

NUXT Docs - Français



Table of contents

• Get started - Installation	5
• Get started - Routing	9
• Get started - Directory structure	11
• Get started - Commands	13
• Get started - Conclusion	16
• Get started - Upgrading	17
• Concepts - Views	18
• Concepts - Context helpers	22
• Concepts - Server side rendering	28
• Concepts - Static site generation	30
• Concepts - Nuxt lifecycle	32
• Features - Rendering modes	37
• Features - Live preview	38
• Features - Transitions	40
• Features - Deployment targets	44
• Features - File system routing	46
• Features - Data fetching	55
• Features - Meta tags seo	60
• Features - Configuration	64
• Features - Loading	70
• Features - Nuxt components	75
• Features - Component discovery	81
• Directory structure - Nuxt	84
• Directory structure - Plugins	86
• Directory structure - Static	93
• Directory structure - Store	95
• Directory structure - Nuxt config	100
• Directory structure - Assets	107
• Directory structure - Components	112
• Directory structure - Content	115
• Directory structure - Dist	123
• Directory structure - Layouts	125
• Directory structure - Middleware	127
• Directory structure - Modules	130

• Directory structure - Pages	139
• Configuration glossary - Configuration alias	145
• Configuration glossary - Configuration extend plugins	147
• Configuration glossary - Configuration generate	148
• Configuration glossary - Configuration global name	155
• Configuration glossary - Configuration head	156
• Configuration glossary - Configuration hooks	157
• Configuration glossary - Configuration ignore	160
• Configuration glossary - Configuration loading	162
• Configuration glossary - Configuration loading indicator	164
• Configuration glossary - Configuration mode	166
• Configuration glossary - Configuration modern	167
• Configuration glossary - Configuration build	169
• Configuration glossary - Configuration modules	187
• Configuration glossary - Configuration modulesdir	189
• Configuration glossary - Configuration plugins	190
• Configuration glossary - Configuration render	192
• Configuration glossary - Configuration rootdir	199
• Configuration glossary - Configuration router	200
• Configuration glossary - Configuration runtime config	208
• Configuration glossary - Configuration server	209
• Configuration glossary - Configuration servermiddleware	211
• Configuration glossary - Configuration srkdir	214
• Configuration glossary - Configuration builddir	216
• Configuration glossary - Configuration ssr	217
• Configuration glossary - Configuration target	218
• Configuration glossary - Configuration telemetry	219
• Configuration glossary - Configuration transition	221
• Configuration glossary - Configuration vue config	223
• Configuration glossary - Configuration watch	224
• Configuration glossary - Configuration watchers	225
• Configuration glossary - Configuration cli	226
• Configuration glossary - Configuration css	228
• Configuration glossary - Configuration components	229
• Configuration glossary - Configuration dev	230
• Configuration glossary - Configuration dir	231

• Configuration glossary - Configuration env	232
• Internals glossary - Context	234
• Internals glossary - Nuxt render	238
• Internals glossary - Nuxt render route	239
• Internals glossary - Nuxt render and get window	240
• Internals glossary - \$nuxt	241
• Internals glossary - Internals	243
• Internals glossary - Internals nuxt	245
• Internals glossary - Internals renderer	246
• Internals glossary - Internals module container	247
• Internals glossary - Internals builder	250
• Internals glossary - Internals generator	251
• Internals glossary - Nuxt	252
• Components glossary - Fetch	253
• Components glossary - Watchquery	255
• Components glossary - Head	256
• Components glossary - Key	257
• Components glossary - Layout	258
• Components glossary - Loading	259
• Components glossary - Middleware	260
• Components glossary - Scrolltotop	261
• Components glossary - Transition	262
• Components glossary - Validate	265

Installation

Découvrez ci-dessous comment créer un projet Nuxt fonctionnel en 4 étapes.

Essayer Nuxt

Vous pouvez essayer Nuxt en ligne sur CodeSandbox ou StackBlitz:

:app-button[Play on CodeSandbox] :app-button[Play on StackBlitz]

Prérequis

- [node](#) - au moins v10.13 *Nous vous recommandons d'installer la dernière version LTS.*
- Un éditeur de texte, nous recommandons [VSCode](#) avec l'extension [Vetur](#) ou [WebStorm](#)
- Un terminal, nous vous recommandons d'utiliser [le terminal intégré de VSCode](#) ou le [terminal de Webstorm](#).

Utiliser create-nuxt-app

:video-player{src="<https://www.youtube.com/watch?v=wHkPjOmJTt0>"}

Pour créer rapidement un nouveau projet, nous pouvons utiliser [create-nuxt-app](#).

Vérifiez que `npx` est installé (`npx` est livré par défaut depuis `npm v5.2.0` ou `npm v6.1` ou `yarn`).

```
yarn create nuxt-app <project-name>
```

```
npx create-nuxt-app <project-name>
```

```
npm init nuxt-app <project-name>
```

Quelques questions nous seront posées (nom, options Nuxt, framework UI, TypeScript, linter, framework de test, etc.). Pour en savoir plus sur toutes les options voir [Create Nuxt App](#).

Une fois les questions répondues, toutes les dépendances seront installées. La prochaine étape est de naviguer dans le répertoire et de démarrer notre application:

```
cd <project-name>
yarn dev
```

```
cd <project-name>
npm run dev
```

Notre application est maintenant en cours d'exécution sur <http://localhost:3000>.

Info

Une autre façon de démarrer avec Nuxt est d'utiliser [CodeSandbox](#) ce qui est un excellent moyen de jouer rapidement avec Nuxt et/ou de partager votre code avec d'autres personnes.

Commencer de zéro

video-player

src: https://www.youtube.com/watch?v=mKV_9AIG70E

Créer un projet Nuxt de zéro demande seulement un répertoire et un fichier.

Dans cet exemple, nous allons utiliser le terminal pour créer les répertoires et fichiers mais nous pouvons utiliser votre éditeur pour les créer.

Première étape : configurer votre projet

Pour commencer, il faut créer un répertoire vide avec le nom du projet et naviguer à l'intérieur de celui-ci :

```
mkdir <nom-du-projet>
cd <nom-du-projet>
```

Remplacer `<nom-du-projet>` avec le nom du projet.

Puis créer un fichier nommé `package.json` :

```
touch package.json
```

Ouvrir le fichier `package.json` avec notre éditeur préféré et ajouter ce contenu JSON :

```
{
  "name": "my-app",
  "scripts": {
    "dev": "nuxt",
    "build": "nuxt build",
    "generate": "nuxt generate",
    "start": "nuxt start"
  }
}
```

`scripts` définit les commandes Nuxt que nous allons lancer avec la commande `npm run <commande>`.

À quoi sert le fichier `package.json` ?

Le fichier `package.json` est comme une carte d'identité pour notre projet. Si on ne sait pas ce qu'est le fichier `package.json`, nous vous recommandons grandement de lire la [documentation de npm](#).

Deuxième étape : installer nuxt

Une fois le fichier `package.json` créé, nous devons ajouter `nuxt` à votre projet avec la commande `npm` ou `yarn` comme ci-dessous :

```
yarn add nuxt
```

```
npm install nuxt
```

Cette commande va ajouter `nuxt` comme une dépendance du projet et va automatiquement l'ajouter à notre `package.json`. Le répertoire `node_modules` va aussi être créé, c'est l'endroit où sont installés tous les packages et leurs dépendances.

Info

Un fichier `yarn.lock` ou `package-lock.json` est aussi créé, ce qui assure une consistance et une compatibilité dans les dépendances installées par votre projet.

Troisième étape : créer notre première page

Nuxt transforme tous les fichiers `*.vue` dans le répertoire `pages` comme une route pour l'application.

Créer le répertoire `pages` dans notre projet :

```
mkdir pages
```

Puis, créer un fichier `index.vue` dans le répertoire `pages` :

```
touch pages/index.vue
```

Il est important que cette page soit appelée `index.vue` car elle sera la page par défaut quand notre application sera ouverte. C'est la page d'accueil et elle doit être appelée index.

Il faut ouvrir le fichier `index.vue` dans notre éditeur et ajouter le contenu suivant :

```
<template>
  <h1>Hello world!</h1>
</template>
```

Étape finale : lancer le projet

On peut lancer notre projet en tapant la commande ci-dessous dans le terminal :

```
yarn dev
```

```
npm run dev
```

Info

La commande `dev` est utilisée lorsque l'application est lancée en mode développement.

L'application est maintenant en cours d'exécution sur <http://localhost:3000>.

On peut l'ouvrir dans notre navigateur en cliquant sur le lien dans le terminal et nous devrions voir le texte "Hello World" que nous avons ajouté dans l'étape précédente.

Info

Une fois Nuxt lancé en mode développement, il écoutera les changements de fichiers dans la plupart des répertoires, il n'est pas nécessaire de redémarrer l'application quand on édite ou ajoute une page.

Warning

Lorsque on lance la commande `dev`, un répertoire `.nuxt` est créé. Ce répertoire devrait être ignoré par Git (ou tout autre système de gestion de versions). On peut faire ceci en créant un fichier `.gitignore` à la racine de votre projet et en y ajoutant `.nuxt`.

Étape bonus

Créer une page nommée `fun.vue` dans le répertoire `pages`.

Ajouter un `<template></template>` et insérer un titre avec une phrase drôle à l'intérieur.

On peut ensuite aller sur le navigateur et regarder la nouvelle page sur <http://localhost:3000/fun>.

Info

Si nous créons un répertoire nommé `more-fun` et ajoutons un fichier `index.vue` à l'intérieur, cela donnera le même résultat que de créer un fichier `more-fun.vue`.

Routage

La plupart des sites Web ont plus d'une page. Par exemple, une page d'accueil, une page à propos, une page de contact, etc. Pour afficher ces pages, nous avons besoin d'un routeur.

Routes automatiques

La plupart des sites Web possèdent plus d'une page (comme une page d'accueil, une page à propos, une page de contact, etc.). Pour afficher ces pages, nous avons besoin d'un routeur. C'est là qu'intervient `vue-router`.

Lorsque nous travaillons avec Vue, nous devons configurer un fichier de configuration (comme `router.js`) et y ajouter manuellement toutes nos routes. Nuxt génère automatiquement la configuration `vue-router` pour nous, en fonction des fichiers Vue fournis dans le répertoire `pages`.

Cela signifie que nous n'avons plus jamais besoin d'écrire une configuration de routeur ! Nuxt nous offre également le découpage dynamique de code pour toutes nos routes.

En d'autres termes, tout ce que nous avons à faire pour avoir un routage dans notre application est de créer des fichiers `.vue` dans le répertoire `pages`.

Next

En savoir plus sur le [Routage](#)

Navigation

Pour naviguer entre les pages de notre application, nous devons utiliser le composant `NuxtLink`. Ce composant est inclus avec Nuxt et nous n'avons donc pas à l'importer comme on le fait avec d'autres composants.

Il est similaire à la balise HTML `<a>`, mais à la place d'utiliser un attribut `href="/about"`, nous utilisons `to="/about"`. Si nous avons déjà utilisé `vue-router`, nous pouvons comparer `<NuxtLink>` au composant `<RouterLink>`.

