

Anglais ▼

Express Tutorial Part 6: Utilisation des formulaires

Présentation: Express Nodejs

Dans ce didacticiel, nous allons vous montrer comment travailler avec des formulaires HTML dans Express à l'aide de Pug. En particulier, nous verrons comment écrire des formulaires pour créer, mettre à jour et supprimer des documents de la base de données du site.

**Conditions
préalables:**

Suivez toutes les rubriques du didacticiel précédentes, y compris le didacticiel express, partie 5: affichage des données de bibliothèque

Objectif:

Pour comprendre comment écrire des formulaires pour obtenir des données des utilisateurs et mettre à jour la base de données avec ces données.

Aperçu

Un formulaire HTML est un groupe d'un ou plusieurs champs / widgets sur une page Web qui peut être utilisé pour collecter des informations auprès des utilisateurs pour les soumettre à un serveur. Les formulaires sont un mécanisme flexible pour collecter les entrées des utilisateurs car il existe des entrées de formulaires appropriées disponibles pour saisir de nombreux types de données différents: zones de texte, cases à cocher, boutons radio, sélecteurs de date, etc. , car ils nous permettent d'envoyer des données dans les POST demandes avec une protection contre la falsification des demandes intersites.

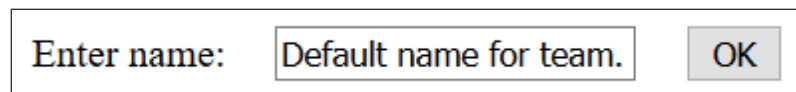
Travailler avec des formulaires peut être compliqué! Les développeurs doivent écrire du HTML pour le formulaire, valider et nettoyer correctement les données entrées sur le serveur (et éventuellement aussi dans le navigateur), republier le formulaire avec des messages d'erreur pour informer les utilisateurs de tout champ invalide, gérer les données lorsqu'elles ont été soumises avec succès, et enfin répondre à l'utilisateur d'une manière quelconque pour indiquer le succès.

Dans ce didacticiel, nous allons vous montrer comment les opérations ci-dessus peuvent être effectuées dans *Express*. En cours de route, nous étendrons le site Web *LocalLibrary* pour permettre aux utilisateurs de créer, modifier et supprimer des éléments de la bibliothèque.

Remarque: Nous n'avons pas examiné comment restreindre des itinéraires particuliers à des utilisateurs authentifiés ou autorisés, donc à ce stade, tout utilisateur pourra apporter des modifications à la base de données.

Formulaires HTML

D'abord un bref aperçu des formulaires HTML. Prenons un simple formulaire HTML, avec un seul champ de texte pour saisir le nom d'une "équipe" et son étiquette associée:



Enter name:

Le formulaire est défini en HTML comme une collection d'éléments à l'intérieur des `<form>...</form>` balises, contenant au moins un `input` élément de `type="submit"`.

```
1 <form action="/team_name_url/" method="post">
2   <label for="team_name">Enter name: </label>
3   <input id="team_name" type="text" name="name_field" value="Default na
4   <input type="submit" value="OK">
5 </form>
```

Bien que nous n'ayons inclus ici qu'un seul champ (texte) pour saisir le nom de l'équipe, un formulaire *peut* contenir n'importe quel nombre d'autres éléments d'entrée et leurs étiquettes associées. L' `type` attribut du champ définit le type de widget qui sera affiché. Le `name` et `id` du champ sont utilisés pour identifier le champ en JavaScript / CSS / HTML, tandis que

`value` définit la valeur initiale du champ lors de son premier affichage. Le libellé de l'équipe correspondante est spécifié à l'aide de la `label` balise (voir "Entrer le nom" ci-dessus), avec un `for` champ contenant la `id` valeur de l'association `input`.

