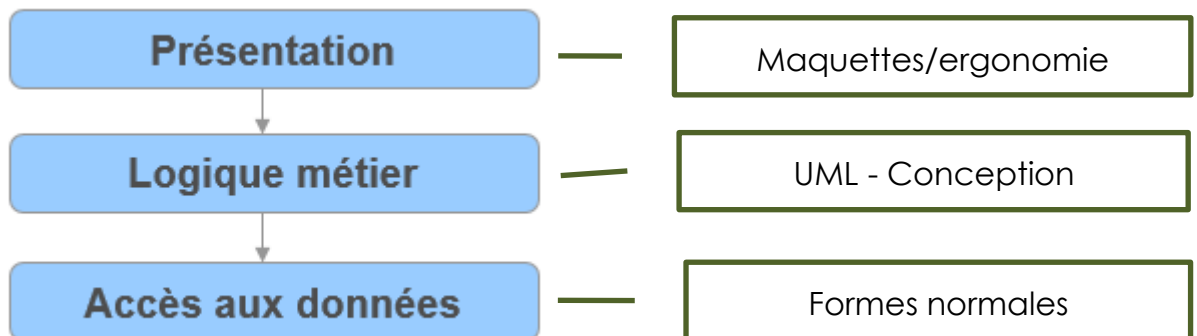
## Les modèles de conception (Design Patterns ou Patrons)

Les **Design Patterns** identifient des problématiques inhérentes à la POO, avec une description non ambiguë, auxquelles une solution validée, ouverte et documentée a été proposée par une communauté d'experts reconnus.

[https://fr.wikipedia.org/wiki/Patron\\_de\\_conception](https://fr.wikipedia.org/wiki/Patron_de_conception)

## Les modèles d'architecture

Au niveau global, une application correctement conçue suit toujours un modèle d'architecture organisé en trois couches (ou **Layers**) principales :



- ❶ La couche de présentation (**UI** ou **IHM**) contient des objets visuels (affichage) et gère le dialogue utilisateur (interactions et contrôles de saisie).
- ❷ La couche métier (**BLL** : Business Logic Layer) contient les objets métiers, c'est à dire spécifiques au domaine ciblé par l'application, avec les règles et les traitements associés.
- ❸ La couche d'accès aux données (**DAL** : Data Access Layer) contient les objets chargés des échanges de données avec un système de stockage (persistance).

Dans la réalité des applications récentes, ces trois couches se subdivisent souvent plus finement, notamment avec une couche de services connectés et des services d'infrastructure (paramétrage, logs, sécurité, etc.).

L'intérêt de ce découpage est double : il permet de :

- ❶ Se concentrer sur chacune des couches, qui peuvent nécessiter des profils de compétences différents.
- ❷ Changer d'IHM ou de DAL sans impacter la BLL.

# Modèles d'architectures applicative

## Problématiques dues au manque d'architecture

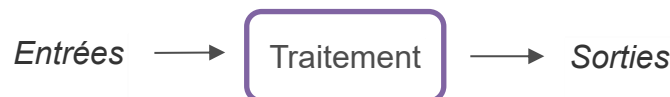
Relativement jeune, La programmation d'applications a commencé au début des années 70, sans aucune abstraction, avec l'assembleur (aussi désigné comme étant le langage machine).

### Besoin d'abstraction

Le langage C voit ensuite le jour et permet d'avoir un premier niveau d'abstraction, avec du code source plus facile à écrire et à relire, nécessitant une compilation pour devenir exécutable.

D'autres langages ont suivi dans les années 80, avec de plus en plus d'abstraction, c'est à dire toujours plus de facilités, tels que Visual Basic, Pascal, etc...

Le nombre d'applications a ainsi pu augmenter beaucoup plus rapidement, mais toujours avec une approche dite "procédurale", intimement liée au fonctionnement de la machine.



### La POO

Lancée par **Smalltalk** au début des années 70, puis intégrée au **C++** un peu plus tard, la POO doit attendre le début des années 2000 pour être adoptée par l'instructrice informatique, notamment avec le lancement de **Java** où elle est intégrée dès le départ, suivi par **.NET** au début des années 2000.

Cette approche s'impose en effet grâce à un paradigme standard, permet de raisonner de manière beaucoup plus intuitive, en regroupant les données et les traitements sous forme d'entités logiques.

### Contraintes de productivité

Grâce à ce niveau d'abstraction, accompagné par l'évolution rapide des outils de développement et des environnements, le nombre d'applications n'a cessé d'augmenter, tout en faisant apparaître un certain nombre de problématiques liées aux difficultés d'évolutions notamment en raison des contraintes techniques historiques (coût de la RAM et du stockage, limitation de taille pour nommer un symbole, approche procédurale facilitée par VB, ...), telles que :

- ❗ Manque de lisibilité du code et de documentation (difficulté de relecture/compréhension).
- ❗ Evolutions de plus en plus difficiles dues à une conception rigide, ou mal organisé (effets de bords indésirés lors d'un changement).
- ❗ Manque de tests ou de difficulté de tester des interface utilisateurs.

[http://fr.wikipedia.org/wiki/Architecture\\_logicielle](http://fr.wikipedia.org/wiki/Architecture_logicielle)

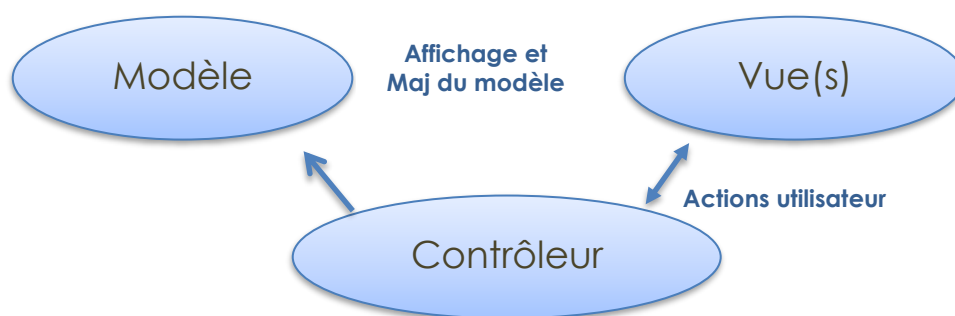
## Le modèle MVC

Cette architecture a été lancée dans les années 80, avec l'objectif d'organiser les applications dotées d'interfaces utilisateur riches.

Il cible les deux couches supérieures du diagramme précédent (**UI** et **BLL**), en partant du constat que la couche **UI**, déjà difficile à tester, l'est d'autant plus si elle contient des opérations de logique applicative.

Ainsi, suivant le principe de séparation des fonctionnalités **SOC**, **MVC** vise à supprimer ce type d'opérations des vues pour les mettre dans des classes sans aucune représentation visuelle. Ces classes se chargent ensuite de manipuler un modèle de données. L'acronyme **MVC** représente cette répartition en trois parties distinctes :

- ❖ Le **Modèle** de données, dont la source peut être quelconque (base de données, service WCF, fichier XML ou autre), qui se présente généralement sous la forme de collections d'objets métiers.
- ❖ Les **Vues** sont chargées d'afficher et de recueillir les données saisies par l'utilisateur ainsi que de transmettre ses actions au **Contrôleur**.
- ❖ Les **Contrôleurs** (généralement un par vue) se chargent quant à eux, de coordonner les échanges entre les vues et les données. Ces données peuvent être affichées (opération de lecture) ou/et modifiées (opération d'écriture) d'après les actions proposées aux utilisateurs dans les vues, transmises et exécutées par les contrôleurs. C'est ce dernier qui se charge d'extraire les données du modèle pour les fournir à la vue en affichage pour une opération de lecture. Dans le cas d'une opération d'écriture, les données sont récupérées dans la vue, validées (contrôlées) et transmises au modèle de données pour mettre à jour la source de données.



<http://fr.wikipedia.org/wiki/Modèle-Vue-Contrôleur>

### Remarques :

- ❶ Ainsi centralisées dans les contrôleurs, les opérations peuvent être facilement testées de manière automatique grâce à des tests unitaires.
- ❶ Le contrôleur communique généralement avec une couche de persistance (DAL) ou/et de services pour charger et mettre les données à jour.
- ❶ Ce modèle est très employé en Web.

## Historique du Web

- ❶ 1970 : La Darpa (Défense des USA) crée **Arpanet**, avec **TCP/IP**.
- ❶ 1984 Le **CERN** (Organisation Européenne pour le Nucléaire) intègre ce protocole en interne.
- ❶ 1991 : Le Web (HTML dont les spec de la V1 n'existent pas) prend naissance au CERN entre 1989 et 1991 avec **Tim Berners-Lee**.
- ❶ 1993 : Navigateur **Mosaic** du **NCSA**, qui crée le tag **<img>** et les formulaires.
- ❶ 1994 : Une partie de l'équipe crée **Netscape** qui ajoute de nouveaux éléments non standards (polices, alignements, clignotement, ...).
- ❶ 1994 **Tim** rejoint le **MIT** et fonde le **W3C**.
- ❶ 1995 : HTML2, avec création de **JavaScript** par **Brendan Eich** pour Netscape. IE V1 & 2 voient le jour avec support des **cookies**, **frames**.
- ❶ 1996 : **CSS 1** avec IE3.
- ❶ 1997 : **HTML 3.2**, avec IE4 qui supprime Netscape et apporte à son tour des éléments non standards (VB Script et Active X).
- ❶ 1997 : **HTML 4** en décembre.
- ❶ 1998 : CSS2, trop complexe, est remplacé par **CSS 2.1**. Le code source de **Netscape** devient libre, mais irrécupérable. Naissance de l'organisation **Mozilla** avec création du moteur **Gecko**.
- ❶ 1999 : **XMLHttpRequest** avec IE5 + **Flash** (FutureWave puis Macromedia puis Adobe).
- ❶ 2000 : Rachat de Netscape par AOL -> déclin.
- ❶ 2001 : IE6 reste majoritaire (jusqu'à 95%) pendant 6 ans !
- ❶ 2002 : Mozilla lance **Firefox**.
- ❶ 2004 : Constitution du groupe de travail **WhatWG** (Web Hypertext Application Technology Working Group) pour faire évoluer le web.
- ❶ 2007 : Lancement de **HTML5** au **W3C**, avec annonce des spécifications en 2022 ! Le W3C et **WhatWG** travaillent en parallèle.
- ❶ 2008 : Sortie de **Chrome**.
- ❶ 2011 : IE9 avec début du support HTML5 – CSS3.
- ❶ 2012 : Lancement de **HTML5 – CSS3** (le WhatWG est intégré au W3C).
- ❶ 2012 : IE10 avec Win8 - IE11 en 2013 avec Win 8.1 (mode de document **EDGE**).
- ❶ 2015 : **EDGE** avec **Win 10**.
- ❶ 2022 : Version 101 de **Chrome**, **FireFox**, et **EDGE**, ...

# HTML5

HTML5 fait évoluer HTML4, en intégrant les éléments permettant de rendre les pages Web interactives et attractives (RIA), sans plug-in. Cette nouvelle version intègre de nouvelles balises, dites « sémantiques », afin d'améliorer la structuration des pages, ainsi que de nouveaux attributs, notamment au niveau des formulaires.

HTML5 s'accompagne de CSS3 afin de combler de nombreuses lacunes de la version antérieure, telles que la gestion des bordures arrondies, des effets de relief avec ombrage, le support de polices de caractères spécifiques, un système de transformations et d'animations, etc.

La partie dynamique repose désormais sur des API's standardisées permettant de gérer la géolocalisation, la mémorisation de données locales, le drag & drop ou l'échange bidirectionnel avec le serveur.

Entièrement compatible avec HTML4, l'intégration de ces nouveautés peut donc se faire progressivement, avec l'une des deux stratégies de compatibilité mentionnées précédemment, généralement basée sur la détection dynamique des fonctionnalités supportées par le navigateur.

## Déclaration du DOCTYPE

La déclaration du DOCTYPE (toujours requise afin d'éviter le mode Quirk), a été simplifiée de façon à devenir ceci :

```
<!doctype html>
```

A la place de ceci :

```
<!doctype html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

**Remarque** : HTML5 reste permissif (un élément peut de pas être refermé par exemple), alors que la déclaration suivante fait passer le document en mode XML, avec une validation, stricte :

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

## Déclaration du jeu de caractères

La déclaration du jeu de caractères (attribut **charset** de la section **<head>**) peut être également simplifiée en ceci ;

```
<meta charset="utf-8" />
```

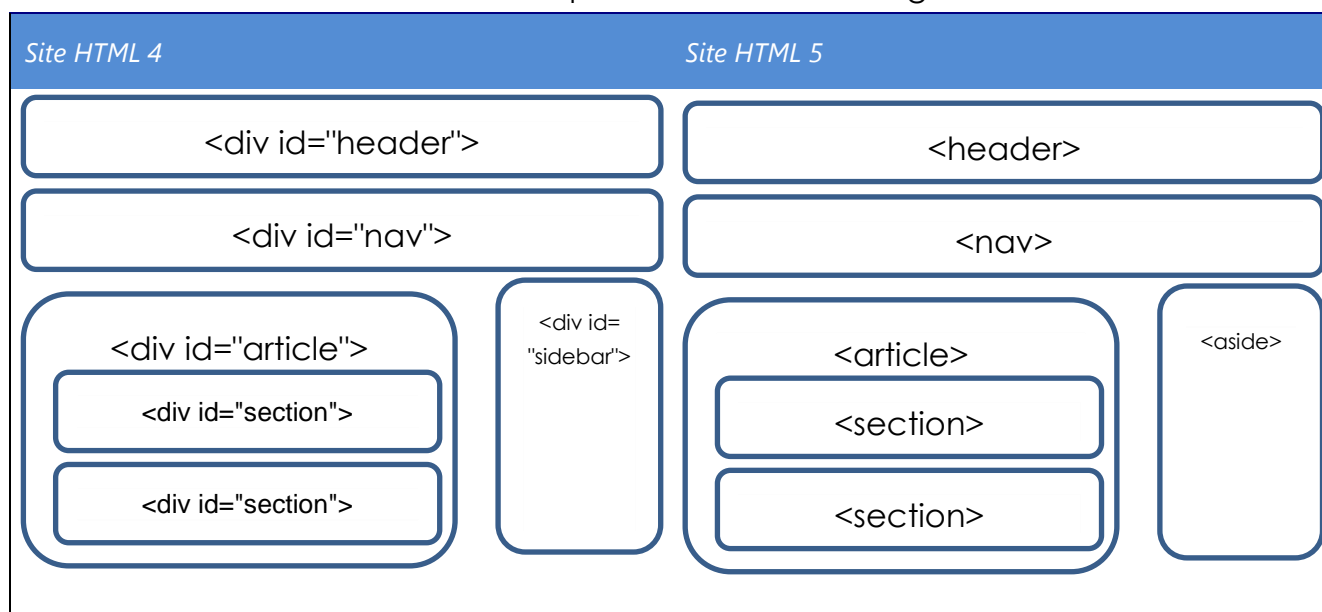
A la place de ceci :

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```



## Balises sémantiques

Après analyse des pages, il s'est avéré que la plupart étaient conçues avec un ensemble d'éléments **div** récurrent, qui sont désormais intégrées à HTML5 :



## Éléments courants

Element	Display	Description
<code>&lt;article&gt;</code>	block	Partie indépendante (article, post de blog, ...)
<code>&lt;section&gt;</code>	block	Partie associée à un titre (chapitre, concept, ...)
<code>&lt;header&gt;</code> , <code>&lt;footer&gt;</code>	block	Section d'en-tête et de fin, pouvant contenir des éléments de navigation. Chaque article ou section peut comporter un élément <b>header</b> et <b>footer</b> .
<code>&lt;nav&gt;</code>	block	Zone comportant des éléments de navigation
<code>&lt;aside&gt;</code>	block	Partie connexe (pub, conseils, statistiques, ...)
<code>&lt;figure&gt;</code>	block	Élément d'illustration (image, vidéo, ...)
<code>&lt;figurecaption&gt;</code>	inline	Libellé descriptif d'un élément <code>&lt;figure&gt;</code>
<code>&lt;mark&gt;</code>	inline	Principe du marqueur, pour faire ressortir un mot ou une expression.
<code>&lt;time&gt;</code>	inline	Affichage d'une date ou/et heure
<code>&lt;hgroup&gt;</code>	block	Permet de regrouper un ensemble de titres.
<code>&lt;address&gt;</code>	block	Coordonnées (adresse postale, tel, email)
<code>&lt;summary&gt;</code> , <code>&lt;details&gt;</code>	block	Résumé et détails (généralement affiché dynamiquement en cliquant sur le résumé).
<code>&lt;code&gt;</code> , <code>&lt;sample&gt;</code>	inline	Code et exemple. Ces deux balises sont sémantiques et de type inline. Contrairement au tag <code>&lt;pre&gt;</code> , les espaces ne sont pas préservés.

<code>&lt;var&gt;</code>	inline	Variable ou équation mathématique (affiché en italique par défaut).
<code>&lt;dfn&gt;</code> , <code>&lt;abbr&gt;</code> , <code>&lt;acronym&gt;</code>	inline	Définition, abréviation et acronyme. Dans une abréviation, l'expression non abrégée est indiquée avec l'attribut <code>title</code> .

A côté des nouveaux éléments apportés par HTML5, un certain nombre d'éléments existants évoluent et s'enrichissent.

### Exemples d'évolutions/améliorations

Élément	Description
<code>&lt;b&gt;</code>	Cet élément était traditionnellement utilisé pour appliquer une mise en forme de mise en gras. Elle reste disponible, mais un texte important doit être délimité par le tag <code>&lt;strong&gt;</code> , qui possède également un rendu en gras par défaut.
<code>&lt;a&gt;</code>	Un lien peut à présent contenir un ensemble d'éléments (à l'exception des éléments interactifs). L'attribut <code>name</code> devient obsolète et doit être remplacé par <code>id</code> .

### Evolution des éléments HTML5

<http://www.w3.org/TR/html5-diff>

## Formulaires

L'élément **input** s'est enrichi d'un grand nombre d'attributs afin de faciliter les saisies.

### Attributs courants

Attribut	Description
<b>autofocus</b>	Active l'élément au chargement de la page : sans valeur.
<b>required</b>	Rend la saisie obligatoire, avec affichage d'un message d'erreur à l'envoi si l'information n'a pas été renseignée : sans valeur.
<b>placeholder</b>	Affiche un message d'aide à la saisie quand l'information n'est pas renseignée. Ce message disparaît automatiquement lorsque l'élément devient actif.
<b>autocomplete</b>	Permet de mémoriser et de proposer les valeurs précédemment saisies : valeur <b>on</b> (défaut) ou <b>off</b> .
<b>pattern</b>	Expression régulière de validation. Cet attribut va de pair avec <b>title</b> et <b>placeholder</b> .

### L'attribut type

Cet attribut enrichit les valeurs traditionnelles (**text**, **password**, **hidden**, **radio**, **button**, **checkbox**, **file** et **submit**) de nombreuses valeurs telles que **number**, **range**, **date**, **phone**, **url**, etc.).

Ces attributs sont gérés différemment selon le navigateur, mais peuvent être utilisés sans risque car ils sont ignorés s'ils ne sont pas supportés.

### Types courants

Type	Description
<b>number, tel, email</b>	Nombre, téléphone, email.
<b>range</b>	Plage de valeurs, nécessite les attributs <b>min</b> et <b>max</b> et éventuellement les attributs <b>step</b> et <b>value</b> pour proposer une valeur par défaut.
<b>date, datetime, month, week time</b>	Affiche un contrôle de saisie d'une date, date avec heure, du mois, d'un numéro de semaine et d'une heure.
<b>search</b>	Zone de recherche (avec une croix pour effacer la saisie et un bouton de recherche selon le navigateur).
<b>pattern</b>	Expression régulière de validation. Cet attribut va de pair avec <b>title</b> et <b>placeholder</b> .
<b>url</b>	Demande une adresse web préfixée par http://.

**Remarque :** Quelques nouveaux éléments tels que **progress** et **meter** font également leur apparition, avec l'attribut **value** à définir en fonction de la progression d'une opération à réaliser (envoi d'un fichier par exemple).

La version 3 des CSS comble de nombreuses lacunes de son prédécesseur, en offrant des possibilités telles que la gestion de polices de caractères spécifiques, les bordures arrondies, ainsi que les effets d'ombrage et d'animations souvent combinées avec des transformations (rotation, inclinaison, changement de taille, etc.).

## Rappels sur les sélecteurs

CSS permet de positionner et de formater les éléments dans une page, à partir de feuilles de styles provenant (par priorité décroissante) :

- ❶ Des styles appliqués par défaut par le navigateur (**User Agent**).
- ❷ Des styles appliqués par l'utilisateur (**User**).
- ❸ Des styles appliqués à la page (**Author**)

Une feuille de styles comporte un ensemble de règles (elles-mêmes constituées de propriétés) appliquées en cascade grâce à des sélecteurs auxquels on affecte un poids afin d'en définir la priorité. Le poids de chaque sélecteur se calcule à l'aide de 3 valeurs déterminant la priorité de la règle correspondante (par ordre décroissant) :

- ❶ Le nombre d'IDs (généralement 0 ou 1).
- ❷ Le nombre de classes (ou pseudo-classes) et d'attributs.
- ❸ Nombre d'éléments (ou pseudo-éléments).

### Exemples :

1 classe et 3 éléments

```
div.header h1 a {  
  Selector specificity: 0, 1, 3 400;  
  font-size: 20px;  
}
```

1 classe + 1 pseudo-classe et 3 éléments