Voici un exemple de l'utilisation du composant `NuxtLink` avec un lien vers la page `index.vue` dans le répertoire `pages` :

```
<template>
  <NuxtLink to="/">Home page</NuxtLink>
</template>
```

Pour tous liens vers des pages de notre site, il faut utiliser `<NuxtLink>`. Si nous avons des liens vers d'autres sites Web, nous devons utiliser la balise `<a>` comme ci-dessous :

```
<template>
  <main>
    <h1>Home page</h1>
```

```
<NuxtLink to="/about">
  About (lien interne vers une page appartenant à notre application Nuxt)
</NuxtLink>
<a href="https://nuxtjs.org">Lien vers une page externe</a>
</main>
</template>
```

Next

En savoir plus sur le [composant NuxtLink](#).

[Go to TOC](#)

Structure des répertoires

La structure par défaut d'une application Nuxt est destinée à fournir un excellent point de départ pour les applications plus ou moins grandes. Nous sommes libres d'organiser notre application comme on le souhaite et nous pouvons créer d'autres répertoires au fur et à mesure que nous en avons besoin.

Créons les répertoires et les fichiers qui n'existent pas encore dans notre projet :

```
mkdir components assets static
touch nuxt.config.js
```

Ce sont les principaux répertoires et fichiers que nous utilisons lors de la construction d'une application Nuxt. Nous pouvons trouver une explication de chacun d'entre eux ci-dessous.

Info

La création de répertoires avec ces noms active les fonctionnalités de votre projet Nuxt.

Répertoires

Le répertoire `pages`

Le répertoire `pages` contient les vues et les routes de notre application. Comme nous l'avons appris dans l'[étape précédente](#), Nuxt lit tous les fichiers `.vue` dans ce répertoire et les utilise pour créer le routeur de l'application.

Next

En savoir plus sur le [répertoire pages](#)

Le répertoire `components`

Le répertoire `components` est l'endroit où nous plaçons tous nos composants Vue.js qui sont ensuite importés dans les pages.

Avec Nuxt, nous pouvons créer nos composants et les importer automatiquement dans les fichiers `.vue`, ce qui signifie qu'il n'est pas nécessaire de les importer manuellement dans la section `script`. Nuxt les analysera et les importera automatiquement pour nous une fois que nous aurons activé cette fonctionnalité dans le fichier de configuration `nuxt.config.js`.

Next

En savoir plus sur le [répertoire des composants](#)

Le répertoire assets

Le répertoire `assets` contient nos fichiers non compilés tels que les styles, images ou polices de caractères.

Next

En savoir plus sur le [répertoire assets](#)

Le répertoire static

Le répertoire `static` est directement servi à la racine du serveur et contient des fichiers qui doivent conserver leur nom (comme `robots.txt`) *ou* qui ne changeront probablement pas (comme le favicon).

Next

En savoir plus sur le [répertoire static](#)

Le fichier nuxt.config.js

Le fichier `nuxt.config.js` est le point unique de configuration pour Nuxt. Si nous souhaitons ajouter des modules ou remplacer les paramètres par défaut, c'est ce fichier qu'il faudra modifier.

Next

En savoir plus sur le [fichier nuxt.config.js](#)

Le fichier package.json

Le fichier `package.json` contient toutes les dépendances et scripts de notre application.

En savoir plus sur la structure du projet

Il existe d'autres répertoires et fichiers utiles, comme `content`, `layouts`, `middleware`, `modules`, `plugins` et `store`. Comme ils ne sont pas nécessaires pour les petites applications, ils ne sont pas traités ici.

Next

Pour en savoir plus sur tous les répertoires en détail, il ne faut pas hésiter à lire le [guide sur la structure des répertoires](#).

[Go to TOC](#)

Nuxt est livré avec un ensemble de commandes utiles, à la fois pour le développement et la production.

Utilisation dans package.json

Vous devriez mettre ces commandes dans `package.json`:

```
"scripts": {
  "dev": "nuxt",
  "build": "nuxt build",
  "start": "nuxt start",
  "generate": "nuxt generate"
}
```

Ensuite, vous pouvez lancer vos commandes via `yarn <command>` ou `npm run <command>` (exemple: `yarn dev` / `npm run dev`).

Environnement de développement

Pour lancer Nuxt en mode développement avec le module de remplacement à chaud sur `http://localhost:3000`:

`yarn dev`

`npm run dev`

Liste des commandes

Vous pouvez exécuter différentes commandes en fonction de la [cible](#):

target: `server` (default value)

- **nuxt dev** - Lancez le serveur de développement.
- **nuxt build** - Créez et optimisez votre application avec Webpack pour la production.
- **nuxt start** - Démarrez le serveur de production (après avoir lancé `nuxt build`). Utiliser pour un hébergement Node.js comme Heroku, Digital Ocean, etc.

target: `static`

- **nuxt dev** - Lancez le serveur de développement.
- **nuxt generate** - Construisez l'application (si nécessaire), générez chaque route sous forme de fichier HTML et exportez statiquement vers le répertoire `dist/` (utilisé pour l'hébergement statique).
- **nuxt start** - servir le répertoire `dist/` comme le ferait votre hébergement statique (Netlify, Vercel, Surge, etc.), idéal pour les tests avant le déploiement.

Webpack Config Inspection

Vous pouvez inspecter la configuration webpack utilisée par nuxt pour créer un projet similaire à [vue inspect](#).

- **nuxt webpack [query...]**

Arguments:

- `--name` : nom du bundle à inspecter (client, serveur, moderne)
- `--dev` : inspecter la configuration webpack pour le mode dev
- `--depth` : inspecter en profondeur. La valeur par défaut est 2 pour empêcher la sortie verbeuse.
- `--no-colors` : désactiver les couleurs ANSI (désactivées par défaut lorsque TTY n'est pas disponible ou quand piping vers un fichier)

Exemples:

- `nuxt webpack`
- `nuxt webpack devtool`
- `nuxt webpack resolve alias`
- `nuxt webpack module rules`
- `nuxt webpack module rules test=.jsx`
- `nuxt webpack module rules test=.pug oneOf use.0=raw`
- `nuxt webpack plugins constructor.name=WebpackBar options reporter`
- `nuxt webpack module rules loader=vue-`
- `nuxt webpack module rules "loader=.*-loader"`

Déploiement de production

Nuxt vous permet de choisir entre les déploiements de serveur ou statique.

Déploiement de serveur

Pour déployer une application SSR, nous utilisons `target: server`, où serveur est la valeur par défaut.

```
yarn build
```

```
npm run build
```

Nuxt créera un répertoire `.nuxt` avec tout ce qu'il contient prêt à être déployé sur votre serveur d'hébergement.

Info

nous recommandons de mettre `.nuxt` dans `.npmignore` ou `.gitignore`.

Une fois votre application créée, vous pouvez utiliser la commande `start` pour voir une version de production de votre application.

```
yarn start
```

```
npm run start
```

Déploiement statique (Pré-rendu)

Nuxt vous donne la possibilité d'héberger votre application Web sur n'importe quel hébergement statique.

Pour déployer un site statique généré, assurez-vous d'avoir `target: static` dans votre `nuxt.config.js`.
(Pour Nuxt >= 2.13:)

```
export default {
  target: 'static'
}
```

```
yarn generate
```

```
npm run generate
```

Nuxt créera un répertoire `dist /` avec tout ce qu'il contient prêt à être déployé sur un service d'hébergement statique.

Depuis Nuxt v2.13, un robot d'exploration est installé qui va maintenant explorer vos liens et générer vos routes lorsque vous utilisez la commande `nuxt generate` basée sur ces derniers.

Info

Attention: les routes dynamiques sont ignorées par la commande `generate` lors de l'utilisation Nuxt <= v2.12: [API de configuration de la génération](#)

Info

Lors de la génération de votre application Web avec `nuxt generate`, le contexte donnée à `asyncData` et `fetch` n'aura pas `req` et `res`.

Échec en cas d'erreur

Pour renvoyer un code d'état différent de zéro lorsqu'une erreur de page est rencontrée et laisser le CI/CD échouer lors du déploiement ou de la construction, vous pouvez utiliser l'argument `--fail-on-error`.

```
yarn generate --fail-on-error
```

```
npm run generate --fail-on-error
```

Et après?

Next

Lisez notre [FAQ](#) pour trouver des exemples de déploiements sur des hôtes connus.

Conclusion

Félicitations, nous avons maintenant créé notre première application Nuxt et nous pouvons maintenant nous considérer comme un Nuxter. Mais il y a tellement plus à apprendre et tellement plus que nous pouvons faire avec Nuxt. Voici quelques recommandations.

Next

Consulter le [guide des concepts](#)

Next

Travailler avec [asyncData](#)

Next

Choisir entre les différents [modes de rendu](#)

`alert{type="star"}`

Vous aimez Nuxt jusqu'à présent ? N'oubliez pas de [mettre en favoris notre projet sur GitHub](#)

[Go to TOC](#)

Mise à jour

La mise à jour de Nuxt est rapide, mais plus complexe que celle de notre package.json

Si nous effectuons une mise à niveau vers Nuxt v2.14 et que nous souhaitons utiliser un hébergement statique, nous devrons ajouter `target:static` dans notre fichier `nuxt.config.js` pour que la commande `generate` fonctionne correctement.

```
export default {
  target: 'static'
}
```

Commencer

1. Vérifier les [notes de publication](#) pour la version que nous souhaitons mettre à niveau pour savoir s'il existe des instructions supplémentaires pour cette version en particulier.
2. Mettre à jour la version spécifiée pour le package `nuxt` dans notre fichier `package.json`.

Après cette étape, les instructions varient en fonction de notre gestionnaire de paquets (Yarn ou npm). *Yarn est l'outil de développement préféré pour travailler avec Nuxt car c'est celui qui a été utilisé lors de l'écriture des tests.*

Yarn

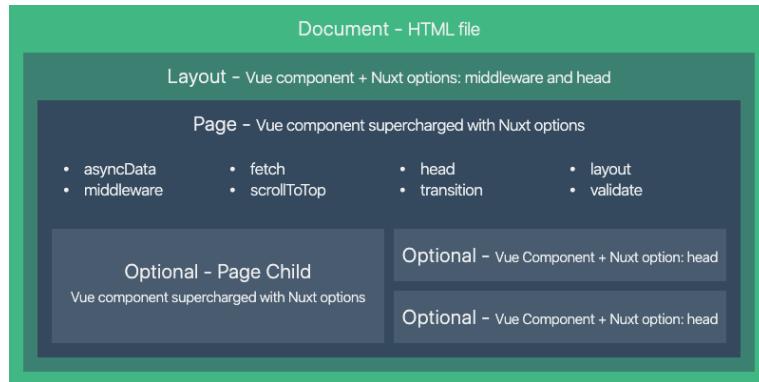
3. Supprimer le fichier `yarn.lock`
4. Supprimer le répertoire `node_modules`
5. Lancer la commande `yarn`
6. Une fois l'installation terminée et les tests exécutés, nous devrions également envisager de mettre à jour d'autres dépendances. La commande `yarn outdated` peut être utilisée.

npm

3. Supprimer le fichier `package-lock.json`
4. Supprimer le répertoire `node_modules`
5. Lancer la commande `npm install`
6. Une fois l'installation terminée et les tests exécutés, nous devrions également envisager de mettre à jour d'autres dépendances. La commande `npm outdated` peut être utilisée.

Vues

La section des vues décrit tout ce que nous avons besoin de savoir pour configurer la data et les vues pour une route spécifique dans notre application Nuxt. Les vues se composent d'un template, d'un layout et de la page actuelle. En plus de cela, nous pouvons définir des méta tags personnalisés pour la section `head` de chaque page. Ces derniers sont importants pour le SEO (référencement naturel).