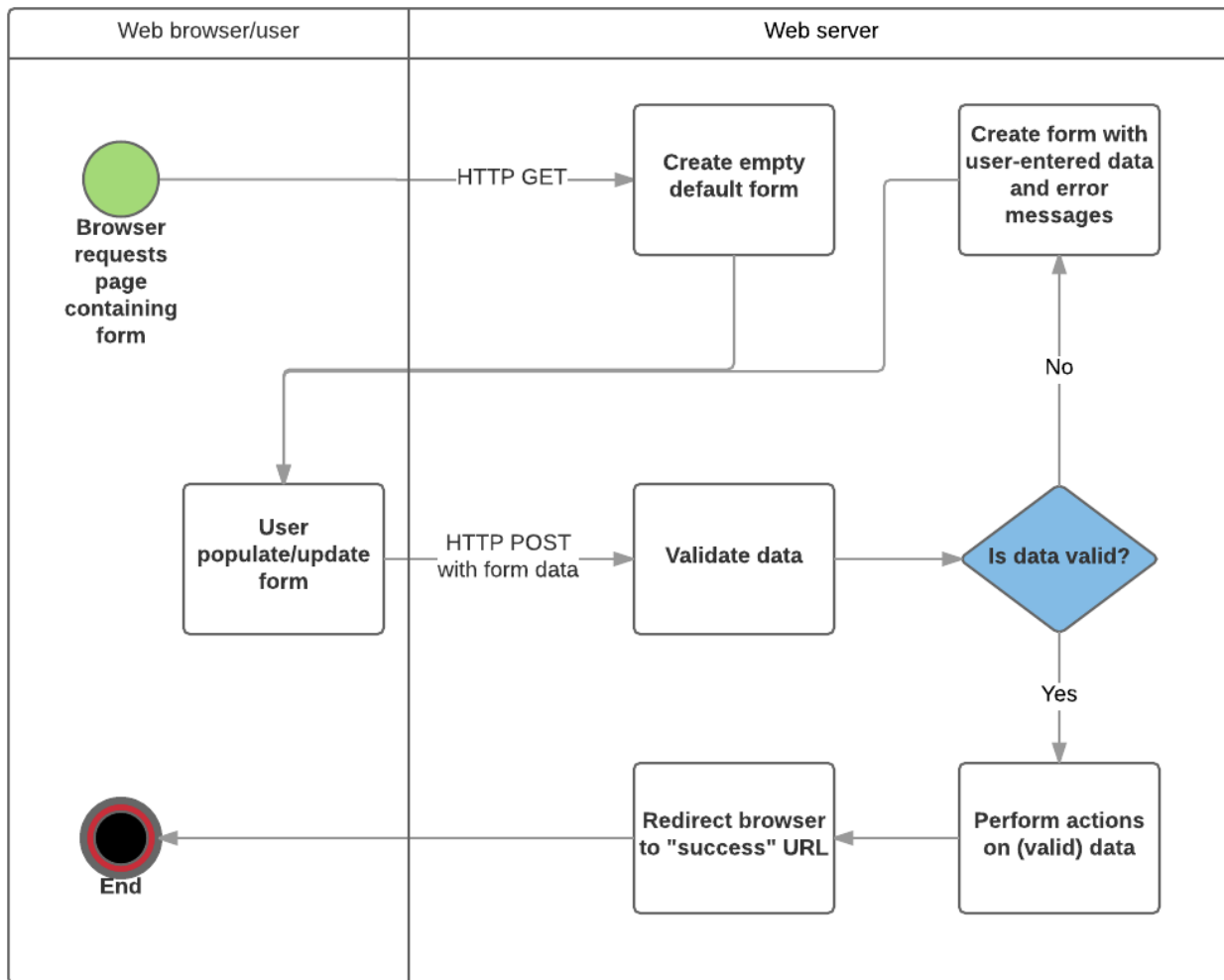
L' `submit` entrée sera affichée sous forme de bouton (par défaut) - l'utilisateur peut appuyer dessus pour télécharger les données contenues par les autres éléments d'entrée sur le serveur (dans ce cas, juste le `team_name`). Les attributs de formulaire définissent le HTTP `method` utilisé pour envoyer les données et la destination des données sur le serveur (`action`):

