

Chapitre 3

Les Web Forms

1. Présentation des Web Forms

Les formulaires web (Web Forms) représentent la partie la plus visible des sites web ASP.NET et par conséquent la plus populaire. Ils reposent sur un partage des responsabilités de type **MVC** : modèle, vue, contrôleur. Lorsqu'un formulaire est écrit en utilisant le style **code séparé**, la page HTML .aspx est chargée de l'affichage (vue), la classe C# porte les données et effectue des calculs (modèle), tandis que le serveur d'applications ASP.NET coordonne l'ensemble (contrôleur). Cette analyse rassurera sans doute les développeurs Java quant à l'organisation des sites web ASP.NET.

D'un autre côté, les formulaires web sont le résultat de la transposition par Microsoft du modèle Visual Basic 6 en une façon originale et productive de développer des interfaces graphiques sur support Internet. Le succès de ce modèle est tel que Sun l'a repris à son compte concernant la technologie développement web JSF (*Java Server Faces*).

1.1 Structure d'une page ASPX

Le chapitre Les sites web ASP.NET a mis à jour la structure d'une page ASPX sous l'angle du modèle de compilation. Il s'agit maintenant d'exposer sa structure logique.

Étudions le code figurant dans une page Default.aspx :

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>

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

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>

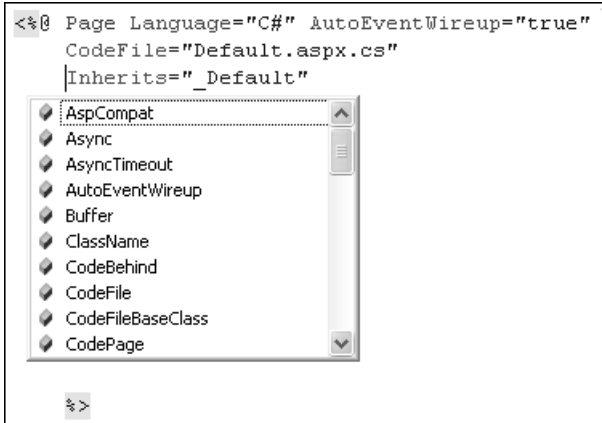
    </div>
  </form>
</body>
</html>
```

Ce code est constitué de trois parties : une directive de page, une déclaration de DTD et du code XHTML.

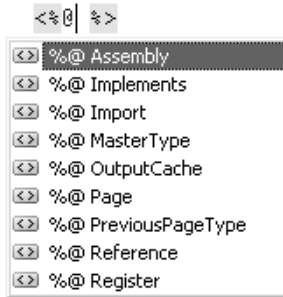
La directive de page

Les directives organisent la lecture d'une page ASPX par le serveur d'applications. Dans la page Default.aspx, l'attribut **Language** désigne le langage – C#, VB.NET, C++ – utilisé pour écrire les scriptlets. D'autres attributs sont également présents, servant à la communication avec la page de code-behind (**AutoEventWireup**, **CodeFile**, **Inherits**), à appliquer des thèmes, à démarquer les traces... Nous découvrirons l'usage de ces attributs au fur et à mesure de notre étude.

Par chance, Visual Studio propose les différents attributs applicables en utilisant la combinaison de touches [Ctrl][Espace].



D'autres directives sont également disponibles pour puiser des ressources dans l'environnement de la page : stratégies de cache, composants, assemblages, types de pages maîtres...



Les DTD

Les définitions de type de documents (*Document Type Definition*) sont établies par le consortium W3C. Il s'agit d'une norme applicable aux documents SGML, XML et HTML qui fixe les règles syntaxiques et sémantiques de construction d'un document à base de tags (marqueurs).

Les navigateurs sont souvent assez tolérants vis-à-vis du respect des DTD. Avec la version ASP.NET 1.X, le flux HTML de sortie était compatible avec la DTD **HTML transitionnel niveau 4**. Excepté l'attribut `MS_POSITIONNING` qui n'était pas filtré, le code HTML était tout à fait standard. Il est vrai qu'une page ASPX contient des balises spéciales (`<asp:label>` par exemple) qui sont traduites en une séquence HTML accessible au navigateur.

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

La version 2.0 apporte la conformité avec **XHTML**, une déclinaison assez stricte du langage HTML. Les puristes peuvent se rendre sur le site du W3C et passer une page ASP.NET à la moulinette de vérification située à l'adresse <http://validator.w3.org>. Les pages doivent être conformes à la DTD annoncée.

■ Remarque

Attention, pour effectuer ce test, il faut enregistrer le flux HTML à partir du bloc-notes ouvert par la commande `affichage source`. La fonction `Enregistrer sous - Page HTML` du navigateur Internet Explorer modifie le fichier et biaise le test.

Pour le développeur de pages web, la conformité avec une version spécifique du langage HTML ne suffit hélas pas à garantir qu'une page aura la même présentation quel que soit le navigateur. D'abord, les navigateurs ont la responsabilité d'interpréter les règles de mise en page comme ils l'entendent. Le langage HTML décrit le contenu mais pas la mise en page. Ensuite, les pages comportent aussi du code JavaScript et des styles CSS qui sont diversement pris en charge par les navigateurs.

Le serveur ASP.NET 2.0 a introduit un autre changement : la notion de schéma de navigateur cible a disparu. Il est vrai que cette directive n'a pas pu accompagner l'évolution des navigateurs cités, sans compter l'apparition d'autres logiciels de navigation. À la place, les sites web ASP.NET possèdent un dossier **App_Browsers** répertoriant les caractéristiques de chaque navigateur. Cet aspect sera étudié en même temps que les composants personnalisés.

Pour certains navigateurs et programmes JavaScript intervenant sur le DOM et qui ne seraient pas compatibles avec la norme XHTML, le serveur d'applications peut être configuré pour revenir au mode HTML transitionnel. La consigne figure dans le Web.config :

