Test technique d'intégration

Le test d'intégration est disponible sous l'url suivante : https://app.thomas-claireau.fr/lego-integration ou en utilisant l'archive envoyée par email (ou les fichiers sur GitHub).

Sommaire

Dans ce readme, je vais revenir globalement sur tous mes choix techniques d'intégration.

Si vous voulez davantage de précisions ou d'autres questions concernant des choix techniques que je n'ai pas abordés, n'hésitez pas à me contacter !

Preloader

La page dispose d'un preloader simple afin de charger la page complètement avant d'afficher les éléments.

Ce preloader a été réalisé en 3 étapes :

1. Ajout d'une div portant la class `preloader` juste après l'ouverture de la balise `body`
2. Ajout de code CSS spécifique à cette div afin qu'elle soit en pleine largeur / hauteur. Son `z-index` doit être plus élevé que n'importe quel autre élément de la page afin qu'elle passe au dessus de tous les éléments.
3. Ajout d'un code JavaScript spécifiant qu'au chargement complet de la page, la div `preloader` disparait progressivement (`fadeOut`).

CSS relatif à la page entière

Afin de respecter la compatibilité avec tous les navigateurs, j'ai effectué un reset des marges sur les balises html et body :

```
html, body {
    margin: 0;
    padding: 0;
}
```

Et pour respecter aussi la largeur du contenu indiquée dans la maquette psd, j'ai créé une classe `container` avec les propriétés css suivantes :

```
.container {
    width: 1220px;
    margin: 0 auto; /* alignement du contenu au centre */
}
```

Header & Menu (durée ≈ 2h)

Le logo est bien intégré en SVG grâce à la balise `<img>`.

Le header a été mis en page grâce à Flexbox. La partie responsive de la page a elle aussi été réalisée avec Flexbox.

L'effet au survol des menu est entièrement géré par le CSS grâce au pseudo-élément `hover`.

Concernant les effets au survol, j'ai prévu une animation simple en css :

```
@keyframes anim-sous-menu {
    from {
        opacity: 0;
    }

    100% {
        opacity: 1;
    }
}
```

Responsive

En dessous de 936px de largeur, le menu actuel est remplacé par un menu "responsive" qui s'affiche en cliquant sur un icone de navigation. Le tout est géré en CSS grâce aux media queries ainsi qu'avec du javascript pour ouvrir le menu au click sur l'icone de navigation

Headings (durée ≈ 30min)

Les titres h1 et h2 sont pourvus de deux lignes dont la largeur s'adapte en fonction de la taille de l'écran ainsi qu'en fonction de la taille qu'occupe le texte.

Cela a été réalisé grâce aux pseudo-éléments `before` et `after` et à la propriété css `display: table` que j'ai choisi pour sa compatibilité avec la plupart des navigateurs.

Pour éviter de me répéter au niveau du code et aussi parce que les titres h2 étaient différents sur la page, j'ai choisi d'utiliser une classe `yellow` pour les titres jaunes et le sélecteur `h2` pour les autres (rouge avec barre en pointillés jaunes)

Slider (durée ≈ 5h)

Cela a été la partie la plus compliquée du test d'intégration.

J'ai réalisé facilement le slider en JavaScript (+ jQuery), mais l'intégration d'une progress bar pour informer l'utilisateur du temps d'affichage de l'item en cours m'a posé quelques soucis.

Après des recherches, j'ai trouvé sur CodePen un exemple de slider incluant une progress bar dont je me suis inspiré.

Dans les parties qui suivent, je reviens brièvement sur l'implémentation des fonctionnalités demandées pour le slider :

Adaptation à un nombre dynamique d'items

Chaque item est dans une div portant la classe commune `diapo` et la classe `diapo-x` ou x est le numéro de la diapo. Par exemple, la structure pour intégrer 3 images au slider est la suivante :

```
<div class="diapo diapo-1"></div>
<div class="diapo diapo-2"></div>
<div class="diapo diapo-3"></div>
```

Les images sont ensuite rajoutées en CSS grâce à la propriété background. Voici un exemple pour la première div de l'exemple précédent :

```
.diapo-1 {
    background: url('mon-image');
}
```

Navigation de droite à gauche avec les flèches

La navigation de droite à gauche avec les flèches est réalisée entièrement en javascript.

Le code HTML des deux flèches est lui aussi ajouté en JavaScript (de façon à éviter d'afficher les flèches si l'utilisateur a désactivé JavaScript).

> Tout le code JS relatif au slider est dans un fichier spécifique : slider.js

Pour afficher les flèches, le code intègre une condition : *Si le nombre de diapo est supérieur à 1, les flèches sont ajoutées*. Si la condition n'est pas remplie, les flèches n'apparaissent pas.

Dans cette même condition, vous pouvez voir l'appel à la fonction `startProgressbar()` qui s'occupe d'afficher la barre de progression. Si la condition précédente n'est pas remplie, la progress bar ne s'affiche donc pas.