```
div.header h1 a:hover {  
  Selector specificity: 0, 2, 3 000;  
}
```

**Remarque :** les styles appliqués localement (dits **inline**) sont toujours prioritaires, mais déconseillés, tandis que les sélecteurs de parenté n'ont aucun poids.

Référence des sélecteurs CSS avec exemples

<http://www.w3.org/TR/selectors/>

## Calcul des valeurs des propriétés au rendu

Il est réalisé en 4 étapes :

- ❶ **Specified value** : valeur spécifiée par une déclaration (ou héritée du parent).
- ❷ **Computed value** : valeur calculée avant rendu (à partir d'un code couleur par exemple).
- ❸ **Used value** : valeur calculée en fonction des dépendances (tailles en pourcentage par exemple).
- ❹ **Actual value** : valeur appliquée (qui peut être adaptée selon les devices).

## La notion de Reset

Pour éviter les variantes d'affichage dues aux styles appliqués par défaut par les navigateurs, il est conseillé d'appliquer une feuille de styles dite de **reset** (ou **normalisation**) afin de les homogénéiser pour tous les navigateurs.

### Resets usuels

<https://meyerweb.com/eric/tools/css/reset>

<https://html5boilerplate.com/>

### Remarques :

- ❖ Les principaux frameworks CSS intègrent leur reset.
- ❖ Le sélecteur **!important** se rencontre souvent dans les frameworks tels que **jQuery**. Il peut être utilisé, mais de manière contrôlée.

## La propriété display

Les éléments affichés dans une page sont disposés en flux basé sur la propriété **display** et leurs dimensions calculées d'après le "*modèle de boîte*". La propriété **display** accepte 2 valeurs principales **inline** et **block** à partir desquelles les éléments **inline** s'affichent les uns à côté, c'est à dire avec passage à la ligne suivante si la largeur n'est pas suffisante, tandis que la valeur **block** les affiche les uns en dessous des autres.

Dans les deux cas, le principe du flux consiste à positionner chaque élément par défaut dans le coin supérieur droit de son frère précédent ou dans le coin supérieur droit de son parent, dans un même plan.

Contrairement aux éléments **inline** (**a**, **span**, **strong** par exemple), les éléments de type **block** tels que **div**, **h1**, **p** ou **ul**, peuvent être dimensionnés.

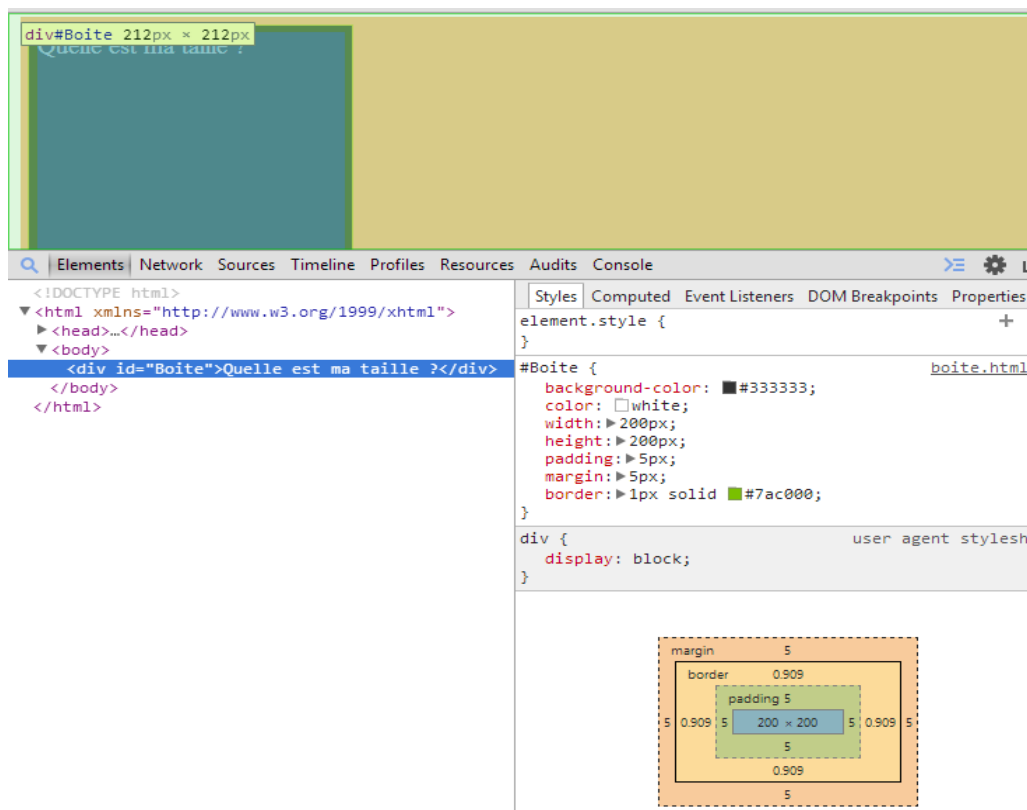
### Valeurs usuelles de la propriété display

Type	Description
<b>block</b>	Positionnement les uns en dessous des autres, avec possibilité de dimensionnement.
<b>inline</b>	Positionnement les uns à côté des autres, sans possibilité de dimensionnement.
<b>Inline-block</b>	Élément <b>inline</b> pouvant être dimensionné ( <b>input</b> par exemple).
<b>table</b>	Analogue au positionnement <b>block</b> , en ajustant automatiquement la largeur au contenu (élément <b>table</b> ).
<b>table-cell</b>	Analogue à la valeur <b>inline-block</b> , en acceptant le centrage vertical avec la propriété <b>vertical-align</b> ( <b>td</b> et <b>th</b> ).
<b>table-row</b>	Analogue à la valeur <b>block</b> , en ajustant automatiquement la hauteur dans l'espace disponible.
<b>list-item</b>	Élément de liste ( <b>li</b> ).

## Le modèle de boîte

Sachant que les éléments de type **block** sont dimensionnables et acceptent un padding, des bordures et des marges. Le système de calcul de la taille de ces éléments d'après ce modèle consiste à additionner la largeur, les marges internes (padding) et l'épaisseur des bordures. Seule la marge, extérieure à l'élément, n'entre pas dans le calcul.

La propriété **width** est donc la largeur du contenu, alors que la taille globale de l'élément = **width** + **padding** + **border**.



### Attention !

- ❗ La taille d'un élément non dimensionné est ajustée à son contenu (shrink-to-fit).
- ❗ Un block dimensionné à 100% peut déborder de son parent s'il a un padding ou des bordures.
- ❗ La propriété **box-sizing** accepte les valeurs **content-box** (CSS2.1) ou **border-box** (CSS 3) qui change le mode de calcul de la largeur en intégrant les bordures.

## La fusion des marges

Lorsque deux blocs frères ou imbriqués possèdent des marges verticales, elles ne s'additionnent pas, mais fusionnent de façon à prendre la marge la plus grande des deux, sauf dans les cas suivants :

- ❗ Une bordure ou un padding vertical est défini sur l'élément parent.
- ❗ La déclaration **overflow:hidden** est définie sur le parent (peut empêcher le débordement des éléments flottants).

## Le positionnement

Le flux des éléments peut être modifié par la propriété **position**, qui accepte les valeurs suivantes :

Valeur	Description
<b>Fixed</b>	Positionnement fixé dans la fenêtre (sort du flux), avec les propriétés <b>top</b> , <b>right</b> , <b>left</b> et <b>bottom</b> .
<b>Absolute</b>	Positionnement dans la page avec les propriétés <b>top</b> , <b>right</b> , <b>left</b> et <b>bottom</b> , par rapport au 1er élément parent positionné avec une valeur autre que <b>static</b> . L'étirement d'un élément devient possible avec <b>left:0</b> et <b>right:0</b> .
<b>Relatif</b>	Déplace l'élément à partir de son emplacement initial (et non pas par rapport à autre élément) dans le flux, avec les propriétés <b>top</b> , <b>right</b> , <b>left</b> et <b>bottom</b> .
<b>Static</b>	Valeur par défaut du positionnement. Sert à rétablir la valeur originale en cas de changement.

## Le flottement

La propriété **float** sert à réorganiser le flux d'un élément avec un rendu flottant à gauche ou à droite dans son conteneur. Il reste aligné sur sa position initiale, mais sa taille n'est plus prise en compte et les éléments frères suivants se réorganisent autour.

Le débordement peut être contrôlé :

- ❶ En donnant la caractéristique d'un **BFC** (Block Formating Context) au conteneur (généralement avec la déclaration **display:table-cell** ou avec la propriété **overflow** si elle possède une valeur différente de **visible**).
- ❷ Avec la propriété **clear** pour annuler le flottement dans la direction spécifiée (ou **both**) sur un élément situé après un élément flottant.

Paragraphe flottant à gauche : débordement du parent, car la hauteur d'un élément flottant n'est pas prise en compte par le conteneur.

Autre paragraphe flottant à gauche : idem.

Div flottant dans un BFC (avec **display:table-cell/inline-block**)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque ornare ipsum at erat. Quisque erat, nec semper dui diam ut libero.



## Media Queries

En CSS 2, un élément **link** de type **text/css** accepte l'attribut **media** de façon à appliquer des styles selon le terminal utilisé (écran, impression, tv, reconnaissance vocale, etc.).

**Exemple** : attribut **media** disponible en CSS2.

```
<link rel="stylesheet" type="text/css" media="screen" href="styles-affichage.css">
<link rel="stylesheet" type="text/css" media="print" href="styles-impression.css">
```

CSS 3 étend cette possibilité avec des blocs **@media device { règles }** nommés **media queries**, dans lesquels les règles de type booléen, s'expriment à l'aide des opérateurs **and**, **only** et **not** appliqués aux caractéristiques du navigateur (taille de la page, orientation, etc.).

**Exemples** :

```
@media screen { body {
    font-size: 1.1em;
}
```

```
@media print {
    body {
        font-size: 10pt;
    }
}
```

```
@media screen and (min-width: 400px) and (max-width: 700px) { ... }
```

### Référence et exemples

[www.w3.org/TR/css3-mediaqueries](http://www.w3.org/TR/css3-mediaqueries)

[www.alsacreations.com/article/lire/930-css3-media-queries.html](http://www.alsacreations.com/article/lire/930-css3-media-queries.html)

[mediaqueriestest.com](http://mediaqueriestest.com) (affichage des caractéristiques du navigateur)

**Remarque** : La reconnaissance des media-queries par la versions 8- de IE, nécessite un polyfill tel que **respond.js**.

**Définition** : Un polyfill est un fichier JavaScript permettant d'implémenter une fonctionnalité standard dans un navigateur qui ne la supporte pas.

## Remarques :

- ❗ **Visual Studio** propose des snippets CSS pour les media queries courantes.
- ❗ Une expression media-querie peut être également appliquée à l'attribut **media** pour charger une feuille de styles spécifique pour des résolutions ciblées :

```
<link rel="stylesheet"
      media="screen and (min-width: 400px) and (max-width: 700px)"
      href="styles-400-700.css" />
```

La propriété **orientation** (enum **portrait** ou **landscape**) permet notamment d'appliquer des règles ou des fichiers CSS en fonction de l'orientation de l'écran.

## Exemple :

```
<link rel="stylesheet" media="(orientation:portrait)" href="portrait.css">
<link rel="stylesheet" media="(orientation:landscape)" href="paysage.css">
```

## La notion de Responsive Web Design

Les média-queries rendent possible la réalisation de sites dits "adaptatifs", c'est à dire capables de s'afficher sur n'importe quel écran, en adaptant automatiquement l'affichage du contenu en fonction de sa résolution.

### Exemples de queries courantes et de sites responsifs

<http://webdesignerwall.com/tutorials/css3-media-queries/>

## Bootstrap

Bootstrap est un Framework CSS et JavaScript permettant de créer des pages Web responsives avec un système de positionnement adaptatif et de composants prédéfinis (messages, menus, boîtes de dialogues, etc.).

### Ressources Bootstrap

<http://getbootstrap.com/>

<https://bootswatch.com/> (thèmes gratuits)

La structure d'une page **Bootstrap** standard comprend la déclaration du fichier **bootstrap.min.css** dans l'en-tête et d'une **div** principale avec la classe **container** dans le document afin de définir la largeur et les marges du document.

Le positionnement des éléments dans la page s'effectue ensuite avec un système de grille organisée en lignes (**div** de class **row**) de 12 colonnes, qui peuvent être distribuées avec des largeurs prédéfinies :

Classe	Description
.col-xs-N	Smartphone : aucune règle (pleine largeur).
.col-sm-N	Petite (tablette) : <b>@media ( min-width : 768 px ) { ... }</b>
.col-md-N	Moyenne : <b>@media ( min-width : 992 px ) { ... }</b>
.col-lg-N	Grande : <b>@media ( min-width : 1200 px ) { ... }</b>

**Remarque** : Les éléments placés librement (en dehors d'une **div** de class **row**) s'étirent sur toute la largeur de la page.

**Exemple** : Page type Bootstrap :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Page type Bootstrap</title>

    <!-- Bootstrap core CSS -->
    <link href="./css/bootstrap.min.css" rel="stylesheet">

    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>

  <body>
    <div class="container">
      <div class="page-header">
        <h1>Exemple de grille Bootstrap</h1>

        <h3>Ligne de 3 colonnes</h3>
        <div class="row">
          <div class="col-md-4">.col-md-4</div>
          <div class="col-md-4">.col-md-4</div>
          <div class="col-md-4">.col-md-4</div>
        </div>
      </div> <!-- /container -->
    </body>
  </html>
```

Système de grille fluide Bootstrap

<https://getbootstrap.com/examples/grid>

# JavaScript

## Qu'est-ce que JavaScript ?

- ❶ **JavaScript** est un langage de scripts basé sur la syntaxe du langage Java, permettant de rendre les pages Web interactives, en manipulant leur **DOM**.
- ❷ JavaScript est orienté objet et s'organise en fonctions.
- ❸ Les scripts sont exécutés par le navigateur, indépendamment du serveur.
- ❹ Des échanges de données peuvent être réalisés avec le serveur avec AJAX.

## Historique

- ❶ Le langage a été créé en 1995 par **Brendan Eich** (Netscape 2).
- ❷ Il est standardisé en 1997 sous le nom de **ECMAScript** (European Computers Manufacturers Association) tandis que le **DOM** a été standardisé en 2008 par le **W3C**.
- ❸ **AJAX** basé sur le composant **XMLHttpRequest** créé par MS en 1998, est adopté en tant que standard de fait. Il est remplacé par l'**API Fetch** depuis ECMAScript 6 en intégrant la notion de promesses.
- ❹ Utilisation en dehors d'un navigateur (notamment sur un serveur) à partir de 2009, avec des Frameworks tels **CommonJS** ou **NodeJS**.

## Remarques :

- ❶ La version 5.1 d'ECMAScript sortie en 2012, est supportée par tous les navigateurs modernes.
- ❷ La version 6 d'ECMAScript est disponible depuis le mois de juin 2015. Elle introduit notamment les notions de classes et de modules.
- ❸ JavaScript évolue chaque année depuis.

## JavaScript est dynamique

Contrairement au **C#** qui type les objets à partir d'une classe, la nature dynamique de JavaScript permet de créer un objet directement, sans classe. De même, des membres peuvent être ensuite ajoutés ou supprimés à un objet existant à tout moment.

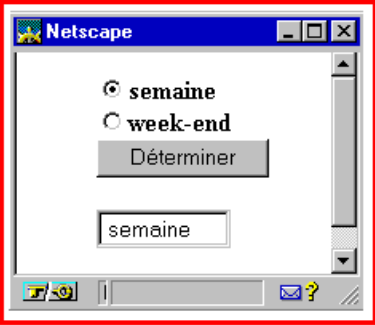
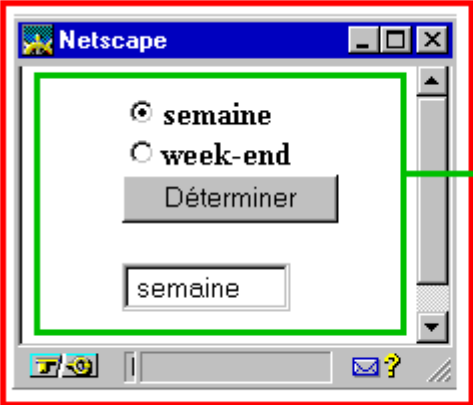
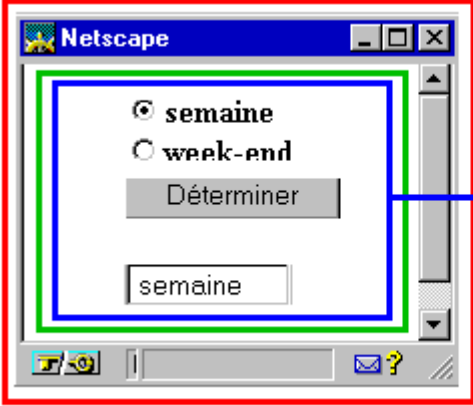
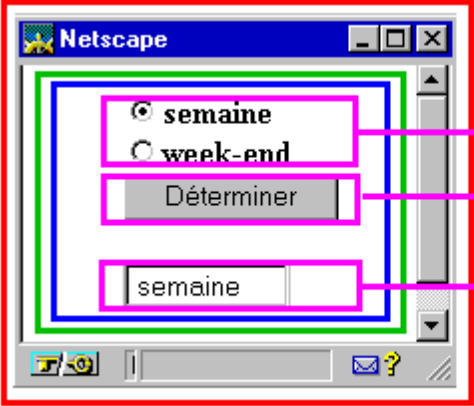
De même, le type d'une variable est dynamique, car il peut varier en fonction du type de la donnée qui lui est affectée.

## Références

<https://developer.mozilla.org/fr/docs/Web/JavaScript>

[https://fr.wikipedia.org/wiki/Document\\_Object\\_Model](https://fr.wikipedia.org/wiki/Document_Object_Model)

## La notion de DOM

 <p>objet fenêtre</p>	<p>Le <b>DOM</b> est une arborescence d'objets dont la racine est l'objet <b>window</b> qui représente la fenêtre du navigateur.</p>
 <p>objet document</p>	<p>La page web affichée dans cette fenêtre est un <b>document</b>.</p>
 <p>objet formulaire</p>	<p>Un <b>document</b> peut ensuite contenir une arborescence de balises (ou tags), avec leurs attributs respectifs.</p>
 <p>objet radio</p> <p>objet bouton</p> <p>objet texte</p>	<p>Un formulaire contient des éléments <b>input</b>. L'accès à un bouton radio dans cet exemple se traduit par l'expression :</p> <pre>(window)     .document.form     .radio[0]</pre> <p>L'objet <b>window</b> est entre parenthèses car il peut être omis. Dans ce cas, il est implicite.</p>

## Le Hello World

Placée dans une balise **<script>** qui peut être placée n'importe où dans la page, une suite d'instructions est exécutée automatiquement lorsqu'elle est reçue par le navigateur.

Sachant que l'objet **window** fournit les méthodes **alert** et **prompt** permettant d'afficher ou de lire un message dans une boîte de dialogue modale :

```
<script>
    alert("Hello JavaScript !");
</script>
```

Ou encore, étant donné que JavaScript permet de modifier les éléments de la page (balises, attributs et styles CSS) via le **DOM** :

### Exemple

```
document.getElementById("msg").innerHTML = "Hello JavaScript !";
```

## Exécution

Le code JavaScript est généralement organisé en fonctions placées dans un fichier externe de façon à les rendre réutilisables.

**Exemple :**

```
<script src="fonctions.js"></script>
```

**Remarque :** HTML5 considère **JavaScript** comme langage par défaut et rend l'attribut **type="text/javascript"** inutile dans la balise **script**.

**Conseil :** placer les scripts dans un fichier externe permet d'optimiser les requêtes HTTP car tout fichier connexe à la page (styles, images, scripts, etc.) est automatiquement mis en cache par le navigateur par défaut.

## Appel d'une fonction avec un événement

L'exécution d'un script peut également se faire en réaction d'un événement déclenché par un élément du DOM (voir section [Gestion des événements](#)).

```
<input type="button" value="Bonjour JavaScript" onclick="bonjourJS();" />
```

## Débogage

Le débogage d'un script peut se faire :

1. Avec le navigateur : tous les navigateurs récents intègrent des outils de développement accessibles par la touche **F12**, destinés à faciliter la mise au point des pages et des scripts. Parmi ces outils, une fenêtre **Console**, permet d'entrer des instructions JS à la volée, ou d'y écrire des informations à partir d'un script avec la méthode `console.log("Un message !")`.
2. Avec Visual Studio.

# Éléments du langage

## Syntaxe

La syntaxe JavaScript se base sur celle du langage C.

- ❶ Sensible à la casse, mots réservés en minuscules.
- ❶ Instructions terminées par un point-virgule.
- ❶ Blocs délimités par des accolades.
- ❶ Commentaires analogues au C#.

## Éléments de base

- ❶ Types primitifs : **Number**, **String**, **Bool**, **null**, **undefined**.
- ❶ Littéraux, variables, opérateurs et expressions.
- ❶ Structures de contrôles : **if**, **while**, **do while**, **for**, **switch**, **break** et **continue**, analogues à celles du C#.
- ❶ Fonctions, tableaux et objets.
- ❶ Un objet peut être créé dynamiquement sans classe, ou à partir d'un constructeur, éventuellement complété par un prototype.

**Remarque** : Les dates sont considérées comme des objets.

## Gestion des erreurs

Les erreurs peuvent être récupérées dans un bloc **try catch** (avec éventuellement un bloc **finally**).

### Exemple

```
function err() {  
    try {  
        // La variable msg n'existe pas  
        alert(msg);  
    } catch (e) {  
        // Afficher l'erreur  
        alert(e);  
    }  
}
```

## Littéraux

- ❗ Une chaîne de caractères se délimite par des guillemets ou des apostrophes.
- ❗ Un nombre s'écrit avec ou sans partie décimale comme en C#, ou en notation scientifique : **265e3** par exemple.
- ❗ Les littéraux booléens sont **true** et **false**.

**Remarque** : la notion de constante est reconnue à partir de la version 6 d'ECMA Script.

## Symboles

Tout symbole (nom de variable, de fonction, d'objet, etc.) doit commencer par une lettre et se composer de lettres, des caractères `_` et `$` (généralement en préfixe) ou de chiffres, sans espace, ni limitation de taille.

**Conseil** : La casse *Camel* est recommandée pour tout symbole.

## Les variables

Une variable peut se déclarer explicitement avec le mot réservé **var** ou de manière implicite, par une affectation (déconseillé).

### Exemple

1. de façon explicite avec le mot réservé **var** :

```
var Nombre = 1  
var Prenom = "Joe"
```

2. de façon implicite, sans le mot réservé **var** :

```
Nombre = 1  
Prenom = "Joe"
```

Une variable aura une portée locale si elle est déclarée dans une fonction ou globale cad au niveau de l'objet racine **window**, si elle est déclarée en dehors.