- `action`: La ressource / URL où les données doivent être envoyées pour traitement lorsque le formulaire est soumis. Si ce n'est pas défini (ou défini sur une chaîne vide), le formulaire sera renvoyé à l'URL de la page actuelle.
- `method`: La méthode HTTP utilisée pour envoyer les données: `POST` ou `GET`.
 - La `POST` méthode doit toujours être utilisée si les données doivent entraîner une modification de la base de données du serveur, car cela peut être rendu plus résistant aux attaques de demandes de contrefaçon intersites.
 - La `GET` méthode ne doit être utilisée que pour les formulaires qui ne modifient pas les données utilisateur (par exemple, un formulaire de recherche). Il est recommandé lorsque vous souhaitez pouvoir ajouter un signet ou partager l'URL.

Processus de traitement des formulaires

La gestion des formulaires utilise toutes les mêmes techniques que nous avons apprises pour afficher des informations sur nos modèles: la route envoie notre demande à une fonction de contrôleur qui effectue toutes les actions de base de données requises, y compris la lecture des données des modèles, puis génère et renvoie une page HTML. Ce qui complique les choses, c'est que le serveur doit également être en mesure de traiter les données fournies par l'utilisateur et d'afficher à nouveau le formulaire avec des informations d'erreur en cas de problème.

Un organigramme du processus de traitement des demandes de formulaire est illustré ci-dessous, en commençant par une demande de page contenant un formulaire (affiché en vert):



Comme le montre le diagramme ci-dessus, les principales tâches que doit exécuter le code de gestion des formulaires sont les suivantes:

1. Affichez le formulaire par défaut la première fois qu'il est demandé par l'utilisateur.
 - Le formulaire peut contenir des champs vides (par exemple, si vous créez un nouvel enregistrement), ou il peut être pré-rempli avec des valeurs initiales (par exemple, si vous modifiez un enregistrement, ou avez des valeurs initiales par défaut utiles).
2. Recevez les données soumises par l'utilisateur, généralement dans une POST requête HTTP .
3. Validez et désinfectez les données.
4. Si des données ne sont pas valides, réaffichez le formulaire, cette fois avec toutes les valeurs renseignées par l'utilisateur et les messages d'erreur pour les champs de problème.

5. Si toutes les données sont valides, effectuez les actions requises (par exemple enregistrer les données dans la base de données, envoyer un e-mail de notification, renvoyer le résultat d'une recherche, télécharger un fichier, etc.)
6. Une fois toutes les actions terminées, redirigez l'utilisateur vers une autre page.

Souvent, le code de gestion de formulaire est implémenté en utilisant une GET route pour l'affichage initial du formulaire et une POST route vers le même chemin pour gérer la validation et le traitement des données de formulaire. C'est l'approche qui sera utilisée dans ce tutoriel.

Express lui-même ne fournit aucun support spécifique pour les opérations de gestion de formulaire, mais il peut utiliser un middleware pour traiter POST et GET paramétrer le formulaire, et pour valider / filtrer leurs valeurs.

Validation et désinfection

Avant de stocker les données d'un formulaire, elles doivent être validées et nettoyées:

- La validation vérifie que les valeurs saisies sont appropriées pour chaque champ (sont dans la bonne plage, le bon format, etc.) et que les valeurs ont été fournies pour tous les champs requis.
- La désinfection supprime / remplace les caractères des données qui pourraient potentiellement être utilisés pour envoyer du contenu malveillant au serveur.

Pour ce didacticiel, nous utiliserons le module de validation express populaire pour effectuer à la fois la validation et la désinfection de nos données de formulaire.

Installation

Installez le module en exécutant la commande suivante à la racine du projet.

```
1 | npm install express-validator
```

Utiliser express-validator

Remarque: Le [guide de validation express](#) sur Github fournit une bonne vue d'ensemble de l'API. Nous vous recommandons de lire cela pour avoir une idée de toutes ses capacités (y

compris l'utilisation de la validation de schéma et la création de validateurs personnalisés).
Ci-dessous, nous *couvrons* juste un sous-ensemble utile pour la *LocalLibrary* .

