



Projet STL :

Rendre une application web configurable

BOUCHENAKI Habib

MEYNIER Florent

Professeur encadrant :

ZIADI Tewfik

Sommaire

I / Introduction	2
II / Contexte et objectif	4
1. Contexte	4
2. Objectif	6
III / Travail réalisé	7
1. Implémentation de l'application web	7
2. Ajout de la variabilité	10
3. Dérivation	11
IV / Conclusion	12

I / Introduction

Dans le monde industriel, on cherche à réduire les coûts et temps de production. Pour se faire, on cherche les similitudes entre différents produits. Ces similitudes peuvent être dans la technique de fabrication, d'assemblage, d'emballage ou d'autres aspects.

On peut prendre l'exemple d'une voiture où un modèle va avoir de très nombreuses pièces et dont la plupart vont être identiques à d'autres modèles issus de la même marque. Comme on peut le voir sur la figure 1, de nombreuses parties peuvent ne pas être spécifiques à ce modèle car sont des pièces qui peuvent très bien être assemblées sur d'autres voitures. Certaines pièces peuvent être modulables et facilement remplacées car sont facilement accessibles, ou ne sont pas complexes dans leur conception et apparence. Par exemple, un rétroviseur peut facilement être remplacé par un autre rétroviseur d'un autre modèle et ne change pas totalement le fonctionnement du véhicule ni son efficacité.



Figure 1 : variabilisation d'une voiture

Ce principe peut être appliqué au Hardware d'un produit, c'est-à-dire au matériel, mais aussi au software. Par exemple, dans une voiture, le logiciel qui va contrôler le limiteur de vitesse peut être utilisé sur différents modèles de véhicules. En effet, le logiciel reçoit des informations, les traite et renvoie des données. Ce comportement sera tout le temps le même, quel que soit le modèle du véhicule, mais les données du véhicule peuvent quant à elles être différentes d'un modèle à un autre, il faut donc au préalable les transformer.

Si on prend l'exemple d'une imprimante, sa fonctionnalité est principalement d'imprimer et de scanner des documents, néanmoins son usage peut être différent en fonction du besoin. En effet, une imprimante d'entreprises doit a priori être plus rapide qu'une imprimante domestique car les besoins sont différents. Mais on peut supposer que le logiciel ne change pas entre ces 2 types d'imprimante, ce qui va changer ce sera la qualité et l'efficacité du matériel et des pièces qui composent l'imprimante, qui pourront se déplacer plus rapidement avec une meilleure précision et meilleure qualité.

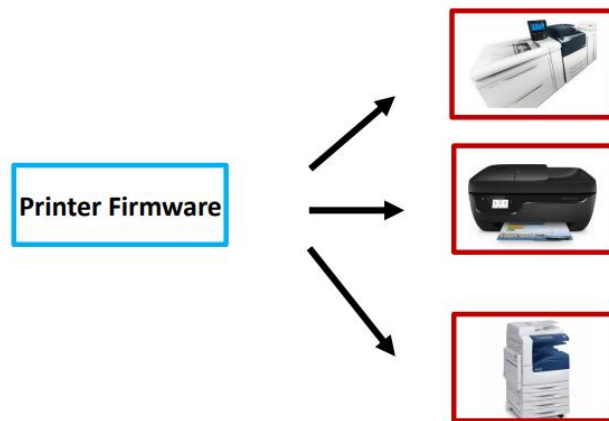


Figure 2 : Un même logiciel peut servir différents usages

Comme on a pu le voir, on a commencé à chercher des manières d’optimiser la création de pièces d’usinage, lorsque des technologies logicielles sont apparues, on a souhaité réutiliser le même logiciel sur différents produits. Dorénavant, on cherche à appliquer ce même concept aux logiciels eux-mêmes.

Existe-t-il des fonctionnalités d’un logiciel qu’on peut récupérer et implémenter dans un autre ?

On peut par exemple supposer que certaines fonctionnalités d’un logiciel de traitement de texte et celles d’un tableur sont identiques. Ce partage d’informations permet de créer du code plus rapidement et d’être plus efficace dans la conception de logiciel.