```
<xhtmlConformance mode="Legacy" />
```

L'attribut mode accepte trois valeurs :

| | |
|--------------|----------------------------------|
| Legacy | Ancien format HTML transitionnel |
| Strict | XHTML strict |
| Transitional | XHTML transitionnel |

Le code XHTML

S'il est vrai que le serveur d'applications ASP.NET 1.X sortait un flux conforme à la DTD HTML 4 transitionnelle, la syntaxe même des pages ASPX mélangeait des séquences HTML avec des séquences XML. Visual Studio 2003 était chargé de contrôler la cohérence de l'ensemble en générant des avertissements si nécessaires, et le serveur d'applications devait opérer une lecture plus qu'attentive (donc coûteuse) pour séparer les séquences HTML et les séquences XML.

Cette fois-ci, l'élément `<html>` contient une référence à l'espace de noms XHTML :

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

En d'autres termes, les balises d'une page ASPX doivent respecter la syntaxe XHTML. Bien entendu, les balises préfixées par `asp` (contrôles web), `uc` (contrôles utilisateurs) ou `cc` (contrôles personnalisés) ne font pas partie du vocabulaire XHTML. Mais au moins, la syntaxe est plus proche et plus précise. Et le flux de sortie reste en tout état de cause conforme à la DTD annoncée.

Enfin, Visual Studio fait de son mieux pour valider à l'avance les séquences HTML figurant dans une page ASPX. Des messages d'avertissement sont générés pour attirer l'attention du développeur sur une non-conformité.

1.1.1 Style imbriqué, en ligne et séparé

L'organisation d'une page dynamique est une simple question de style. Suivant la nature de la séquence HTML à décrire, il est préférable d'opter pour la version imbriquée ou pour la version en ligne (inline). Seul le style séparé (code-behind) change radicalement et apporte une distinction nette entre la présentation et le calcul. C'est la raison pour laquelle il est privilégié par Visual Studio.

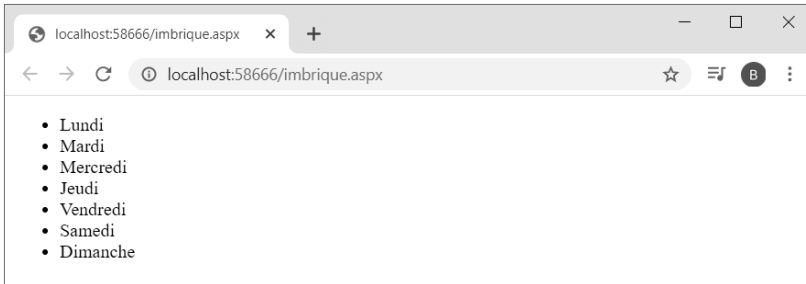
Le style imbriqué

Ce sont les premières générations de pages dynamiques (ASP, PHP) qui ont imposé le style imbriqué. Avec les modèles de composants web ASP.NET, celui-ci n'a plus vraiment cours mais reste applicable. Il peut également servir dans le cas de contrôles à base de modèles tels que des Repeater ou des Data List.

Voici un exemple de code basé sur ce style :

