

Insertion d'image flexible

Insertion d'images flexibles dans Castopod

Objectif :

Permettre à l'utilisateur d'insérer des images dans Castopod, même si elles ne respectent pas les ratios (1:1) et la taille minimale requise (1400px), en fournissant une interface de recadrage et en ajustant automatiquement les dimensions si nécessaire.

Explication :

A la base, le formulaire de la page envoie les données de l'épisode (dont l'image) à un contrôleur (`EpisodeController`) via la route `episode-create` :

```
<form action="<?= route_to('episode-create', $podcast->id) ?>" method="POST" enctype="multipart/form-data" class="flex flex-col w-full max-w-xl mt-6 gap-y-8" id="episode-form">
```

```
$routes->post(
    'new',
    'EpisodeController::attemptCreate/$1',
    [
        'filter' => 'permission:podcast#.episodes.create',
    ],
);
```

Il fallait donc proposer à l'utilisateur cette interface, changer l'image du formulaire avant de l'envoyer, et gérer ce que renvoie le contrôleur.

Pour gérer l'interface, il faut du JavaScript (JS) car c'est du code dynamique, qui va changer tout le temps, contrairement à PHP qui charge juste au début de la page. Le script Croppie permet de faire cette interface, de manière simple.

```

// Fonction pour afficher l'interface lorsqu'une image est insérée
var $uploadImage = document.getElementById('cover');
var $imagePreview = document.getElementById('imagePreview');
var croppie = new Croppie($imagePreview, {
  viewport: { width: 200, height: 200, type: 'square' },
  boundary: { width: 300, height: 300 },
});

$uploadImage.addEventListener('change', function () {
  $imagePreview.classList.remove('hidden');
  var reader = new FileReader();
  reader.onload = function (e) {
    croppie.bind({
      url: e.target.result,
    });
  };
  reader.readAsDataURL(this.files[0]);
});

// Fonction pour obtenir l'image croppée sous forme de fichier
function getCroppedImageAsFile(croppieInstance) {
  return new Promise(resolve => {
    croppieInstance.result({
      type: 'blob',
      format: 'jpeg',
      size: 'original',
    }).then(blob => {
      const file = new File([blob], 'cropped_image.jpeg', { type: 'image/jpeg' });
      resolve(file);
    });
  });
}

```

Pour gérer la taille de l'image, on utilise la fonction `resizeImage`, qui permet de changer la taille d'une image.

Une fois que l'utilisateur soumet le formulaire, nous bloquons l'envoi direct par la page. Étant donné que la nouvelle image est générée par JavaScript, l'envoi direct des données n'est pas possible. Nous sommes donc contraints de gérer l'envoi de la requête en JavaScript et de traiter la réponse. Comme cette réponse ne peut pas être un objet PHP, nous devons la gérer entièrement côté client.

```

document.getElementById('episode-form').addEventListener('submit', async function (event) {
    event.preventDefault();

    var formData = new FormData(this);

    if(document.getElementById('cover').value !== ""){
        // Récupérez l'image croppée sous forme de fichier
        const croppedFile = await getCroppedImageAsFile(croppie);

        // Redimensionnez l'image à une largeur de 1400px si nécessaire
        const resizedFile = await resizeImage(croppedFile, 1400);

        // Ajoutez le fichier redimensionné au formulaire FormData
        formData.set('cover', resizedFile);
    }

    try {
        const response = await fetch("<?= route_to('episode-create', $podcast->id) ?>", {
            method: "POST",
            body: formData,
        });
    }

```

Auparavant, lorsque les données étaient envoyées directement depuis la page sans JavaScript, l'objet renvoyé était un `RedirectResponse`, géré en arrière-plan par un contrôleur. Cependant, dans ce scénario, cette approche n'est pas applicable. Nous devons plutôt utiliser un type d'objet compatible avec JavaScript, comme un JSON. Ce JSON peut prendre deux formes :