Mais, on peut aussi supposer autre chose : Est-il intéressant de créer un logiciel avec beaucoup de caractéristiques et de contenu et permettre à un client de choisir les fonctionnalités et l’apparence de l’application ?

II / Contexte et objectif

1. Contexte

Les entreprises

La demande en logiciel de la part d'entreprises et de clients est de plus en plus grande. Dans un monde où la digitalisation prend de plus en plus de place, il est important pour ces entreprises, qu'elles soient locales ou internationales, de posséder un site internet. Ce site peut avoir de nombreux intérêts et possibilités en fonction du besoin de l'entreprise.

Il existe par exemple des sites qui permettent de créer facilement un portfolio, en pouvant personnaliser l'apparence du site, la possibilité d'y vendre ses créations. D'une autre manière, il existe des sites internet qui permettent de créer des sites simplement à l'aide de template déjà créée et la possibilité de personnaliser son propre site comme bon nous semble. Ces sites sont très utiles pour les personnes ayant peu de moyens et de connaissances pour créer un site mais reste assez limité pour des entreprises qui souhaiteraient avoir besoin d'un site avec des fonctionnalités plus poussées avec une interaction client/serveur.

Les entreprises font donc appel à des professionnelles afin que l'application soit faite tout en correspondant à un cahier des charges. Cette application sera unique car correspondra aux demandes du client mais aura un coût de production élevé. Néanmoins, en fonction du besoin, une entreprise peut acheter une application déjà produite afin de l'utiliser et l'exploiter. Cette application (ou logiciel) ne sera pas personnelle à l'entreprise (si le vendeur le décide) et peut être peu personnalisable. C'est pourquoi une autre approche du monde des logiciels et à envisager : les lignes de produits.

En effet, on peut récupérer certains modes de fonctionnement du monde de l'usinage et du commerce afin de les adapter au monde des logiciels et applications afin de pouvoir proposer des produits d'un nouveau type. L'idée derrière ce concept de ligne de produit dans les logiciels est de créer une application avec beaucoup de fonctionnalités dont le choix des fonctionnalités pourra être fait par le client. De plus, l'aspect visuel de l'application est configurable, c'est-à-dire que le client peut choisir les logos, images, textes, et aspects généraux de cette application. Ainsi, une même application vendue à deux clients différents, pourra proposer des services différents et pourra ne rien à voir visuellement.

Sur la figure 3, nous avons une application qui est créée. Chaque rectangle correspond à une fonctionnalité, les blancs sont celles obligatoires, qui seront communes à toutes les applications produites, et les rouges sont celles qui peuvent être activées/désactivées de l'application résultat. Comme on peut le voir parmi les produits créés, tous ont une base commune qui correspond aux rectangles blancs, mais on peut voir que parmi les fonctionnalités sélectionnées, toutes n'ont pas pris les mêmes. Ces différences vont rendre une même application initiale totalement différente à l'arrivée. En plus de la personnalisation côté back-end (qui correspond aux fonctionnalités proposées), ces applications peuvent aussi être personnalisées au niveau du front-end. Ainsi, 2 applications comportant exactement les mêmes fonctionnalités, pourront avoir des aspects totalement différents.

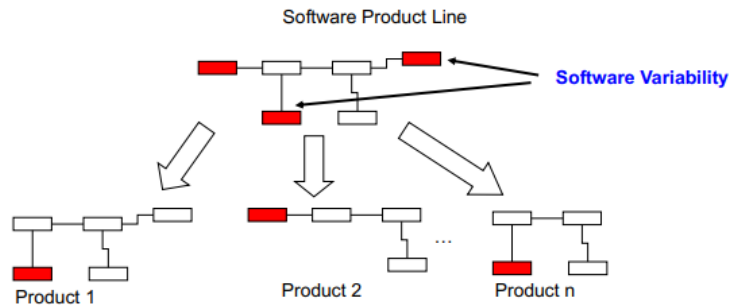


Figure 3 : Variabilisation d'une application

Grâce à ceci, une entreprise peut acquérir et utiliser une application à moindre frais car le vendeur peut vendre plusieurs unités de cette application à différents clients et avoir une application qui peut davantage ressembler à l'entreprise.