Composition of a View in Nuxt

Composition d'une vue dans Nuxt

Pages

Chaque composant Page est un composant Vue mais Nuxt ajoute un attribut spécial et des fonctions pour rendre le développement de notre application aussi facile que possible.

```

<template>
  <h1 class="red">Hello World</h1>
</template>

<script>
  export default {
    head() {
      // Configurer les méta tags pour cette page
    }
    // ...
  }
</script>

<style>
  .red {
    color: red;
  }
</style>
  
```

Propriétés du composant page

Il y a beaucoup de propriétés possibles sur un composant Page comme le `head` dans l'exemple juste au dessus.

Next

Se référer à [la documentation sur la structure des répertoires](#) pour en apprendre davantage sur les propriétés que nous pouvons utiliser sur notre page.

Layouts

Les layouts sont un moyen efficace quand nous voulons changer l'apparence et le comportement de notre app Nuxt. Par exemple, inclure une sidebar ou avoir des layouts bien distincts entre une vue mobile et bureau.

Layout par défaut

Nous pouvons définir un layout par défaut en ajoutant un fichier `default.vue` dans le répertoire des `layouts`. Ceci sera utilisé pour toutes les pages qui n'ont pas de layout déjà spécifié. La seule chose qu'il sera nécessaire d'inclure dans le layout est le composant `<Nuxt />` qui s'occupera de render le composant page.

```
<template>
  <Nuxt />
</template>
```

Next

Se référer au [composant Nuxt](#) dans le chapitre des composants.

Layout personnalisé

Nous pouvons créer des layouts personnalisés en ajoutant un fichier `.vue` dans le répertoire des `layouts`. Pour utiliser ce layout personnalisé, nous aurons besoin de spécifier la propriété `layout` dans le composant page dans lequel nous voulons utiliser ce layout. La valeur sera le nom du layout personnalisé que nous avons créé.

Pour créer un layout de blog, ajoutez un fichier `blog.vue` au répertoire `layouts` :

```
<template>
  <div>
    <div>La barre de navigation de mon blog</div>
    <Nuxt />
  </div>
</template>
```

Info

Il faut faire bien attention à ajouter le composant `<Nuxt />` lorsque nous créons le composant pour y inclure le contenu de la page.

Par la suite, il suffit d'attribuer la valeur `blog` à la propriété `layout` dans la page où nous souhaitons que le layout soit utilisé.

```
<template>
  <!-- notre template -->
</template>
<script>
  export default {
    layout: 'blog'
    // ici se trouvent les définitions du composant page
  }
</script>
```

Info

Si on ne spécifie pas la propriété `layout` sur notre page (ex: `layout: 'blog'`) alors le layout `default.vue` sera utilisé.

Page d'erreur

La page d'erreur est le *composant page* qui sera toujours affiché lors d'une erreur (ce n'est pas le cas lors d'une utilisation en mode server-side rendering).

Info

Même si ce fichier sera placé dans le répertoire `layouts`, il devra être considéré comme une page.

Comme dit ci-dessus, ce layout est spécial car il ne devra pas contenir de composant `<Nuxt />` dans son `template`. Ce layout devra être vu comme un composant affiché lorsque des erreurs surviennent (`404`, `500`, etc.). Comme pour les autres composants page, Nous pouvons appliquer un layout personnalisé pour la page erreur.

Nous pouvons personnaliser la page erreur en ajoutant un fichier `layouts/error.vue`:

```
<template>
  <div>
    <h1 v-if="error.statusCode === 404">Page non trouvée</h1>
    <h1 v-else>Une erreur est survenue</h1>
    <NuxtLink to="/">Page d'accueil</NuxtLink>
  </div>
</template>

<script>
  export default {
    props: ['error'],
    layout: 'error' // nous pouvons définir un layout personnalisé pour la page
    erreur
  }
</script>
```

Document: App.html

Le template `app` est utilisé pour créer le cadre HTML pour le document de notre application Nuxt, ce dernier s'occupera d'injecter le contenu ainsi que les variables pour le `head` ainsi que le `body`. Ce fichier est automatiquement créé pour nous et a rarement besoin d'être modifié. Nous pouvons customiser le template HTML `app` utilisé par Nuxt pour inclure des scripts ou des classes CSS conditionnelles en créant un fichier `app.html` dans le répertoire source de notre projet, qui par défaut, n'est autre que la racine de votre projet.

Le template par défaut utilisé par Nuxt est:

```
<!DOCTYPE html>
<html {{ HTML_ATTRS }}>
  <head {{ HEAD_ATTRS }}>
    {{ HEAD }}
  </head>
  <body {{ BODY_ATTRS }}>
    {{ APP }}
  </body>
</html>
```

Un cas d'usage de personnalisation du template app serait d'ajouter des classes CSS conditionnelles pour IE:

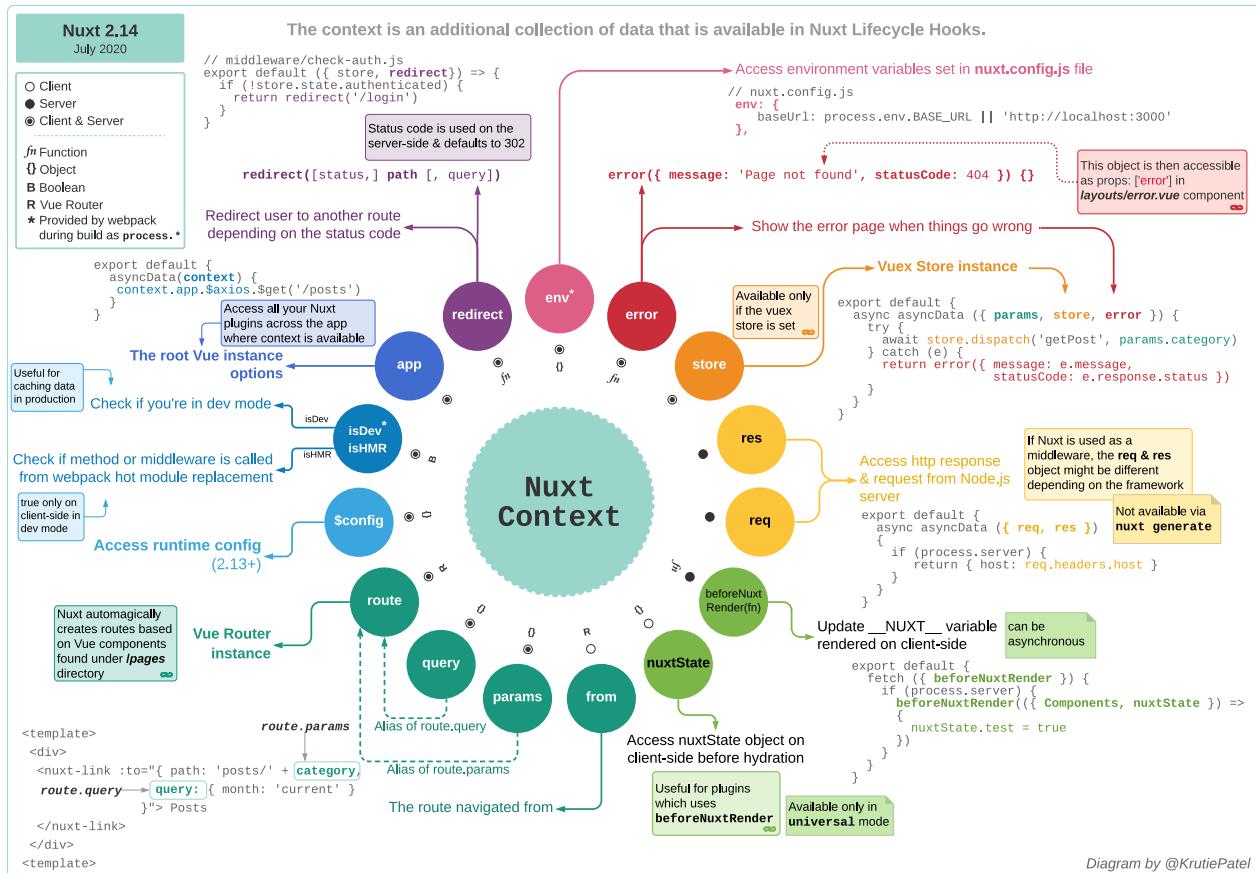
```
<!DOCTYPE html>
<!--[if IE 9]><html class="lt-ie9 ie9" {{ HTML_ATTRS }}><![endif]-->
<!--[if (gt IE 9)!!(IE)]><!--><html {{ HTML_ATTRS }}><!--<![endif]-->
  <head {{ HEAD_ATTRS }}>
    {{ HEAD }}
  </head>
  <body {{ BODY_ATTRS }}>
    {{ APP }}
  </body>
</html>
```

Info

Même si nous pouvons ajouter du JavaScript et des fichiers CSS dans `app.html`, il est plutôt recommandé d'utiliser `nuxt.config.js` pour cet usage !

Helpers et Contexte

Le contexte fournit des informations *additionnelles* et souvent optionnelles de la requête envoyée à l'application.



Nuxt context keys

L'objet `context` est disponible dans des fonctions spécifiques de Nuxt comme `asyncData`, `plugins`, `middleware` et `nuxtServerInit`. Le contexte fournit des informations *additionnelles* et souvent optionnelles de la requête envoyée à l'application.

Tout d'abord, le contexte est utilisé pour permettre un accès aux autres parties d'une application Nuxt, comme par exemple un store Vuex ou l'instance `connect` sous-jacente. Ainsi, nous avons les objets `req` et `res` disponibles dans le contexte du côté serveur et le `store` de toujours disponible. Avec le temps, le contexte fut étendu avec plein d'autres variables bien pratiques et de raccourcis. Ainsi, nous avons accès au fonctionnalités du HMR en mode `development`, la `route` actuelle, les `params` de la page, la `query`

(URL), ainsi que la possibilité d'avoir accès à des variables d'environnement à travers le contexte. En plus de cela, les modules de fonctions et les helpers peuvent être exposés à travers le contexte pour être disponibles sur les deux - côté client et serveur.

Toutes les propriétés du contexte qui sont présents par défaut

```
function (context) {
  // toujours disponibles
  const {
    app,
    store,
    route,
    params,
    query,
    env,
    isDev,
    isHMR,
    redirect,
    error,
    $config
  } = context

  // disponible uniquement côté serveur
  if (process.server) {
    const { req, res, beforeNuxtRender } = context
  }

  // disponible uniquement côté client
  if (process.client) {
    const { from, nuxtState } = context
  }
}
```

Warning

Le `context` dont nous parlons ici ne doit pas être confondu avec l'objet `context` disponible dans les [Actions Vuex](#) ou celui disponible dans la fonction `build.extend` dans notre `nuxt.config.js`. Ils n'ont absolument aucun rapport entre eux !

Découvrez davantage de propriétés disponibles dans le contexte dans le [Glossaire des propriétés internes](#)

Exemples

Utiliser les paramètres de la page pour notre requête API

Le contexte expose directement tous les paramètres dynamiques possibles de la route via `context.params`. Dans l'exemple suivant, on appelle une API grâce au module `nuxt/http` en utilisant un paramètre dynamique de la page dans l'URL. Les modules tels que `nuxt/http` peuvent exposer leurs propres fonctions qui sont par la suite disponibles à travers l'objet `context.app`.