```
<body>
  <form id="form1" runat="server">
    <ul>
      <%
        int i;
        string[] jours = { "Lundi", "Mardi", "Mercredi", "Jeudi",
"Vendredi", "Samedi", "Dimanche" };
        for(i=0; i<jours.Length; i++)
        {
          <li><%= jours[i] %></li>
        } %>
    </ul>
  </form>
</body>
```

Le développeur doit faire de son mieux pour aligner son code comme s'il s'agissait d'un programme totalement écrit en C#.



Le style en ligne (inline)

Le style imbriqué est plutôt dévolu à la présentation. Il ne convient pas lorsque des traitements sont envisagés. La version en ligne sépare le code C# et le code HTML dans deux parties du même fichier .aspx. Des balises `<script runat="server">` indiquent au compilateur qu'il s'agit de code C#, mais elles pourraient être remplacées par des scriptlets `<% %>`.

```
<%@ Page Language="C#" %>
<script runat="server">
    // contient le code événementiel
    void traiter_click(object sender, EventArgs e)
    {
        message.Text = "Vous avez cliqué !";
    }
</script>

<!-- limite entre le code C# et le code HTML -->

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

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Style en-ligne</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="message" runat="server"></asp:Label>
            <asp:Button ID="cmd" runat="server" Text="Cliquer ici">

```

Chapitre 4 Dockerfile

1. Principes et syntaxe

Vous êtes maintenant armés pour pouvoir manipuler Docker grâce au CLI. Cependant, vous avez jusqu'à présent utilisé uniquement des images officielles publiées sur le Docker Hub.

Pour pouvoir créer ces images, ceux qui les ont publiées ont dû les construire au préalable. Ainsi, pour avoir la possibilité d'exploiter vos créations à l'aide de Docker vous allez devoir, à votre tour, créer vos propres images Docker. Ceci est possible grâce à un fichier appelé **Dockerfile**.

Dans la logique, il faut voir le Dockerfile comme étant une succession d'instructions ordonnées, analogue à une recette de cuisine, nécessaire pour créer l'image. Ce fichier, sans extension, se comporte comme un fichier texte (il peut être édité simplement avec un bloc-notes). Pour suivre l'exercice, il est conseillé de se placer dans un dossier vide individuel et d'y créer un fichier nommé "Dockerfile" sans extension.

■ Remarque

Appeler le fichier "Dockerfile" est une convention et n'est pas forcément nécessaire. On peut très bien l'appeler autrement et renseigner ce nom lors des instructions de génération de l'image. L'avantage de suivre cette convention est que l'on va éviter l'obligation d'ajouter un argument supplémentaire à notre ligne de commande. Il est recommandé de la suivre s'il n'y a qu'un seul fichier Dockerfile dans le dossier.

Explorons les différentes instructions nécessaires pour construire un Dockerfile.

1.1 Instructions FROM et WORKDIR

■ Remarque

Par convention, les instructions du Dockerfile s'écrivent en majuscule. Il s'agit uniquement d'une convention et ce n'est pas obligatoire. Cependant, adopter cette pratique améliorera à la lisibilité de votre fichier.

Pour commencer, il est souhaitable de partir d'une image existante pour pouvoir construire le Dockerfile. Il faut savoir qu'il est possible de partir d'un très bas niveau, auquel cas il faudra choisir une image correspondant à un système Linux, comme **Debian** ou **Alpine**. Cependant, en tant que développeur, ce choix sera rarement effectué, car il existe des images préconfigurées de vos environnements favoris (que ce soient les SDK, mais aussi les *runtimes*).

Pour pouvoir travailler efficacement, partir d'une image déjà composée est recommandé pour gagner du temps. Dans le cas des images .NET Core par exemple, Microsoft en propose deux types de base : le runtime ou le SDK. Le **runtime** sera préféré pour construire une image prête à l'exploitation, ne comprenant que les fichiers binaires de l'application finale, alors que le SDK sera utilisé pour toutes les opérations entrant dans le cadre du développement (phases de *build*, de tests, etc.).

L'instruction FROM du Dockerfile est systématiquement la première ligne dans nos fichiers, et elle détermine à partir de quelle image vous souhaitez commencer.

Par exemple, pour partir du SDK de .NET Core 3.1, on utilisera l'instruction FROM suivante :

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1
```

■ Remarque

On constate ici que la version de l'outil est précisée. À l'instar de ce que l'on a vu dans le chapitre précédent avec la version latest, il est encore plus important dans le cas d'un Dockerfile de préciser la version, par mesure de sécurité et pour les performances. Ce chapitre expliquera plus en détail comment le cache de Docker se comporte avec les différentes lignes d'instructions d'un Dockerfile.