Le développement

Comme dit précédemment, l'idée est de créer une application avec beaucoup de fonctionnalités différentes et permettre au client de pouvoir choisir celles qu'il souhaite. Lors du développement, il faut implémenter toutes les fonctionnalités de l'application. Une fois cette phase de développement faite, il faut faire la liste de ces fonctionnalités et déterminer si elles sont obligatoires, optionnelles et parmi ces fonctionnalités, définir si 2 fonctionnalités ne sont pas compatibles entre elles. Pour cela, on va dresser un *feature model* de l'application, un schéma de ces fonctionnalités (figure 4).

Une fois cette liste et ce schéma effectué, il va falloir détecter chacune de ces fonctionnalités dans le code et les annoter afin de pouvoir supprimer celles qui ne seront pas sélectionnées par le client (figure 5).

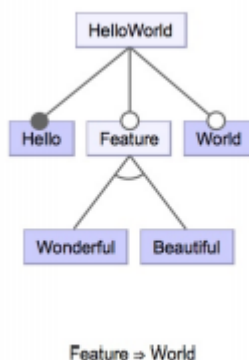


Figure 4 : exemple d'un feature model

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("Hello");
    #ifdef Beautiful
        printf(" beautiful");
    #endif
    #ifdef Wonderful
        printf(" wonderful");
    #endif
    #ifdef World
        printf(" world");
    #endif
    return 0;
}
```

Figure 5 : exemple d'un code C annoté
Correspondant au feature model

Sur la figure 4, nous avons un exemple d'un *feature model* d'un programme qui va afficher un message. Le contenu de ce message va être défini par ce que le développeur a choisi comme possibilité de variabilisation et du choix du client par la suite. Dans cet exemple, le minimum de fonctionnalités que l'on peut avoir c'est « Hello » quoi qu'il arrive, c'est une fonctionnalité obligatoire. Les autres sont optionnelles. La fonctionnalité « Feature » est optionnelle, si on décide de l'utiliser, il faut choisir entre « Wonderful » et « Beautiful », on ne peut pas avoir les deux simultanément. Et pour finir « World » est aussi une fonctionnalité optionnelle.

Sur la figure 5, nous avons le code source d'un programme qui va afficher ce message. On peut voir que chaque affichage défini dans le *feature model* est présent dans le code source est inversement. La fonctionnalité « Hello » qui correspond à l'affichage du mot « Hello » est obligatoire et n'est pas annotée, contrairement à toutes les autres qui sont optionnelles et donc annotées.

En combinant le schéma de la figure 4 avec le code de la figure 5, on peut en déduire qu'il existe 6 affichages de texte différents : « Hello », « Hello wonderful », « Hello beautiful », « Hello wonderful world », « Hello beautiful world » et « Hello world ».

On peut voir ici qu'avec quelques fonctionnalités, on a plusieurs possibilités de variabilisation et donc d'affichage.

2. Objectif

L'objectif est d'implémenter une ligne de produit d'une application web. Cette application web est un réseau social d'entreprise en se basant sur le cahier des charges de l'application Hive.

L'application propose à des personnes au sein d'une même entreprise de poster des messages, faire des sondages ou encore proposer des réunions.

Cette application est une application web développée en C# en se basant sur le motif d'architecture MVC (Model View Controller) composé de 3 modules :

- Les modèles (Model) : contient les données à afficher
- Les vues (View) : contient la présentation de l'interface graphique
- Les contrôleurs (Controller) : contient la logique concernant les actions effectuées par l'utilisateur

La base de données est une base de données MongoDB et le front-end est en CSS.

Le travail a été réalisé en 2 étapes :

- 1 Implémentation de l'application web
- 2 Ajout de la variabilité

III / Travail réalisé

Lien vers le github : <https://github.com/bouchenakihabib/HivePSTL>

1. Implémentation de l'application web

La première étape de développement a été d'implémenter l'application web en se basant sur le cahier des charges de l'application Hive.

Nous avons commencé par le module *Model* de l'architecture MVC en implémentant chacune des données que nous souhaitons avoir dans notre application.

Les modèles

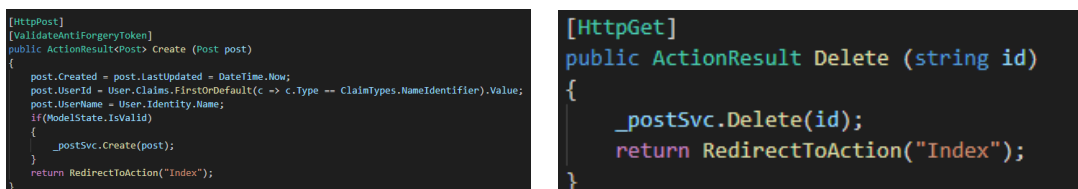
Le premier modèle qui a été implémenté a été le *UserProfile*. Ce modèle comporte toutes les données relatives à un utilisateur. Certaines données sont obligatoires telles que le nom et le prénom de l'utilisateur et son adresse mail. Un utilisateur a la possibilité de préciser son numéro de téléphone, son adresse, son métier, son pays et sa ville.

Le modèle *Post* permet de stocker toutes les informations relatives à un post. Chaque Post est défini par un identifiant unique. Cette donnée ainsi que le contenu du Post sont obligatoires. Le nom de l'auteur du post ainsi que son contenu, sa date de publication, et sa dernière date de modification sont aussi stockées.

La base de données *MongoDB* possède plusieurs collections correspondant à chacun des modèles. On a accès à chacune de ces collections grâce à la classe qui gère cette base de données.

Les controllers

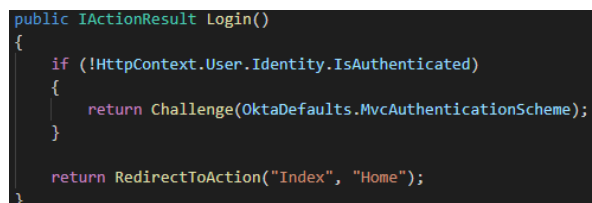
Toutes les fonctionnalités que vont utiliser les données des modèles sont présentes dans les controllers. Tout ce qui est méthode de création, suppression et modification sont gérées par les controllers.