## Activer le mode Strict

Il est fortement conseillé de toujours utiliser **var** pour déclarer une variable. Dans le cas contraire, elle aura une portée globale, même si elle est définie dans une fonction. Pour éviter cela, la 5<sup>ème</sup> édition d'ECMA Script introduit le **Mode Strict** afin d'imposer la déclaration des variables avec l'instruction suivante :

```
"use strict";
```



## Les types

Contrairement à un langage fortement typé où une variable ne peut contenir qu'un type de donnée bien déterminé, le type d'une variable JavaScript est dynamique et peut changer par inférence à chaque affectation.

En dehors des types primitifs **String**, **Number**, **Boolean** et des valeurs spéciales **null** et **undefined**, les autres sont des objets, soit spécifiques (**Object**), soit prédéfinis (**Date**, **Array**, **RegExp**, **Math**, etc.)

**Remarque :** Une variable peut tout aussi bien référencer un type primitif, un tableau, un objet ou une fonction.

## Les opérateurs

Les opérateurs sont analogues à ceux du C, à quelques nuances près, notamment concernant les opérateurs de comparaison **==** et **!=** dont les variantes **===** et **!==** (voir section [Les booléens](#)) permettent de vérifier également le type des données comparées et non pas seulement les valeurs qui peuvent résulter d'une conversion de type implicite.

### Référence et exemples

[http://www.w3schools.com/jsref/jsref\\_operators.asp](http://www.w3schools.com/jsref/jsref_operators.asp)

# Fonctions

Une fonction peut :

- ❶ Retourner un résultat avec l'instruction **return**. En l'absence de cette instruction, son résultat est **undefined**.
- ❷ Accepter des arguments. Si le nombre d'arguments reçus ne correspond pas lors d'un appel au nombre d'argument déclarés, les arguments en trop sont ignorés, tandis que ceux qui ne sont pas fournis sont **undefined**.
- ❸ Être référencée par une variable.
- ❹ En contenir d'autres (sous-fonctions). Dans ce cas, les fonctions imbriquées ne sont plus accessibles directement en dehors de la fonction parente.
- ❺ Portée des variables (rappel) : la portée d'une variable déclarée explicitement dans une fonction est locale. Les autres sont globales (**window**).
- ❻ Être anonyme.

Une **fonction anonyme** permet de regrouper un ensemble d'instructions sans avoir à créer une fonction de portée globale et peut être auto-exécutée si elle utilisée en tant qu'instruction après avoir été entourée de parenthèses.

Exemple :

```
// Fn. anonyme exécutée directement
(function (msg) {
    alert(msg);
})("Message pour la fn anonyme");
```

**Remarque** : Les arguments de type **object** (ainsi que les tableaux) sont transmis par référence, tandis que le reste est transmis par valeur.

## Attention

- ❶ La notion de signature n'existe pas en JavaScript. Si deux fonctions portent le même nom avec une liste d'arguments différents, c'est la fonction la plus proche de l'appelant qui sera utilisée, indépendamment des paramètres transmis par l'appelant.
- ❷ Les blocs ne limitent pas la portée des variables (sauf à partir de la version 6 d'ECMA Script avec le mot réservé **let**).

La liste d'arguments fournie par l'appelant est disponible par la propriété **arguments** implicitement associée à toute fonction.

# Objets prédéfinis

## Les nombres

Un nombre (type **Number**) est stocké sur 8 octets, avec partie décimale flottante.

- ❶ L'objet **Math** global fournit les fonctions mathématiques standards.
- ❷ La valeur spéciale **NaN** s'obtient lorsqu'une opération donne un résultat indéfini ou une erreur (une division par 0 donne la valeur **Infinity**).

## Les chaînes de caractères

Une chaîne de caractères (objet **String**) :

- ❶ Se concatène avec l'opérateur **+**.
- ❷ Se manipule par des méthodes prédéfinies (**indexOf("chaîne")**, **charAt(index)**, **join**, **substring**, **replace**, etc.).
- ❸ Les méthodes **test**, **search**, **match** et **replace** acceptent les expressions régulières, qui doivent être délimitées par le caractère **/**. A l'instar de la méthode **indexOf**, la méthode **search** est sensible à la casse et donne l'index de la chaîne recherchée ou **-1** en cas d'échec.
- ❹ Un littéral se délimite par des guillemets ou des apostrophes et peut contenir des caractères spéciaux préfixés, comme en C, par le caractère d'échappement **\** (**\n** représente un saut de ligne par exemple.)

### Exemples :

```
var texte1 = "JavaScript est un langage 'dynamique' !";
var texteAvecNombre = "Visual Studio 2020 sera dans le cloud !";

var index = texte1.indexOf("Java");           // 0
index = texte1.search(/Java/);               // 0

// Recherche d'une série de 4 chiffres
index = texteAvecNombre.search(/[0-9]{4}/);  // 14
```

### Référence et exemples

[www.regexlib.com](http://www.regexlib.com)

## Les Booléens

Le type booléen accepte les valeurs prédéfinies **true** et **false** et se combine avec les opérateurs **==**, **===**, **!=**, **!==**, **&&** (Et), **||** (Ou) et **!** (Not).

### Remarques :

- ❶ L'opérateur **?** : fonctionne comme en C (ou C#).
- ❷ Les valeurs spéciales **null**, **undefined**, **NaN**, **""** et **0** sont évaluées à **false**, alors que l'évaluation de toute autre expression non booléenne donne toujours **true**.
- ❸ Les opérateurs **===** et **!==** permettent de réaliser une comparaison stricte, en tenant compte du type tandis que les opérateurs **==** et **!=** font des conversions implicites et comparent la valeur du résultat.

### Exemples :

<pre>0 == false // true 0 === false // false</pre>	<pre>0 != false) // false (0 !== false) // true</pre>	<pre>n = 10; n == "10" // true n === "10" // false</pre>
--	---	--

## Les dates

Une date s'obtient avec l'un des 4 constructeurs suivants :

```
var d = new Date();
var d = new Date(milliseconds);
var d = new Date(dateString);
var d = new Date(year, month, day, hours, minutes, seconds, milliseconds);
```

Les différentes parties d'une date se modifient avec les méthodes **setPartie** (**setFullYear**, **setMonth**, **setHours**, etc.) et se lisent avec les méthodes **getPartie**.

## Les fonctions globales

A l'instar des objets prédéfinis, JavaScript propose des fonctions prédéfinies, dites "globales", telles que **encodeURIComponent**, **decodeURI**, **isNaN**, ou **Number** pour convertir une donnée en nombre, **parseInt** et **parseFloat** pour convertir une String en entier ou en **float**, **String** pour convertir un **object** en chaîne de caractères, ou encore **eval** (à éviter) pour interpréter et exécuter une chaîne de caractères en tant que JavaScript.

### Référence et exemples

[https://www.w3schools.com/jsref/jsref\\_obj\\_number.asp](https://www.w3schools.com/jsref/jsref_obj_number.asp)

[https://www.w3schools.com/jsref/jsref\\_obj\\_string.asp](https://www.w3schools.com/jsref/jsref_obj_string.asp)

[https://www.w3schools.com/jsref/jsref\\_obj\\_boolean.asp](https://www.w3schools.com/jsref/jsref_obj_boolean.asp)

[https://www.w3schools.com/jsref/jsref\\_obj\\_date.asp](https://www.w3schools.com/jsref/jsref_obj_date.asp)

[https://www.w3schools.com/jsref/jsref\\_obj\\_global.asp](https://www.w3schools.com/jsref/jsref_obj_global.asp)

# Objets et tableaux

Un objet peut :

- ❶ Comporter des données (propriétés ou attributs) et des méthodes (voir remarques ci-dessous).
- ❷ L'accès à une propriété se fait soit par un point `objet.membre`, soit avec la syntaxe d'un dictionnaire `objet["membre"]`.
- ❸ Une propriété peut être scalaire ou complexe (objet, tableau, ou fonction).
- ❹ Une propriété peut être ajoutée ou supprimée dynamiquement.
- ❺ Une propriété qui n'existe pas est **undefined**.
- ❻ L'existence d'une propriété sur objet peut être testée avec la méthode **hasOwnProperty**.

## Remarques :

- ❶ Une méthode est une fonction associée à un objet.
- ❷ Une fonction JavaScript est également considérée comme un objet, que l'on peut référencer avec une variable.
- ❸ Un objet peut être créé directement (dynamiquement), sans notion de classe.

## Exemple :

```
// Objet littéral (JSON)
var contact = { nom: "Martin", prenom: "Jean", age: 38 };
```

## Remarques :

- ❶ La syntaxe d'initialisation d'un objet par des valeurs **NomValeur:Valeur** spécifiées entre accolades et séparées par une virgule est celle de **JSON** (JavaScript Object Notation).

**Attention !** Les chaînes de caractères doivent être délimitées par des guillemets et non pas avec des apostrophes !

- ❶ Un objet vierge peut être également créé par l'instruction `var o = new Object();`
- ❷ Les propriétés d'un objet peuvent être parcourues avec une boucle **for**.

## Exemple :

```
for (var pte in contact) {
    console.log(pte + "=" + contact[pte]);
}
```

## Référence

<http://json.org>

## La primitive null

Comme en C# pour les types référence, cette primitive représente l'absence de valeur.

## La primitive undefined

Cette primitive indique l'absence d'un membre sur un objet.

## Définir des méthodes

Une méthode se définit (ou s'ajoute) à un objet avec la même syntaxe qu'une propriété, dont la valeur est une fonction.

## Le mot réservé this

Ce mot réservé ne se comporte pas comme en C#. En JavaScript, il fait référence au contexte de la fonction, qui peut être un objet s'il s'agit d'une méthode, de l'élément HTML ayant déclenché l'événement s'il s'agit d'un gestionnaire d'événement ou de l'objet **window** (objet global) sinon.

**Exemple** : Créer une méthode affichant des attributs de l'objet **contact**.

```
contact.methode = function () {  
    alert(this.nom + " " + this.prenom + " a " + this.age + " ans !");  
}
```

**Remarque** : Il est principalement utilisé dans les fonctions de constructeur (voir sections [Les méthodes call et apply](#) et [Techniques avancées](#)).

## Tableaux

Un tableau permet de regrouper un ensemble de données accessibles par un index, avec un comportement qui s'apparente à celui d'une collection, en acceptant l'ajout ou la suppression d'éléments.

### Un tableau :

- ❶ Peut contenir des données de différents types et se déclare avec l'expression `[]` et non pas avec `new`.
- ❷ Est indexé en base 0.
- ❸ S'alimente par l'ajout d'une donnée avec la méthode `push`, qui peut être supprimée ensuite avec la méthode `pop`.
- ❹ Peut être également alimenté par l'affectation de l'index voulu et se dimensionne automatiquement d'après l'index le plus élevé.
- ❺ Indique le nombre d'éléments qu'il contient avec la Pté `length`.
- ❻ Se parcourt avec la boucle `for` ou avec la méthode  `[].forEach(fonction)` pour exécuter une fonction sur chaque élément du tableau. Cette dernière reçoit l'index et l'élément parcouru en argument.

### Exemples :

```
/* Dimensionnement automatique d'après l'index le plus élevé */
var tabDyn = [];
tabDyn[0] = 0;
tabDyn[1] = 1;
tabDyn[3] = 2; // tabDyn[2] == undefined;

/* Parcours avec for + index */
for (var i = 0; i < tabDyn.length; i++) {
    afficherMessage("tabDyn[" + i + "] : " + tabDyn[i]);
}

/* Parcours avec for ~ foreach */
for (var v in tabDyn) {
    console.log(v);
}

/* Parcours avec la Mt [].forEach */
var listeValeurs = ["valeur1", "valeur2", "valeur3"];

function afficherValeur(element, index) {
    console.log("listeValeurs[" + index + "] = " + element);
}

// Le tableau n'est pas modifié
listeValeurs.forEach(afficherValeur);
```

**Remarque :** La méthode `forEach` accepte un tableau en second argument, pour récupérer le résultat de la fonction appelée sur chaque élément en l'affectant à l'expression `this[index]`.

## La sérialisation JSON

L'objet global **JSON** sert à sérialiser et désérialiser un objet JSON en chaîne de caractères, avec les méthodes `stringify(objetJSON)` et `parse(String)`.

**Exemple :**

```
var stringJSON = '{ "prenom": "Jean", "nom": "MARTIN",  
                  "email": "jean.martin@exemple.com" }';  
var jsonParse = JSON.parse(stringJSON);  
  
var objetJSON = { "prenom": "Jean", "nom": "MARTIN",  
                  "email": "jean.martin@exemple.com" };  
var chaineJSON = JSON.stringify(objetJSON);
```

La méthode `stringify` accepte les arguments optionnels `replacer` et `space`. Le 1<sup>er</sup> permet d'appliquer une conversion sur chaque propriété avec une fonction ou de sélectionner les propriétés à sérialiser si c'est un tableau. Le second permet de préciser comment séparer les valeurs à sérialiser, soit avec une chaîne de caractères, soit avec un entier afin d'indiquer un nombre d'espaces.

**Exemples :**

```
// Sérialiser les ptés spécifiées avec un tableau  
var ptes = ["nom", "prenom"];  
var chaineJsonReplacer = JSON.stringify(objetJSON, ptes);  
// Même exemple, en séparant les valeurs avec un tag <br/>  
var chaineJsonSpace = JSON.stringify(objetJSON, ptes, "<br/>");
```

De même, la méthode `parse` accepte une fonction en argument optionnel, de façon à appliquer un traitement sur chaque valeur désérialisée :

**Exemple :** Mettre la propriété `nom` en majuscule de l'objet désérialisé.

```
var resultat = JSON.parse(stringJSON, function (key, val) {  
    if (key === "nom") return val.toUpperCase();  
    return val;  
});
```



## DOM & BOM

L'intérêt premier de JavaScript est de pouvoir manipuler une page Web via l'API du **DOM** (Document Object Model) prévu pour modifier/créer/supprimer n'importe quel type d'élément dans une page Web c'est à dire les balises et leur contenu (texte, attributs et styles).

De même, l'API **BOM** (Browser Object Model) permet d'interagir avec le navigateur (obtention de la taille de la fenêtre, navigation, méthodes **prompt** et **alert**, **timer**, **cookies**, etc.).

### DOM

Le DOM représente l'arborescence des éléments (parfois appelés nœuds dans la terminologie XML) de la page représentée par l'objet **document**, dont les membres caractéristiques sont les suivants :

- ❶ **head, title, body** : en-tête, titre et corps du document.
- ❷ **URL, referrer, domain** : URL du document, du document précédent et domaine émetteur du document.
- ❸ **images, links, forms, scripts** : ensemble des images, des liens, des éléments de formulaire et des scripts.
- ❹ **readyState** : l'une des 4 valeurs **uninitialized, loading, interactive** ou **complete** de façon à indiquer la progression du chargement du document.
- ❺ **lastModified** : date de la dernière mise à jour du document.
- ❻ **inputEncoding** : encodage (**charset**) des caractères du document.
- ❼ **doctype** : DOCTYPE du document.

### Manipulation des éléments de la page

La plupart des opérations permettant de modifier un élément nécessitent de le référencer avec la méthode **getElementById**, sachant qu'il est également possible d'atteindre un ensemble d'éléments par leur tag ou par une classe CSS avec les méthodes **getElementsByTagName** et **getElementsByClassName**.

**Exemple** : parcourir et afficher le type et la valeur des éléments **input** de la page.

```
var divMsg = document.getElementById("msg");
function listeInputs() {
    var elements = document.getElementsByTagName("input");
    for (var i = 0; i < elements.length; i++) {
        afficherMessage(elements[i].type + " " + elements[i].value);
    }
}
```

La méthode **querySelector** permet en outre d'obtenir le premier élément (ou **null** en cas d'échec) d'après un sélecteur CSS, sachant que la méthode **querySelectorAll** fournit la liste.

## BOM

**Définition** : La taille de la zone d'affichage de la page du navigateur (sans tenir compte des barres d'outils ni des scrollbars), s'appelle le **Viewport**.

L'API BOM est un standard de fait supporté par tous les navigateurs, dont les membres caractéristiques sont les suivants :

- ❶ La racine du **BOM** est l'objet **window**.
- ❷ Le **viewport** (espace d'affichage disponible pour la page) du navigateur s'obtient par les propriétés **window.innerWidth** et **window.innerHeight**.
- ❸ Les méthodes **open**, **close** et **resizeTo** de l'objet **window** permettent respectivement d'ouvrir une nouvelle fenêtre, de fermer et de redimensionner la fenêtre active.
- ❹ La propriété **location** fait référence à un objet dont les membres fournissent les informations relatives à l'url de la page (ptés **href** (url), **hostname** (domaine), **pathname** (path + page sans le domaine ainsi que **protocol** (http:// ou https://) et de rediriger vers une nouvelle page avec la méthode **assign()**.
- ❺ Les propriétés **self** et **top** font respectivement référence à la fenêtre en cours et à la fenêtre racine en cas d'imbrication dans un **iframe**.
- ❻ Le comportement des boutons précédent/suivant est programmable avec les méthodes **back()** et **forward()** de l'objet référencé par la propriété **history**.
- ❼ Le navigateur est décrit quant à lui par les propriétés **userAgent**, **appName**, **appCodeName** (nom du navigateur), **product** (nom du moteur de rendu du navigateur), **cookieEnabled** (acceptation des cookies), **platform** (OS) et **language** de l'objet référencé par la propriété **navigator**.
- ❽ Les méthodes **alert("Message")**, **prompt("Message", "Valeur par défaut")** et **confirm("Message")** sont disponibles pour afficher une boîte de dialogue modale. La méthode **prompt** retourne la chaîne saisie par l'utilisateur s'il a cliqué sur OK ou **null** s'il a cliqué sur Annuler. La méthode **confirm** retourne un booléen.

Obtention de la largeur du viewport pour tous les navigateurs :

```
var largeur=window.innerWidth || document.documentElement.clientWidth || document.body.clientWidth;
```

### Remarques :

- ❶ Toute variable globale devient implicitement une propriété de l'objet **window**.
- ❷ L'objet **document** du DOM est lui aussi associé implicitement à l'objet **window**. Les expressions *Pte*, **document.Pte** et **window.document.Pte** sont donc équivalentes.

## Gestion des événements

La plupart des éléments du DOM-BOM déclenchent des événements tels que le **clic** sur un élément **input** de type **button**, auxquels il est possible de s'abonner avec un gestionnaire d'événement (**EventListener**). Ils sont préfixés par **"on"** (**onclick** par exemple) pour les distinguer des autres types de membres. Un **EventListener** est une fonction standard, sans signature particulière.

Les événements courants sont :

- ❶ Le chargement de la page avec les événements **onload** (chargement) et **onunload** (sortie) de l'objet **body**.
- ❷ Le passage de la souris sur un élément déclenche les événements **onmouseenter**, **onmouseleave** et **onmouseover**.
- ❸ L'élément devient actif/inactif avec les événements **onfocus** et **onblur**.
- ❹ Le changement de valeur d'un élément **input** par l'utilisateur déclenche l'événement **onchange**.
- ❺ La saisie d'un caractère dans un élément **input** de type texte déclenche les événements **onkeydown**, **onkeyup** et **onkeypress**.

Exemple :

```
<body onload="alert('Chargement de la page');">
```

## La méthode addEventListener

Cette méthode permet de s'abonner dynamiquement à un événement, sans avoir à imbriquer du script dans la partie HTML.

Obtention de la largeur du viewport pour tous les navigateurs :

```
document.getElementById("btn").addEventListener("click", eventListener);
```

**Conseil** : Il est recommandé de créer une fonction exécutée au chargement de la page afin d'attacher tous les gestionnaires d'événement.

## Object vs Array

- ❗ L'accès à un élément se réalise par son index s'il est stocké dans un tableau, tandis qu'il possède un nom dans un objet.
- ❗ Un tableau se déclare avec les crochets `[]` (et non pas avec l'opérateur `new`).
- ❗ La méthode `Array.isArray()` permet de différencier un tableau d'un objet.

### Exemple :

```
// Différencier un tableau d'un objet
console.log((typeof contact) === "object");
console.log(Array.isArray(listeValeurs));
```

## L'objet event

Cet objet est disponible dans un gestionnaire d'événement et fournit les méthodes `preventDefault` et `stopPropagation`. La 1<sup>ère</sup> sert à Inhiber l'action par défaut (la navigation en cliquant sur un lien ou empêcher le post lorsqu'un bouton de type `submit` est cliqué par exemple), tandis que la seconde stoppe la propagation de l'événement aux éléments parents (*bubbling*).

**Exemple :** Empêcher la navigation lorsqu'un lien est cliqué.

```
document.getElementById("idElement").
  addEventListener("click", function (event) {
    event.preventDefault();
  });
```

**Remarque :** Ces deux méthodes sont notamment utilisées avec l'API **Drag & Drop**.

Il donne également accès à des informations telles que les coordonnées de la souris (`pageX`, `pageY`) dans la page lorsque l'événement se produit, le `timeStamp` de l'événement (nombre de secondes écoulées depuis une date de référence), etc.

## Les méthodes call et apply

Ces deux méthodes s'appliquent aux fonctions de façon à leur fournir un objet accessible avec le mot réservé **this**. Les arguments sont transmis normalement avec **call** et sous forme de tableau avec **apply**.

**Exemples :**

```
var objet1 = { pte: "Valeur Pté Objet1" };
var objet2 = { pte: "Valeur Pté Objet2" };

function fnAvecContexteObjet(arg1, arg2) {
    var msg = "";

    if (this.pté != undefined) {
        msg = "Parametres: " + arg1 + ", " + arg2 +
            " - Attribut: " + this.pté;
    } else {
        msg = "Parametres: " + arg1 + ", " + arg2 +
            " pas de contexte objet.";
    }

    return msg;
}

afficherMessage("Call avec objet1 : " +
    fnAvecContexteObjet.call(objet1, "P1", "P2"));

afficherMessage("Call avec objet2 : " +
    fnAvecContexteObjet.call(objet2, "P1", "P2"));

afficherMessage("Apply avec objet1 : " +
    fnAvecContexteObjet.apply(objet1, ["P1", "P2"]));

afficherMessage("Apply avec objet2 : " +
    fnAvecContexteObjet.apply(objet2, ["P1", "P2"]));

afficherMessage("Call sans objet : " +
    fnAvecContexteObjet("P1", "P2"));
```

**Remarque :** Ces techniques s'emploient notamment pour faire de l'héritage.

## Créer un constructeur d'objet

Bien que la notion de classe n'existe pas en JavaScript, un constructeur peut se réaliser à l'aide d'une fonction chargée de standardiser la création d'un ensemble d'objets dont le nom devient celui de la "classe" (voir remarques ci-dessous).