De plus, nous allons entourer l'appel d'API d'une instruction `try/catch` pour gérer le cas d'erreurs potentielles. Avec la fonction `context.error`, nous pouvons directement afficher la page d'erreur de Nuxt et lui passer l'erreur survenue.

```

export default {
  async asyncData(context) {
    const id = context.params.id
    try {
      // ici, on utilise le module nuxtjs/http disponible via context.app
      const post = await context.app.$http.$get(
        `https://api.nuxtjs.dev/posts/${id}`
      )
      return { post }
    } catch (error) {
      context.error(error) // affiche la page d'erreur de nuxt avec l'erreur
    correspondante
    }
  }
}

```

Grâce à l'[ES6](#), on peut utiliser la syntaxe pour déstructurer l'objet `context`. On peut passer les objets aux-
quels nous voulons avoir accès et ensuite les utiliser dans notre code sans même utiliser le mot `context`.

```

export default {
  async asyncData({ params, $http, error }) {
    const id = params.id

    try {
      // ici on utilise le module nuxtjs/http depuis le contexte
      const post = await $http.$get(`https://api.nuxtjs.dev/posts/${id}`)
      return { post }
    } catch (error) {
      error(error) // affiche la page d'erreur de nuxt avec l'erreur
    correspondante
    }
  }
}

```

Si nous préférons utiliser les paramètres de la requête, nous pouvons alors utiliser `context.query.id`.

Rediriger les utilisateurs et accéder au store

Accéder au store Vuex (quand nous l'avons configuré dans le répertoire `store`) est aussi possible via le contexte. Cela fournit un objet `store` qui est équivalent à `this.$store` dans les composants Vue. De plus, on peut utiliser la méthode `redirect`, un helper disponible via le contexte, pour rediriger l'utilisateur dans le cas où le state `authenticated` est `falsy`.

```

export default {
  middleware({ store, redirect }) {
    // on déstructure les objets avec leurs propriétés
    const isAuthenticated = store.state.authenticated
    if (!isAuthenticated) {
      return redirect('/login')
    }
  }
}

```

Les helpers

En plus des raccourcis dans le contexte, il y a aussi de petits helpers présents dans notre application Nuxt.

\$nuxt : Le helper Nuxt

Le helper `$nuxt` est prévu pour améliorer l'expérience utilisateur et peut être une trappe de secours dans certains cas. Il est accessible via `this.$nuxt` dans les composants Vue et via `window.$nuxt` du côté client.

Vérification de la connexion

Le helper `$nuxt` fournit un moyen rapide pour voir si l'utilisateur a de la connexion ou pas: nous avons accès aux booléens `isOffline` et `isOnline`. On peut utiliser ceux-ci pour montrer un message dès que l'utilisateur est hors ligne (par exemple).

```
<template>
  <div>
    <div v-if="$nuxt.isOffline">Vous êtes hors-ligne</div>
    <Nuxt />
  </div>
</template>
```

Accéder à l'instance racine

Au delà de l'aspect DX/UX, le helper `$nuxt` fournit aussi un raccourci à l'instance racine de notre application à partir de n'importe quel composant. Mais ce n'est pas tout - nous pouvons aussi avoir accès au helper `$nuxt` via `window.$nuxt`, ce qui peut être pratique comme solution de secours pour avoir accès à des méthodes de module comme `$axios` depuis l'extérieur de nos composants Vue. Nous pouvons utiliser cela judicieusement et seulement en **dernier recours**.

Actualiser la data de la page

Quand nous voulons actualiser la page actuelle pour l'utilisateur, nous ne voulons pas recharger la page intégralement et entraîner une nouvelle requête au serveur ainsi qu'une nouvelle initialisation de l'intégralité de notre application Nuxt. À la place, on préférera souvent juste actualiser la data, que l'on peut récupérer avec `asyncData` ou `fetch`.

Nous pouvons faire cela en utilisant `this.$nuxt.refresh()` !

```
<template>
  <div>
    <div>{{ content }}</div>
    <button @click="refresh">Actualiser</button>
  </div>
</template>

<script>
  export default {
    asyncData() {
      return { content: 'Créé le: ' + new Date() }
    },
    methods: {
      refresh() {
        this.$nuxt.refresh()
      }
    }
  }
</script>
```

```

        }
    }
</script>

```

Contrôler la barre de progression

Avec `$nuxt`, nous pouvons aussi contrôler la barre de progression de manière programmatique via `this.$nuxt.$loading`.

```

export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}

```

Plus d'informations sont disponibles dans le [chapitre sur le chargement](#)

Le helper onNuxtReady

Si nous voulons exécuter certains scripts *après* que notre application Nuxt soit chargée et prête, nous pouvons utiliser la fonction `window.onNuxtReady`. Cela peut être utile si nous souhaitons exécuter une fonction côté client sans augmenter le temps à l'interactivité (TTI).

```

window.onNuxtReady(() => {
  console.log('Nuxt est monté et prêt')
})

```

Les helpers de Process

Nuxt injecte trois valeurs booléennes dans l'objet global `process`, ce dernier va permettre de déterminer si l'app a été render côté serveur ou intégralement côté client, ainsi que vérifier la génération du site en version statique. Ces helpers sont disponibles à travers notre application et sont communément utilisés dans un bloc `asyncData`.

```

<template>
  <h1>Je suis render côté {{ renderedOn }}</h1>
</template>

<script>
  export default {
    asyncData() {
      return { renderedOn: process.client ? 'client' : 'serveur' }
    }
  }
</script>

```

Dans l'exemple, `renderedOn` aura la valeur `'serveur'` dans le cas d'un render côté serveur quand un utilisateur arrivera sur la page directement. Quand l'utilisateur voudra naviguer jusqu'à la page à partir d'un autre endroit de l'application, par exemple en cliquant sur un `<NuxtLink>`, la valeur sera à `'client'`.

[Go to TOC](#)

Rendu

Le rendu côté serveur (SSR), est la capacité qu'une application a à générer une page web sur le serveur au lieu de la générer dans le navigateur. Le rendu côté serveur renvoie une page complète au client; le JavaScript côté client prend ensuite le relais et permet à Vue.js d'[hydrater](#) notre application.

Serveur Node.js requis

Un environnement JavaScript est requis pour rendre notre page web.

Un serveur Node.js a besoin d'être configuré pour exécuter notre application Vue.js.

Contrôler et étendre les possibilités de notre serveur

Nous pouvons étendre les possibilités de notre serveur avec `serverMiddleware` et contrôler les routes avec `middleware`.

```
export default function (req, res, next) {
  console.log(req.url)
  next()
}

export default {
  serverMiddleware: ['~/server-middleware/logger']
}
```

Différence entre serveur et navigateur

Comme nous sommes dans un environnement Node.js nous avons accès aux objets Node.js tels que `req` et `res`. Nous n'avons pas accès aux objets `window` ou `document` vu qu'ils appartiennent au navigateur. Nous pouvons cependant utiliser `window` ou `document` en les utilisant dans les hooks `beforeMount` ou `mounted`.

```
beforeMount{
  window.alert('hello');
}
mounted{
  window.alert('hello');
}
```

Les étapes pour faire du rendu côté serveur avec Nuxt

Étape 1: Du navigateur au serveur

Quand le navigateur envoie la requête initiale, cela va atterrir sur le serveur Node.js. Nuxt va alors générer le HTML et le renvoyer au navigateur en prenant en compte le résultat des fonctions telles que `asyncData`, `nuxtServerInit` ou `fetch`. Les fonctions présentes dans les hooks seront aussi exécutées.

Étape 2: Du serveur au navigateur

Le navigateur reçoit la page avec le HTML généré depuis le serveur. Le contenu est affiché et l'hydratation de Vue.js entre en scène, rendant le tout réactif. Après ce processus, la page est interactive.

Étape 3: Du navigateur au navigateur

Naviguer entre les pages à l'aide de `<NuxtLink>` est fait du côté client afin que nous n'ayons pas besoin de faire une requête au serveur, sauf si nous faisons un rafraîchissement manuel de notre navigateur.

Génération de site statique

Avec la génération statique nous pouvons générer notre application durant la phase de build et déployer notre site sur n'importe quel hébergeur tel que Netlify, Github pages, Vercel etc... Ce qui veut dire qu'il n'y a pas besoin de serveur pour pouvoir déployer notre application.

Générer notre site

Lorsque nous déployons notre site avec `target:static`, toutes nos pages `.vue` seront générées dans des fichiers HTML et JavaScript. Tous les calls aux API seront faits et mis en cache dans un répertoire nommé `static` à l'intérieur de notre contenu généré pour le côté client, ainsi il n'y aura pas besoin de faire d'appels à notre API par la suite.

Étape 1: Du navigateur au CDN

Lorsqu'un navigateur enverra la requête initiale, il va accéder au CDN.

Étape 2: du CDN au navigateur

Le CDN va envoyer le HTML préalablement généré, ainsi que le JavaScript et toutes les ressources statiques au navigateur. Le contenu sera affiché et l'hydratation de Vue.js commencera à faire effet afin de rendre le tout réactif. À la suite de ce processus, la page sera interactive.

Étape 3: du navigateur au navigateur

Naviguer entre les pages à l'aide de `<NuxtLink>` est fait du côté client afin que nous n'ayons pas besoin de refaire une requête au CDN et les appels à l'API seront chargés à partir du répertoire des ressources mis en cache et ce même si nous rafraîchissons manuellement notre page.

Solution de secours: la SPA

Les pages qui auront été exclues de la génération en utilisant la propriété `generate.exclude` se rabattront sur la version SPA de notre application. Ces pages n'existeront donc pas sur le CDN et seront régénérées sur le client (dans le navigateur) à chaque fois qu'un utilisateur naviguera sur ces pages.

Warning

Pour en savoir davantage sur la propriété `generate`

Mettre à jour notre contenu

Afin de récupérer le nouveau contenu sur notre site à partir de l'API, nous aurons besoin de régénérer le site à nouveau. Avec la plupart des services d'hébergement, nous pouvons parvenir à cela en poussant nos modifications sur la branche `main` avec git ou via une pull request.

Mode de prévisualisation

Le mode de prévisualisation va appeler notre API ou notre CMS afin que nous puissions voir les changements en live avant de déployer. Se référer au [mode de prévisualisation](#) pour activer cette fonctionnalité.