```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create (Post post)
{
    post.Created = post.LastUpdated = DateTime.Now;
    post.UserId = User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier).Value;
    post.UserName = User.Identity.Name;
    if (ModelState.IsValid)
    {
        _postSvc.Create(post);
    }
    return RedirectToAction("Index");
}
```

```
[HttpGet]
public ActionResult Delete (string id)
{
    _postSvc.Delete(id);
    return RedirectToAction("Index");
}
```

Figure 6 : Création et suppression d'un Post



```
public IActionResult Login()
{
    if (!HttpContext.User.Identity.IsAuthenticated)
    {
        return Challenge(OktaDefaults.MvcAuthenticationScheme);
    }
    return RedirectToAction("Index", "Home");
}
```

Figure 7 : Login d'un User

Le code de la figure 6 représente le code du controller d'un Post. La méthode *Create* permet de créer un Post. On récupère la date de création et d'update, lors de la publication, ces deux dates sont identiques. A partir du *UserId* on récupère le nom de l'utilisateur puis on crée le Post à partir de ces données tout en respectant les contraintes du *Model*. La méthode *Delete* permet de supprimer un Post. On récupère l'identifiant de l'utilisateur et on vérifie si c'est celui du Post. Si c'est le cas, on supprime ce Post à partir de son identifiant.

Le code de la figure 7 est la méthode *Login* du modèle *User*. Dès qu'un utilisateur veut se login, il est redirigé vers le gestionnaire de connexion Okta ou Google en fonction de son choix. S'il se connecte, il est redirigé vers la page d'accueil du site.

Les views

Nous avons au moins un view par *Controller* (*Account*, *Home*, *Post*). Les views gèrent l'affichage du client.

