Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/gil

Gestion d'identité en ligne

TP Web : Création d'un site web personnel

Objectifs de ce TP :
- Poser les bases du développement d'un site web personnel avec HTML et CSS.
- Utiliser des scripts pour se faciliter la vie.
- Utiliser un `Makefile` pour automatiser la génération du site.
- **Rappel :** travaillez dans un dossier approprié par exemple `~/s1/gil/siteweb/`.

Rappel des liens de documentation :
- HTML : `https://developer.mozilla.org/fr/docs/Web/HTML`
- CSS : `https://developer.mozilla.org/fr/docs/Web/CSS`
- Makefile : `https://makefiletutorial.com/`

Validateurs de code :
- HTML : `https://validator.w3.org/`
- CSS : `https://jigsaw.w3.org/css-validator/`

Exercice 0.

Structure du site web et première version des pages.

1. L'objectif est de réaliser une site web personnel de quelques pages en utilisant HTML et CSS.

   Commençons par la page d'accueil, `index.html`. Comme on l'a vu en cours, celle-ci doit déclarer dans son entête son encodage et un titre avec votre nom, et son corps doit contenir trois éléments : 1- une section de titre (balise `header`) contenant elle-même un titre de premier niveau (balise `h1`), 2- une section de contenu principal de la page (balise `main`) contenant elle-même un titre de deuxième niveau (`h2`) "Bienvenue" puis des informations vous présentant succinctement, 3- une section de pied de page (balise `footer`) contenant elle-même un paragraphe rappelant votre nom et la dernière date de mise à jour de la page.

   → Créez un fichier `index.html` (sans passer trop de temps sur le fond de votre présentation, vous aurez le temps plus tard pour ça).

2. En plus de la page d'accueil, on veut au moins des pages pour les projets, le CV, les liens.

   → Créez rapidement ces pages et ajoutez à chacune dans leur corps, juste après le titre de premier niveau à l'intérieur du `header`, un élément de navigation (balise `nav`) contenant une liste (balises `ul` et `li`) de liens (balise `a`) vers les différentes pages de votre site.

   Pensez à ajouter ce menu dans le fichier `index.html` également.

   Pensez à adapter le titre déclaré dans l'entête de chaque page, et à ajouter une classe (attribut `class`) "`current`" sur le lien de la page actuelle dans le menu de navigation, ce qui permettra plus tard de personnaliser son apparence avec CSS.

3. On va maintenant s'occuper un peu de la présentation du site.

   → Créez un fichier `styles.css` vide pour l'instant, et liez le à vos pages web (balise `link` dans l'entête).

4. Vous pourrez laisser libre court à votre imagination et adapter votre site à votre personnalité plus tard, mais pour l'instant suivez le guide ci-dessous.

   (a) → Choisissez une couleur principale et stockez la dans la variable `--accent-color` que vous déclarez dans votre fichier CSS pour l'élément racine (`html`). Vous pourrez ainsi utiliser sa valeur partout ailleurs avec la syntaxe `var(--accent-color)`.

   (b) → Donnez une largeur maximale (`max-width`) raisonnable (800 pixels par exemple) au corps de la page, et centrez son affichage dans la fenêtre en lui mettant des marges (`margin`) horizontales automatiques.

   (c) → Centrez le titre principal (`text-align`), affichez le de la couleur principale (`color`) dans une police sans sérifs (`font-family`).

   (d) → Mettez à zéro les marges de la liste de navigation et indiquez que ses éléments doivent être affichés (`display`) en ligne.

   (e) → Stylisez les liens du menu de navigation en les affichant de la couleur principale, sans décoration (`text-decration`), et en gras (`font-weight`), avec un peu (1em) de remplissage (`padding`) latéral, et des bordures (`border`) continues de de 2px de large et de la couleur principale.

(f) → Rendez le menu de navigation un peu plus interactif en mettant 70% d'opacité (`opacity`) à ses liens par défaut, et revenez à 100% d'opacité pour le lien de la page courante et pour celui sur lequel passe la souris.