Pour ce faire, la fonction initialise les propriétés de l'objet avec ses arguments, en utilisant le mot réservé **this**.

### Remarques :

- i** Il est assez courant de rencontrer le mot "classe", pour désigner cette technique.
- i** Ceci permet d'implémenter l'encapsulation de propriétés, en le préfixant par **this** pour les rendre publiques et non préfixées pour les propriétés privées.

### Exemple :

```
function Contact(nom, prenom, age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;

    this.descriptif = function () {
        return this.prenom + " " + this.nom +
            " a " + this.age + " ans !";
    }
}

var contact1 = new Contact("Martin", "Jean", 27);
afficherMessage("Contact1 avec Ctor : " + contact1.descriptif());

var contact2 = new Contact("Doo", "John", 34);
afficherMessage("Contact2 avec Ctor : " + contact2.descriptif());
```

**Conseil** : La casse Pascal est recommandée pour les fonctions utilisées en tant que constructeurs.

Les méthodes de l'objet peuvent être ensuite créées suivant le même principe, avec des fonctions anonymes (méthode **descriptif** dans l'exemple ci-dessus).

**Attention !** Cette technique n'est pas très optimale en termes d'allocation mémoire, car les méthodes sont dupliquées dans chaque instance. Elle peut convenir avec un faible nombre, sinon, il est préférable d'opter pour le mécanisme de prototype.

## La notion de prototype

L'inconvénient concernant la duplication des méthodes placées dans un constructeur peut être évité grâce à la propriété **prototype** associé implicitement à toute fonction utilisée en tant que classe, de façon à centraliser la déclaration de ses membres, avec la syntaxe JSON.

### Exemple :

```
function ContactAvecProto(nom, prenom, age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
}

ContactAvecProto.prototype = {
    descriptif: function () {
        return this.prenom + " " + this.nom + " a " + this.age + " ans";
    },
    ste: "Ets Martin"
}

var contact1 = new ContactAvecProto("Martin", "Jean", 27);
afficherMessage("Contact1 avec proto : " + contact1.descriptif());

var contact2 = new ContactAvecProto("Doe", "John", 34);
afficherMessage("Contact2 avec proto : " + contact2.descriptif());

afficherMessage("Sté contact2 avec proto : " + contact2.ste);
```

**Remarque :** ES6 intègre le mot réservé `class` permettant de créer des classes plus facilement en JavaScript.

Technologies AJAX et propriétés de l'objet XHR

[https://www.w3schools.com/js/js\\_class\\_intro.asp](https://www.w3schools.com/js/js_class_intro.asp)

## Recommandations

- ❶ Déclarer toutes les variables avec **var** (systématiser l'usage du mode **Strict**).
- ❶ Eviter les variables globales et privilégier les variables locales ou les closures.
- ❶ Restreindre l'usage de l'opérateur **new** aux constructeurs (utiliser l'affectation directe, avec les crochets pour un tableau ou la notation JSON pour un objet dans les autres cas).
- ❶ Utiliser la casse *Camel* par défaut et la casse *Pascal* uniquement pour les fonctions utilisées en tant que constructeurs.
- ❶ Utiliser les opérateurs de comparaison strictes **===** et **!==** à la place de **==** et **!=** pour obtenir une comparaison de la valeur et du type.
- ❶ Ne pas terminer une liste de valeurs d'un tableau ou d'un objet avec une virgule.
- ❶ Dans la définition des fonctions, placer l'accolade ouvrante à droite (sur la même ligne).
- ❶ Délimiter les chaînes de caractères externes avec des guillemets et internes avec des apostrophes.
- ❶ Définir des valeurs par défaut dans les fonctions qui peuvent recevoir des arguments optionnels :

```
function gn(x, y) {  
    if (y === undefined) {  
        y = 0;  
    }  
}
```

- ❶ Eviter l'usage de la fonction **eval()** pour des raisons de sécurité.



# jQuery

**jQuery** est un Framework JavaScript cross-browser disponible en téléchargement ou sous forme de package **NuGet**, dont la vocation consiste à faciliter et étendre les manipulations du **DOM**.

Une fois le package NuGet installé, les 2 fichiers suivants sont placés dans le dossier `\wwwroot\lib\jquery\dist` :

Fichier	Description
<code>jquery-[version].js</code>	<b>Version</b> Debug (~250 ko).
<code>jquery-[version].min.js</code>	<b>Version</b> Release, <b>c'est-à-dire minifiée</b> (~ 80 ko).

**Définition** : un fichier est dit minifié, lorsqu'il a été réduit en taille, en retirant tous les caractères inutiles à son utilisation, notamment les espaces et les tabulations.

## Fonctionnalités principales

- ❶ Référence des éléments du **DOM** avec des sélecteurs riches (par id, tag, classe CSS, attribut, etc.).
- ❶ Gestion des événements simplifiée et améliorée (notamment pour le chargement du document).
- ❶ Manipulation complète du **DOM** (ajout/suppression de nœuds, changement d'attributs, gestion des styles CSS, animations, etc.).
- ❶ AJAX

### Remarque :

- ❶ De nombreux Frameworks JavaScript s'appuient sur **jQuery**, notamment **jQuery UI** qui enrichit les éléments HTML interactifs, **Bootstrap** et **AngularJS**.

### Référence et exemples

[www.jquery.com](http://www.jquery.com)

[api.jquery.com](http://api.jquery.com)

[code.jquery.com](http://code.jquery.com) (CDN)

[learn.jquery.com](http://learn.jquery.com)

## Hello JQuery

Une déclaration doit être faite dans la page sur le fichier `jquery-[version].[min].js`.

Le caractère `$` fait ensuite référence à l'espace de noms **jQuery** donnant accès à l'ensemble des méthodes fournies, avec la syntaxe `$.methode()` ou `$(selecteur).methode()`. Dans ce dernier cas, l'accès à la sélection ciblée par le sélecteur se réalise par le mot réservé **this** dans la méthode.

### Remarque :

- De nombreux arguments de méthodes sont des références à des fonctions, qui peuvent être anonymes.

**Exemple :** Afficher une chaîne de caractères dans une **div** nommée `div1`.

```
<script>
  function bonjourJQ() {
    $("#div1").text("Bonjour jQuery !");
  }
</script>
```

▲ 1 sur 2 ▼ jQuery text(String textString)  
Set the content of each element in the set of matched elements to the specified text.  
**textString:** A string of text to set as the content of each matched element.

Dans cet exemple, on distingue :

- Le caractère `"$"` qui fait référence à l'espace de noms **jQuery** (le mot **jQuery** pourrait être écrit à la place de `$`).
- L'expression `#div1` qui sélectionne un élément **div** nommé `div1`.
- La méthode **text** chargée de renseigner la propriété **innerText** de l'élément sélectionné.

## La méthode ready

Il est souvent nécessaire d'attendre que l'ensemble de la page soit disponible, avec l'événement **window.onload** avant de manipuler le **DOM** de la page.

**jQuery** permet d'optimiser cette opération avec la méthode `$(document).ready()`, qui s'exécute dès que le DOM du document est prêt, sans attendre le chargement des images. Elle accepte la fonction à exécuter en argument, qui peut être anonyme.

**Exemple :** Afficher "ready" dans la console au chargement de la page.

```
<script>
  $(document).ready(function () { console.log("ready!"); });
</script>
```

**Remarque :** L'expression `$(document).ready(function () { ... });` se simplifie souvent en :

```
$(function () { ... });
```

## Gestion des événements

**jQuery** offre la possibilité de définir une fonction en tant que gestionnaire d'événement sur les éléments ciblés par un sélecteur, avec l'une des syntaxes suivantes :

```
$(selector).nomEvenement(gestionnaireEvenement);  
$(selector).on("nomEvenement", gestionnaireEvenement);
```

Où "**nomEvenement**" est similaire à celui du DOM, sans le préfixe "**on**" (**click** pour **onclick** par exemple).

**Exemple** : Associer une fonction **bonjourJQ** à l'événement **clik** d'un bouton nommé **btnBonjourJQ**.

```
// Avec une fn. nommée bonjourJQ  
$("#btnBonjourJQ").on("click", bonjourjQuery);  
  
// Avec une fn anonyme :  
$("#btnBonjourJQ").on("click", function () { alert("Bonjour jQuery !");  
});
```

**Conseil** : Cette approche est à privilégier de façon à éviter l'approche **obtrusive** telle que :

```
<input type="button" value="Bouton" onclick="fonction();" />
```

De même, il est conseillé d'utiliser la méthode **ready** pour attacher les gestionnaires d'événement au chargement de la page :

```
$(function () {  
    $("#btnBonjourJQ").click(bonjourjQuery);  
});
```

## Remarques

- ❶ La méthode **on** possède les variantes **one** et **off** qui permettent respectivement d'exécuter un gestionnaire d'événement une seule fois et de se désabonner.
- ❷ La méthode **on** autorise l'abonnement d'un gestionnaire à plusieurs événements comme ceci :

```
$("#div1").on("mouseenter mouseleave", function () {  
    console.log("La souris entre ou sort de div1");  
});
```

## Les sélecteurs

L'accès au DOM est simplifié à l'expression `$(selecteur)` dans laquelle le sélecteur étend les sélecteurs standard CSS (par type d'élément, par ID, par classe CSS, par attribut ou par lien de parenté).

### Sélecteurs d'éléments courants

Sélecteur	Description
<code>\$("*")</code>	Tous les éléments.
<code>\$("#ID")</code>	Élément nommé avec l' <i>Id</i> spécifié.
<code>\$("typeElement")</code>	Tous les éléments du type spécifié.
<code>\$(".nomClasse")</code>	Éléments dont la classe CSS est <i>nomClasse</i> .

### Sélecteurs par relation

Ce type de sélecteur reprend le principe des sélecteurs de parenté CSS.

Sélecteur	Description
<code>\$("parent enfant")</code>	Descendants, même indirects de parent.
<code>\$("parent &gt; enfant")</code>	Descendants directs de parent.
<code>\$("element + frere")</code>	Élément frère voisin direct de <i>element</i> .
<code>\$("element ~ frere")</code>	Élément frère voisin direct ou non de <i>element</i> .

### Sélecteurs d'attributs

Cette famille permet de sélectionner des éléments d'après leurs attributs, avec possibilité d'utiliser des expressions régulières.

Sélecteur	Description
<code>\$("[attr]")</code>	Éléments possédant l'attribut spécifié, sans tenir compte de la valeur.
<code>\$("[attr]='valeur'")</code>	Éléments possédant l'attribut dont la valeur est "valeur".
<code>\$("[attr] != 'valeur'")</code>	Éléments possédant l'attribut dont la valeur est différente de "valeur".
<code>\$("[attr] ^= 'valeur'")</code>	Éléments possédant l'attribut dont la valeur commence par "valeur".
<code>\$("[attr] \$= 'valeur'")</code>	Éléments possédant l'attribut dont la valeur se termine par "valeur".
<code>\$("[attr] ~= 'valeur'")</code>	Éléments possédant l'attribut dont la valeur contient "valeur".

**Exemple :** Sélectionner les éléments **div** pourvus de l'attribut **class** dont la valeur commence par **div**.

```
$("div[class^='div']");
```




## Pseudo-sélecteurs

Cette famille de sélecteurs reprend le principe des pseudo-classes CSS et se caractérise par le préfixe ":".

### Pseudo-sélecteur courants :

Sélecteur	Description
<code>:contains("valeur")</code>	Éléments contenant le mot spécifié.
<code>:first</code> <code>:last</code>	Premier ou dernier élément. Exemple pour sélectionner le 1 <sup>er</sup> paragraphe : <code>\$("p:first")</code> .
<code>:event</code> <code>:odd</code>	Éléments pairs ou impairs. Exemple pour sélectionner les paragraphes pairs : <code>\$("p:even")</code> .
<code>:eq(n)</code>	N ième élément, en base 0. Exemple pour sélectionner le 2ème paragraphe : <code>\$("p:eq(1)")</code> .
<code>:lt(n)</code> <code>:gt(n)</code>	Éléments dont l'index est strictement inférieur/supérieur à N. Exemple pour sélectionner les 2 premiers paragraphes : <code>\$("p:lt(2)")</code> .
<code>:first-child</code> <code>:last-child</code>	Premier ou dernier fils.
<code>:nth-[last-]child(n)</code>	Nième fils ( <code>:nth-last-child(n)</code> pour partir du dernier).
<code>:first</code> <code>:last</code> <code>:nth-of-type(n)</code>	1 <sup>er</sup> , dernier ou Nième élément du type spécifié.
<code>:elementInput</code>	Sélectionne l'élément de formulaire spécifié. Exemple pour sélectionner les inputs de type <b>button</b> : <code>\$(":button")</code> .

### Remarques :

-  jQuery étend les pseudo-sélecteurs CSS standard avec des sélecteurs tels que **:odd**, **:even**, **gt(index)**, **lt(index)**, **:visible**, **:parent**, **:first()**, **:last()** pour le premier ou dernier élément de la sélection, **:eq(index)** pour le nième élément de la sélection, etc.
-  Le pseudo-sélecteur **contains** est déprécié en CSS3, mais toujours supporté par jQuery.
-  Le pseudo sélecteur **visible** se base sur l'affichage d'un élément et non pas sur la valeur de l'attribut **display**.

### Exemples récapitulatifs

Exemple	Description
<code>\$("input[name='nom']");</code>	Éléments <b>input</b> possédant l'attribut <b>name</b> avec la valeur <b>"nom"</b> .
<code>\$("#liste ul.themes li");</code>	Éléments <b>li</b> situés dans une liste <b>ul</b> de classe <b>themes</b> , elle-même située dans un parent nommé <b>liste</b> .
<code>\$("#nomDiv p:first")</code>	1 <sup>er</sup> paragraphe d'une div dont l'id= <b>nomDiv</b> .

## Gestion de la sélection

Ces sélecteurs retournent un ensemble d'objets jQuery et non pas les objets DOM eux-mêmes.

Pour vérifier la présence d'éléments dans cet ensemble, il convient de tester la propriété **length** du résultat et non pas le sélecteur.

**Exemple :**

```
if ($("#div").length) { }
```

et non pas :

```
if ($("#div")) { }
```

Le résultat d'un sélecteur n'étant pas mémorisé, il peut l'être avec une variable (souvent préfixée par le caractère **\$** par convention), sans oublier qu'il s'agit d'un objet jQuery (et non pas les éléments DOM recherchés).

**Exemple :**

```
var $divs = $("#div");

if ($divs.length > 0) {
    alert("Nbr de divs : " + $divs.length);
}
```

## Récupérer l'objet DOM d'une sélection

La récupération des éléments DOM fournis par un sélecteur se fait par la méthode **get(N)** qui donne la Nième entrée du résultat.

**Exemple :** Différencier l'objet jQuery de l'objet DOM.

```
// Acces DOM avec la méthode get()
$("#btnObjetDOM").on("click", function () {
    var $objetJQ = $("#div1");
    var objetDOM = $("#div1").get(0);
    console.log($objetJQ);
    console.log(objetDOM);
});
```

**Exemple :** Afficher un message dans le 1<sup>er</sup> paragraphe de la page.

```
var $paragraphes = $("p");
$paragraphes.get(0).innerHTML = "1er paragraphe";
```

**Attention !** La méthode **eq** (voir [page suivante](#)) à la place de **get** ne conviendrait pas car elle donne un objet jQuery qui ne possède pas la propriété **innerHTML**, mais une méthode **html()**.

# Manipuler le DOM

## Manipuler les éléments

**jQuery** simplifie la modification du document avec les méthodes suivantes :

Méthode	Description
<code>\$(element).insertAfter(sélection)</code>	Insère un élément après la sélection et fournit une référence sur l'élément inséré en retour.
<code>\$(élément).insertBefore(sélection)</code>	Insère un élément avant la sélection et fournit une référence sur l'élément en retour.
<code>\$(element).appendTo(sélection)</code>	Insère un élément à la fin <u>de l'intérieur</u> de la sélection et fournit une référence sur l'élément inséré en retour.
<code>\$(element).prependTo(sélection)</code>	Insère un élément au début <u>de l'intérieur</u> de la sélection et fournit une référence sur l'élément inséré en retour.

**Exemple** : Ajouter un paragraphe à une **div** nommée *div1*.

```
function HtmlDynamique() {  
    $("<p>Paragraphe ajouté par la méthode appendTo</p>").appendTo("#div1");  
}
```

**Remarque** : Ces méthodes possèdent les variantes **after()**, **before()**, **append()** et **prepend()** qui inversent les deux parties : l'élément est précisé en argument et s'invoque sur une sélection, sans retourner de résultat.

**Exemple** : Idem avec la variante **append**.

```
$("#div1").append("<p>Paragraphe ajouté par la méthode append.</p>");
```

## Manipuler les attributs

Avec la méthode **attr()**, un attribut se lit si elle est appelée avec un seul argument permettant de spécifier le nom de l'attribut et se modifie en indiquant la valeur à affecter en 2<sup>ème</sup> argument :

### Exemples :

```
// Lire le titre du document
$(document).attr("title");

// Modifier le titre du document
$(document).attr("title", "Titre du document MAJ");
```

Plusieurs attributs peuvent être modifiés ou ajoutés s'ils n'existent pas, avec la syntaxe **JSON** :

### Exemple :

```
$("#lienW3").attr({
  title: "Apprendre le Web avec W3 School !",
  href: "http://www.w3schools.com/"
});
```

Le principe est identique avec les méthodes suivantes :

Méthode	Description
<b>css()</b>	Lit/modifie une ou un ensemble de propriétés CSS.
<b>width(), height()</b>	Lit/définit la largeur/hauteur de l'élément.
<b>position()</b>	Lecture des coordonnées de l'objet par rapport à son 1 <sup>er</sup> ancêtre positionné).
<b>html(), text()</b>	Lit ou définit le contenu html ou le texte de l'élément.
<b>val()</b>	Lit ou définit la valeur d'un l'élément de formulaire.

La suppression d'un attribut se réalise avec la méthode **removeAttr** en indiquant son nom en argument :

```
$("#lien").removeAttr("title");
```

**Remarque** : La modification d'un attribut qui n'existe pas le crée, tandis que la suppression d'un attribut qui n'existe pas est ignorée.



# AJAX

Une requête HTTP classique déclenche un aller-retour (requête-réponse) de l'intégralité de la page correspondante (avec éventuellement les fichiers d'images, CSS, Scripts, etc. associés en fonction de la gestion du cache du navigateur) alors que dans la plupart des cas, lorsque la page a déjà été chargée une 1<sup>ère</sup> fois, une partie seulement doit être actualisée.

Créé en 1998 par Microsoft, l'objet **XMLHttpRequest (XHR)** élimine ce problème avec des requêtes exécutées de manière asynchrones afin de pouvoir modifier le DOM uniquement avec les nouvelles données reçues.

Intégré à **ECMAScript** entre 2003 et 2005, cette technique a été adoptée par tous les navigateurs, sous l'acronyme de **d'AJAX (Asynchronous JavaScript and XML)**, qui a été remplacée depuis, par l'**API Fetch** intégrée à **ECMAScript 6**.

**Remarque** : Pour optimiser le volume des données échangées, le format XML a été progressivement remplacé par le format **JSON** depuis.

## Principe

Grâce à cet objet **XHR**, la mise en œuvre d'AJAX est relativement simple, mais diffère selon le navigateur utilisé.

**Exemple** : récupérer un bloc HTML situé dans un fichier [infos-ajax.html](#) (qui doit figurer dans le même domaine que la page) et l'afficher dans une div nommée *divMsg*.

```
function chargerFichier() {
    var xmlhttp;
    if (window.XMLHttpRequest) {
        // code pour IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
    }
    else { // code pour IE6, IE5
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    xmlhttp.onreadystatechange = function () {
        if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
            document.getElementById("divMsg").innerHTML =
                xmlhttp.responseText;
        }
    }
    // 2ème argument à true pour faire un appel asynchrone
    xmlhttp.open("GET", "infos-ajax.html", true);
    xmlhttp.send();
}
```

Cet exemple montre les propriétés **readyState** et **status** de l'objet XHR, permettant de connaître respectivement l'état de la requête et son résultat.

## Technologies AJAX et propriétés de l'objet XHR

[http://fr.wikipedia.org/wiki/Ajax\\_\(informatique\)](http://fr.wikipedia.org/wiki/Ajax_(informatique))

# Le Framework AJAX jQuery

AJAX est intégré à de nombreux Frameworks, notamment à **jQuery**, dont l'intérêt est d'en simplifier l'usage de manière cross-browser.

## Méthodes AJAX jQuery

Méthode	Description
<code>load()</code>	Récupère un fichier contenant un bloc HTML. Cette méthode (c'est la seule) s'applique à un sélecteur.
<code>get()</code> , <code>getJSON()</code>	GET HTTP pour récupérer un fichier ou des données au format spécifié (html, <b>xml</b> , <b>json</b> , <b>script</b> ) avec <b>get</b> et au format JSON avec <b>getJSON</b> .
<code>post()</code>	Idem à <b>get()</b> avec un POST HTTP.
<code>ajax()</code>	Méthode principale permettant de configurer l'échange en détail (méthode, format des données échangées, mise en cache de la réponse, callback d'erreur, etc.  En fait, toutes les méthodes précédentes font appel à la méthode <b>ajax()</b> , avec des configurations prédéfinies.

## La méthode `load()`

Cette méthode permet de récupérer un fichier contenant un bloc HTML avec un **GET** et de le placer dans l'élément ciblé par un sélecteur, avec la syntaxe suivante :

```
$(selector).load(URL[,data][,callback]);
```

L'URL du fichier doit être spécifiée en 1<sup>er</sup> argument (le seul obligatoire). Les arguments **data** et **callback** permettent respectivement de fournir des données et de prévoir une fonction de callback qui sera exécutée une fois le chargement effectué.