```
<h1>User Profile</h1>

<div>
  <dl class="row">
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Profile.FirstName)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Profile.FirstName)
    </dd>
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Profile.LastName)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Profile.LastName)
    </dd>
    <dt class="col-sm-2">
      @Html.DisplayNameFor(model => model.Profile.Email)
    </dt>
    <dd class="col-sm-10">
      @Html.DisplayFor(model => model.Profile.Email)
    </dd>
  </dl>
</div>
```

Figure 8 : View du Profile

On affiche la liste des informations du *Profile* fourni par l'API Okta ou Google sous forme de Description List (dl).

```
<form asp-action="Edit">
  <fieldset class="idea-form">
    @Html.HiddenFor(m => m.Created)
    @Html.HiddenFor(m => m.UserId)
    @Html.HiddenFor(m => m.UserName)
    @Html.LabelFor(m => m.Content)
    @Html.TextAreaFor(m => m.Content)
    <input type="submit" name="Update"/>
  </fieldset>
</form>
```

Figure 9 : View d'édition d'un Post

Pour l'édition d'un post, l'utilisateur ne voit qu'une box d'édition où il peut modifier son *Post* ainsi qu'un bouton *Update*. Dans le code, *HiddenFor* permet de cacher ce qui est calculé en interne tel que le *last Updated*.