En utilisant cette première instruction, vous allez définir le contexte d'exécution qui statuera que cette image est la vôtre, tout en partant d'une base existante. Cette instruction FROM nous permet de nous assurer que nous avons le SDK de .NET Core 3.1, avec la dernière version de patch et avec tous les outils nécessaires.

■ Remarque

Si vous avez besoin de partir sur une création totalement vierge, utilisez l'instruction FROM scratch.

Si on se contente d'enchaîner les instructions, elles vont dorénavant être exécutées dans le contexte d'exécution issu du lancement de cette image. Dans le cas du SDK .NET Core par exemple, c'est à la racine du dossier home de l'utilisateur root courant que seront exécutées la totalité des instructions. Un développeur va plutôt souhaiter se placer dans un répertoire de travail vierge, afin de ne pas être parasité.

Cette opération peut être réalisée grâce à l'instruction WORKDIR, qui permet de spécifier le nom du répertoire de travail.

```
WORKDIR [NOM_DOSSIER]
```

À la suite de cela, on se retrouve dorénavant dans un répertoire (créé automatiquement s'il n'existe pas) où nous avons une totale liberté pour notre travail. Cette commande, bien que facultative, est souvent recommandée pour éviter de rencontrer des conflits, souvent dus à la présence de fichiers indésirables propres à l'image qui se mélangent à nos propres fichiers, ou encore à des problèmes de droits d'écriture/exécution.

■ Remarque

WORKDIR définit le répertoire courant. Il est dès lors possible de l'utiliser plusieurs fois pour naviguer de répertoire en répertoire au fil des instructions.

1.2 Instruction RUN

Une fois que le contexte de travail est défini (image de base et répertoire de travail), nous pouvons exécuter des commandes au sein de l'image. Pour faire un parallèle, il faut voir l'instruction RUN comme l'équivalent de l'exécution d'une ligne de commande sur un terminal. Ainsi, cette instruction va permettre d'exécuter des commandes pour la bonne création de l'image. Par exemple, il serait possible de préinstaller certains outils sur une image Debian grâce à cette commande en lançant l'outil `apt-get`.

Lors de chaque instruction RUN, Docker crée un conteneur temporaire, permettant l'exécution de cette commande, qui sera utilisé comme base pour l'instruction RUN suivante. Ainsi, il est généralement préférable de chaîner les commandes de console dans une seule instruction RUN plutôt que de multiplier ces dernières, afin de profiter du mécanisme de cache pour gagner en vitesse. En effet, les reliquats de chaque conteneur intermédiaire vont se retrouver dans le conteneur final de façon cumulative, ce qui pourra avoir un impact sur la taille de l'image.

Chapitre 4

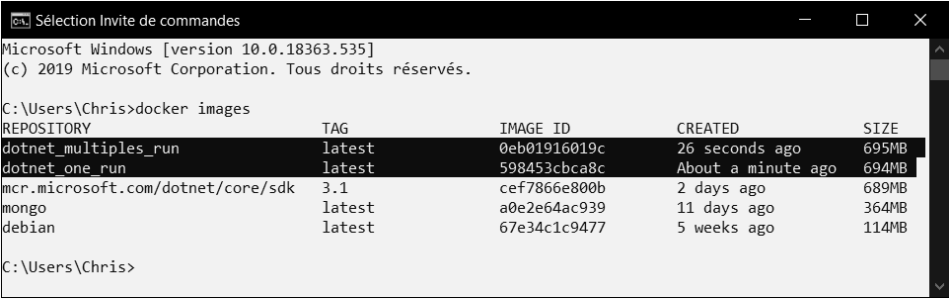
Dans notre exemple, étant donné que la base du Dockerfile est l'image du SDK .NET Core, il est possible de demander la création d'un nouveau projet, puis d'exécuter les commandes `restore`, `build` et `publish` à l'aide du CLI .NET Core :

```
RUN dotnet new console && \  
dotnet restore && \  
dotnet build && \  
dotnet publish -o publish  
WORKDIR publish  
RUN dotnet testproject.dll # affichera "Hello World!" Sur la console
```