**Exemple** : récupérer un fichier nommé `aide.html` et l'afficher dans une div nommée `divMsg`.

```
$("#divMsg").load("aide.html");
```

**Exemple** : Même exemple, avec une fonction de callback de façon à afficher un message fourni par l'objet **XHR** transmis en argument, en cas d'erreur.

```
$("#divMsg").load("aide.html", null,
    function (response, status, xhr) {
        if (status == "error") {
            $("#divMsg").html("Erreur de chargement : " +
                xhr.status + " " + xhr.statusText);
        }
    });
```

## La méthode get()

Cette méthode émet également une requête **GET**, en offrant la possibilité de préciser le format ("**html**", "**xml**", "**json**" ou "**script**") des données à récupérer avec l'argument **dataType** :

```
$.get(URL[,data][,callback][,dataType]);
```

**Remarque** : Une fonction de callback est nécessaire pour récupérer des données.

**Exemple** : Même exemple que celui de la méthode **load()** :

```
$("#btnChargerHTML").click(function () {  
    $.get("aide.html", function (data) {  
        $("#divMsg").html(data);  
    });  
});
```

## La méthode post()

Calquée sur la méthode **get()**, la méthode **post()** utilise le verbe **HTTP POST** à la place de GET et s'emploie généralement pour s'affranchir de la limitation de taille des données envoyées avec un GET.

```
$.post(URL[,data][,callback][,dataType]);
```

## La méthode ajax()

Cette méthode offre toutes les options pour configurer les échanges avec le serveur. En réalité, les méthodes **load()**, **get()**, **getJSON()** et **post()** sont présentées dans la documentation comme étant des méthodes dites "**Shorthands**" (raccourcis) car elles y font appel, avec un ensemble d'options prédéfinies.

A titre d'exemple, les méthodes **get()** et **post()** sont équivalentes à :

<pre>\$.ajax({     type: "GET",     url: url,     data: data,     success: success,     dataType: dataType });</pre>	<pre>\$.ajax({     type: "POST",     url: url,     data: data,     success: success,     dataType: dataType });</pre>
Pour la méthode <b>get()</b>	Pour la méthode <b>post()</b>

## Les promesses

Les callbacks peuvent être remplacés par des promesses à partir de jQuery 2.x. Une promesse s'exécute de manière asynchrone et fournit une nouvelle promesse en retour, ce qui permet de les enchaîner avec une syntaxe dite "fluent".

L'exemple suivant montre comment appeler chacune des opérations en utilisant les promesses **done**, **fail** et **always** qui remplacent respectivement les callbacks **success**, **error**, et **complete**.

```
function afficherErreur(jqXHR, textStatus, errorThrown) {
    $("#divMsg").html("textStatus : " + textStatus + "<br/>" +
        "errorThrown:" + errorThrown + "<br/>" +
        "jqXHR.responseText" + jqXHR.responseText);
}

$("#btnCreer").click(function () {
    var article = { Designation: "Nouvel article" };

    $.ajax({
        type: "POST",
        url: "/api/articles",
        data: article,
        dataType: "json"
    })
    .done(function () {
        $("#divMsg").html("Article ajouté !");
    })
    .fail(afficherErreur);
});
```

## L'API Fetch

Toujours disponible, l'objet XHR a été remplacé et intégré à JavaScript avec l'API **Fetch**, similaire, indépendamment de jQuery.

Cette API se base nativement sur les promesses (objet **Promise** avec les méthodes **then()** ou **catch()**).

**Exemple** : récupérer un fichier image.

```
const request = new Request('/images/oiseau.jpg');
const image = document.querySelector('#image');

var btnRequete = document.getElementById("btnChargerImage");
btnRequete.addEventListener('click', () => { chargerImage(); });

function chargerImage() {
    fetch(request)
        .then(
            response => {
                if (response.status === 200) {
                    return response.blob();
                } else {
                    console.log('Le fichier n\'existe pas !');
                }
            }
        )
        .then(blob => {
            const objectURL = URL.createObjectURL(blob);
            image.src = objectURL;
            btnRequete.hidden = true;
        }).catch(error => {
            console.log(error + "!!!");
        });
}
```

La méthode accepte un objet **Request** en premier argument, avec un deuxième argument optionnel (généralement un objet de configuration JSON) permettant de configurer cette requête, notamment avec la méthode, des en-têtes spécifiques et des données (POST, PUT).

La réponse fournit une promesse dont la résolution est une requête fournissant un ensemble de méthodes permettant d'obtenir les données attendues (**text()** pour du texte, **json()** pour du JSON, **blob()** pour un fichier, etc.)

### Référence et exemples

[https://developer.mozilla.org/fr/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/fr/docs/Web/API/Fetch_API)

## Aperçu des API's WEB standard

Au-delà des enrichissements apportés HTML5 et CSS3, la branche **Web Applications Working Group** du W3 est chargée de proposer de spécifications pour un ensemble d'API's Web standards qui passent par les étapes de validations suivantes :

1. *Working Draft (WD)* (brouillon de travail),
2. *Last Call Working Draft* (dernier appel),
3. *Candidate Recommendation (CR)* (candidat à la recommandation),
4. *Proposed Recommendation (PR)* (recommandation proposée),
5. *W3C Recommendation (REC)* (recommandation du W3C)

Milestones					
Specification	FPWD	LC	CR	PR	Rec
DeviceOrientation Event Specification	Q2 2011	Q2 2014	Q4 2014	Q2 2015	Q3 2015
Geolocation API L2	Q2 2014	Q1 2015	Q3 2015	Q2 2016	Q2 2016
Note: This table will be updated periodically as the Working Group identifies new milestones.					

### API's courantes

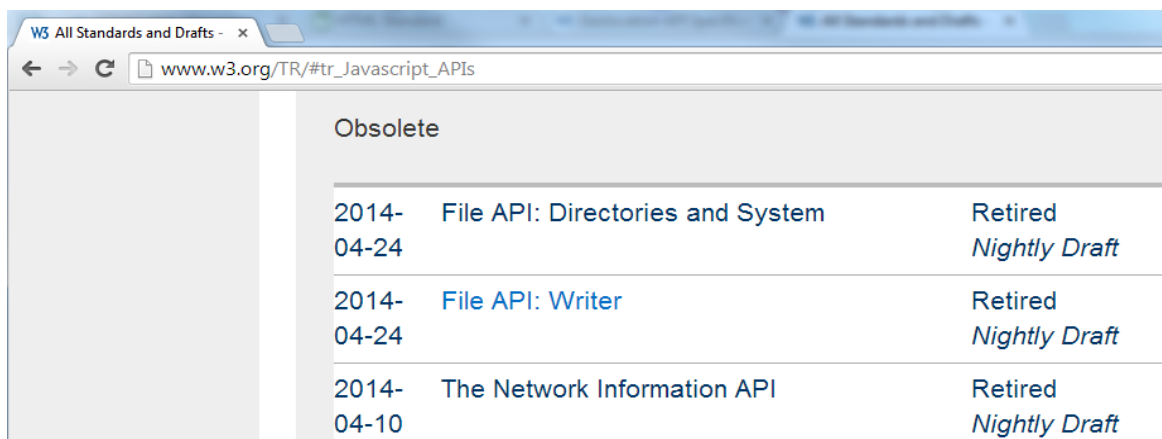
Nom API	Description
Canvas	Dessin de formes bitmap 2D et 3D créés avec du JavaScript.
SVG	Dessin vectoriel de formes 2D et 3D créées de manière déclarative.
Communication	Communication entres onglets ou iFrames.
Web Sockets	Communications bidirectionnelles entre le navigateur et le serveur.
Drag & Drop	Glisser déposer d'éléments dans la page.
Geolocalisation	Gestion de la localisation géographique.
Web Storage	Espace de stockage de données organisées en dictionnaires côté client.
IndexedDB	Espace de stockage d'objets indexés côté client.
Web Workers	Opérations exécutées en tâche de fond de manière asynchrone.
Cache	Navigation hors connexion.

### WhatWG

<http://www.whatwg.org/>

<https://html.spec.whatwg.org/multipage>

Certaines d'entre elles sont abandonnées telles que l'API **File**, partiellement supportée par **Chrome**, mais en règle générale, une spécification est utilisable à partir de la version **CR**.



The screenshot shows a web browser window with the address bar displaying 'www.w3.org/TR/#tr\_Javascript\_APIs'. The page content is titled 'Obsolete' and lists three entries:

Obsolete		
2014-04-24	File API: Directories and System	Retired Nightly Draft
2014-04-24	File API: Writer	Retired Nightly Draft
2014-04-10	The Network Information API	Retired Nightly Draft

## Etat des lieux des API's WEB

[http://www.w3.org/TR/#tr\\_Javascript\\_APIs](http://www.w3.org/TR/#tr_Javascript_APIs)

L'interface (format **IDL**) des objets rencontrés lorsque l'on utilise ces API's est décrite dans les spécifications que l'on trouve sur le site du **W3C** ou celui du **WhatWG**.

**Exemple** : Options disponibles avec les méthodes **getCurrentPosition** et **watchPosition** de l'API **Geolocation**.

### 5.2 PositionOptions interface

The [getCurrentPosition\(\)](#) and [watchPosition\(\)](#) methods accept [PositionOptions](#) objects as their third argument.

In ECMAScript, [PositionOptions](#) objects are represented using regular native objects with optional properties named [enableHighAccuracy](#), [timeout](#) and [maximumAge](#).

```
[NoInterfaceObject]
interface PositionOptions {
  attribute boolean enableHighAccuracy;
  attribute long timeout;
  attribute long maximumAge;
};
```

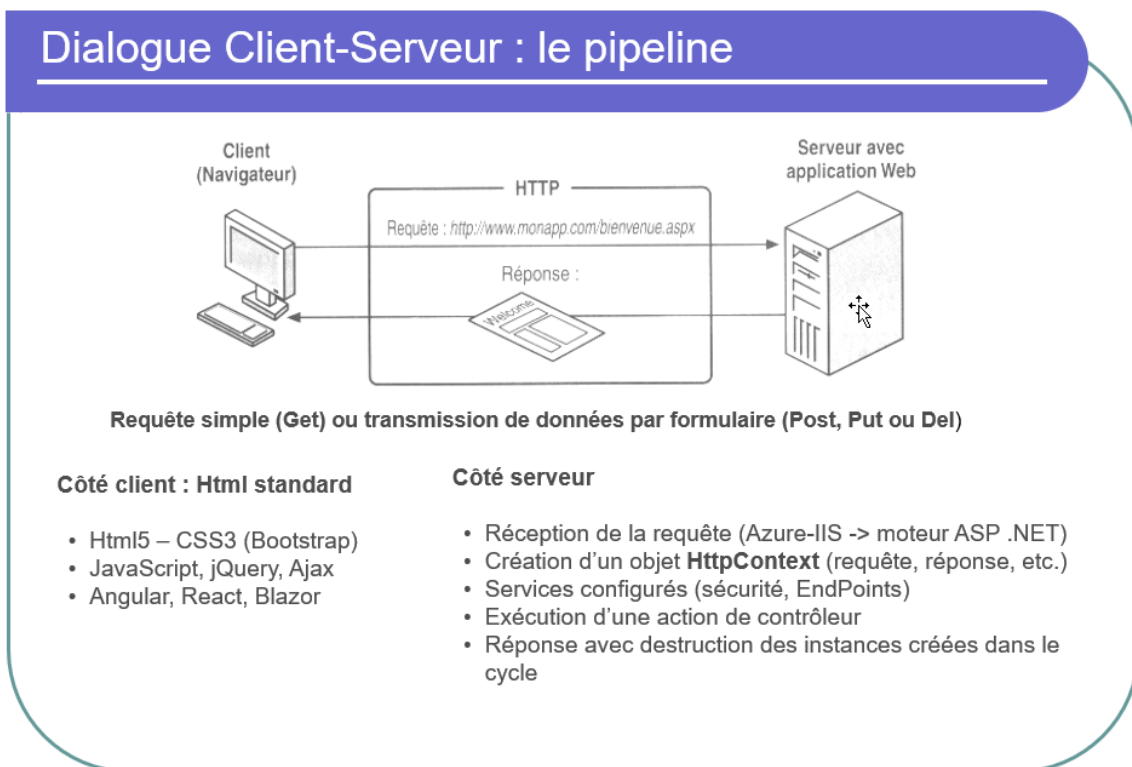
# Application MVC

## Introduction

### La notion de Pipeline

Ce type d'application se base sur le modèle d'architecture **MVC**, avec des classes de type POCO pour le modèle de données, des vues pour les interfaces HTML et des contrôleurs chargés d'interpréter les requêtes reçues par un navigateur afin d'exécuter des actions et renvoyer la vue correspondante au navigateur.

A partir de la réception d'une requête, jusqu'à l'envoi du rendu HTML au navigateur, elle est traitée par une succession d'étapes qui peuvent se configurer au démarrage de l'application. L'exécution du contrôleur fait ainsi partie de cette suite qui se déroule dans un ordre bien déterminé. On parle ainsi de **Pipeline** pour décrire cette suite d'étapes qui se répètent pour chaque requête, que l'on peut représenter sous forme de cycle.



Un contrôleur est capable de traiter différents types de requêtes (**Get**, **Post**, **Put**, **Del**) pour différentes vues. Il peut ainsi comporter un ensemble d'opérations nommées actions chargées de répondre à ces différents types de requêtes.

Ces quatre types de requêtes http permettent de réaliser les opérations d'édition désignées par l'acronyme **CRUD** (Create Read Update et Delete), avec les correspondances suivantes : **Get** pour une lecture, **Pos** pour une création, **Put** pour une mise à jour et **Del** pour une suppression.



## Organisation de l'application

L'application créée par défaut comporte un ensemble d'assemblés regroupés en deux frameworks `Microsoft.NetCore.App` et `Microsoft.AspNetCore.App`, qui peut être ensuite enrichie par des packages Nuget.

Il s'agit en réalité d'une application de type **Console**, avec une méthode `Main` dans un fichier `Program`, exécutant :

```
using IntroASPNET.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the DI container.
builder.Services.AddControllersWithViews();
var app = builder.Build();

var env = app.Environment.EnvironmentName;

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
}

app.UseDefaultFiles();
app.UseStaticFiles();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
```

L'environnement de l'application représentée par la classe **WebApplication** est créé par la méthode `CreateBuilder` afin de définir l'ensemble des services utilisés par l'application dans conteneur (propriété **Service**) lié à système d'injection de dépendances (DI) intégré ainsi que les différentes étapes (**Pipeline**) de traitement des requêtes reçues.

La génération produit une bibliothèque de classe qui sera utilisée sur un serveur et un exécutable permettant de lancer un serveur local, avec une URL de type <https://localhost:NNNN> où NNNN est le numéro de port à entrer dans la barre d'adresse d'un navigateur.

## Système d'injection de dépendances (DI)

Ce pattern consiste quant à lui, à réduire les dépendances entre les classes, en les remplaçant, idéalement par des interfaces afin de proposer les opérations attendues, indépendamment de toute implémentation (SOLID).

Un service ainsi configuré au démarrage s'obtient ensuite en précisant son interface dans le Ctor de la classe intéressée, notamment dans un contrôleur.

**Exemple** : Sans DI, la classe est liée (couplage) à l'implémentation de sa dépendance.

```
public class MainController {
    private readonly Service _sce;

    public MainController() {
        _sceData = new Service();
    }
}

public class MainController {
    private readonly IService _sce;

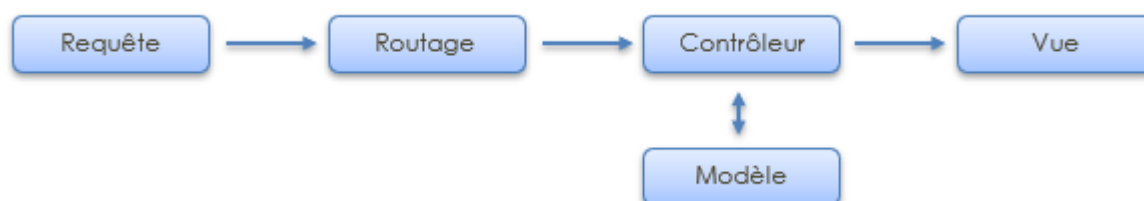
    public MainController(IService sce) {
        _sce = sce;
    }
}
```

La création des instances du service dépend de la méthode choisie lors de sa configuration, qui peut être :

- ❖ A chaque appel avec la méthode **AddTransient**.
- ❖ Pour le cycle de la requête avec la méthode **AddScoped**.
- ❖ Permanent avec la méthode **AddSingleton**.

# Configuration des routes

Dans une application ASP.NET MVC, l'accès aux actions des contrôleurs se réalise par le service **MapControllerRoute**.



L'exemple suivant montre comment définir une route nommée **"default"**, avec une url comprenant des segments optionnels s'ils ont une valeur par défaut ou suivis par un "?".

```
app.MapControllerRoute(  
    name: "default",  
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Cette route consiste à proposer des Urls formées de la manière suivante :

[www.site.com/Contrôleur/Action/Id](http://www.site.com/Contrôleur/Action/Id)

L'**Id** étant optionnel, l'action **Index** du contrôleur **Home** sera exécutée si rien n'est spécifié après le nom de domaine.

[www.site.com](http://www.site.com)

Le modèle MVC ASP.NET est basé sur un ensemble de conventions de nommage :

- i Les contrôleurs sont automatiquement invoqués par leur nom. Par exemple **ClientController** est automatiquement invoqué avec **Client** dans le segment **{controller}**.
- i Une action de contrôleur est invoquée automatiquement lorsque son nom figure dans le segment **{action}**.
- i Les vues doivent être placées dans un dossier **Views**, avec un sous-dossier par contrôleur. Les vues placées dans le sous-dossier **Shared** peuvent être partagées par plusieurs contrôleurs.
- i Une vue peut être typée avec un modèle de données spécifique. Dans ce cas, elle peut être générée automatiquement (Scaffolding) à partir de ce modèle.
- i Un contrôleur est une classe dérivée de **Controller** (EN **Microsoft.AspNetCore.Mvc**) comportant des méthodes retournant un **ActionResult**. Cette classe peut se dériver selon le type de données retourné par l'action (une vue, du **Json**, un **StatusCode** HTTP, etc.).
- i Un contrôleur fournit un objet **HttpContext** donnant accès au niveau HTTP (**Request**, **Response**, **Connexion**, etc.).

## Web API REST

Basée sur l'architecture **REST** (Representational State Transfer) ce type d'API sert à réaliser les opérations CRUD de manipulation de données en interprétant les verbes disponibles **GET** (lecture), **POST** (création), **PUT** (modification) et **DELETE** (suppression) pour une requête HTTP.

Dans une application ASP.NET, elle se crée à partir de la classe **ControllerBase** (dont la classe **Controller** dérive). Le principe est en effet le même, mais un contrôleur standard répond par des **ViewResult** tandis qu'un contrôleur d'API répond avec des données JSON par défaut.

Il a donc accès au **HttpContext** et peut recevoir des services par DI de la même façon qu'un contrôleur standard. La classe **ControllerBase** autorise la configuration de routes spécifiques avec des attributs.

Généré par Visual Studio, l'exemple suivant montre comment implémenter les opérations **CRUD** sur une liste de valeurs :

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET: api/<ValuesController>
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/<ValuesController>/5
    [HttpGet("{id}")]
    public string Get(int id)
    {
        return "value";
    }

    // POST api/<ValuesController>
    [HttpPost]
    public void Post([FromBody] string value)
    {
    }

    // PUT api/<ValuesController>/5
    [HttpPut("{id}")]
    public void Put(int id, [FromBody] string value)
    {
    }

    // DELETE api/<ValuesController>/5
    [HttpDelete("{id}")]
    public void Delete(int id)
    {
    }
}
```

L'attribut **Route** est ainsi défini au niveau de la classe de même qu'un attribut indique le verbe HTTP permettant d'atteindre chaque action.

## Syntaxe Razor

Côté serveur, une vue peut être statique (Html, CCS et JS exécuté par le navigateur), ou dynamique, notamment si elle affiche des données associées à un modèle de données. S'il s'agit d'une liste de données, il faut pouvoir la parcourir dans une boucle. Pour ce faire, une vue Razor est un fichier d'extension **.cshtml** qui peut comporter à la fois des éléments HTML et une logique écrite en **C#**, dans des blocs délimités par **@{ // code C# }**. De même il est possible de placer une donnée disponible dans la page derrière le caractère **@** (liaison de données).

### Exemples :

```
@foreach (var contact in Contacts)
{
    <p>Name: @contact.Nom</p>
}

<p>@DateTime.Now</p>
```

Une page peut également comporter des directives avec le caractère **@**, notamment pour déclarer un modèle (**@model**) afin de typer les données affichées et déclarer des espaces de noms (**@using**).

Il est également possible de créer des pages Razor suivant le même principe, mais contrairement aux vues Razor qui sont affichées en réponse d'actions de contrôleurs, une page peut s'afficher directement. Pour ce faire, une page razor doit comporter la directive **page** avec l'url relative de la page afin de la rendre accessible sans système de routage.

Elles sont notamment utilisées dans les applications **Blazor**. Dans une application MVC, elles sont disponibles en activant les services **AddRazorPages** et **MapRazorPages** dans le pipeline.

**Remarque** : Une vue/page peut également afficher des données en dehors d'un modèle, avec le dictionnaire **ViewData** ou de manière dynamique avec le **ViewBag** commun aux contrôleurs et aux vues/pages.

## Html et Tag Helpers

La création dynamique d'éléments Html standard se réalise grâce à des **HtmlHelpers** (EN **Microsoft.AspNetCore.Mvc.Rendering**) ou à des **TagHelpers** (EN **Microsoft.AspNetCore.Mvc.TagHelpers**).