Pour utiliser le validateur dans nos contrôleurs, nous devons *exiger* les fonctions que nous voulons utiliser des modules **'express-validator / check '** et **'express-validator / filter '**, comme indiqué ci-dessous:

```
1 | const { body, validationResult } = require('express-validator/check');  
2 | const { sanitizeBody } = require('express-validator/filter');
```

Il existe de nombreuses fonctions disponibles, vous permettant de vérifier et de nettoyer les données des paramètres de demande, du corps, des en-têtes, des cookies, etc., ou de tous en même temps. Pour ce tutoriel, nous allons d'abord être à l'aide `body`, `sanitizeBody` et `validationResult` (comme « nécessaire » ci-dessus).

Les fonctions sont définies comme suit:

- `body([fields, message])`: Spécifie un ensemble de champs dans le corps de la demande (un POST paramètre) à valider ainsi qu'un message d'erreur facultatif qui peut s'afficher en cas d'échec des tests. Les critères de validation sont connectés en chaîne à la `body()` méthode. Par exemple, la première vérification ci-dessous vérifie que le champ "nom" n'est pas vide et définit un message d'erreur "Nom vide" s'il ne l'est pas. Le deuxième test vérifie que le champ `age` est une date valide et utilise `optional()` pour spécifier que les chaînes nulles et vides n'échoueront pas à la validation.

```
1 | body('name', 'Empty name').isLength({ min: 1 }),  
2 | body('age', 'Invalid age').optional({ checkFalsy: true }).isISO8601;
```

Vous pouvez également chaîner différents validateurs en chaîne et ajouter des messages qui s'affichent si les validateurs précédents sont vrais.

```
1 | body('name').isLength({ min: 1 }).trim().withMessage('Name empty').  
2 |   .isAlpha().withMessage('Name must be alphabet letters.'),
```

Remarque: Vous pouvez également ajouter des désinfectants en ligne comme `trim()`, comme indiqué ci-dessus. Cependant, les désinfectants appliqués ici

s'appliquent UNIQUEMENT à l'étape de validation. Si vous souhaitez désinfecter la sortie finale, vous devez utiliser une méthode de désinfection distincte, comme indiqué ci-dessous.

- `sanitizeBody(fields)` : Spécifie un champ corporel à nettoyer. Les opérations de désinfection sont ensuite connectées en guirlande à cette méthode. Par exemple, l' `escape()` opération de nettoyage ci-dessous supprime les caractères HTML de la variable de nom qui pourraient être utilisés dans les attaques de script intersite JavaScript.

```
1 | sanitizeBody('name').trim().escape(),
2 | sanitizeBody('date').toDate(),
```

- `validationResult(req)` : Exécute la validation, rendant les erreurs disponibles sous la forme d'un `validation` objet de résultat. Ceci est appelé dans un rappel séparé, comme indiqué ci-dessous:

```
1 | (req, res, next) => {
2 |   // Extract the validation errors from a request.
3 |   const errors = validationResult(req);
4 |
5 |   if (!errors.isEmpty()) {
6 |     // There are errors. Render form again with sanitized val
7 |     // Error messages can be returned in an array using `errc
8 |   }
9 |   else {
10 |     // Data from form is valid.
11 |   }
12 | }
```

Nous utilisons la `isEmpty()` méthode du résultat de la validation pour vérifier s'il y a eu des erreurs et sa `array()` méthode pour obtenir l'ensemble des messages d'erreur. Voir l' API Validation Result pour plus d'informations.

Les chaînes de validation et de nettoyage sont des middlewares qui doivent être transmis au gestionnaire de route Express (nous le faisons indirectement, via le contrôleur). Lorsque le middleware s'exécute, chaque validateur / désinfectant est exécuté dans l'ordre spécifié.

Nous couvrirons quelques exemples réels lorsque nous implémenterons les formulaires *LocalLibrary* ci-dessous.