(g) → Mettez les mêmes bordures qu'aux liens du menu de navigation à la section principale, et une couleur de bordure du bas (`border-bottom-color`) blanche au lien de la page courante dans le menu de navigation, pour simuler des onglets. Mettez aussi un remplissage de 1.5em à la section principale pour éviter que le texte ne soit trop collé au bord.

(h) → Pour rendre plus sympathique le rendu, mettez des bords arrondis de 0.5em à la section principale (`border-radius`), et faites pareil pour les coins haut-gauche et haut-droite (`border-top-left-radius`, `border-top-right-radius`) des liens du menu de navigation.

(i) → Toujours pour améliorer le rendu, supprimez les marges des titres de second niveau dans votre section principalen et mettez leur une bordure du bas (`border-bottom`) en pointillés, large de 2 pixels, et de la couleur principale.

(j) → Par cohérence thématique, faites s'afficher les liens contenus de la section principale de la couleur principale.

(k) → Enfin, centrez et affichez en gris et en plus petit (`font-size`) le texte du paragraphe du pied de page.

Exercice 1.

Factoriser le code commun et scripter la génération des pages.

1. À part notre code CSS qui est déjà commun à toutes les pages du site, il y a beaucoup de répétitions entre celles-ci : l'entête est le même sauf pour le contenu de la balise `title`, le titre du site est le même, le menu aussi, et le pied de page également.

   Si on veut rajouter une page au site par exemple, ce n'est vraiment pas pratique : il faut aller modifier le menu autant de fois qu'il y a de pages, sans en oublier aucune.

   Le but de cet exercice va être d'écrire un script permettant de factoriser le code commun de nos pages.

   On va commencer par créer les dossiers nécessaires :
   — `layout/` contiendra les parties factorisées du code,
   — `pages/` contiendra les parties spécifiques à chaque page,
   — `assets/` contiendra les ressources (feuilles de styles, images, etc.),
   — `public/` contiendra le site une fois construit,
   — `scripts/` contiendra les scripts que l'on va écrire.
   → Créez ces dossiers avec la commande `mkdir`.

2. On va découper nos fichier HTML en trois :
   — tout le début commun, jusqu'au `<main>` inclus, va dans `layout/before.html`,
   — la partie spécifique du fichier, le contenu de l'élément `main`, va dans le dossier `pages/`,
   — toute la fin commune, à partir du `</main>` inclus, va dans `layout/after.html`.
   → Commencez par découper de la sorte votre fichier `index.html`, en mettant son contenu principal dans `pages/index.html`.

3. → Créez dans le dossier `pages/` les fichiers correspondant à chacune de vos pages avec le contenu de leur section principale.

4. On va maintenant créer une première version de notre script.

   (a) → Copiez le code ci-dessous dans le fichiers `scripts/buildpage.sh`. N'hésitez pas à consulter les pages de manuel des commandes ou à chercher sur le web pour comprendre ce qui s'y passe.

```
#!/bin/bash

function buildpage {
  cat layout/before.html
  cat "${1}"
  cat layout/after.html
}

if [[ "${1}" = "" ]]; then # premier argument obligatoire : le nom de la page
  echo "Usage: ${0} PAGE" >&2
  exit 1
elif test ! -f "${1}"; then # et doit correspondre à une page existante
  echo "${0}: error: ${1}: no such file" >&2
  exit 1
else # si tout va bien on construit la page dans public/
  FILE="${1#pages/}"
  buildpage "${1}" > "public/${FILE}"
fi
```

   (b) → Rendez le exécutable avec la commande `chmod`.

(c) → Normalement si vous appelez le script `scripts/buildpage.sh` en lui passant comme argument le fichier `pages/index.html`, il devrait produire dans le dossier `public/` une copie de votre fichier `index.html` original avant la découpe.

(d) → Quel est le soucis pour les autres fichiers ?

5. Il nous faut un moyen d'adapter à la volée les fichiers du dossier `layout` au moment de la construction de la page. On va pour cela pourvoir utiliser la commande `sed`, le "stream editor", qu'on peut combiner à d'autres commandes grâce aux *pipes* (`|`). Une des choses que sait faire `sed`, c'est de rechercher et remplacer des motifs.