Pour naviguer d'un item à l'autre, dans un sens comme dans l'autre, j'ai "écouté" l'évènement click sur les flèches droites et gauches.

Si le click est détecté, la fonction associée fait plusieurs choses :

Sur la flèche de droite :

- On incrémente de 1 une valeur `i` initialement à 0 qui va nous servir à sélectionner l'image suivante du slider.
- Si la valeur de i est inférieure ou égale au nombre d'images dans le slider, on décide de cacher toutes les images sauf celle sélectionnée grâce à la fonction JS `eq()`. Voici un exemple :

```
$img.css('display', 'none');
$currentImg = $img.eq(i);
$currentImg.fadeIn("slow");
```

- Sinon, si la valeur de i est supérieure au nombre d'image dans le slider, on réinitialise la valeur de i à 0 puis on affiche l'image comme précédemment.

Sur la flèche de gauche :

- On décrémente de 1 une valeur `i` initialement à zéro qui va nous servir à sélectionner l'image suivante du slider.
- Si la valeur de i est supérieure ou égal à 0, on décide de cacher toutes les images sauf celle sélectionnée grâce à la fonction JS `eq()`. Voici un exemple :

```
$img.css('display', 'none');
$currentImg = $img.eq(i);
$currentImg.fadeIn("slow");
```

- Sinon, on réinitialise la valeur de i à 0 puis on affiche l'image comme précédemment.

Navigation d'un item à l'autre via les dots en bas

Tout d'abord, il est nécessaire de pouvoir changer dynamiquement le nombre de dots en fonction du nombre d'images dans le slider.

J'ai réalisé ceci grâce à une boucle for et à une variable `addId` initialisée à 1.

Tout comme les flèches, les dots sont rajoutés via JavaScript à la page.

La balise `ul` (parente des balises `li` qui représente les dots) est soumise à la même condition que les flèches. Elle ne s'affichera pas sur la page si le slider ne comporte qu'une image.

Ensuite, chaque balise `li` est ajoutée dynamiquement en fonction du nombre d'item du slider, grâce à une boucle `for`. De plus, chaque balise `li` possède un identifiant unique de la forme `carousel-x` où x est le numéro du dot (qui est le même que l'image en cours d'affichage sur le slider). Voici la boucle for :

```
let addId = 1;

for (let a = 0; a <= indexImg; a++) {
```

```
    $('.dots').append('<li class="carousel-buttons" id="carousel' + addId++ + '">
</li>');
}
```

Grâce au système d'identifiant unique associé au numéro de l'image, chaque dot est cliquable et renvoi vers l'image sélectionnée.

Pour faire ceci, plusieurs étapes :

1. Je récupère l'identifiant unique de chaque dot avec la fonction `attr()`
2. J'utilise la fonction `split()` pour transformer l'id en un tableau de sous-chaine.
3. Cela me permet ensuite de sélectionner seulement le numéro x, comme je le ferais avec un tableau
4. Il suffit ensuite de procéder comme avec les flèches et de sélectionner l'image à afficher avec `eq()`. Seule différence : il faut soustraire 1 au numéro récupéré à l'étape 3 car la fonction `eq()` récupère les images à partir du numéro d'index 0 :

- 0 : image n°1
- 1 : image n°2
- ...

Durée d'affichage d'un item et progress bar

Cette partie a été compliquée pour moi car j'avais commencé à construire mon code autour de la fonction `slideImg()` qui s'occupait d'activer le défilement automatique (sans progress bar).

En implémentant une progress bar, cette dernière devient une sorte de "contrôle" qui décide de faire défiler les images ou non.

J'ai donc changé mon code de cette façon :

La fonction `startProgressBar()` permet d'activer le défilement automatique avec une durée d'affichage de 4 secondes par slide.

Du coté HTML, j'ai intégré une div avec la classe `progress`, parente d'une div avec la classe `bar`, initialement à `width: 0%`.

Le but ici est de remplir la progress bar sur 4 secondes, puis de passer les images une par une dès que la largeur de la div avec la classe `bar` est égal à 100%.

Pour rendre la progression de la bar fluide, je me suis inspiré de l'exemple sur CodePen qui a utilisé les pourcentages.

Je l'ai modifié pour que la progression soit égale à 4 secondes. Explication :

- La fonction qui englobe tout le code du slider demande une valeur `timer` : c'est une valeur en milliseconde qui indique l'intervalle de répétition de la fonction `interval()`. Plus la valeur `timer` est élevée, moins la progress bar est fluide, et inversement.
- La fonction `interval()` se charge donc d'incrémenter de 0.5% la largeur de la bar toutes les 20 millisecondes. On arrive donc à 100% au bout de 4 secondes.