Conception de formulaire

De nombreux modèles de la bibliothèque sont liés / dépendants. Par exemple, a *Book* *requiert* un *Author* et *peut* également en avoir un ou plusieurs *Genres*. Cela soulève la question de savoir comment traiter le cas où un utilisateur souhaite:

- Créez un objet lorsque ses objets associés n'existent pas encore (par exemple, un livre où l'objet auteur n'a pas été défini).
- Supprimez un objet qui est toujours utilisé par un autre objet (par exemple, supprimez un Genre qui est toujours utilisé par un Book).

Pour ce projet, nous simplifierons la mise en œuvre en déclarant qu'un formulaire ne peut que:

- Créez un objet en utilisant des objets qui existent déjà (les utilisateurs devront donc créer toutes les instances *Author* et *Genre* instances requises avant de tenter de créer des *Book* objets).
- Supprimez un objet s'il n'est pas référencé par d'autres objets (par exemple, vous ne pourrez pas supprimer un *Book* tant que tous les *BookInstance* objets associés n'auront pas été supprimés).

Remarque: Une implémentation plus "robuste" peut vous permettre de créer les objets dépendants lors de la création d'un nouvel objet et de supprimer n'importe quel objet à tout moment (par exemple, en supprimant des objets dépendants ou en supprimant des références à l'objet supprimé de la base de données).

Itinéraires

Afin d'implémenter notre code de gestion de formulaire, nous aurons besoin de deux routes qui ont le même modèle d'URL. La première GET route () est utilisée pour afficher un nouveau formulaire vide pour créer l'objet. Le deuxième itinéraire (POST) est utilisé pour valider les données saisies par l'utilisateur, puis enregistrer les informations et les rediriger vers la page de détail (si les données sont valides) ou pour réafficher le formulaire avec des erreurs (si les données ne sont pas valides).

Nous avons déjà créé les routes pour toutes les pages de création de notre modèle dans **/routes/catalog.js** (dans un tutoriel précédent). Par exemple, les itinéraires de genre sont présentés ci-dessous:

```
1 // GET request for creating a Genre. NOTE This must come before route tha
2 router.get('/genre/create', genre_controller.genre_create_get);
3
4 // POST request for creating Genre.
5 router.post('/genre/create', genre_controller.genre_create_post);
```

Formulaires express sous-articles

Les sous-articles suivants nous guideront tout au long du processus d'ajout des formulaires requis à notre exemple d'application. Vous devez les lire et les parcourir tour à tour, avant de passer au suivant.

1. Formulaire Créer un genre - Définition d'une page pour créer des Genre objets.
2. Formulaire Créer un auteur - Définition d'une page pour créer des Author objets.
3. Créer un formulaire livre - Définition d'une page / formulaire pour créer des Book objets.
4. Créer un formulaire BookInstance - Définition d'une page / d'un formulaire pour créer des BookInstance objets.
5. Formulaire de suppression d'auteur - Définition d'une page pour supprimer des Author objets.
6. Formulaire de mise à jour du livre - Définition d'une page pour mettre à jour des Book objets.

Relevez le défi

Mettre en oeuvre les pages de suppression pour les Book , BookInstance et les Genre modèles, les reliant des pages de détail associées de la même manière que notre

Auteur delete page. Les pages doivent suivre la même approche de conception:

- S'il existe des références à l'objet à partir d'autres objets, ces autres objets doivent être affichés avec une note que cet enregistrement ne peut pas être supprimé tant que les objets répertoriés n'ont pas été supprimés.
- S'il n'y a pas d'autres références à l'objet, la vue doit vous inviter à le supprimer. Si l'utilisateur appuie sur le bouton **Supprimer**, l'enregistrement doit alors être supprimé.

Quelques conseils:

- Supprimer un Genre revient à supprimer un Author car les deux objets sont des dépendances de Book (dans les deux cas, vous ne pouvez supprimer l'objet que lorsque les livres associés sont supprimés).
- La suppression d'un Book est également similaire, mais vous devez vérifier qu'aucun n'est associé BookInstances.
- Supprimer un BookInstance est le plus simple de tous car il n'y a pas d'objets dépendants. Dans ce cas, vous pouvez simplement trouver l'enregistrement associé et le supprimer.