   La syntaxe pour ça est `s~MOTIF~REMPLACEMENT~`, à passer en premier argument de la commande `sed`.

   Par exemple si j'ai un fichier `toto.txt` et que je veux l'afficher mais en ayant remplacé dans donc contenu les occurrences de "`xxx`" par "`yyy`", je peux écrire la commande : `cat toto.txt | sed "s~xxx~yyy~"`.

   → Commencez par préparer le fichier `layout/before.html` en retirant le "` class="current"`" du lien vers le fichier `index.html` et en mettant "`{{TITLE}}`" dans la balise `title` là où vous voulez que le titre de chaque page vienne se mettre.

   Même chose pour le fichier `layout/after.html` en mettant "`{{DATE}}`" dans le pied de page là où vous voulez que la date de dernière mise à jour de la page vienne se mettre.

6. Commençons par la classe "`current`" à mettre sur le lien du menu de navigation vers la page en cours de construction, pour que notre rendu visuel en onglets fonctionne de nouveau :
   — On veut chercher dans le fichier `layout/before.html` le texte "`href="`*`FICHIER`*`"`" (si on est en train de construire *`FICHIER`*), et le remplacer par lui même suivi du texte "` class="current"`" pour ajouter la classe au lien.
   — On a besoin du nom du fichier pour construire le motif, notre fonction `buildpage` reçoit le chemin vers le fichier (avec le dossiers "`pages/`" avant le nom du fichier proprement dit) donc on peut le récupérer à partir de là.
   — Comme on manipule des chaînes HTML avec des guillemets doubles (`"`) dedans, il faut les *échapper* avec un backslash (`\`) à l'intérieur de la chaîne donnée en argument à `sed` pour que ça ne soit pas confondu avec la fin de l'argument.
   — Dans la texte de remplacement de la commande `sed`, on peut utiliser le symbole `&` pour remettre tel quel le motif.

   → Dans la fonction `buildpage`, créez une variable `FILE` contenant le nom du fichier en construction, puis utilisez la pour faire le remplacement dans le contenu de `layout/before.html` avec la commande `sed`.

   Vérifiez que cela fonctionne correctement avec toutes vos pages.

7. Continuons avec le titre. Il nous faut un moyen de le définir pour chaque page et de le récupérer dans notre script. Le plus simple est de décider d'une convention, par exemple de mettre le titre sur la toute première ligne de nos fichiers dans le dossier `pages/`. Par exemple le fichier `pages/index.html` commençait jusque là par une ligne "`<h2>Bienvenue</h2>`", et on va maintenant placer avant cette ligne une nouvelle première ligne avec simplement "`Accueil`".

   Sachant que :
   — On peut récupérer dans une variable le résultat d'une commande avec la syntaxe *`VAR`*`=$(`*`COMMAND`*`)`.
   — La commande `head` permet d'afficher le début d'un fichier, on peut lui spécifier le nombre de ligne qu'on veut (par défaut c'est 10).
   — La commande `tail` permet d'afficher la fin d'un fichier, on peut lui spécifier le nombre de ligne qu'on veut depuis la fin ou à partir de la combientième ligne en partant du début on veut commencer l'affichage jusqu'à la fin.

   → Dans la fonction `buildpage`, créez une variable `TITLE` contenant le titre de la page récupéré sur la première ligne du fichier reçu en argument, puis utilisez la pour faire le remplacement de "`{{TITLE}}`" dans le contenu de `layout/before.html` avec la commande `sed`.

   Pensez aussi à remplacer la commande `cat` qui affiche le contenu principal de la page par une commande permettant de sauter la première ligne du fichier.

   Évidemment, vérifiez que cela fonctionne correctement avec toutes vos pages.

8. Enfin, on va faire le remplacement de "`{{DATE}}`" par la date du jour, qu'on peut récupérer avec la commande `date` (voir sa page de manuel pour personnaliser le format).