Les premiers sont des méthodes de l'objet **@Html** :

```
@Html.DisplayFor(model => model.Ville)
```

Les seconds permettent d'introduire des attributs interprétés côté serveurs dans des tags Html standard :

```
<span asp-validation-for="Message"></span>
```

## Vues partagées

Les éléments communs à un ensemble de pages peuvent être factorisés de deux façons :

- ❖ Au niveau d'une page, avec un **Layout** (propriété de la classe **RazorPageBase**) contenant la structure globale de la page, avec la méthode **@RenderBody()** permettant d'indiquer l'emplacement du rendu de la vue.
- ❖ Avec des vues partielles qui peuvent contenir n'importe quel fragment HTML à placer dans différentes pages. L'emplacement de ces vues se déclare dans un **Layout** avec la méthode **@await RenderSectionAsync("NomSection", required: false)**. Une vue partielle peut être ensuite intégrée à une vue à cet emplacement, avec un bloc **@section NomSection** et la méthode **RenderPartialAsync("NomVuePartielle", required: false)**.

**Exemple :**

```
@section Scripts {  
    @{await Html.RenderPartialAsync("_NomVuePartielle");}  
}
```

Les vues partagées se situent dans le sous-dossier **Views/Shared** et sont préfixées par **"\_"** par convention.

Il est ainsi possible de plusieurs Layout, mais le fichier **\_ViewStart.cshtml** permet de définir un Layout par défaut.

De même, étant donné qu'une page razor peut contenir du code C#, les espaces de noms communs aux différentes Vues/Pages peuvent être factorisés dans le fichier **\_ViewImports.cshtml**.

# Validations des données

L'espace de noms **System.ComponentModel.DataAnnotations** sert de nouveau ici pour réaliser des validations, aussi bien dans les vues générées côté client que dans le code des actions POST et PUT côté serveur.

**Remarque** : Si la partie validation doit être séparée de la définition du modèle, il est possible de créer une classe intermédiaire (principe du [ViewModel](#)).

**Exemple** :

```
public class ContactViewModel
{
    [Required]
    [StringLength(20, MinimumLength = 5)]
    public string Nom { get; set; }
    [Required(ErrorMessage = "Email requis")]
    [EmailAddress]
    public string Email { get; set; }
    [Required]
    [MaxLength(30)]
    public string Sujet { get; set; }
    [Required]
    [MaxLength(100)]
    public string Message { get; set; }
}
```

**Remarque** : La propriété **ErrorMessage** peut être remplacée par la propriété **ErrorMessageResourceName**.

Les validations peuvent être intégrées aux vues avec les attributs **asp-validation-summary** **asp-for** et **asp-validation-for** des tags helpers :

```
@model ContactViewModel
@{ ViewBag.Title = "Nous contacter"; }
<h1>Contact</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <label asp-for="Nom">Votre nom :</label>
    <input asp-for="Nom" name="nom" />
    <span asp-validation-for="Nom"></span>
    <label asp-for="Email">Email :</label>
    <input asp-for="Email" name="email" />
    <span asp-validation-for="Email"></span>
    <label asp-for="Sujet">Sujet :</label>
    <input asp-for="Sujet" name="sujet" />
    <span asp-validation-for="Sujet"></span>
    <label asp-for="Message">Message :</label>
    <input asp-for="Message" name="message" />
    <span asp-validation-for="Message"></span>
    <input type="submit" value="Envoyer" />
</form>
```

Ensuite, les validations doivent toujours être vérifiées côté serveur, grâce à la propriété **ModelState.IsValid** du contrôleur :

```
[HttpPost("contact")]
public IActionResult Contact(ContactViewModel modele)
{
    if (ModelState.IsValid)
    {
        _messengerie.EnvoyerEmail($"{modele.Sujet},{modele.Message} de {modele.Nom}-{modele.Email}", "webmaster@app.com");
        ModelState.Clear();
        ViewBag.Message = "Message envoyé !";
    }
    return View();
}
```

## Validation unobtrusive

La validation des données reçues peut être optimisée avec des scripts exécutés côté client, fournis par jQuery, de manière unobtrusive, c'est à dire sans à avoir à placer les scripts dans la page, mais dans des fichiers séparés.

```
@section Scripts {
    <script
        src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
    <script
        src="~/lib/jquery-validation-
            unobtrusive/jquery.validate.unobtrusive.js"></script>
}
```

**Remarque** : Ces scripts sont intégrés par défaut et disponibles séparément sous forme de packages Nuget **jquery-validation** et **jquery-validation-unobtrusive**.



## Protections contre les attaques

Différents types d'attaques peuvent avoir lieu en Web, notamment par :

Nom	Description
<b>XSS</b>	Scripts malveillants.
Solution	Les données affichées par la syntaxe Razor avec @ sont automatiquement encodées si elles contiennent des caractères tels que "<" et ">".
<b>XSRF ou CSRF</b>	Scripts intersites injectés dans une page à partir de saisies non validées.
Solution	Un champ caché nommé <b>"_RequestVerificationToken"</b> est automatiquement ajouté dans les pages contenant un formulaire. Dans les autres cas, l'attribut <b>[ValidateAntiForgeryToken]</b> peut être ajouté explicitement sur l'action à protéger.
<b>SurPost</b>	Réception de données supplémentaires non désirées par un POST.
Solution	Spécifier la liste des noms des données attendues avec l'attribut <b>[Bind]</b> dans la signature de l'action.
<b>Injection SQL</b>	Par des chaînes SQL créées par concaténations à partir de saisies utilisateurs.
Solution	Paramétrage des requêtes ou procédures stockées.
<b>HTTP</b>	Non sécurisé par défaut.
	HTTPS avec ajout d'une redirection systématique au début du pipeline avec <b>app.UseHttpsRedirection()</b>

## Authentification interne par Identity

L'authentification des utilisateurs est prise en charge par ASP.NET Core Identity, sous forme d'API couplée aux vues permettant aux utilisateurs de se connecter.

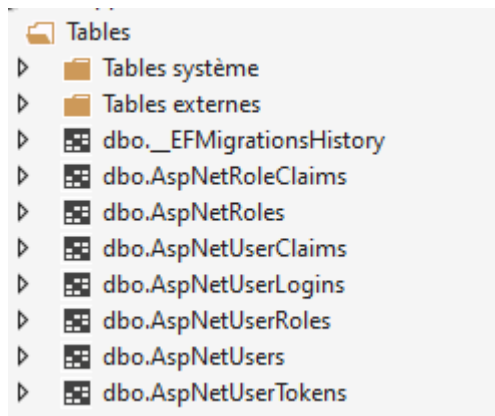
Cette Api gère les utilisateurs avec leurs mots de passe chiffrés, des informations de profil et des rôles. Elle permet d'utiliser un fournisseur de connexions externe, tels que Facebook, Google, Twitter et Microsoft, ou interne dans une base de données SQL proposée par défaut.

Ce dernier s'obtient par l'option **Comptes individuels** proposée dans le **Type d'authentification** par Visual Studio à la création d'un projet. Elle s'appuie sur Entity Framework, ainsi que sur les packages Nuget :

- ❖ `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- ❖ `Microsoft.AspNetCore.Identity.UI`

**Remarque** : Le package `Microsoft.EntityFrameworkCore.Tools` est également requis pour obtenir le scaffolding de vues en général et pour les vues liées à l'Api Identity générées automatiquement dans le cas contraire ([/Identity/Account/Login](#) [/Identity/Account/Logout](#), etc.)

Un contexte dérivé de **IdentityDbContext** est alors proposé, avec un script de migration permettant de générer la base, qui peut être complétée avec un modèle de données personnalisé. Une fois la base générée, les tables nécessaires sont prêtes à l'emploi.



Le service **Identity** s'active ensuite dans le pipeline, en ajoutant les deux méthodes suivantes avant les **Endpoints** :

```
app.UseAuthentication();  
app.UseAuthorization();
```

L'API se présente ensuite avec cet ensemble de classes :

Classe	Description
<b>IdentityUser</b>	Représente un utilisateur ( <b>Id</b> , <b>UserName</b> , <b>Email</b> , etc.). Cette classe peut être dérivée pour ajouter des informations de profil spécifiques.
<b>userManager</b>	Gestion des utilisateurs (création, suppression, changement d'email, recherche par nom ou par email, etc.)
<b>SignInManager</b>	Gestion des connexions (méthodes <b>SignIn</b> , <b>SignOut</b> , <b>LockedOut</b> , etc.)

**Remarque** : La classe **HttpContext** contient la propriété **User** de type **ClaimsPrincipal** dont la propriété **Identity** permet à son tour de savoir s'il est authentifié (propriété **IsAuthenticated**) et si c'est le cas, son nom.

L'accès aux actions de contrôleurs peut être restreint de manière déclarative avec les attributs **[Authorize]** et **[AllowAnonymous]** (espace de noms **Microsoft.AspNetCore.Authorization**). L'attribut **[Authorize]** peut être également placé sur le contrôleur pour restreindre l'accès à l'ensemble de ses actions.

## Persistence des données de session

Des données peuvent être mémorisées côté serveur pour l'ensemble des pages consultées par un navigateur, dans des sessions. Pour ce faire, les packages Nuget **Microsoft.AspNetCore.Session** doit être installé et le service **app.UseSession()** placé dans le pipeline avant celui des routes, éventuellement configuré au préalable.

**Exemple** : Configurer le délai d'expiration des sessions à 3 minutes après la dernière requête.

```
services.AddSession(options => {
    options.IdleTimeout = TimeSpan.FromMinutes(3);
});
```

Un **Id** est automatiquement attribué à chaque session, accessible par la propriété **Session** du **HttpContext**, afin de mémoriser un dictionnaire de données (**Int** ou **string**) avec une clé de type **string** (un nom) pour chaque donnée, avec les méthodes **SetInt32**, **SetString**, **GetInt32** et **GetString**.

Cette collection peut être remise à zéro avec la méthode **Clear** et une session peut être supprimée par la méthode **Remove(SessionId)**.

## Journalisation

L'environnement configuré par défaut au démarrage de l'application avec **Host.CreateDefaultBuilder()** ajoute automatiquement un service de journalisation dans la fenêtre de sortie (**Console**) du projet. Il peut être supprimé avec la méthode **ClearProviders** et remplacé par d'autres systèmes avec l'implémentation de l'interface **ILogger**, (espace de noms **Microsoft.Extensions.Logging**) configurés en tant que service DI.

Il s'obtient avec une injection par Ctor avec le nom de la classe afin de préciser la catégorie des informations à enregistrer.

```
private readonly ILogger<HomeController> _logger;

public HomeController(ILogger<HomeController> logger)
{
    _logger = logger;
}
```

Ces informations à enregistrer peuvent être classifiées par niveau (**Trace**, **Déboguer**, **Informations**, **Avertissement**, **Erreur** ou **Critique**) avec les méthodes correspondantes.

## Filtrage et gestion des exceptions

L'espace de noms **Microsoft.AspNetCore.Mvc.Filters** permet de créer des filtres qui pourront être appliqués sous forme d'attributs à une action ou à un contrôleur. Il propose des attributs prêts à l'emploi et des interfaces qui peuvent être implémentées de manière personnalisée.

L'interface **ExceptionHandler** permet par exemple d'implémenter la méthode **OnException** :

**Exemple** : Créer un filtre pour enregistrer les erreurs dans un fichier.

```
public class LogExceptionHandler : IExceptionHandler {
    private readonly IWebHostEnvironment _env;

    public LogExceptionHandler(IWebHostEnvironment hostingEnvironment) {
        _env = hostingEnvironment;
    }

    public void OnException(ExceptionContext context) {
        var ex = context.Exception;

        string infosLog = $"{DateTime.Now}{Environment.NewLine}{ex.GetType()}
                           {Environment.NewLine}{ex.Message}{Environment.NewLine}";
        infosLog += "-".PadRight(150, '-') + Environment.NewLine;

        using var sw = new StreamWriter(Parametres.FichierLogEx, true);
        sw.Write(infosLog);
    }
}
```

# Préparation du déploiement

## Gestion des configurations du projet

L'interface **IWebHostEnvironment** reçue par injection de dépendance dans la méthode **Configure** de la classe **Startup** fournit un ensemble de booléens **IsDevelopment**, **IsStaging**, **IsProduction** permettant de configurer le pipeline en conséquence.

La valeur de ce booléen se définit avec les **Variables d'environnement** disponibles dans l'onglet **Déboguer** des propriétés du projet. D'autres valeurs peuvent être renseignées si besoin et testées avec la méthode **IsEnvironnement** ou la propriété **EnvironmentName**.

## Configuration des pages d'erreurs

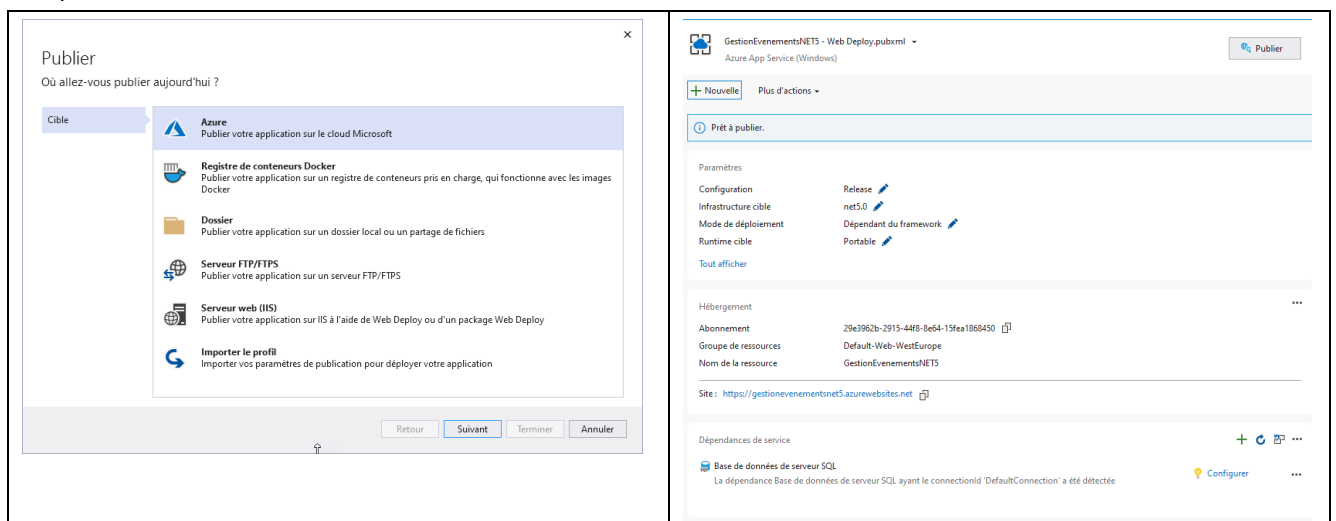
Si l'environnement n'est pas celui de développement, la page d'erreur à afficher à l'utilisateur doit être activée par la méthode **app.UseExceptionHandler**, avec une route (généralement **/Home/Error**) dédiée à l'affichage d'une page sans aucune information technique.

Le service **ILogger** reçu par le contrôleur **Home** par défaut, permet d'enregistrer les informations concernant l'erreur récupérée par la méthode :

```
HttpContext.Features.Get<IExceptionHandlerPathFeature>()
```

## Publication

Une fois l'application générée en mode release et les paramètres renseignés selon l'environnement cible, différents profils de déploiements sont proposés par l'assistant de publication de Visual Studio.



L'application peut être notamment déployée de manière autonome (avec toutes ses dépendances), notamment dans un conteneur **Docker**, ou liée à un environnement existant sur la machine cible.

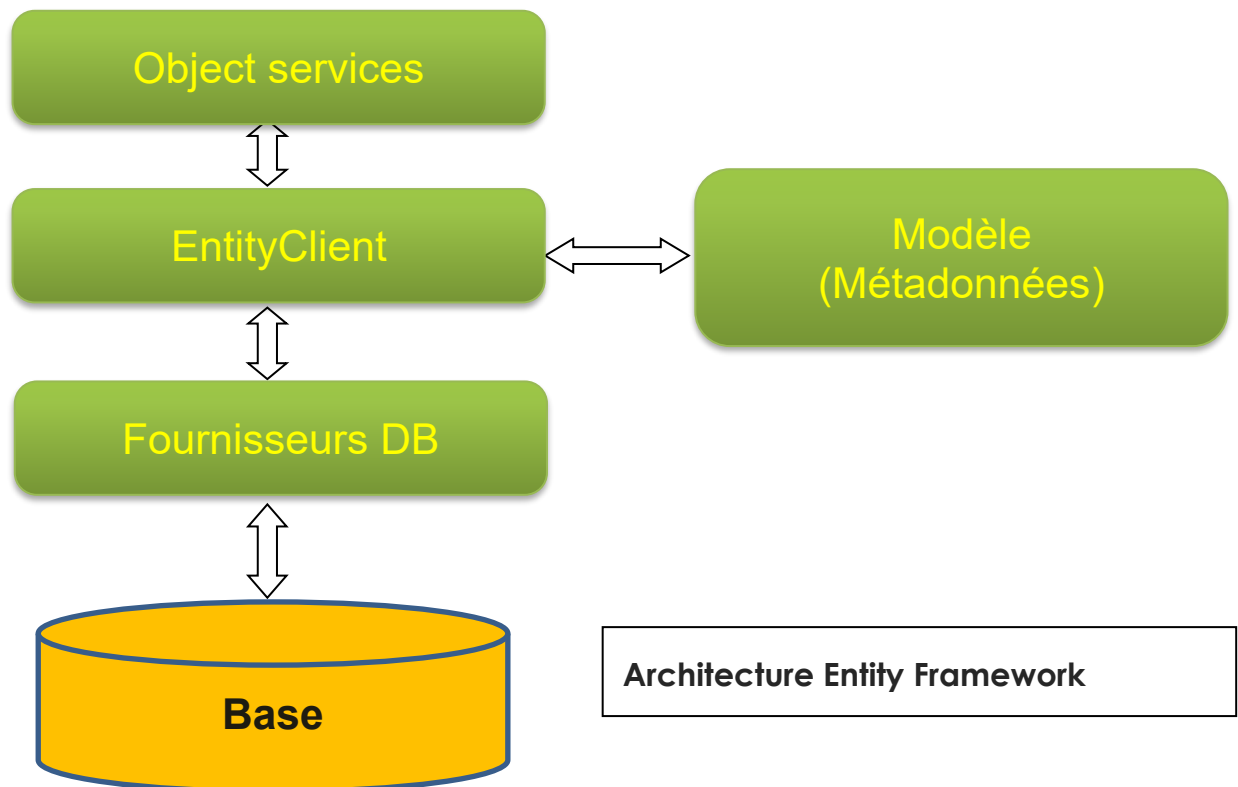
Une fois le profil créé, la publication peut être lancée, en remplaçant l'ensemble des éventuels fichiers préexistants ou seulement les fichiers plus récents.

# Entity Framework Core

Entity Framework est la principale solution **ORM** (Object Mapping Relational) proposée en .NET, intégrée par défaut aux applications ASP.NET Core, avec le support de nombreux fournisseurs de données (**Oracle**, **SQL Server**, **DB2-Informix**, **PostgreSQL**, **MySQL**, **Cosmos DB**, **Sqlite**, **Firebird**, **Access**, etc.).

## Introduction

- ❖ Mapping riche (1-N, N-M, 1-1, héritage, types complexes, énumérations, etc.).



- Le niveau **Object Services** assure la gestion des changements réalisés sur les entités (tracking) chargées en mémoire.
- Le niveau **EntityClient** se charge du modèle de données (mapping entre la base et les classes d'entités).

## Définitions

- ❖ Une entité est une classe comportant des données, avec une clé logique mise en correspondance avec la clé primaire de la table sous-jacente.
- ❖ Une entité est une classe dite **POCO** (Plain Old CLR Object), c'est à dire indépendante de EF.

## Fonctionnalités

- ❖ Possibilité de générer la base à partir du modèle, en se basant sur des conventions pour le nommage et la définition des contraintes (clés, existence, intégrité, etc.).
- ❖ Possibilité de mapping pour les autres situations.
- ❖ Possibilité de générer le modèle à partir d'une base existante.
- ❖ Entity Framework est en open-source, disponible sous la forme d'un package NuGet.
- ❖ Fonctionnement en mode déconnecté, avec génération SQL dynamique pour les accès aux données, avec trace active par défaut en développement.
- ❖ Support du pattern [Async/Await](#).

### Principaux packages Nuget :

- ❖ [Microsoft.EntityFrameworkCore](#)
- ❖ [Microsoft.EntityFrameworkCore.Provider](#) (SqlServer, SqlLite, Cosmos, ..)
- ❖ [Oracle.EntityFrameworkCore](#) pour Oracle.
- ❖ [Microsoft.EntityFrameworkCore.Tools](#) pour les outils de migration.
- ❖ [Microsoft.EntityFrameworkCore.Proxies](#) pour le LazyLoading.

## Définition du modèle

La définition du modèle se réalise à partir d'un ensemble de classes POCO associées à un **DbContext** chargé de leur persistance dès lors qu'elles sont référencées par des **DbSet**.

Exemple d'un modèle [ArticlesDbContext](#) avec une entité :

```
using Microsoft.EntityFrameworkCore;

public class ArticlesDbContext : DbContext {
    public DbSet<Article> Articles { get; set; }
}

public class Article
{
    public int ArticleId { get; set; }
    public string Designation { get; set; }
}
```

## Générer la base

Le contexte propose un ensemble de méthodes virtuelles permettant de le configurer.

**Exemple** : définir la chaîne de connexion.

```
public class ArticlesDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
                                   optionsBuilder) {
        optionsBuilder.UseSqlServer(@"Data Source=(localdb)\MSSqlLocalDb;
                                   Initial Catalog=DbArticles");
    }
}
```

Si la base n'existe pas, elle peut être créée par la commande de migration **update-database** ou automatiquement par code avec la méthode **EnsureCreated()** de la propriété **Database** du contexte.

```
class Program
{
    static void Main(string[] args)
    {
        using (var context = new ArticlesDbContext())
        {
            context.Database.EnsureCreated();
        }

        Console.WriteLine("La base existe !");
        Console.Read();
    }
}
```

## Alimenter la base à sa création

De même, la méthode **OnModelCreating** permet de configurer le modèle de manière spécifique par rapport aux conventions appliquées par défaut et d'initialiser la base avec un ensemble de données exprimées sous forme d'instances d'entités avec la méthode **modelBuilder.Entity<T>().HasData()**.

**Exemple** : Créer trois articles s'ils n'existent pas.

```
modelBuilder.Entity<Article>().HasData(art1, art2, art3);
```

L'existence de données s'effectue par la présence ou non de la clé de chaque entité en base.