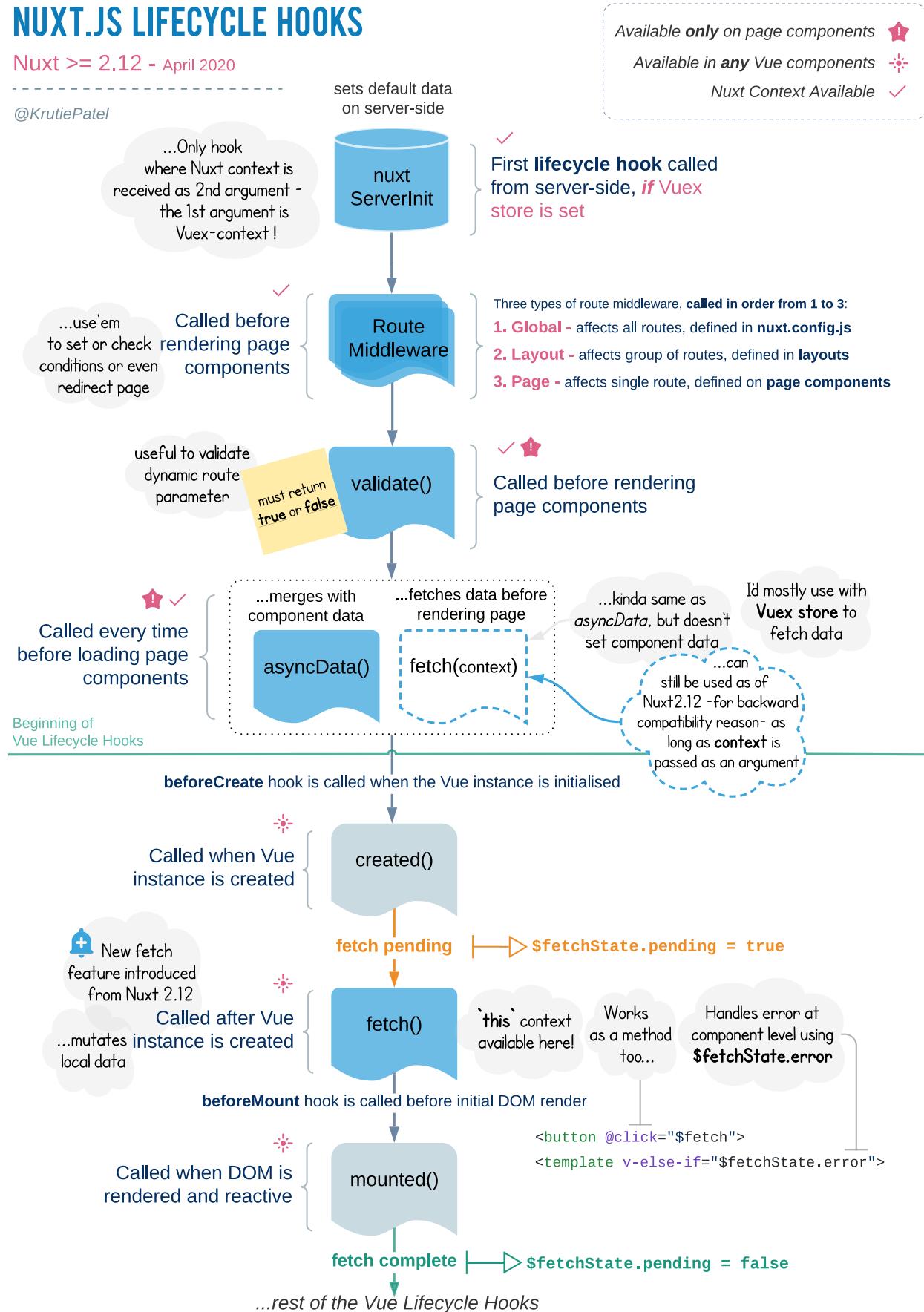
Cycles de vie de Nuxt

Peu importe l'outil que nous utilisons, nous serons davantage en confiance lorsque nous serons capable de comprendre comment il marche dans les détails. Il en est de même pour Nuxt. Le but de ce chapitre est de nous donner une vue d'ensemble des différentes parties du framework, leur ordre d'exécution ainsi que la façon dont ils interagissent entre eux.

NUXT.JS LIFECYCLE HOOKS

Nuxt >= 2.12 - April 2020

@KrutiePatel



Les lifecycles de Nuxt 2.12

Le cycle de vie de Nuxt décrit les différentes étapes qui ont lieu après la phase de build, une fois que notre application a été bundlée, chunkée et minifiée. Ce qu'il se passe après cette phase dépend de si nous avons activé le rendu côté serveur ou pas. Si c'est le cas, il faut ensuite regarder le type de rendu serveur que nous avons choisi:

Rendu dynamique côté serveur (SSR) (`nuxt start`)

ou Génération statique de site (SSG) (`nuxt generate`).

Cycle de vie

Serveur

Pour le SSR, les étapes suivantes seront exécutées pour chaque requête sur notre app

- Le serveur démarre (`nuxt start`)

Lorsque nous utilisons la génération statique de notre site, les étapes qui ont lieu sur le serveur ne seront exécutées que lors de la phase de build et une fois pour chaque page générée.

- Le processus de génération commence (`nuxt generate`)
- Hooks Nuxt
- serverMiddleware
- Les plugins de Nuxt côté serveur
 - en suivant l'ordre défini dans `nuxt.config.js`
- `nuxtServerInit`
 - action Vuex qui est appelée seulement côté serveur pour pré-remplir le store
 - le premier argument dans le **context de Vuex**, le second argument dans le **context de Nuxt**
 - on dispatch les autres actions d'ici → c'est seulement le "point d'entrée" pour des actions ultérieures du côté serveur
 - peut seulement être défini dans le `store/index.js`
- Middleware
 - Global middleware
 - Layout middleware
 - Route middleware
- `asyncData`
- `beforeCreate` (cycle de vie de Vue)
- `created` (cycle de vie de Vue)

- Le nouveau fetch (du haut vers le bas, en parallèle si ce sont des voisins)
- Sérialisation du state (hook `render:routeContext` de Nuxt)
- le rendu HTML commence (hook `render:route` de Nuxt)
- `render:routeDone` hook quand le HTML a été envoyé au navigateur
- `generate:before` hook de Nuxt
- les fichiers HTML sont générés
 - **génération intégralement statique**
 - ex. les payloads statiques sont extraits
 - `generate:page` (HTML éditable)
 - `generate:routeCreated` (Route générée)
- `generate:done` quand tous les fichiers HTML ont été générés

Client

Cette partie du cycle est entièrement exécutée dans le navigateur, peu importe le mode que nous avons choisi pour Nuxt.

- Reçoit le HTML
- Charge les ressources (ex. Javascript)
- Hydratation de Vue
- Middleware
 - Global middleware
 - Layout middleware
 - Route middleware
- `asyncData` (bloquant)
- plugins clients de Nuxt
 - en suivant l'ordre défini dans `nuxt.config.js`
- `beforeCreate` (méthode du cycle de vie de Vue)
- `created` (méthode du cycle de vie de Vue)
- Le nouveau fetch (du haut vers le bas, en parallèle si ce sont des voisins)
- `beforeMount` (méthode du cycle de vie de Vue)
- `mounted` (méthode du cycle de vie de Vue)

Naviguer en utilisant le composant NuxtLink

De la même façon que pour la partie *client*, tout se passe dans le navigateur mais seulement lors d'une navigation avec `<NuxtLink>`. De plus, aucun contenu n'est affiché sur la page tant qu'il reste des tâches *bloquantes*.

Info

Se référer au chapitre sur les composants pour plus d'informations sur `<NuxtLink>`.

- middleware (bloquant)
 - Global middleware
 - Layout middleware
 - Route middleware
- asyncData (bloquant)
- asyncData (bloquant) [ou chargement entièrement statique du payload]
- beforeCreate & created (méthode du cycle de vie de Vue)
- fetch (non-bloquant)
- beforeMount & mounted

Ensuite ?

Next

Plus d'informations dans le [recueil sur les features](#).

[Go to TOC](#)

Les modes de rendus

Rendu côté serveur et sites statiques

Les sites de rendus côté serveur sont rendus sur le serveur chaque fois que l'utilisateur demande une page, par conséquent un serveur est nécessaire pour pouvoir servir la page à chaque demande.

Les sites statiques sont très similaires aux applications rendues côté serveur, la principale différence étant que les sites statiques sont rendus au moment de la construction, aucun serveur n'est donc nécessaire. La navigation d'une page à l'autre se fait alors côté client.

Voir les [cibles de déploiements](#) pour plus d'informations sur l'hébergement statique et serveur.

```
export default {
  ssr: true // default value
}
```

Info

Vous n'avez pas besoin d'ajouter `ssr: true` à votre configuration nuxt pour activer le rendu côté serveur vu qu'il est activé par défaut.

Rendu côté client uniquement

Avec le rendu côté client uniquement, il n'y a pas de rendu côté serveur. Le rendu côté client signifie le rendu du contenu dans le navigateur à l'aide de JavaScript. Au lieu d'obtenir tout le contenu du HTML, nous obtenons simplement un document HTML de base avec un fichier JavaScript qui rendra ensuite le reste du site à l'aide du navigateur. Pour le rendu côté client, définissez `ssr` sur `false`.

```
export default {
  ssr: false
}
```

Next

[La propriété ssr](#)

Mode de prévisualisation

Avec Nuxt et une génération axée sur de l'intégrallement statique, nous pouvons maintenant utiliser la prévisualisation en temps réel, sans configuration additionnelle. Cela nous permettra d'appeler notre API ou notre CMS afin que nous puissions voir les changements avant de déployer.

Info

Seulement disponible dans le cas d'un `target:static`

Le mode de prévisualisation va s'occuper de rafraîchir automatiquement la data de la page vu qu'il utilise `$nuxt.refresh` (et donc fait les appels de `nuxtServerInit`, `asyncData` et `fetch`) du côté client.

Pour activer la prévisualisation en temps réel, il faudra ajouter le plugin suivant:

```
export default function ({ query, enablePreview }) {
  if (query.preview) {
    enablePreview()
  }
}
```

Info

`EnablePreview` est seulement disponible dans l'objet contexte des plugins. Les prévisualisations sont générées côté client et donc, le plugin doit s'exécuter sur le client: `preview.client.js`

```
export default {
  plugins: ['~/plugins/preview.client.js']
}
```

Une fois que l'on a ajouté le plugin, on peut maintenant générer et servir notre site.

```
yarn generate
yarn start
```

```
npm run generate
npm run start
```

Nous pouvons maintenant voir la page de prévisualisation en ajoutant une query param à la fin de la page qui nous intéresse:

```
?preview=true
```

Info

`enablePreview` doit être testé localement avec `yarn start` et non `yarn dev`

Prévisualiser les pages qui ne sont pas encore générées

Pour les pages qui ne sont pas encore générées, la solution de rechange de la SPA va continuer de s'occuper de faire un appel à l'API avant de montrer une page 404 vu que ces pages existent sur l'API mais ne sont pas encore générées.

Si nous avons configuré un crochet de validation (webhook), on aura sans doute besoin de le modifier afin d'accommorder le fait qu'il ne redirige plus vers une 404 en mode de prévisualisation.

```
validate({ params, query }) {  
  if (query.preview) {  
    return true  
  }  
}
```

Passer la data à enablePreview

On peut passer de la data à la fonction `enablePreview`. La data sera ainsi disponible sur le helper de contexte `$preview` et sur `this.$preview`.

Et ensuite ?

Next

Se référer au [document sur la structure des répertoires](#).

Transitions

Nuxt utilise le composant `<transition>` pour permettre de créer de superbes transitions/animations entre nos routes.

Pour définir une transition personnalisée pour une route spécifique, il faut ajouter la propriété `transition` au composant page.

```
export default {
  // peut être une chaîne de caractères
  transition: ''
  // ou un objet
  transition: {}
  // ou une fonction
  transition (to, from) {}
}
```

Chaîne de caractères

Si la propriété `transition` est une chaîne de caractères, ce sera utilisé comme `transition.name`.

```
export default {
  transition: 'home'
}
```

Nuxt va utiliser ces paramètres pour configurer le composant comme suit:

```
<transition name="home"></transition>
```

Info

Ceci est fait automatiquement pour nous et nous n'avons pas besoin d'ajouter de composant `<transition>` à nos pages ou layouts.