Il est également possible de décomposer cette instruction en plusieurs instructions `RUN` indépendantes :

```
RUN dotnet new console  
RUN dotnet restore  
RUN dotnet build  
RUN dotnet publish -o publish  
WORKDIR publish  
RUN dotnet testproject.dll # affichera "Hello World!" Sur la console
```

Cependant, sur le résultat de l'image finale, on peut constater une différence en termes de place sur le disque local :



| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-----------------------------------|--------|--------------|--------------------|-------|
| dotnet_multiples_run | latest | 0eb01916019c | 26 seconds ago | 695MB |
| dotnet_one_run | latest | 598453cbca8c | About a minute ago | 694MB |
| mcr.microsoft.com/dotnet/core/sdk | 3.1 | cef7866e800b | 2 days ago | 689MB |
| mongo | latest | a0e2e64ac939 | 11 days ago | 364MB |
| debian | latest | 67e34c1c9477 | 5 weeks ago | 114MB |

Figure 1 : Différence de build avec instruction `RUN` unique et multiple

Remarque

La différence n'est pas ici extrêmement significative (1 Mo), car les couches intermédiaires créées par chaque instruction `RUN` ne sont pas lourdes. Dans le cas d'opérations plus coûteuses, cela peut vraiment faire la différence.

1.3 Docker build

Si vous avez suivi les instructions jusqu'ici, vous devriez avoir un fichier appelé `Dockerfile` dans votre répertoire. Cependant, vous ne pouvez encore l'utiliser avant de l'avoir transformé en image. La commande qui permet de réaliser ceci est l'instruction `build` du CLI.

Cette commande n'a pas été vue dans le chapitre précédent, car il n'y avait pas matière à l'exploiter sous forme d'exercice. Elle est donc détaillée maintenant.

La syntaxe de cette commande ressemble à toute commande du CLI :

```
docker build [ARGUMENTS] [PATH]|[URL]|-
```

Comme on peut le voir, elle présente cependant une spécificité : elle prend en paramètre final un **chemin** (PATH), une **URL** ou un **flux** (caractère -). Si l'on souhaite réaliser un *build* de l'image du répertoire actuel, on pourrait utiliser au choix :

- `docker build .`
- (PowerShell) `Get-Content Dockerfile | docker build -`
- (CMD) `more Dockerfile | docker build -`

Le résultat sera totalement identique, le CLI analysera les instructions de votre `Dockerfile` pour les traiter de façon séquentielle.

Une option très intéressante lorsque l'on réalise un *build*, pour s'assurer qu'on n'utilise pas d'anciens reliquats de cache, est l'option `--no-cache`. En toute logique, cette option indiquera au CLI d'effectuer la totalité des opérations sans tenir compte du cache local, pour s'assurer de construire l'image sur une base neuve.

À la suite de cette opération, si vous n'avez précisé aucun argument, vous obtiendrez une image "anonyme" sur votre système (vous pourrez le constater grâce à la commande qui liste les images présentes, une ou plusieurs images taguées <none> seront listées). Bien évidemment, on va vouloir différencier nos images avec un nom facile à utiliser pour n'importe quelle opération en ayant besoin (comme le lancement d'un conteneur par exemple). Cela nous permettra de travailler avec ce nom plutôt qu'avec un ID. Cette opération correspond à "taguer" son image.

1.3.1 Tag des images

Afin de pouvoir aborder cette section, il est nécessaire de comprendre le fonctionnement de l'organisation du Docker Hub ainsi que les informations remontées par un listing des images.

À l'exécution de la commande `docker images`, ce sont les deux premières colonnes qui nous intéressent dans le cas présent. Il s'agit des colonnes **repository** et **tag**. Jusqu'ici, nous nous sommes contentés de lancer des commandes `docker run` avec un nom d'image simple, comme par exemple `docker run debian`. Implicitement, cela signifie qu'on veut récupérer l'image Debian depuis la racine du Docker Hub et non dans une organisation particulière.

En effet, Docker Hub est une place d'échange publique. Ainsi, chacun peut librement pousser ses images pour que d'autres puissent les télécharger et les utiliser. Cela signifie également qu'il faut pouvoir distinguer les images produites par chacun. Pour répondre à cette problématique, Docker Hub introduit une notion **d'organisation**. Dès lors que vous avez un compte sur Docker Hub avec un nom d'utilisateur, il correspond à votre organisation personnelle.