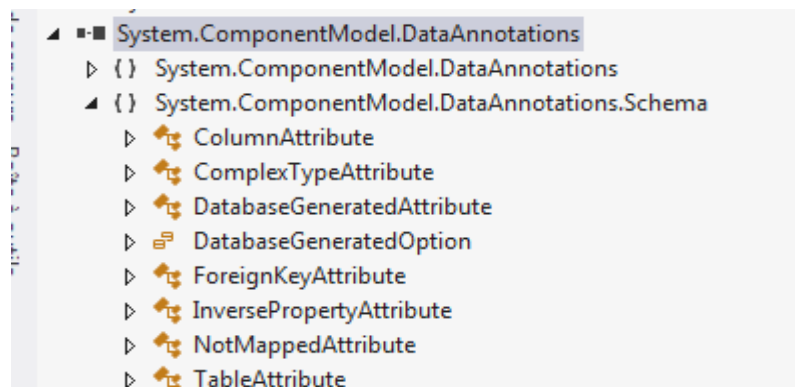


## Conventions

Par défaut, le schéma de la base générée s'appuie sur un ensemble de conventions à partir des entités du modèle :

- ❖ Nom des tables pluralisé (EN)
- ❖ Clés (**Identity**) obtenues par la base d'après les colonnes dont le nom est *NomEntiteId* ou *Id*
- ❖ Correspondances des types / fournisseur
- ❖ Type string : **nvarchar(MAX) NULL**
- ❖ Types valeurs **NOT NULL** (NULL pour nullables)
- ❖ Type **byte[]** : **varbinary(MAX)**

Ces conventions peuvent être ensuite configurées soit par les annotations disponibles dans l'espace de noms **System.ComponentModel.DataAnnotations**, soit par code avec **l'API Fluent** après avoir installé le package Nuget de même nom.



Exemple d'annotations :

```
[Table("TableArticles")]
public class Article {
    public int ArticleId { get; set; }
    [Required]
    [StringLength(50)]
    public string Designation { get; set; }
    [StringLength(300)]
    public string Description { get; set; }
    public Categorie Categorie { get; set; }
    public decimal Prix { get; set; }
    [Column("Promo")]
    public bool Promotion { get; set; }
    [Column(TypeName = "Image")]
    public byte[] Photo { get; set; }
}
```

Pour obtenir davantage de souplesse, **l'API Fluent** autorise la configuration du mapping par programmation, en redéfinissant la méthode **OnModelCreating** du contexte, dont l'argument **ModelBuilder** permet de configurer le modèle de données.

### Exemple : configuration des propriétés par l'Api Fluent

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Article>().
        Property(p => p.Designation).IsRequired();
    modelBuilder.Entity<Article>().
        Property(p => p.Designation).HasMaxLength(50);
}
```

Cette approche peut convenir pour un petit modèle, mais à partir d'une certaine taille, il est préférable de créer une configuration par entité à l'aide d'une classe de type **IEntityTypeConfiguration<TEntity>**, qu'il suffit ensuite d'ajouter à la collection **Configurations** du *modelBuilder* :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new CategorieConfiguration());
    // ...
}

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

class CategorieConfiguration : IEntityTypeConfiguration<Categorie>
{
    public void Configure(EntityTypeBuilder<Categorie> builder)
    {
        builder.Property(p => p.LibelleCategorie).IsRequired();
        builder.Property(p => p.LibelleCategorie).HasMaxLength(50);
    }
}
```

## Configuration des tables

Si le nom d'une entité doit être différent de celui de la table correspondante, il est possible de :

- ❖ Préciser le nom de la table à générer s'il doit différer de celui de l'entité avec l'attribut/méthode **Table/ToTable**.
- ❖ Soustraire une classe au modèle avec l'attribut/méthode **NotMapped/Ignore<NomClasse>**.

# Configuration des propriétés

## Nom de colonne

Une propriété peut être associée à un nom de colonne différent avec l'attribut/méthode **Column/HasColumnName**, via l'espace de noms **System.ComponentModel.DataAnnotations.Schema**.

Exemple :

<pre>[Column("Promo")] public bool Promotion { get; set; }</pre>	<pre>Property(p =&gt; p.Promotion).     HasColumnName("Promo");</pre>
--	---

## Type de colonnes

Possibilité de spécifier un type de colonne compatible avec l'attribut/méthode **Column/ HasColumnType** :

Exemple : convertir un `byte[]` en type `Image` :

<pre>[Column(TypeName = "Image")] public byte[] Photo { get; set; }</pre>	<pre>Property(p =&gt; p.Photo).     HasColumnType("Image");</pre>
---	---

## Clé

Les types de données acceptés pour une clé sont les types primitifs (généralement `int`, `string` ou `Guid`).

La configuration d'une clé est utile lorsque le nom de la colonne de clé de l'entité ne correspond pas à la convention (propriété nommée *NomEntiteId* ou *Id*), et se réalise avec l'attribut/méthode **Key/HasKey**.

Exemple :

<pre>[Key] public string CodeArticle { get; set; }</pre>	<pre>Entity&lt;T&gt;().     HasKey(p =&gt; p.CodeArticle)</pre>
--	---

## Existence

Par convention, la valeur NULL est autorisée pour les types référence et pour les types valeur nullables.

Une colonne de type référence (ainsi que le type `string`) peut être rendue obligatoire avec l'attribut/méthode **Required/IsRequired**.

Exemple :

<pre>[Required] public string CodeArticle { get; set; }</pre>	<pre>Entity&lt;T&gt;().     IsRequired(p =&gt; p.CodeArticle)</pre>
---	---

## Type string

Configuration	Description
Convention	MAX
Annotation	MinLength( <i>n</i> ) - MaxLength( <i>n</i> ) - StringLength( <i>n</i> )
Fluent	Property( <i>t</i> => <i>t</i> .PropertyName).HasMaxLength( <i>n</i> )

### Remarques :

- ❖ Les attributs **MaxLength** et **StringLength** sont équivalents.
- ❖ L'attribut/méthode **MinLength/HasMinLength** sert à faire de la validation, sans affecter le schéma de la base.
- ❖ L'API Fluent propose la méthode **IsUnicode(bool)** permettant de créer une colonne qui n'est pas Unicode (sans équivalent par les annotations).

## Colonne de type Timestamp

Ce type de colonne doit être configuré explicitement sur une unique propriété de type **byte[]** par entité, avec l'attribut **TimeStamp** ou la méthode **IsRowVersion**.

Ce type de colonne sert généralement à gérer les accès concurrentiels optimistes en ajoutant la valeur de la colonne à la clause **WHERE** de chaque **Update**.

## Type complexe

Plusieurs colonnes peuvent être regroupées sous la forme d'un type complexe, qui peut être réutilisé sur différentes entités. Il se définit par une classe dotée des caractéristiques suivantes :

Un type complexe

- ❖ Ne doit pas comporter de clé.
- ❖ Ne peut contenir que des types primitifs.
- ❖ Ne peut pas être utilisé dans une collection référencée par une propriété d'une autre entité.

**Conseil** : Créer une instance du type complexe dans le Ctor de la classe parente pour éviter d'avoir à gérer les nulls.

### Notes

# Configuration des relations

## Relations 1-N

Une relation [0-1]-N est créée lorsqu'il existe une propriété **List<T>** du côté principal (entité principale comportant la clé primaire) et une référence sur l'autre côté (entité dépendante) de la relation.

Dans ce cas une colonne (clé externe) nommée **[NomEntitePrincipale]\_[NomCléEntitePrincipale]** (*Categorie\_CategorieId* dans l'exemple) est ajoutée automatiquement à la table dépendante contenant la clé externe.

Les collections d'entités dépendantes (côté 1) et la référence vers l'entité principale (côté N) sont des propriétés de navigation permettant de charger les entités connexes.

Une relation 1-N peut être obtenue en imposant la présence d'une référence avec l'attribut **Required**.

Avec l'API Fluent, la multiplicité s'exprime de manière plus générale avec la méthode **Has[Multiplicité](Pté)** suivie de **With[Multiplicité](Pté)**. La méthode **IsRequired** permet de préciser si une référence est obligatoire.

**Exemple :** Rendre la catégorie obligatoire dans la classe *Article*.

```
modelBuilder.Entity<Article>()  
    .HasOne(p => p.Categorie)  
    .WithMany(b => b.Articles)  
    .IsRequired();
```

**Conseil :** Déclarer explicitement la clé externe dans le modèle avec le nom *EntitePrincipaleID* pour qu'elle soit reconnue en tant que telle à la place de la clé générée automatiquement. Dans ce cas, la présence d'une référence sur l'entité principale peut se gérer avec la nullabilité sans qu'il soit nécessaire de configurer la multiplicité.

**Attention !** Une référence obligatoire active automatiquement la suppression en cascade des entités associées. Ce comportement par défaut peut être désactivé par l'API Fluent uniquement, à partir d'une relation configurée, suivi de la méthode **OnDelete** appelée avec une valeur de l'énum **deleteBehavior**.

**Exemple :**

```
modelBuilder.Entity<Article>()  
    .HasOne(p => p.Categorie)  
    .WithMany(b => b.Articles)  
    .OnDelete(DeleteBehavior.SetNull);
```

Avec les valeurs **NoAction** et **ClientSetNull**, une tentative de suppression d'une entité principale avec des entités associées provoquera une exception.

## Relations 1-1

Ce type de relation est créé lorsqu'il existe une référence croisée entre 2 entités partageant la même clé. Entity Framework crée dans ce cas une relation de dépendance entre les deux entités, avec une clé externe générée automatiquement qui peut être configurée explicitement avec les méthodes **HasOne** suivie de **WithOne**, suivie de **HasForeignKey**.

## Relations N-M

Ce type de relation est créé lorsqu'il existe une collection **List<T>** croisée entre 2 entités. Par convention, la table intermédiaire générée se nomme **EntiteNEntiteM**, et contient une paire de clé nommées **NomEntite\_NomCle**. Cette convention peut être redéfinie par l'API Fluent, avec la méthode **HasOne**, suivie de **WithMany**, puis **HasForeignKey** pour chaque entité.

**Conseil** : Créer une entité intermédiaire avec la clé externe et la référence des deux entités connexes.

### Notes

---

# Configuration des hiérarchies

## TPH (Table Par Hierarchie)

Ce type de hiérarchie s'obtient par convention, en dérivant une entité existante.

**Exemple :**

```
public class Promotion : Article
{
    public DateTime DateDebut { get; set; }
    public DateTime DateFin { get; set; }
}
```

Le schéma ainsi obtenu en base se fonde sur une seule table, à laquelle un champ discriminant nommé **Discriminator** de type **nvarchar** est ajouté. A sa création, le type de l'entité est automatiquement stocké dans ce champ afin de différencier le type de la classe de base et celui des classes dérivées.

Seule l'API Fluent autorise le choix du nom de cette colonne avec la méthode **HasDiscriminator**, éventuellement suivie par la méthode **HasValue** afin d'indiquer la valeur associée à chaque type.

Les entités dérivées s'obtiennent avec la méthode **OfType** :

```
var promos = contexte.Voyages.OfType<Promo>();
```

## L'héritage TPT (Table Par Type)

Cette hiérarchie se crée de manière analogue à celle du **TPH**, en décorant la sous-classe de l'attribut **Table** (ou par la méthode **ToTable** avec l'API Fluent), de façon à obtenir une table par classe, avec ajout automatique de la clé de la classe de base dans la table correspondant à la sous-classe.

**Même exemple avec le TPT :**

```
[Table("Promotions")]
public class Promotion : Article
{
    public DateTime DateDebut { get; set; }
    public DateTime DateFin { get; set; }
}
```

En cas d'ajout d'une entité dérivée, l'entité principale est d'abord créée pour en récupérer la clé afin de pouvoir créer ensuite l'entité dérivée.

Pour le requêtage, on obtient les entités dérivées grâce à la méthode **OfType** de l'entité de base :

```
var croisieres = contexte.Voyages.OfType<Croisiere>();
```

## Principe du SQL dynamique

Les opérations de chargement et d'édition des données sont réalisées par l'intermédiaire du contexte, qui se charge de générer automatiquement les instructions SQL reconnues par le fournisseur sous-jacent.

### Interface **IQueryable<T>**

Les données sont chargées en mémoire grâce aux **DbSet**(s) associés au contexte, qui implémentent les interfaces **IQueryable<T>** et **IEnumerable<T>**, où **T** représente une classe d'entité.

- ❖ Grâce à l'interface **IEnumerable<T>**, il est possible de traiter un **DbSet** comme une séquence, en appliquant les opérateurs de **LINQ To Objects**.
- ❖ Les requêtes **LINQ To Entities** retournent en réalité des objets de type **IQueryable<T>**, qui représentent une requête typée sur une entité du modèle.

### Paramétrage et exécution

- ❖ Si les paramètres des requêtes fournis par l'application sont des variables, ils sont automatiquement convertis en paramètres SQL (et non pas sous forme de chaînes de caractères) afin d'éviter les problèmes d'injection SQL.
- ❖ Les requêtes (projections, restrictions, etc.) effectuées sur un **DbSet** sont exécutées côté serveur.
- ❖ Les opérateurs d'agrégation sont à exécution immédiate.
- ❖ Les autres sont exécutés de manière différée. Aucune connexion n'est ouverte avant l'exécution de la requête (objet **IQueryable<T>**) avec un opérateur tel que **ToList()**, une liaison de données ou une itération, en générant dynamiquement l'instruction SQL à exécuter en base.
- ❖ Entity Framework fonctionne de manière déconnectée. Chaque opération (ou ensemble d'opérations) de lecture ou de mise à jour doit se faire avec une connexion spécifique (c'est à dire un contexte) ouverte et refermée automatiquement aussitôt l'opération terminée.

#### Notes



## LINQ To Entities

**Exemple :** LINQ To Entities (espace de noms **System.Linq**)

```
var req = contexte.Voyages.Where(v => v.Categorie.CategorieId == 1);  
List<Voyage> resultat = req.ToList();
```

Cet exemple montre la définition de la requête, suivie de son exécution par l'opérateur Linq **ToList()**.

## La méthode Find

Un **DbSet** propose la méthode **Find** qui simplifie l'usage de la méthode **SingleOrDefault**, avec un argument correspondant à la clé (ou à une liste de valeurs pour une clé multiple) de l'entité recherchée à la place d'une expression lambda.

**Exemple :** Rechercher un **Voyage** dont la clé est **"AUS"** :

```
Voyage voyage = contexte.Voyages.Find("AUS");
```

Si l'entité recherchée a déjà été chargée, elle est récupérée en mémoire, sans requêter la base.

## La propriété Local

Toujours au niveau du **DbSet**, la propriété **Local** permet d'obtenir les entités chargées en mémoire, sans avoir à le faire explicitement (avec l'opérateur **ToList** par exemple). L'alimentation d'un **DbSet** peut en outre se faire directement par la méthode **Load**.

**Exemple :** Charger le **DbSet Voyages** avec la méthode **Load** et afficher le nombre d'entités chargées en mémoire avec la propriété **Local**.

```
// Charger les données du DbSet en mémoire  
contexte.Voyages.Load();  
  
// Nouvelle requête en base  
int n = contexte.Voyages.Count();  
  
// Plus de requêtage en base avec Local  
int n = contexte.Voyages.Local.Count();
```

## Remarques :

- ❖ La méthode **Load** est une méthode d'extension de **IQueryable** définie dans l'espace de noms **Microsoft.EntityFrameworkCore**. Elle peut donc se combiner avec un **Where**.
- ❖ Si le **DbSet** contient déjà entités lorsque la méthode **Load** est invoquée, les nouvelles entités chargées sont ajoutées aux entités existantes.

## Chargement des propriétés de navigation

Par défaut le chargement des propriétés de navigation (collections ou références) doit être fait explicitement en invoquant la méthode **Load** sur l'objet **EntityEntry** sur ces dernières. Cet objet fait partie des services concernant les opérations effectuées sur les entités chargées en mémoire (tracking).

```
// Charger une collection faisant partie des propriétés de navigation
context.Entry(voyage).Collection(c => c.Reservations).Load();

// Idem avec une référence
context.Entry(voyage).Reference(v => v.Categorie).Load();
```

### Activer le LazyLoading

Ce mécanisme permet de charger automatiquement les entités connexes en parcourant les propriétés de navigation. Il s'active avec le package Nuget **Microsoft.EntityFrameworkCore.Proxy**, en déclarant l'ensemble des propriétés de navigation en **virtual** et l'appel de la méthode **optionsBuilder.UseLazyLoadingProxies()** dans la configuration du contexte.

**Note** : Pour ce faire, les entités chargées sont encapsulées dans des classes Proxies.

Ce mécanisme peut être activé/désactivé à la demande au niveau du contexte :

```
// Désactiver le LazyLoading des propriétés de navigation
context.ChangeTracker.LazyLoadingEnabled = false;
```

### EagerLoading

Un **DbSet** propose également la méthode **Include** afin de pouvoir charger les entités associées aux propriétés de navigation en même temps que les entités parentes, c'est à dire avec une seule requête.

**Exemple** : Charger les réservations associées aux clients en une seule requête.

```
List<Client> liste =
    context.Clients.Include(c => c.Reservations).ToList();
```

**Remarque** : Les données d'un **Include** ne peuvent pas être filtrées par défaut, ce que permet le package Nuget **Z.EntityFramework.Plus.EFCore**, grâce à la méthode d'extension **IncludeFilter**.

## Le pattern async/await

L'API du **DbContext** bénéficie du pattern **async/await** apporté par le Framework .NET 4.5, permettant d'améliorer les performances des opérations coûteuses, telles que les opérations d'accès aux données.

Ce pattern consiste à exécuter ce type d'opération de manière asynchrone, en se basant sur le système de **Tasks** (espace de noms **System.Threading.Tasks**), avec une syntaxe simplifiée :

Les méthodes proposant ce pattern sont suffixées par **Async** et retournent une **Task**, dont le résultat s'obtient avec le mot réservé **await**.

**Rappel** : Pour ce faire, toute méthode contenant une instruction comportant ce dernier, doit être déclarée avec le mot réservé **async**.

**Exemple** : Récupération du résultat en un seul appel, dans une méthode asynchrone :

```
List<Article> liste = await _ctx.Articles.ToListAsync();
```

Si l'appelant souhaite exécuter d'autres opérations en attendant le résultat, l'appel peut être scindé en deux étapes, avec une référence sur la tâche fournie par la méthode asynchrone.

**Exemple** : Obtention du résultat d'une liste en deux étapes :

```
using System.Threading.Tasks;

Task<List<Article>> taskListe = context.Articles.ToListAsync();
// Continuation ...

// Attente et récup du résultat
List<Article> liste = await taskListe;
```

### Notes

## Validations locales

EF intègre deux méthodes de validation locales basées sur l'assembly **System.ComponentModel.DataAnnotations.dll** :

- ❖ Au niveau d'une propriété, par des attributs, c'est à dire de manière déclarative ou par code avec l'API Fluent, suivant le principe de la définition du schéma.
- ❖ Au niveau de l'entité, en implémentant l'interface **IValidatableObject**.

### Validation d'une propriété

Les attributs courants sont **Required** (valeur obligatoire), **Range** (plage de valeurs), **StringLength** (longueur min et max d'une chaîne de caractères), **RegularExpression** (expression régulière), **Url**, **EmailAddress** et **CustomValidation** (validation personnalisée).

```
public class Article
{
    [Required]
    [Range(1, 4, ErrorMessage =
        "La valeur {0} doit être comprise entre {1} et {2}.")]
    public int ArticleId { get; set; }

    [Required(ErrorMessage = "Designation obligatoire")]
    public string Designation { get; set; }

    [Required]
    [Range(1, 5000, ErrorMessage =
        "La valeur {0} doit être comprise entre à {1} et {2}.")]
    public decimal Prix { get; set; }
}
```

Dans les messages d'erreur, la valeur peut être affichée avec l'index 0 entre accolades. Il en est de même pour les index qui suivent, pour afficher les éventuels arguments de l'attribut.

#### Remarques :

- ❖ La propriété **ErrorMessage** peut être remplacée par la propriété **ErrorMessageResourceName**.
- ❖ Un type valeur est implicitement obligatoire s'il n'est pas nullable.

## Validation d'une entité

Si les validations au niveau des propriétés ne suffisent pas, l'interface **IValidatableObject**, constituée d'une unique méthode **Validate**, permet d'effectuer une validation au niveau de l'entité.

### Exemple :

```
public partial class Article : IValidatableObject
{
    public IEnumerable<ValidationResult>
        Validate(ValidationContext validationContext)
    {
        if (CategorieId != 1 && Prix < 500)
        {
            yield return new ValidationResult(
                "Le prix doit être supérieur à 500 pour
                cette catégorie d'article",
                // Associer l'erreur aux Ptés
                new[] { "CategorieId", "Prix" });
        }
    }
}
```

Cette méthode retourne une séquence (**IEnumerable**) d'objets **ValidationResult** fournis par l'opérateur **yield**.

Un objet **ValidationResult** est constitué d'un message d'erreur et d'une séquence **IEnumerable<String>** contenant le nom des propriétés concernées.

## Gérer les erreurs de validation

Les annotations sont prises en compte pour générer des validations dans une application Web.

Elles peuvent être également détectées explicitement par la méthode statique **TryValidateObject** de l'objet **Validator** pour chaque entité à enregistrer.

```
// Validations locales
var entries = context.ChangeTracker.Entries();
bool erreurs = false;

foreach (var entry in entries) {
    var entity = entry.Entity;
    var results = new List<ValidationResult>();
    bool isValid = Validator.TryValidateObject(entity,
        new ValidationContext(entity), results);

    if (!isValid) {
        erreurs = true;
        foreach (var validationResult in results)
        {
            foreach (var membre in validationResult.MemberNames) {
                Console.WriteLine(
                    $"{validationResult.ErrorMessage} sur {membre}");
            }
        }
    }
}

if (!erreurs) {
    context.SaveChanges();
    Console.WriteLine("Enregistrement terminé.");
}
```

Les erreurs de validations de chaque entité sont obtenues en paramètre de sortie dans une séquence d'objets **ValidationResult**.