Maintenant, il ne nous reste plus qu'à créer la nouvelle classe pour nos transitions.

```
<styles>
  .home-enter-active, .home-leave-active { transition: opacity .5s; }
  .home-enter, .home-leave-active { opacity: 0; }
</styles>
```

Objet

Si la propriété `transition` est un objet:

```
export default {
  transition: {
    name: 'home',
```

```
    mode: 'out-in'
  }
}
```

Nuxt va utiliser ces paramètres pour configurer le composant comme suit:

```
<transition name="test" mode="out-in"></transition>
```

L'objet `transition` peut avoir des propriétés telles que le nom, le mode, le CSS, la durée et encore tout plein d'autres ! La [documentation de Vue](#) a plus d'informations sur ce sujet. On peut aussi définir des méthodes dans la propriété `transition` de la page.

```
export default {
  transition: {
    afterLeave(el) {
      console.log('afterLeave', el)
    }
  }
}
```

Mode de transition

Info

La mode de transition par défaut pour les pages diffère de celui de Vue.js. Ici, le mode par défaut est à `out-in`. Si l'on souhaite avoir des transitions d'entrée/sortie simultanées, il faudra changer le mode en lui attribuant une chaîne de caractères vide `mode: ''`.

```
export default {
  transition: {
    name: 'home',
    mode: ''
  }
}
```

Function

Si la propriété `transition` est une fonction:

```
export default {
  transition(to, from) {
    if (!from) {
      return 'slide-left'
    }
    return +to.query.page < +from.query.page ? 'slide-right' : 'slide-left'
  }
}
```

Voici ce que cela va générer comme transitions:

- `/` to `/posts` => `slide-left`
- `/posts` to `/posts?page=3` => `slide-left`
- `/posts?page=3` to `/posts?page=2` => `slide-right`

Paramètres globaux

Le nom de la transition par défaut de Nuxt est "page". Pour ajouter une transition d'estompage à chaque page de notre application, nous avons seulement besoin d'un fichier CSS qui est partagé entre toutes les routes.

Notre fichier global CSS `assets/main.css` :

```
.page-enter-active,
.page-leave-active {
  transition: opacity 0.5s;
}
.page-enter,
.page-leave-to {
  opacity: 0;
}
```

Nous l'ajoutons à notre tableau CSS dans le fichier `nuxt.config.js` :

```
export default {
  css: ['~/assets/main.css']
}
```

Paramètres de configuration

La propriété `layoutTransition`

`layoutTransition` est utilisée pour définir les propriétés par défaut pour les transitions du layout.

Les paramètres par défaut sont :

```
{
  name: 'layout',
  mode: 'out-in'
}
```

```
.layout-enter-active, .layout-leave-active {
  transition: opacity .5s
}
.layout-enter, .layout-leave-active {
  opacity: 0
}
```

Pour les changer, rendons-nous dans le fichier `nuxt.config.js` :

```
export default {
  layoutTransition: 'mes-layouts'
  // ou
  layoutTransition: {
    name: 'mes-layouts',
    mode: 'out-in'
  }
}
```

```
.mes-layouts-enter-active,
.mes-layouts-leave-active {
  transition: opacity 0.5s;
}
.mes-layouts-enter,
.mes-layouts-leave-active {
  opacity: 0;
}
```

La propriété pageTransition

Les paramètres par défaut sont:

```
{
  name: 'page',
  mode: 'out-in'
}
```

Pour les changer, rendons-nous dans le fichier `nuxt.config.js`

```
export default {
  pageTransition: 'ma-page'
  // ou
  pageTransition: {
    name: 'ma-page',
    mode: 'out-in',
    beforeEnter (el) {
      console.log('Avant de rentrer...');
    }
  }
}
```

Si l'on modifie `pageTransition`, il faudra faire attention à renommer les classes CSS qui vont avec.

```
.ma-page-enter-active,
.ma-page-leave-active {
  transition: opacity 0.5s;
}
.ma-page-enter,
.ma-page-leave-to {
  opacity: 0;
}
```

Cibles de déploiement

Hébergement statique

Avec de l'hébergement statique, nous n'avons pas besoin de serveur, nous pouvons choisir d'héberger une SPA ou une MPA (`multiple page application`), en somme: des sites statiques. Ce qui veut dire que nous pouvons envoyer le tout sur un hébergement serverless ou un CDN et ce, de manière gratuite. Pour de l'hébergement statique, la propriété `target` doit être mise sur `static` dans le fichier `nuxt.config`.

```
export default {
  target: 'static' // la valeur par défaut est 'server'
}
```

SPA

Les Single Page Applications (SPA) sont des pages qui sont render seulement du côté client sans le besoin d'un serveur. Pour déployer une SPA, il faut mettre le `mode` à `spa` et ensuite utiliser la commande `build` pour générer notre application.

```
export default {
  target: 'static' // la valeur par défaut est 'server'
  mode: 'spa'
}
```

Sites Statiques

Comme Nuxt marche aussi en tant que générateur de sites statiques, nous pouvons générer notre application en tant que tel. Générer notre application Nuxt et bénéficier de tous les avantages d'une application universelle sans avoir besoin d'un serveur. La commande `nuxt generate` va s'occuper de générer l'HTML pour chacune de nos routes et mettre le tout dans le répertoire `dist`.

Globalement, chaque fichier `.vue` qui sera placé à l'intérieur du répertoire `pages` sera généré en tant que page HTML statique. Cela va améliorer la performance, le SEO (référencement naturel) ainsi que permettre un meilleur support en mode hors-ligne.

Info

Les sites statiques fonctionnent avec le mode `universel` par défaut.

Utiliser la commande `nuxt dev` avec `static` comme cible de déploiement peut améliorer l'expérience développeur.

- Enlève `req` & `res` du `context`
- Solution de secours sur le rendu côté client dans le cas d'une 404, d'erreurs et de redirections [voir les solutions de secours d'une SPA](#)
- `$route.query` sera toujours égal à `{}` sur le render côté serveur
- `process.static` est vrai

Info

Nous mettons de plus `process.target` à disposition pour les auteurs de modules qui souhaiteraient ajouter un comportement particulier en fonction de l'utilisateur visé.

Hébergement avec serveur

L'hébergement avec serveur requiert un serveur et a pour but d'être utilisé dans le cas d'applications SSR. Le rendu côté serveur aussi connu sous le nom de SSR, signifie que notre page est générée sur le serveur à chaque fois qu'un utilisateur en fait la requête. Quand l'utilisateur ouvre une de nos pages, son navigateur va envoyer une requête au serveur pour lui demander cette page. Une fois render sur le serveur, la page est renvoyée au navigateur avec tout le contenu.

Pour de l'hébergement avec serveur, la propriété `target` doit être mise à `server` (valeur par défaut). On utilise la commande `build` pour générer l'application.

```
export default {  
  target: 'server'  
}
```

Info

Les SSR fonctionnent avec le mode [universel](#) par défaut.

Système de route par fichier

Nuxt s'occupe de générer automatiquement la configuration pour vue-router en fonction de l'arborescence des fichiers Vue à l'intérieur du répertoire des pages. Il suffit de créer un fichier .vue à l'intérieur du répertoire des pages pour que le routage soit fonctionnel, nul besoin de configuration.

Parfois nous aurons besoin de créer une route dynamique ou une route imbriquée ou encore de configurer notre router de manière plus poussée. Ce chapitre a pour but de tout passer en revue afin de tirer le maximum de notre routeur.

Info

Nuxt nous offre du code splitting automatique pour nos routes, sans configuration supplémentaire.

Info

Utilisez le composant [NuxtLink](#) pour naviguer entre les pages.

```
<template>
  <nuxt-link to="/">Page d'accueil</nuxt-link>
</template>
```

Routes Basiques

Cette structure de fichier:

```
pages/
--| user/
-----| index.vue
-----| one.vue
--| index.vue
```

va automatiquement générer ceci:

```
router: {
  routes: [
    {
      name: 'index',
      path: '/',
      component: 'pages/index.vue'
    },
    {
      name: 'user',
      path: '/user',
      component: 'pages/user/index.vue'
    },
    {
      name: 'user-one',
      path: '/user/one',
      component: 'pages/user/one.vue'
    }
  ]
}
```

```

    }
]
```

Routes Dynamiques

Parfois il n'est pas possible de connaître le nom de la route. Un call vers une API pour récupérer une liste d'utilisateurs ou d'articles de blog par exemple. C'est ce que l'on appelle des routes dynamiques. Pour créer une route dynamique, il faut ajouter un underscore (_) avant le nom du fichier `.vue` ou devant le nom du répertoire. Nous pouvons nommer le fichier ou le répertoire de la façon dont nous voulons mais il doit être préfixé d'un underscore.

Cette structure de fichiers:

```

pages/
--| _slug/
----| comments.vue
----| index.vue
--| users/
----| _id.vue
--| index.vue
```

va automatiquement générer ceci:

```

router: {
  routes: [
    {
      name: 'index',
      path: '/',
      component: 'pages/index.vue'
    },
    {
      name: 'users-id',
      path: '/users/:id?',
      component: 'pages/users/_id.vue'
    },
    {
      name: 'slug',
      path: '/:slug',
      component: 'pages/_slug/index.vue'
    },
    {
      name: 'slug-comments',
      path: '/:slug/comments',
      component: 'pages/_slug/comments.vue'
    }
  ]
}
```

Info

Comme nous pouvons le voir, la route nommée `users-id` a la valeur `:id?` dans son chemin, ce qui en fait une valeur optionnelle. Si nous souhaitons rendre ce paramètre obligatoire, il faut créer un fichier `index.vue` dans le répertoire `users/_id`.

Info

Depuis Nuxt >= v2.13, un crawler va détecter nos liens et générer des routes dynamiques à partir de ceux-ci. Cependant, si nous avons des pages qui ne sont pas liées (comme une page secrète), nous aurons besoin de spécifier manuellement que nous souhaitons générer cette route dynamique.

Next

Générer des routes dynamiques pour des sites statiques.

Accéder aux paramètres de la route localement

Nous pouvons accéder aux paramètres de la route actuelle, depuis notre page locale ou notre composant en appelant `this.$route.params.{nomDuParamètre}`. Par exemple, si nous avons une page dynamique avec la liste de nos utilisateurs (`users_{id}.vue`), nous pouvons accéder au paramètre `id` pour charger un utilisateur en particulier, pour cela nous pouvons utiliser: `this.$route.params.id`.

Routes Imbriquées

Nuxt nous permet de créer des routes imbriquées grâce à `vue-router`. Pour définir le composant parent d'une route imbriquée, nous avons besoin de créer un fichier Vue du même nom que le répertoire qui va contenir les vues imbriquées.

Info

Il ne faut pas oublier d'inclure le composant `NuxtChild` à l'intérieur du composant parent (fichier `.vue`).

Cette structure de fichiers:

```
pages/
--| users/
-----| _id.vue
-----| index.vue
--| users.vue
```

va automatiquement générer ceci:

```
router: {
  routes: [
    {
      path: '/users',
      component: 'pages/users.vue',
      children: [
        {
          path: '',
          component: 'pages/users/index.vue',
          name: 'users'
        },
        {
          path: ':id',
          component: 'pages/users/_id.vue',
          name: 'users-id'
        }
      ]
    }
  ]
}
```

```

    }
]
}

```

Routes dynamiques imbriquées

Ce n'est pas très commun mais il se peut que nous ayons envie d'avoir des enfants dynamiques dans des parents dynamiques, c'est possible avec Nuxt.