```
<footer class="footer text-muted" style="background: lightgrey">
  <div id="social-buttons">
    <a href="https://facebook.com/" class="fa fa-facebook"></a>
    <a href="https://twitter.com/" class="fa fa-twitter"></a>
    <a href="https://instagram.com/" class="fa fa-instagram"></a>
  </div>
  <div class="container">
    &copy; @DateTime.Now.Year - HivePSTL - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
  </div>
</footer>
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@RenderSection("Scripts", required: false)
```

Figure 10 : View *Shared*

La view *Shared* est une view qui est partagée et commune à toutes les autres view. On peut voir que le site a aussi des ressources tels que le logo, les réseaux sociaux ainsi qu'implémenter

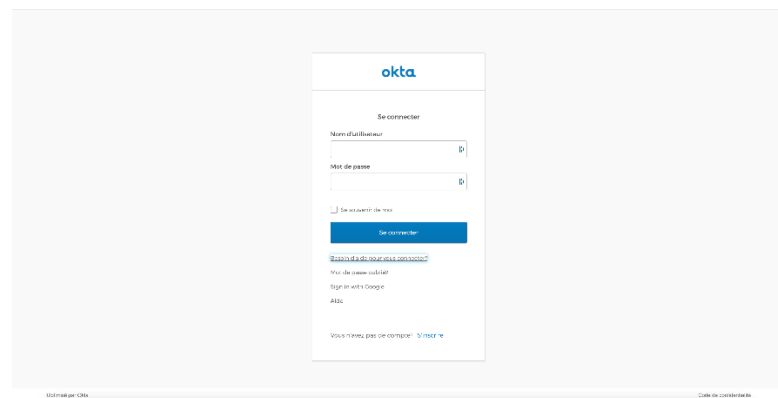


Figure 11 : Page de connexion

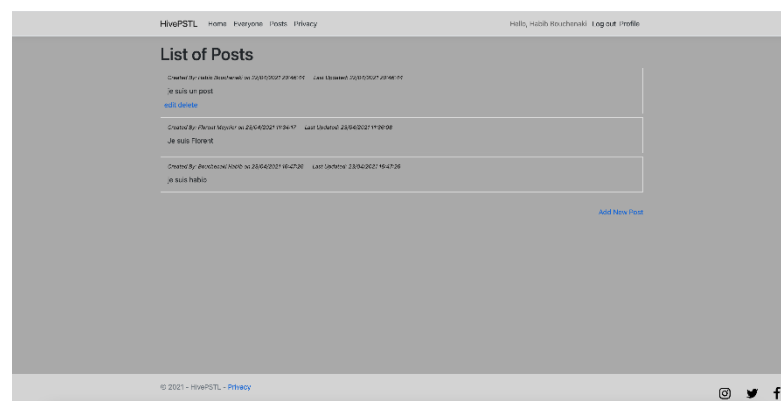


Figure 12 : Page des Posts

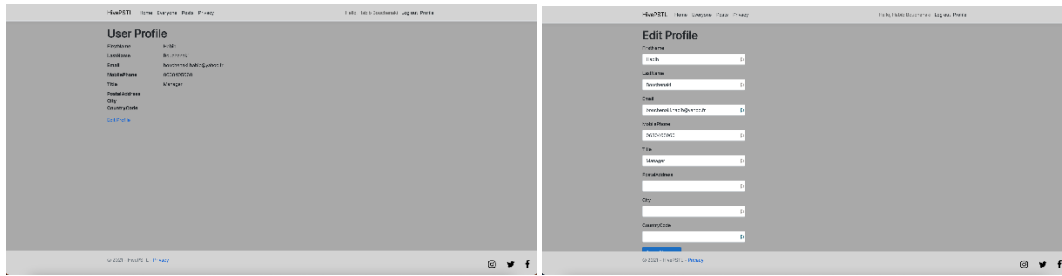


Figure 13 : Page du Profile

La figure 11 présente la page de connexion. Nous utilisons l'API Okta pour se connecter, mais il est aussi possible de se connecter grâce à Google.

La figure 12 présente la liste des Posts. Les posts sont triés par date de publication. On peut voir qui a publié le Post, à quelle date et quand a été faite la dernière modification. Si l'utilisateur connecté est celui qui a publié le Post, il a la possibilité de l'éditer ou de le supprimer.

La figure 13 présente le profil d'un utilisateur. Il a la possibilité d'éditer ses informations.

2. Ajout de la variabilité

Une fois l'application finale terminée, la partie variabilisation débute. Pour cela, nous devons détecter toutes les fonctionnalités et ressources de l'application puis créer le *feature model* de l'application.

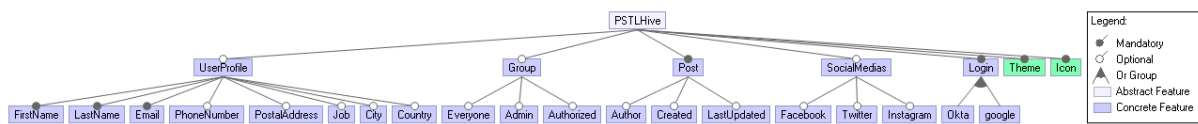


Figure 14 : Feature Model de l'application

Comme on peut le voir sur la figure 14, nous avons plusieurs fonctionnalités. Les fonctionnalités *Post*, *Login*, *Theme* et *Icon* sont obligatoires et *Group*, *SocialMedia* sont optionnelles. Certaines fonctionnalités possèdent d'autres fonctionnalités qui sont elles aussi variabilisables. La fonctionnalité *User* n'est pas obligatoire mais si elle est sélectionnée, les fonctionnalités *FirstName*, *LastName* et *Email* sont obligatoires alors que *PhoneNumber*, *PostalAdress*, *Job*, *City* et *Country* ne le sont pas. La fonctionnalité *Login* est obligatoire et il est nécessaire d'avoir au moins un type de connexion (Okta ou Google ou les deux). Les fonctionnalités qui sont en vert sont les ressources. Ces ressources sont obligatoires et peuvent aussi être modifiées par le client.

Une fois les fonctionnalités détectées, il faut annoter celles qui sont optionnelles et les ressources qui sont configurables. Pour cela, on met 2 commentaires qui encadrent tous les endroits du code où une référence à cette fonctionnalité est faite.