- En l'absence d'erreur : il contient un champ "id" qui représente l'URL de l'épisode créé. Dans ce cas, l'URL retournée est chargée.

```

const idUrl = responseData.id;
console.log(idUrl);
// Open the new episode view page
const baseUrl = '<?= base_url('/') ?>'.slice(0, -1);
window.location.href = baseUrl + idUrl;

```

- En présence d'une erreur : un champ error qui contient les erreurs retournées par le contrôleur gérant l'upload de l'épisode. Dans ce cas, la page est rafraîchie avec un champ expliquant l'erreur au début de la page, à l'aide d'une variable `GET error` contenant l'erreur.

```

<?php
// Vérifier s'il y a une erreur, et dans ce cas l'afficher
if (isset($_GET['error'])) {
    // Décoder la chaîne URL avant de la diviser
    $decodedError = urldecode($_GET['error']);

    // Diviser la chaîne en un tableau basé sur le caractère "+"
    $errorMessages = explode('+', $_GET['error']);

    // Supprimer les éléments vides du tableau
    $errorMessages = array_filter($errorMessages);

    // Afficher un div contenant un paragraphe pour chaque élément du tableau
    echo '<div class="flex flex-col gap-x-2 gap-y-4 md:flex-row">';
    echo '<fieldset class="w-full p-8 bg-red-200 border-3 flex flex-col items-start border-red-600 rounded-xl ">';
    foreach ($errorMessages as $errorMessage) {
        echo '<p> Erreur : ' . $errorMessage . '</p>';
    }
    echo '</fieldset>';
    echo '</div>';
}
?>

```

Un problème rencontré en cas d'erreur est la perte des données saisies par l'utilisateur car la page est rechargée. Pour remédier à cela, nous effectuons des vérifications avant d'envoyer les données du formulaire. Nous vérifions que les champs obligatoires sont correctement remplis par l'utilisateur. Si un champ (ou plusieurs) est manquant, une erreur est affichée et la page est automatiquement ramenée à l'emplacement du champ manquant.

```

const audioFile = document.getElementById('audio_file').value;
const title = document.getElementById('title').value;
const description = document.getElementById('description').value;

// Vérifiez si les champs requis sont vides
const emptyFields = findEmptyFields([
    { id: 'audio_file', label: 'Audio File' },
    { id: 'title', label: 'Title' },
    { id: 'description', label: 'Description' }
]);

```

```
if (emptyFields.length > 0) {  
    const errorMessage = `Veuillez remplir les champs obligatoires : ${emptyFields.map(field =>  
field.label).join(', ')}.`;  
    alert(errorMessage);  
  
    // Faites défiler la page jusqu'au premier champ manquant  
    const firstEmptyField = emptyFields[0].id;  
    const element = document.getElementById(firstEmptyField);  
    const elementPosition = element.getBoundingClientRect().top + window.scrollY - window.innerHeight / 2;  
    window.scrollTo({ top: elementPosition, behavior: 'smooth' });  
}
```

En conclusion, pendant que le script JavaScript gère l'envoi du formulaire et la réponse associée, nous désactivons le bouton d'envoi dès que l'utilisateur appuie dessus. Cette désactivation évite que l'utilisateur n'appuie plusieurs fois sur le bouton, ce qui pourrait entraîner l'envoi répété de la même requête. De plus, un indicateur visuel sous forme d'un rond de chargement apparaît pour informer l'utilisateur que le traitement est en cours.

```
// Baisser l'opacité de la page sauf pour le cercle de chargement  
document.getElementById('all').style.opacity = '0.5';  
  
// Désactiver le bouton  
const submitButton = document.getElementById('submit-button');  
submitButton.disabled = true;  
  
// Afficher le cercle de chargement  
const loadingCircle = document.getElementById('loading-circle');  
loadingCircle.style.display = 'block';
```

Revision #4

Created 29 April 2024 15:18:22 by Ulysse

Updated 2 May 2024 11:06:23 by Ulysse