Cette structure de fichiers:

```

pages/
--| _category/
----| _subCategory/
| _id.vue
| index.vue
----| _subCategory.vue
----| index.vue
--| _category.vue
--| index.vue

```

va automatiquement générer ceci:

```

router: {
  routes: [
    {
      path: '/',
      component: 'pages/index.vue',
      name: 'index'
    },
    {
      path: '/:category',
      component: 'pages/_category.vue',
      children: [
        {
          path: '',
          component: 'pages/_category/index.vue',
          name: 'category'
        },
        {
          path: ':subCategory',
          component: 'pages/_category/_subCategory.vue',
          children: [
            {
              path: '',
              component: 'pages/_category/_subCategory/index.vue',
              name: 'category-subCategory'
            },
            {
              path: ':id',
              component: 'pages/_category/_subCategory/_id.vue',
              name: 'category-subCategory-id'
            }
          ]
        }
      ]
    }
  ]
}

```

Routes dynamiques imbriquées inconnues

Si nous ne connaissons pas la profondeur d'une URL, nous pouvons utiliser `_.vue` pour dynamiquement faire correspondre les chemins imbriqués. Cela va matcher tant qu'il n'y a pas de *requête plus spécifique*.

Cette structure de fichiers:

```
pages/
--| people/
-----| _id.vue
-----| index.vue
--| _.vue
--| index.vue
```

va automatiquement générer ceci:

```
/ -> index.vue
/people -> people/index.vue
/people/123 -> people/_id.vue
/about -> _.vue
/about/careers -> _.vue
/about/careers/chicago -> _.vue
```

Info

Gérer les pages 404 ne fait pas partie du travail d'une page `_.vue`.

Personnaliser le routeur

Il y a plusieurs façons de personnaliser notre routeur avec Nuxt:

- `router-extras-module` pour personnaliser les paramètres dans la page
- le composant `@nuxtjs/router` pour écraser le routeur Nuxt et écrire notre propre fichier `router.js`
- utiliser la propriété `router.extendRoutes` dans notre fichier `nuxt.config.js`

La propriété router

La propriété `router` nous permet de personnaliser le routeur Nuxt (`vue-router`).

```
export default {
  router: {
    // personnalisons le routeur de Nuxt
  }
}
```

Base:

L'url de base de notre application. Par exemple, si nous voulons que notre Single Page App soit sous le chemin `/app/`, alors nous devons lui attribuer la valeur `'app/'`.

Next

[Propriété base du routeur.](#)

extendRoutes

Nous voulons peut-être personnaliser les routes créées par Nuxt, pour cela nous pouvons utiliser l'option `extendRoutes`.

Exemple d'ajout d'une route personnalisée:

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        name: 'custom',
        path: '*',
        component: resolve(__dirname, 'pages/404.vue')
      })
    }
  }
}
```

Si nous souhaitons trier nos routes, nous pouvons utiliser la fonction `sortRoutes(routes)` à partir de `@nuxt/utils`:

```
import { sortRoutes } from '@nuxt/utils'
export default {
  router: {
    extendRoutes(routes, resolve) {
      // ajouter nos routes ici ...

      // trions-les
      sortRoutes(routes)
    }
  }
}
```

Info

Les routes doivent respecter le schéma du [vue-router](#).

Info

Lorsque l'on ajoute les routes qui utilisent des [routes nommées](#), il ne faut pas oublier d'ajouter les `chunkNames` des `components`.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        path: '/users/:id',
        components: {
          default: resolve(__dirname, 'pages/users'), // ou
          routes[index].component
          modal: resolve(__dirname, 'components/modal.vue')
        },
      })
    }
  }
}
```

```

        chunkNames: {
          modal: 'components/modal'
        }
      })
    }
  }
}

```

Next

[propriété extendRoutes](#).

fallback

S'occupe du routeur, si ce dernier doit se rabattre en mode `hash` lorsque le navigateur ne supporte pas `history.pushState` alors que le mode est configuré sur `history`.

Next

[propriété fallback](#).

mode

Nous pouvons changer le mode du routeur, ce n'est cependant pas recommandé en raison du Server Side Rendering (SSR).

Next

[propriété mode](#).

parseQuery / stringifyQuery

Fournit des fonctions de *parsing* / *stringification* personnalisées.

Next

[propriété parseQuery / stringifyQuery](#).

routeNameSplitter

Nous pourrions vouloir changer le séparateur entre les noms des routes que Nuxt utilise, pour cela nous pouvons utiliser l'option `routeNameSplitter` dans notre fichier de configuration. Prenons par exemple le fichier page `pages/posts/_id.vue`. Nuxt va générer le nom de la route de manière programmatique, ici ce sera `posts-id`. Passer le `routeNameSplitter` à `/` va changer le nom en `posts/id`.

```

export default {
  router: {
    routeNameSplitter: '/'
  }
}

```

scrollBehavior

L'option `scrollBehavior` nous laisse le choix pour définir un comportement personnalisé pour la position du défilement entre les routes. Cette méthode est appelée à chaque fois qu'une page est render.

Next

Pour plus d'informations, veuillez consulter [la documentation sur le scrollBehavior de vue-router](#).

Disponible depuis la version `2.9.0` :

Dans Nuxt, nous pouvons utiliser un fichier pour écraser le `scrollBehavior` du routeur. Ce fichier doit être placé dans le répertoire `app`.

`~/app/router.scrollBehavior.js`.

Un exemple de comment forcer la position du défilement à se retrouver tout en haut pour chaque route:

```
export default function (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

Next

fichier `router.scrollBehavior.js` par défaut de Nuxt.

Next

propriété `scrollBehavior`.

trailingSlash

Disponible depuis la version `2.10` :

Si cette option est passée à `true`, des slashes traînantes seront suffixés pour chaque route. Sinon, ils seront enlevés.

```
export default {
  router: {
    trailingSlash: true
  }
}
```

Info

Cette option doit être changée avec une certaine préparation et une sequence de tests conséquents. Lorsque l'on définit `router.trailingSlash` à quelque chose d'autre que la valeur par défaut (`undefined`), la route opposée cessera de fonctionner. Il doit donc y avoir des redirections 301 et nos *liens internes* doivent s'adapter eux aussi. Si nous passons `trailingSlash` à `true`, alors seulement `example.com/abc/` vont marcher mais pas `example.com/abc`. Dans le cas d'un `false`, c'est l'inverse.

Next

[propriété trailingSlash.](#)

[Go to TOC](#)

Récupération de la data

Dans Nuxt, nous avons 2 façons de récupérer de la data depuis une API. Nous pouvons utiliser la méthode `fetch` ou bien `asyncData`.

Nuxt prend en charge les modèles Vue traditionnels pour le chargement de données dans votre application côté client, comme la récupération de données dans le hook `mounted()` d'un composant. Cependant, les applications universelles doivent utiliser des hooks spécifiques à Nuxt pour pouvoir restituer les données lors du rendu côté serveur. Cela permet à votre page de s'afficher avec toutes ses données requises présentes.

Nuxt a deux hooks pour le chargement asynchrone des données:

- Le hook `fetch` (Nuxt 2.12+). Ce hook peut être placé sur n'importe quel composant et fournit des raccourcis pour le rendu des états de chargement (pendant le rendu côté client) et des erreurs.
- Le hook `asyncData`. Ce hook ne peut être placé que sur les composants *page*. Contrairement à `fetch`, ce hook n'affiche pas d'espace réservé de chargement pendant le rendu côté client: à la place, ce hook bloque la navigation de l'itinéraire jusqu'à ce qu'il soit résolu, affichant une erreur de page en cas d'échec.

Info

Dans les versions de Nuxt antérieures à la 2.12, le hook `fetch` fonctionnait un peu comme `asyncData` aujourd'hui. Cette fonctionnalité est toujours supportée aujourd'hui pour des raisons de rétrocompatibilité: si un argument `context` est accepté dans votre `fetch()`, il sera considéré comme un hook de récupération "hérité". Cette fonctionnalité est obsolète et doit être remplacée par `asyncData(context)` ou par un `middleware anonyme` en utilisant `middleware(context)`.

Ces hooks peuvent être utilisés avec *toute bibliothèque de récupération de données* que vous choisissez. Nous vous recommandons d'utiliser [@nuxt/http](#) ou [@nuxt/axios](#) pour faire des requêtes HTTP aux API. Vous trouverez plus d'informations sur ces bibliothèques, telles que des guides de configuration, des en-têtes d'authentification, dans leur documentation respective.

Le hook `fetch`

Info

Ce hook est uniquement disponible à partir de Nuxt 2.12+.

Le hook `fetch` de Nuxt est appelé après que l'instanciation du composant soit faite côté serveur: `this` est disponible à l'intérieur.

```
export default {
  async fetch() {
    console.log(this)
  }
}
```

Info

`fetch(context)` a été déprécié dans nos pages, il faut utiliser un [middleware anonyme](#) à la place: `middleware(context)`

Quand utiliser `fetch`?

À chaque fois que nous avons besoin d'avoir de la data asynchrone. `fetch` est appelé côté serveur lors du render de la route et côté client lors de la navigation.

Cela expose `$fetchState` au niveau du composant et avec les propriétés suivantes:

- `pending` est un Booléen, il permet d'afficher un placeholder quand `fetch` est appelé côté *client*.
- `error`, il peut être soit à `null` soit à `Error` et nous permet d'afficher un message
- `timestamp` est l'horodatage du dernier `fetch`, cela peut s'avérer utile pour faire du [cache avec keep-alive](#)

Nous avons aussi accès à `this.$fetch()`, utile si nous voulons appeler le hook `fetch` dans notre composant.

```
<template>
<p v-if="$fetchState.pending">Récupération des montagnes... 🏔</p>
<p v-else-if="$fetchState.error">Une erreur est survenue :(</p>
<div v-else>
  <h1>Montagnes Nuxt</h1>
  <ul v-for="mountain of mountains">
    <li>{{ mountain.title }}</li>
  </ul>
  <button @click="$fetch">Actualiser</button>
</div>
</template>

<script>
export default {
  data() {
    return {
      mountains: []
    }
  },
  async fetch() {
    this.mountains = await fetch(
      'https://api.nuxtjs.dev/mountains'
    ).then(res => res.json())
  }
}
</script>
```

Info

On peut accéder au `contexte` Nuxt à l'intérieur du hook `fetch` avec `this.$nuxt.context`.

Options

`fetchOnServer`: Un booléen ou une fonction (par défaut: `true`), utile si on veut appeler `fetch()` lors du render de la page côté serveur.

`fetchDelay` : Un entier (par défaut: `200`), définit une limite minimale exprimée en millisecondes de temps d'exécution (pour éviter les flashes).

Lorsque `fetchOnServer` est falsy (`false` ou retourne quelque chose de faux), `fetch` sera appelé uniquement côté client et `$fetchState.pending` retournera `true` lors du render du composant côté serveur.

```
export default {
  data() {
    return {
      posts: []
    }
  },
  async fetch() {
    this.posts = await fetch('https://api.nuxtjs.dev/posts').then(res =>
      res.json()
    )
  },
  // on appelle fetch uniquement du côté client
  fetchOnServer: false
}
```

Écoute des modifications des query strings

Le hook `fetch` n'est pas appelé sur les modifications des query strings par défaut. Pour surveiller ces modifications, nous pouvons ajouter un watcher sur `$route.query` et appeler `$fetch` :

```
export default {
  watch: {
    '$route.query': '$fetch'
  },
  async fetch() {
    // appelé aussi lors des modifications de query
  }
}
```

Mise en cache

Nous pouvons utiliser la directive `keep-alive` dans les composants `<nuxt/>` et `<nuxt-child>` pour sauvegarder le résultat des appels de `fetch` dans les pages déjà visitées:

```
<template>
  <nuxt keep-alive />
</template>
```

Nous pouvons aussi spécifier les `props` passées à `<keep-alive>` en passant la prop `keep-alive-props` au composant `<nuxt>`.