Enfin, la progress bar doit se réinitialiser (revenir à une largeur de 0%) à :

- chaque click sur une des flèches de défilement
- chaque click sur un dot
- chaque fois qu'une image défile automatiquement

Pour faire ceci, à chaque évènement click sur les flèches ou les dots, ainsi que sur chaque défilement en automatique, je fais appel à la fonction `startProgressbar()` qui fait deux choses :

1. Appel à la fonction `resetProgressbar` qui remet la largeur de la bar à 0% et stoppe le `setInterval()`
2. Relance le `setInterval()` qui incrémente de 0.5% la largeur de la bar, toutes les 20 millisecondes.

Bloc 3 colonnes (durée ≈ 1h)

Les 3 blocs de largeurs égales ont été mis en page avec Flexbox.

Les traits de séparation entre les images ont été réalisés avec le pseudo-élément `after`.

Pour coller le plus fidèlement possible à la maquette, j'ai choisi de garder les images déjà présentes et d'intégrer les animations qui étaient proposées dans les exemples.

Pour chaque image, j'ai recréé une deuxième image légèrement différente :

- Image n°1 : j'ai repris la brique de Lego qui sourit, et j'ai changé le sourire par un sourire plus grand
- Image n°2 : j'ai empilé les briques sur Photoshop
- Image n°3 : j'ai repris le bonhomme Lego, et je lui fait lever le bras droit

Au survol de chaque bloc, un script jQuery avec une fonction `animImg()` se déclenche et permet de remplacer l'image par celle modifiée. Lorsqu'on quitte le survol, l'image 1 revient à la même place.

Iframe Youtube (durée ≈ 20min)

J'ai encadré l'iframe YouTube dans une div portant la classe `video` qui m'a permis de fixer une largeur max de 1100px ainsi qu'un `margin:auto` pour centrer la vidéo.

A partir de là, quand on redimensionne, on perd le ratio d'origine et des bordures noires apparaissent quand on réduit la taille de la fenêtre.

Pour éviter cela, il faut encadrer l'iframe dans une deuxième div avec le CSS suivant :

```
.video div {
    position: relative;
    height: 0;
    padding-bottom: 56.25%;
}
```

Puis de rajouter les règles css suivantes directement à l'iframe :

```
.video iframe {
    position: absolute;
```

```
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    border: none;
}
```

La position `absolute` de l'iframe permet de la sortir du flux de la page et le fait de rajouter une largeur et une hauteur de 100% permet d'adapter automatiquement la taille de la vidéo en fonction de la taille de la fenêtre, exactement comme le comportement d'une image.

Bloc "Parallax" & boutons (durée ≈ 40min)

Tous les boutons (button, a, input) possèdent des styles interchangeables.

Le parallax sur l'image de fond a été créé en trois étapes :

1. Ajout d'une classe `paralax` sur la balise section
2. Ajout de l'image grâce à la propriété `background`. L'image de fond est assombrie avec `linear-gradient`
3. Il suffit ensuite de définir la règle css `background-attachment` sur `fixed` pour fixer l'image pendant le scroll

Mosaïque (durée ≈ 1h30)

Chaque logo de licence est entouré d'une div portant une classe du nom de la licence (starwars, avenger...)

Pour des raisons de simplicité, les briques grisées derrière les logos sont tirées de la maquette directement (format png).

Pour créer la mosaïque, j'ai utilisé une nouvelle fois Flexbox mais d'une manière différente. Conscient qu'il existe plusieurs façons de faire, j'ai tenté de faire au plus simple / efficace :

- Les divs portant les classes `starwars` et `bigbang` sont entourées d'une div avec une classe `col1`, symbolisant la 1ère colonne
- Les divs portant les classes `avenger` et `batman` sont entourées d'une div avec une classe `col2`, symbolisant la 2ème colonne
- Les divs portant les classes `indiana` et `harry` sont entourées d'une div avec une classe `col3`, symbolisant la 3ème colonne
- La div portant la classe `lord` est entourée d'une div avec une classe `col4`, symbolisant la 4ème colonne

Ensuite ces 4 divs (col1, col2...) sont mises en page grâce à FlexBox.

Le rendu n'étant pas parfaitement conforme à la maquette, j'ai rectifié certains logos manuellement en css avec des règles comme `top` ou `left`.

Footer (durée ≈ 2h)

La disposition des 3 blocs du footer a été réalisée avec FlexBox.

La validation du formulaire a été réalisée en JavaScript via le fichier `form-validator.js`

J'ai d'abord vérifié chaque champ (nom, email et message) en écoutant la frappe du clavier avec `keyup()`. Quand une des conditions n'était pas remplie par l'utilisateur, j'affiche un message d'erreur invitant l'utilisateur a validé le champ.

Ensuite, j'effectue toutes les vérifications demandées grâce à la fonction `check()` que j'appelle au submit du formulaire. Avant l'appel de cette fonction, j'annule l'envoi du formulaire grâce à `preventDefault()` pour rester en phase de test.