   → Dans la fonction `buildpage`, créez une variable `DATE` contenant la date actuelle récupérée grâce à la commande `date`, puis utilisez la pour faire le remplacement de "`{{DATE}}`" dans le contenu de `layout/after.html` avec la commande `sed`.

   Vérifiez que cela fonctionne bien avec toutes vos pages.

9. Et voilà ! Maintenant, pour construire votre site il suffit :
   — de copier le contenu du dossier `assets/` dans le dossier `public/`,
   — d'appeler le script `scripts/buildpage.sh` pour chaque page.

On peut automatiser tout ça en écrivant un script qui va supprimer complètement le dossier public, le recréer vide, puis faire les opérations décrites ci-dessus à l'aide de la commande **cp** pour le premier point, et à l'aide d'une boucle **for** pour le second.

Vous pouvez recopier le script suivant dans **scripts/buildsite.sh** et le rendre exécutable pour pouvoir vous en servir en attendant d'avoir mieux (cf exercice suivant) :

```
#!/bin/bash

echo "Rebuilding site."
rm -rf public/                  # tout supprimer !
mkdir public                    # recréer le dossier
cp -r assets/* public/          # copier les ressources
for page in pages/*.html; do    # pour chaque page ...
  echo -n "> building ${page#pages/}... "
  scripts/buildpage.sh "$page"  # ... la construire avec notre script
  echo "done."
done
echo "Finished!"
```

Exercice 2.

Utiliser un **Makefile** pour automatiser la génération du site.

1. Le problème avec le script **buildsite.sh**, c'est qu'il refait tout à chaque fois, même quand on a modifié qu'une seule page par exemple.

   Il existe une solution toute faite et très pratique pour ce genre de situation : l'utilitaire **make**.

   La commande **make** sert à construire et reconstruire des projets en ne (re)faisant que le strict nécessaire pour arriver au résultat final. Pour cela elle a besoin de "recettes" (appelée "règles") qu'on lui indique dans un fichier "**Makefile**".

   Chaque recette/règle est indiquée dans le fichier **Makefile** par la syntaxe suivante [1] :

```
cible: dépendances
        commandes
```

   La *cible* est le fichier à produire, les *dépendances* sont les fichiers dont la cible dépend, au sens où il faut la reconstruire si et seulement si les dépendances ont été modifiées plus récemment qu'elle, et les *commandes* sont la ou les commandes permettant de construire la cible à partir des dépendances.

   Si il y a plusieurs cibles définies dans un **Makefile**, la commande **make** va lancer la construction de la première par défaut, et sinon on peut lui dire laquelle on veut sur sa ligne de commande.

   Par exemple dans notre cas, le fichier **public/index.html** dépend des fichiers **pages/index.html** mais aussi de **layout/before.html** et **layout/after.html**, et pour le fabriquer on utilise notre script en lançant la commande **./scripts/buildpage.sh pages/index.html**. En syntaxe **Makefile**, cela donne la règle suivante :

```
public/index.html: pages/index.html layout/before.html layout/after.html                # code 1 #
        ./scripts/buildpage.sh pages/index.html
```

   On pourra alors lancer la construction de cette page avec la commande **make public/index.html**. Cette commande ne fera rien si le fichier **public/index.html** est déjà plus récent que ses dépendances, elle n'exécutera notre script **buildpage.sh** que si c'est nécessaire.

   C'est déjà ça, mais pour l'instant, ça ne nous avance pas beaucoup en terme d'automatisation ! Il nous faut un moyen de lister automatiquement les pages existantes et de construire toutes celles qui ont besoin de l'être.