```

//if[UserProfile]
public class UserProfileViewModel
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    //if[mobilePhone]
    public string MobilePhone { get; set; }
    //endif[mobilePhone]

    public string PostalAddress { get; set; }

    //if[title]
    public string Title { get; set; }
    //endif[title]

    //if[city]
    public string City { get; set; }
    //endif[city]

    //if[countrycode]
    public string CountryCode { get; set; }
    //endif[countrycode]
}
//endif[UserProfile]

```

Figure 15 : Variabilisation du modèle
UserProfile

Comme on peut le voir sur la figure 15, le modèle *UserProfile* a été annoté en fonction des fonctionnalités qui sont variabilisables. Les champs *FirstName*, *LastName*, et *Email* qui sont obligatoires ne sont pas annotés alors que *MobilePhone* par exemple est précédé d'un « *//if[mobilePhone]* » et suivi d'un « *//endif[mobilePhone]* ». Cela signifie que si la fonctionnalité *PhoneNumber* n'est pas sélectionnée, alors tout ce qui se trouve entre le balisage correspondant à cette fonctionnalité sera supprimé. Si autre part dans le code, il y a une référence au numéro de téléphone, cette référence doit être annotée avec le même mot clé (*mobilePhone* par exemple pour le numéro de téléphone). Néanmoins, sachant que la fonctionnalité *UserPorfile* est optionnelle, si celle-ci n'est pas sélectionnée, tout ce qui est à l'intérieur des balises *Profile* sera supprimé. Même chose concernant les autres fonctionnalités qui sont variabilisables.

Cette suppression ou (modification pour les ressources) de code est faite à l'étape suivante qui s'appelle : la dérivation.

3. Dérivation

La dernière partie pour la création d'un site variabilisable est la dérivation. La dérivation ne fait pas partie de notre projet mais nous allons expliquer son fonctionnement.

Lorsque le client effectue sa variabilisation, il doit choisir quelles fonctionnalités il souhaite avoir ou ne pas avoir, mais aussi les ressources qu'il souhaite modifier avec quels fichiers css/contenu. Une fois son choix fait, un outil génère un fichier au format JSON qui contient chacune des fonctionnalités avec une information concernant son existence ou non ainsi que les ressources avec le chemin vers les nouvelles ressources.

Une fois ce fichier généré, un script va s'exécuter et parcourir chacune des fonctionnalités du JSON, puis parcourir le code source de l'application afin de supprimer ou modifier les fonctionnalités en fonction du contenu du fichier JSON.

IV / Conclusion

La première étape, consistait à créer une application en se basant sur le cahier des charges de l'application Hive. La création de notre application a nécessité de nombreuses recherches sur les nouvelles technologies nécessaires. Cette étape est une étape classique de développement d'application web, mais il a fallu implémenter davantage de fonctionnalités que nécessaire pour une variabilisation intéressante.

La seconde étape a consisté à variabiliser cette application en la rendant configurable. Cette variabilisation a été précédée de recherches concernant cette nouvelle approche qui nous ont aidé dans cette étape. Suite à cela, notre application web peut être configurée par plusieurs personnes et peut donc avoir un intérêt différent ainsi qu'une apparence différente.

Dans une vision plus générale, la variabilisation d'une application peut être intéressante afin de concevoir une seule application ayant des usages différents pour différentes entreprises ou clients. Cette technologie peut permettre à un client qui n'est intéressé que par une petite partie de l'application, de pouvoir supprimer toutes les fonctionnalités qui ne l'intéressent pas mais aussi lui permettre de lui donner l'apparence qu'il souhaite.

La réalisation de ce projet est passée par l'utilisation de nouvelles technologies telles que le Framework ASP.NET MVC, MongoDB Atlas et l'utilisation d'API telle qu'Okta ou Google. Ceci a contribué à élargir notre bagage en tant que développeurs. Suite à toutes les étapes décrites précédemment nous sommes parvenus à réaliser l'objectif principal de ce projet, à savoir implémenter une ligne de produit d'une application web. Ainsi, maintenant que nous avons réalisé une application web variabilisable, il serait pertinent de s'intéresser à un outil permettant de sélectionner et de déployer rapidement les fonctionnalités voulus ou sélectionnées par le client.