Mettre en oeuvre les pages de mise à jour pour les BookInstance, Author et les Genre modèles, les reliant des pages de détail connexes de la même manière que notre *mise à jour du livre page*.

Quelques conseils:

- La *page de mise à jour du livre* que nous venons de mettre en place est la plus difficile! Les mêmes modèles peuvent être utilisés pour les pages de mise à jour des autres objets.
- Les Author champs de date de décès et de date de naissance et le BookInstance champ `due_date` ne sont pas le bon format à saisir dans le champ de saisie de date du formulaire (il nécessite des données sous la forme "AAAA-MM-JJ"). Le moyen le plus simple de contourner ce problème consiste à définir une nouvelle propriété virtuelle pour les dates qui formate les dates de manière appropriée, puis à utiliser ce champ dans les modèles de vue associés.
- Si vous êtes bloqué, il y a des exemples de pages de mise à jour dans l'exemple ici .

Résumé

Les packages *Express* , *Node* et tiers sur NPM fournissent tout ce dont vous avez besoin pour ajouter des formulaires à votre site Web. Dans cet article, vous avez appris à créer des formulaires à l'aide de *Pug* , à valider et à nettoyer les entrées à l'aide de *validateur express* , et à ajouter, supprimer et modifier des enregistrements dans la base de données.

Vous devez maintenant comprendre comment ajouter des formulaires de base et du code de gestion de formulaires à vos propres sites Web de nœuds!

Voir également

- [validateur express \(docs npm\)](#).



Présentation: Express Nodejs



Dans ce module

- [Introduction à Express / Node](#)
- [Configuration d'un environnement de développement Node \(Express\)](#)
- [Tutoriel Express: le site Web de la bibliothèque locale](#)
- [Express Tutorial Part 2: Création d'un site Web squelette](#)
- [Express Tutorial Part 3: Utilisation d'une base de données \(avec Mongoose\)](#)
- [Express Tutorial Part 4: Routes et contrôleurs](#)
- [Express Tutorial Part 5: Affichage des données de bibliothèque](#)
- [Express Tutorial Part 6: Utilisation des formulaires](#)

- Express Tutorial Part 7: Déploiement en production
-

Dernière modification: 5 juin 2020, par les contributeurs MDN

Rubriques connexes

Les débutants complets commencent ici!

- Débuter avec le Web

HTML - Structurer le Web

- Introduction au HTML
- Multimédia et intégration
- Tableaux HTML

CSS - Styliser le Web

- Premières étapes CSS
- Blocs de construction CSS
- Texte de style
- Disposition CSS

JavaScript - Script dynamique côté client

- Premiers pas avec JavaScript
- Blocs de construction JavaScript
- Présentation des objets JavaScript
- JavaScript asynchrone
- API Web côté client

Formulaires Web - Utilisation des données utilisateur

- Parcours d'apprentissage des formulaires de base
- Articles de formulaires avancés

Accessibilité - Rendre le Web utilisable par tous

- Guides d'accessibilité
- Évaluation de l'accessibilité

Outils et tests

- Tests inter-navigateurs
- Git et GitHub
- Introduction aux frameworks côté client
- Réagir
- Ember
- Vue

Programmation de site Web côté serveur

- Premiers pas
- Framework web Django (Python)
- ▼ Express Web Framework (node.js / JavaScript)

Présentation d'Express Web Framework (Node.js / JavaScript)

Introduction à Express / Node

Configuration d'un environnement de développement Node (Express)

Tutoriel express: le site Web de la bibliothèque locale

Express Tutorial Part 2: Création d'un site Web squelette

Express Tutorial Part 3: Utilisation d'une base de données (avec Mongoose)

Express Tutorial Part 4: Routes et contrôleurs

Express Tutorial Part 5: Affichage des données de bibliothèque

Express Tutorial Part 6: Utilisation des formulaires

Express Tutorial Part 7: Déploiement en production

Autres ressources

▸ Questions courantes

Comment contribuer



Apprenez le meilleur du développement Web

Recevez les dernières nouveautés de MDN directement dans votre boîte de réception.

S'inscrire maintenant