   Pour ça on peut définir une variables **PAGES** qui liste tous les fichiers HTML dans le dossier **pages/**, et ensuite créer une première cible *factice* [2] **all** qui aura pour dépendances toutes nos pages construites dans le dossier **public/** :

```
PAGES=$(wildcard pages/*.html)         # wildcard permet d'utiliser l'étoile pour lister des fichiers
all:: $(subst pages/,public/,$(PAGES)) # subst fait une substitution dans chaque élément de la liste
```

   Avec ça et en mettant des règles pour chaque fichier HTML à l'instar de celle donnée dans le code 1 pour notre fichier **index.html** ci-dessus, on a déjà une génération automatisée qui fonctionne bien pour les pages HTML, mais ça fait encore beaucoup de code répétitif :

```
public/index.html: pages/index.html layout/before.html layout/after.html                # code 2 #
        ./scripts/buildpage.sh pages/index.html
public/cv.html: pages/cv.html layout/before.html layout/after.html
        ./scripts/buildpage.sh pages/cv.html
```

1. Notez que les **commandes** doivent être indentées avec une vraie tabulation.
2. Une cible *factice* est une cible qui ne correspond pas à un vrai fichier ou dossier, on l'indique en la faisant suivre d'un double deux-points plutôt que d'un simple.

```
5 public/projets.html: pages/projets.html layout/before.html layout/after.html
6         ./scripts/buildpage.sh pages/projets.html
7 public/liens.html: pages/liens.html layout/before.html layout/after.html
8         ./scripts/buildpage.sh pages/liens.html
```

Il est possible de factoriser ce code en utilisant le *joker* `%` et des *variables magiques* des `Makefile` :

— Le symbole `%`, quand il est utilisé dans une cible et dans ses dépendances, est remplacé automatiquement par la même chose des deux côtés pour toutes les possibilités. Par exemple "`%.png: %.jpg`" permet d'exprimer n'importe quelle cible en ".png" dont la dépendance a le même nom mais en ".jpg" (par exemple pour une commande de conversion de formats d'image).

— Il y a quatre variables magiques utilisables dans les commandes :
  — `$@` est remplacée par le nom de la cible ;
  — `$^` est remplacée par la liste de toutes les dépendances ;
  — `$<` est remplacée par la première dépendance ;
  — `$?` est remplacée par la liste des dépendances plus récentes que la cible.

→ Écrivez un `Makefile` pour la génération des pages HTML de votre site en utilisant les astuces que l'on vient d'évoquer pour remplacer le code 2 ci-dessus par une règle seule générique.

Vérifiez bien sûr que votre code fonctionne correctement : chaque fois que vous appelez la commande `make` il ne doit reconstruire que les pages web dont vous avez mis à jour le fichier dans le dossier `pages/` ou toutes les pages web si vous mettez à jour un des deux fichiers du dossier `layout/`.

2. Il nous reste maintenant à gérer la mise à jour des ressources du site (feuilles de styles, images, etc.) contenues dans le dossier `assets/`. Il s'agit de construire le dossier `public/` à jour, avec toutes les ressources dedans. C'est en fait techniquement une dépendance de la construction de notre site : le dossier doit exister avant qu'on puisse construire nos pages.

   (a) → Ajoutez `public` comme première dépendance sur la règle `all`.

   (b) → Ajoutez une nouvelle règle pour la construction de `public`. Cette cible doit dépendre de l'ensemble des ressources (fichiers du dossier `assets/`) et doit consister en la création du dossier `public/` avec `mkdir` si celui-ci n'existe pas (et sans erreur si il existe déjà), puis en la copie des ressources plus récentes que le dossiers `public/`, puis en la mise à jour de la date de modification du dossier `public/` avec la commande `touch`. Pensez à utiliser les variables magiques si besoin !

3. C'est toujours une bonne idée d'avoir un moyen de nettoyer un projet des fichiers générés automatiquement. La convention est d'appeler cette règle "`clean`". Ici il s'agit de supprimer complètement le dossier `public/` et tout son contenu.

   → Ajoutez une cible factice "`clean`" dans votre `Makefile` qui nettoie le projet.

4. (Bonus) Pour un site statique les navigateurs sont capables de lire directement les fichiers sur votre ordinateurs, mais c'est toujours une bonne idée de tester votre site à travers un véritable serveur web.

   Vous pouvez utiliser par exemple le serveur web fourni avec Python dans son module `http.server`.

   → Ajoutez une cible factice "`test`" dans votre `Makefile` qui lance un serveur web en local pour vous permettre de tester votre site en condition réelle. Cette règle doit dépendre de la cible `all` pour être sûr que vous testez une version à jour de votre site.