La première boucle **foreach** parcourt l'ensemble des entités à enregistrer obtenues la méthode **Entries** du service de tracking lié au contexte. Si la méthode **TryValidateObject** échoue, la séquence des objets **ValidationResult** de l'entité concernée, est parcourue dans une seconde boucle. Une troisième boucle se charge pour terminer, d'afficher la liste **MemberNames** des noms des propriétés en erreur.

### La classe **EntityEntry**

La gestion des changements effectués dans les entités chargées en mémoire (tracking) s'effectue avec la classe **EntityEntry**, (espace de noms **Microsoft.EntityFrameworkCore.ChangeTracking**) qui s'obtient à partir de la méthode **Entry** du contexte, en précisant l'entité souhaitée en argument.

**Exemple** : Récupérer l'objet **EntityEntry** d'une entité.

```
var v = contexte.Articles.Find(208);  
EntityEntry entry = contexte.Entry(v);
```

Cet objet expose les membres caractéristiques suivants :

Propriété	Description
<b>IsKeySet</b>	Indique l'existence d'une clé ( <b>false</b> en cas de création non enregistrée si la clé est générée par la base et <b>true</b> dans les autres cas).
<b>Entity</b>	Référence sur l'entité associée.
<b>State</b>	Enumération <b>EntityState</b> .
<b>CurrentValues</b>	Liste ( <b>DbPropertyValues</b> ) des valeurs en cours de l'entité.
<b>OriginalValues</b>	Liste ( <b>DbPropertyValues</b> ) des valeurs de la base au moment du chargement de l'entité en mémoire. Cette propriété n'est pas disponible pour les nouvelles entités ( <b>State=Added</b> ) et provoque une exception si elle utilisée dans ce cas.



La propriété **State** étant de type **enum**, sa valeur n'est pas mise à jour automatiquement en cas de changement sur l'entité correspondante.

Méthode	Description
<b>GetDataBaseValues</b>	Récupère les valeurs de la base dans une liste <b>PropertyValues</b> .
<b>Reload</b>	Remplace les valeurs en cours et originales par celles de la base.

## La méthode SaveChanges

La mise à jour en base des entités créées, modifiées ou supprimées se réalise en invoquant la méthode **SaveChanges** de l'objet **DbContext**.

Pour ce faire, une commande de création (INSERT), de mise à jour (UPDATE) ou de suppression (DELETE) est générée dynamiquement pour chaque entité possédant l'un des 3 états **Added**, **Deleted**, et **Modified** de la propriété **State** de chaque **EntityEntry**.

Les opérations d'ajout et de suppression se réalisent respectivement par les méthodes **Add** et **Remove** du **DbSet**.

```
// Création d'une nouvelle entité
var v1 = new Voyage { CodeVoyage = "NEW", Destination = "Dest. new",
                    Description = "Description new", Prix = 100 };
contexte.Voyages.Add(v1);

// Récup. de l'entité si elle n'est pas déjà chargée en mémoire
// pour une maj/suppression

var v2 = contexte.Voyages.Find("MAJ");
v2.Description = "Nouvelle description"; // Modification

var v3 = contexte.Voyages.Find("DEL");
contexte.Remove(v3); // Suppression
contexte.SaveChanges(); // Enregistrement des changements
```

### Remarques :

- ❗ La méthode **SaveChanges** est virtuelle : elle peut être redéfinie.
- ❗ La méthode **SaveChanges** s'exécute de manière transactionnelle. Une transaction peut néanmoins être effectuée sur un ensemble de mises à jour avec un objet **TransactionScope**.
- ❗ Une fois la méthode **SaveChanges** appelée, les entités mises à jour repassent à l'état **Unchanged**.

**Remarque :** La propriété **Database** du **DbContext** propose la méthode **ExecuteSqlRaw** afin d'être en mesure d'exécuter des commandes SQL, sans passer par le modèle.



## Contrôler la création avec des associations

La création d'un objet lié par intégrité référentielle à d'autres entités connexes peut s'effectuer de différentes manières. Si l'entité connexe à ajouter est chargée dans le même contexte que l'entité à créer, son état est connu et possède la valeur **Unchanged** si elle n'est pas modifiée. Dans ce cas, la création de l'entité principale se déroule correctement, avec affectation de la clé externe de l'entité connexe pour assurer l'intégrité référentielle.

**Exemple :** Création d'une entité avec une association chargée dans le même contexte.

```
using (var contexte = new ArticlesDbContext()) {
    var categorie = contexte.Categories.Find(1);
    var new = new Article { CodeArticle = "V1 ", Categorie = categorie };

    contexte.Articles.Add(new);
    contexte.SaveChanges();
}
```

En revanche, si l'entité connexe provient d'un autre contexte, elle est détachée et son affectation à une propriété de navigation d'une entité dont l'état est **Added**, la fait également passer à l'état **Added**.

**Exemple :** Création incorrecte, provoquant la duplication de l'entité connexe.

```
Categorie categorie = null;

using (var contexte1 = new ArticlesDbContext()) {
    categorie = contexte1.Categories.Find(1);
}

using (var contexte2 = new ArticlesDbContext()) {
    var new = new Article { CodeArticle = "A108", Categorie = categorie };
    contexte2.Articles.Add(new);
    contexte2.SaveChanges();
}
```

Bien qu'il soit possible de modifier la valeur de la propriété **State** de chaque entité pour contrôler les instructions qui seront générées, il est plus simple d'utiliser la valeur de clé externe et non pas la propriété de navigation pour éviter ce problème :

```
var new = new Article { CodeArticle = "A108",
    CategorieId = categorie.CategorieId};
```

# Migrations

Les migrations permettent d'automatiser la mise à jour de la base en cas de changement du modèle. Elles se réalisent à l'aide des outils fournis par le package Nuget **Microsoft.EntityFrameworkCore.Tools**. Une fois installé, l'ensemble des commandes disponibles peuvent être affichées par la commande suivante dans la fenêtre **Console de gestionnaires de packages** de Visual Studio, désigné par l'acronyme **PMC** (Package Manager Console) :

```
PM > Get-Help about_EntityFrameworkCore
```

Les commandes proposées s'exécutent dans cette même fenêtre, en sélectionnant le projet dans lequel se trouve le contexte dans la liste **Projet par défaut**.

**Exemple** : Génération d'un script de migration avec le nom *InitialCreate*.

```
PM > Add-Migration InitialCreate
```

Un sous-dossier **Migrations** est alors créé s'il n'existe pas, avec deux fichiers de code générés. Un fichier *NomContextModelSnapshot.cs* qui reflète l'état du modèle en cours et un fichier avec le nom de migration choisi, préfixé avec sa date de création. Une fois la migration générée, elle peut être appliquée en base soit par la commande **update-database**, soit par code avec la méthode **Migrate** de l'objet **DbContext.Database**, soit en exécutant un script SQL obtenu par la commande **Script-Migration**.

Lors de la création de la base ou à la première migration, la table **\_MigrationHistory** est créée dans la base, avec la liste des migrations qui ont été appliquées à la base, ainsi que la version de Entity Framework utilisée.

**Exemple** : Appliquer la dernière migration (ou créer la base si elle n'existe pas).

```
PM > update-database
```

Grâce à cette table, les migrations qui n'ont pas été appliquées sont automatiquement exécutées.

## Remarques :

- ❖ La génération de la dernière migration peut être supprimée par la commande **remove-migration**.
- ❖ Une migration peut être également ciblée explicitement en indiquant son nom après la commande **Update-Database**.
- ❖ La commande **Scaffold-DbContext** permet d'obtenir un modèle à partir d'une base existante, en indiquant une chaîne de connexion et le nom du fournisseur de données (**Microsoft.EntityFrameworkCore.SqlServer**) pour **Sql Server**.
- ❖ Une exception est levée si le schéma de la base ne correspond bien pas au modèle.

**Exemple** : Code de migration généré après avoir ajouté une colonne *Stock* de type *int* dans une entité *Article* :

```
using Microsoft.EntityFrameworkCore.Migrations;

namespace DemosEF.Migrations
{
    public partial class Stock : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<decimal>(
                name: "Stock",
                table: "Articles",
                type: "int",
                nullable: false,
                defaultValue: 0);
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "Stock",
                table: "Articles");
        }
    }
}
```

Le code généré pour chaque migration propose ainsi les méthodes *Up* et *Down* dans un fichier nommé *DateMigration\_NomMigration.cs* placé dans le sous-dossier **Migrations**, dont le nom est enregistré dans la table **\_MigrationHistory** par la commande **update-database**.

## Notes

---

# Application Blazor

## Introduction

Blazor permet de créer des interfaces utilisateur (Web en .NET) dynamiques avec du C# à la place de JavaScript, avec la syntaxe **Razor**. Deux modèles sont proposés.

Le modèle **WebAssembly Client** s'appuie sur l'Api Web standard **Web Assembly**, qui consiste à exécuter du JavaScript compilé dans un assembly exécutable par le navigateur. Cet assembly est généré automatiquement à partir du C#.

Ce type d'application s'apparente aux applications **SPA** (Single Page App) qui fonctionnent de manière déconnectée, sans même que .NET soit installé sur le serveur, avec des temps de réponse très rapides.

En contrepartie, les fonctionnalités sont limitées à celles qui sont supportées par le navigateur et ne permet pas d'accéder à celles du serveur, sauf par l'intermédiaire d'un service. Le temps de chargement est également proportionnel à la taille de l'exécutable.

Le modèle **Blazor serveur** (hosting serveur) nécessite la présence de .Net sur le serveur, afin d'établir une connexion **SignalR**. Chaque opération modifiant l'interface nécessite une connexion car elle est exécutée côté serveur.

Le démarrage de l'application est très rapide (pas de chargement initial) et les fonctionnalités du serveur sont entièrement disponibles.

Dans les deux cas, il est possible de faire des appels (dans les deux sens) avec du JavaScript (**JSInterop**), notamment pour pouvoir interagir avec les Apis Web Standard.

# Organisation de l'application

Dans les deux cas, la conception est la même et se base sur un ensemble de composants et de pages Razor (fichiers d'extension **.razor**).

**Remarque** : Un composant se distingue d'une page s'il n'a pas de directive **@page**.

Exemple de page :

```
@page "/counter"
<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount() {
        currentCount++;
    }
}
```

## Blazor WebAssembly

Dans le cas d'une application cliente, l'application démarre de façon à configurer le service **HttpClient** proposé par défaut :

```
public static async Task Main(string[] args) {
    var builder = WebAssemblyHostBuilder.CreateDefault(args);
    builder.RootComponents.Add<App>("#app");

    builder.Services.AddScoped(sp => new HttpClient {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

    await builder.Build().RunAsync();
}
```

Elle s'intègre ensuite dans une page Html de la manière suivante, dans laquelle le code généré dans fichier **blazor.webassembly.js** alimentera la **div** dont l'**id** est **"app"** lorsqu'il a été chargé dans le navigateur.

```
<body>
  <div id="app">Loading...</div>
  <div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
  </div>
  <script src="_framework/blazor.webassembly.js"></script>
</body>
```

## Blazor Serveur

Dans le cas d'une application serveur, l'application démarre dans une application serveur standard, avec la classe **Startup** afin de définir le service et les **Endpoints Blazor** :

```
services.AddServerSideBlazor();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

Le Endpoint **MapFallbackToPage** fait référence à un fichier Razor **\_Host.chnml**, intégrant l'application avec le script **blazor.server.js** chargé d'établir la connexion **SignalR**. Ce fichier intègre l'application dans le composant **<component>** prévu à cet effet, avec l'attribut **render-mode="ServerPrerendered"** indiquant de générer le code HTML côté serveur.

```
@page "/"
@namespace BlazorAppServeur.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{ Layout = null; }

<!DOCTYPE html>
<html><head>
    <meta charset="utf-8" />
    <title>BlazorAppServeur</title>
    <base href="~/>
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="BlazorAppServeur.styles.css" rel="stylesheet" />
</head>
<body>
    <component type="typeof(App)" render-mode="ServerPrerendered" />

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

<script src="_framework/blazor.server.js"></script>
</body></html>
```

L'attribut **<base>** permet de spécifier l'url de base de l'application, qui peut être un sous-dossier si elle s'intègre à une application MVC par exemple.

## Système de routage

Dans les deux cas, le système l'application est représentée par le fichier `App.razor` qui définit le système de routage et la présentation générale des pages (fichier `MainLayout.razor` suivant le même principe que le fichier `_Layout.cshtml` dans le dossier `Shared` d'un projet MVC).

```
<Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

De même, dans les deux modèles, il est possible de :

- ❖ Créer une bibliothèque de composants.
- ❖ Déclarer globalement les espace de noms dans le fichier `_Imports.razor` (même principe que le fichier `_ViewImports.chnl` d'une application MVC).
- ❖ Définir plusieurs routes dans une page car il n'est pas possible de définir des routes avec des segments optionnels.
- ❖ Faire du débogage dans Visual Studio avec le navigateur **Edge**. Les outils de débogage des navigateurs sont disponibles dans tous les cas.

## La notion de composant

Un composant dérive de la classe **ComponentBase** permettant de gérer le cycle son exécution avec les méthodes **OnInitialize**, **OnParametersSet** et **OnAfterRender**(*bool firstRender*), également disponibles en version Async.

La méthode **StateHasChanged** peut être invoquée lorsque l'actualisation de l'affichage doit être réalisée explicitement.

**Remarque** : Le code peut figurer dans un fichier **.cs** avec une classe partielle de même nom que le composant.

### Paramétrage

Un composant peut accepter des paramètres, avec des propriétés décorées de l'attribut **[Parameter]**.

```
public partial class ComposantHello {  
    [Parameter]  
    public string Message { get; set; } = "Composant Hello !";  
}
```

La valeur d'un paramètre peut être renseigné par un composant parent de manière déclarative ou par code, via le système de navigation intégré.

```
<ComposantHello Message="Message défini par le parent"></ComposantHello>
```

### Injection de dépendances

Des services peuvent être également reçus et utilisés par injection de dépendance, mais par l'attribut ou par la directive **@inject** (pas par Ctor de la classe). Les services de navigation (méthode **NavigateTo**(*url*) et **HttpClient** sont intégrés par défaut :

```
[Inject]  
public NavigationManager NavigationManager { get; set; }
```

La classe **HttpClient** fournit les méthodes **Get**, **Post**, **Put** et **Del** (également en version Async) d'une chaîne JSON.

```
@inject HttpClient Http  
@code {  
    private WeatherForecast[] forecasts;  
  
    protected override async Task OnInitializedAsync() {  
        forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");  
    }  
}
```

### Isolation CSS

Une feuille de styles CSS peut être appliquée à chaque composant avec un fichier de même nom, avec l'extension.css : **MainLayout.razor.css** par exemple.



## Liaisons de données

Les tag helpers sont remplacés par des composants de saisie (**EditForm**, **ValidationSummary**, **InputText**, **InputCheckbox**, **InputNumber**, etc. (espace de noms **Microsoft.AspNetCore.Components.Forms**) et de validation intégrés. Une liaison de donnée en affichage se réalise en préfixant l'expression par @, comme dans une vue Razor. Une expression de liaison en lecture-écriture se définit l'attribut **@bind-value**.

Exemple de formulaire de saisie avec validations intégrées et appel à une méthode *Valider* :

```
<EditForm Model="@vm" OnValidSubmit="@Valider">
  <DataAnnotationsValidator />
  <ValidationSummary />
  <div class="form-group row">
    <label for="Designation" class="col-sm-3">Désignation : </label>
    <InputText id="Designation" class="form-control col-sm-8"
      @bind-Value=@vm.Designation placeholder="Entrer une désignation"></InputText>
    <ValidationMessage class="offset-sm-3 col-sm-8" For="@(() => vm.Designation)" />
  </div>
  <div class="form-group row">
    <label for="Description" class="col-sm-3">Description : </label>
    <InputText id="Description" class="form-control col-sm-8"
      @bind-Value=@vm.Description></InputText>
    <ValidationMessage class="offset-sm-3 col-sm-8" For="@(() => vm.Description)" />
  </div>
  <div class="form-group row">
    <label for="Duree" class="col-sm-3">Durée : </label>
    <InputNumber id="Duree" class="form-control col-sm-8"
      @bind-Value=@vm.Duree></InputNumber>
  </div>
  <div class="form-group row">
    <label for="Prix" class="col-sm-3">Prix : </label>
    <InputNumber id="Prix" class="form-control col-sm-8"
      @bind-Value=@vm.Prix @bind-Value:format="C"></InputNumber>
  </div>
  <div class="form-group row">
    <label for="Promo" class="col-sm-3">Promotion : </label>
    <InputCheckbox id="Promo" class="form-control col-sm-8"
      @bind-Value=@vm.Promotion></InputCheckbox>
  </div>
  <button type="submit" class="btn btn-primary">Enregistrer</button>
</EditForm>
```

# Interopérabilité JavaScript

Le service **IJSRuntime** (espace de noms **Microsoft.JSInterop**) donne accès au runtime JS du navigateur, avec la méthode **InvokeAsync** ou **InvokeVoidAsync** afin de pouvoir invoquer une fonction JavaScript avec ou sans résultat. Ce service est intégré par défaut et s'obtient dans une page par injection de dépendance avec la directive **@inject** :

```
@inject IJSRuntime jsr
```

La méthode accepte le nom de la fonction en premier argument et les arguments à transmettre à la fonction ensuite.

```
await jsr.InvokeVoidAsync("messagePopup", "Hello JS !");  
// L'objet global window est directement accessible  
//await jsr.InvokeVoidAsync("alert", "Hello JS !");  
  
var contact = new Contact { Prenom = "Jean", Nom = "Martin",  
                             Email = "jm@dom.com" };  
await jsr.InvokeVoidAsync("receptionObjet", contact);
```

Avec la fonction JS :

```
function objetRetour(prenom, nom) {  
    return { prenom, nom, email: prenom + "." + nom + "@dom.com" };  
}  
  
Contact _contact;  
private async Task ObjetRetour()  
{  
    _contact = await jsr.InvokeAsync<Contact>("objetRetour",  
                                              "jean", "martin");  
}
```

## Appel de méthodes C# à partir d'une fonction JS

L'interopérabilité fonctionne également en sens inverse de façon à pouvoir appeler une méthode C# décorée par l'attribut **JSInvokable** à partir de JavaScript, à partir d'un objet global **DotNet** ajouté à l'objet **window**.

```
[JSInvokable]
public static int AdditionI(int a, int b) {
    return a + b;
}

function appelAddition () {
    var promesse = DotNet.invokeMethodAsync("InteropFromJS",
                                           "Addition", 2, 5);
    promesse.then(resultat => alert(resultat));
}
```

Cet objet permet d'appeler directement une méthode statique, en indiquant le nom de l'application (son espace de noms) en premier argument, le nom de la fonction à appeler et les arguments éventuels attendus par la méthode.

L'appel étant asynchrone, l'éventuel résultat s'obtient par la méthode **then** de la promesse obtenue en retour.

Pour invoquer une méthode d'instance, l'appel doit être effectué à partir d'une référence fournie par la méthode **DotNetObjectReference.Create(this)**.

```
private async Task TransfertRefDotNet()
{
    var refDN = DotNetObjectReference.Create(this);
    await jsr.InvokeVoidAsync("interop.appelAdditionI", refDN);
}

function appelAdditionI(refDotNet) {
    var promesse = refDotNet.invokeMethodAsync("AdditionI", 2, 5);
    promesse.then(resultat => alert(resultat));
}
```

## Intégration à une application MVC

L'intégration d'une application Blazor à une application serveur est implicite dans le modèle Serveur (PN `Microsoft.AspNetCore.Identity.UI`). Pour le modèle **WebAssembly**, elle est proposée à la création du projet avec le modèle d'authentification par Identity avec le package Nuget `Microsoft.AspNetCore.Components.WebAssembly.Authentication`.

### Intégration du modèle d'authentification Identity

Dans les deux cas, la classe `Startup` du projet Web configure le pipeline avec :

```
app.UseAuthentication();
app.UseAuthorization();
```

Etant donné le fonctionnement autonome du modèle **WebAssembly**, une passerelle doit être configurée sur le serveur avec un contrôleur `OidcConfigurationController` et les services :

```
builder.Services
    .AddIdentityServer()
    .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();

builder.Services
    .AddAuthentication()
    .AddIdentityServerJwt();
```

Côté client, les communications avec ce contrôleur se réalisent par les services :

```
builder.Services
    .AddHttpClient("BlazorAppClientAuth.ServerAPI",
        client => client.BaseAddress =
            new Uri(builder.HostEnvironment.BaseAddress))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

// Supply HttpClient instances that include access tokens when
// making requests to the server project
builder.Services
    .AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>())
    .CreateClient("BlazorAppClientAuth.ServerAPI");

builder.Services.AddApiAuthorization();
```

Le composant **LoginDisplay** se charge de faire appel au service d'authentification de l'application serveur (directement ou par la passerelle avec un composant **Authentication** pour un **WebAssembly**).

Le système de routes défini dans le fichier **App.razor** doit s'imbriquer dans un élément **CascadingAuthenticationState** afin de pouvoir restreindre l'accès aux pages aux utilisateurs authentifiés.

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
        <NotAuthorized>
          <h1>Sorry, you're not authorized to view this page.</h1>
        </NotAuthorized>
      </AuthorizeRouteView>
    </Found>
    <NotFound>
      <LayoutView Layout="@typeof(MainLayout)">
        <p>Sorry, there's nothing at this address.</p>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

**Remarque** : Quel que soit le modèle choisi, le **HttpContext** de l'application hôte n'est pas disponible dans l'application Blazor (Aide [Avancé/Accéder à HttpContext](#)).

Le composant **AuthorizeView** peut être ensuite utilisé de façon à gérer l'affichage en fonction de l'authentification.

```
<AuthorizeView>
  <Authorized>
    <a href="Identity/Account/Manage">Hello, @context.User.Identity.Name!</a>
    <form method="post" action="Identity/Account/Logout">
      <button type="submit" class="nav-link btn btn-link">Log out</button>
    </form>
  </Authorized>
  <NotAuthorized>
    <a href="Identity/Account/Register">Register</a>
    <a href="Identity/Account/Login">Log in</a>
  </NotAuthorized>
</AuthorizeView>
```

De plus, l'accès aux pages peut être restreint avec l'attribut **Authorize** :

**@attribute** [Authorize]

Et bonne continuation en ASP.NET !