```
<nuxt keep-alive :keep-alive-props="{ max: 10 }" />
```

Le code ci-dessus va garder en mémoire 10 composants `page`.

Utiliser le hook `activated`

Nuxt va directement remplir `this.$fetchState.timestamp` (horodatage) avec le dernier appel de `fetch` (cela inclut le SSR). Nous pouvons utiliser cette propriété en combinaison avec le hook `activated` pour ajouter 30 secondes de cache à `fetch`:

```
<template> ... </template>

<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    activated() {
      // appeler fetch de nouveau si le dernier appel date de plus de 30 secondes
      if (this.$fetchState.timestamp <= Date.now() - 30000) {
        this.$fetch()
      }
    },
    async fetch() {
      this.posts = await fetch('https://api.nuxtjs.dev/posts').then(res =>
        res.json()
      )
    }
  }
</script>
```

La navigation à la même page ne va pas rappeler `fetch` tant que le dernier call ne date pas d'au moins 30 secondes.

Async Data

Info

`asyncData` est seulement disponible pour les [pages](#) et nous n'avons donc pas accès à `this` à l'intérieur du hook.

La différence principale avec `fetch` est que vous n'avez pas à gérer les status d'erreur ou en cours. Nuxt va attendre que le hook `asyncData` soit terminé avant de procéder à la navigation sur la page suivante ou afficher la [page d'erreur](#)

Ce hook reçoit [le contexte](#) en tant que premier argument. Nous pouvons l'utiliser pour aller chercher de la data et Nuxt va automatiquement fusionner l'object retourné avec le `data` du composant.

```
<template>
  <h1>{{ project }}</h1>
</template>

<script>
  export default {
    async asyncData(context) {
      return {
        project: 'nuxt'
      }
    }
  }
</script>
```

```
        }
    }
</script>
```

Dans les exemples qui suivent nous utilisons `@nuxt/http`, que nous recommandons pour aller récupérer de la data d'une API.

Écouter les changements de query

La méthode `asyncData` n'est pas appelée par défaut sur les modifications liées aux query strings. Si nous souhaitons changer ce comportement, pour par exemple créer un composant de pagination, nous pouvons configurer des paramètres qui seront par la suite écoutés grâce à la propriété `watchQuery` sur le composant de notre page.

Info

Pour en apprendre davantage sur la propriété `watchQuery` et voir la liste des clés disponibles dans le contexte.

Balises meta et référencement SEO

Nuxt nous permet de définir les balises `<meta>` par défaut pour notre application dans le fichier `nuxt.config.js` en utilisant la propriété `head`. Cela permet d'ajouter un titre et une description par défaut pour le référencement ou bien définir la fenêtre d'affichage ou ajouter la favicon.

Nuxt nous propose trois façons différentes d'ajouter des métadonnées à l'application :

```
div{.d-heading-description .leading-6}
```

1. Utiliser globalement le fichier `nuxt.config.js`
2. Utiliser localement la propriété `head` comme un objet
3. Utiliser localement la propriété `head` comme une fonction pour que nous ayons accès aux données et aux propriétés calculées.

Paramètres globaux

Nuxt nous permet de définir les balises `<meta>` par défaut pour notre application dans le fichier `nuxt.config.js` en utilisant la propriété `head`. Cela permet d'ajouter un titre et une description par défaut pour le référencement ou bien définir la fenêtre d'affichage ou ajouter la favicon.

```
export default {
  head: {
    title: 'mon titre de site web',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      {
        hid: 'description',
        name: 'description',
        content: 'ma description de site web'
      }
    ],
    link: [{ rel: 'icon', type: 'image/x-icon', href: '/favicon.ico' }]
  }
}
```

Info

Cela nous donnera le même titre et la même description sur chaque page.

Configuration locale

Nous pouvons également ajouter des titres et des meta pour chaque page en utilisant la propriété `head` à l'intérieur de notre balise de script sur chaque page.

```
<script>
export default {
  head: {
    title: "Page d'accueil",
    meta: [
```

```

    {
      hid: 'description',
      name: 'description',
      content: "Description de la page d'accueil"
    }
  ],
}
</script>

```

Info

Utiliser `head` comme objet pour définir un titre et une description uniquement pour la page d'accueil.

```

<template>
  <h1>{{ title }}</h1>
</template>
<script>
  export default {
    data() {
      return {
        title: "Page d'accueil"
      }
    },
    head() {
      return {
        title: this.title,
        meta: [
          {
            hid: 'description',
            name: 'description',
            content: "Description de la page d'accueil"
          }
        ]
      }
    }
  }
</script>

```

Info

Utiliser `head` comme une fonction pour définir un titre et une description uniquement pour la page d'accueil. En utilisant une fonction, nous avons accès aux données (`data`) et aux propriétés calculées (`computed`).

Nuxt utilise `vue-meta` pour mettre à jour l'en-tête et les méta data de notre application.

Info

Pour éviter toute duplication lors de l'utilisation de composants enfants, veuillez donner un identifiant unique avec la clé `hid` à la méta description. De cette façon, `vue-meta` saura qu'il doit écraser la balise par défaut.

Next

Pour en savoir plus sur les options disponibles pour la propriété `head`, on peut consulter la documentation de [vue-meta](#).

Ressources externes

Nous pouvons inclure des ressources externes telles que des scripts et des polices de caractères en les ajoutant globalement au fichier `nuxt.config.js` ou localement dans l'objet ou la fonction `head`.

Info

Nous pouvons également passer à chaque ressource un `body : true` optionnel afin d'inclure la ressource avant la balise de fermeture `</body>`.

Configuration globale

```
export default {
  head: {
    script: [
      {
        src: 'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
      }
    ],
    link: [
      {
        rel: 'stylesheet',
        href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
      }
    ]
  }
}
```

Configuration locale

```
<template>
  <h1>Page avec jQuery et la police police de caractère Roboto</h1>
</template>

<script>
  export default {
    head() {
      return {
        script: [
          {
            src:
              'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
          }
        ],
        link: [
          {
            rel: 'stylesheet',
            href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
          }
        ]
      }
    }
  }
</script>

<style scoped>
  h1 {
```

```
    font-family: Roboto, sans-serif;  
}  
</style>
```

Conseils pour les ressources

Ajoutez des liens de `prefetch` et de `preload` pour accélérer le temps de chargement de la page initiale.

Nous voudrons peut-être désactiver cette option si nous avons beaucoup de pages et de routes.

Next

[Conseils pour les ressources](#).

Configuration

Par défaut, Nuxt est configuré pour couvrir la plupart des cas d'usage. Cette configuration par défaut peut être écrasée par un fichier `nuxt.config.js`.

La propriété `css`

Nuxt nous permet de définir globalement les fichiers/modules/librairies CSS que nous voulons (ceux qui seront inclus dans chaque page).

Info

Dans le cas où nous souhaitons utiliser du `SASS`, il faut vérifier que nous avons bien installé les paquets `node-sass` et `sass-loader`.

Dans notre fichier `nuxt.config.js`, il faut ajouter les ressources CSS:

```
export default {
  css: [
    // charge un module Node.js directement (ici c'est un fichier SASS)
    'bulma',
    // fichier CSS dans notre projet
    '~/assets/css/main.css',
    // fichier SCSS dans notre projet
    '~/assets/css/main.scss'
  ]
}
```

Info

Nuxt va automatiquement deviner le type de fichier par son extension et utiliser le loader du pré-processeur adéquat pour webpack. Nous aurons cependant besoin d'installer le loader requis si nous avons besoin de l'utiliser.

Extensions de Style

Nous pouvons omettre l'extension de nos fichiers CSS/SCSS/Postcss/Less/Stylus/... listés dans notre tableau `css`.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

Info

Si nous avons deux fichiers du même nom, par ex `main.scss` et `main.css` et si nous ne spécifions pas l'extension dans notre tableau `css` par ex `css: ['~/assets/css/main']` alors seulement un fichier sera chargé en fonction de l'ordre défini par `styleExtensions`. Dans ce cas, le fichier `css` sera chargé et le `scss` sera ignoré parce que l'extension `css` arrive en premier dans le tableau `styleExtension`.

Ordre par défaut: `['css', 'pcss', 'postcss', 'styl', 'stylus', 'scss', 'sass', 'less']`

Pré-processeurs

Grâce au [loader Vue](#), nous pouvons utiliser n'importe quel type de pré-processeur pour notre `<template>` ou `<style>`, il faut utiliser l'attribut `lang` pour cela.

Voici l'exemple d'un fichier `pages/index.vue` utilisant du [pug](#) et du [SASS](#):

```
<template lang="pug">
  h1.red Salut {{ name }} !
</template>

<style lang="scss">
  .red {
    color: red;
  }
</style>
```

Pour utiliser ces pré-processeurs, nous avons besoin d'installer leurs loaders Webpack:

```
yarn add --dev pug pug-plain-loader
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev pug pug-plain-loader
npm install --save-dev sass sass-loader@10
```

JSX

Nuxt utilise [@nuxt/babel-preset-app](#), qui est basé sur l'officiel [@vue/babel-preset-app](#) pour une configuration de Babel par défaut, afin que nous puissions utiliser du JSX dans nos composants.

Nous pouvons aussi utiliser du JSX dans la méthode `render` des composants:

```
export default {
  data () {
    return { name: 'Le monde' }
  },
  render (h) {
    return <h1 class="red">{this.name}</h1>
  }
}
```

Abréger `createElement` par `h` est une convention commune que nous pouvons voir dans l'écosystème Vue mais c'est en fait optionnel pour du JSX car ce dernier injecte automatiquement `const h = this.$createElement` dans n'importe quelle méthode et getter (pas les fonctions ni les fonctions fléchées) à condition qu'ils soient définis dans la syntaxe ES2015. Si c'est le cas, nous pouvons omettre le paramètre `h` dans le cas où nous utilisons du JSX.

Nous pouvons en apprendre davantage sur le fonction du JSX dans cette [section](#) de la documentation de Vue.

Ignorer des fichiers

.nuxtignore

Nous pouvons utiliser un fichier `.nuxtignore` pour dire à Nuxt d'ignorer des `layout`, `page`, `store` et/ou `middleware` à la racine de notre projet (`rootDir`) durant la phase de build. Le fichier `.nuxtignore` est sujet aux mêmes spécifications qu'un `.gitignore` ou un `.eslintignore`, dans le sens où chaque ligne est un schéma (glob pattern) indiquant quels fichiers devraient être ignorés.

```
# ignore le layout foo.vue
layouts/foo.vue

# ignore les fichiers layout dont le nom finit par -ignore.vue
layouts/*-ignore.vue

# ignore la page bar.vue
pages/bar.vue

# ignore les pages à l'intérieur du répertoire ignore
pages/ignore/*.vue

# ignore le store baz.js
store/baz.js

# ignore les fichiers du store qui correspondent avec _.test.-
store/ignore/_.test.-

# ignore les fichiers middleware à l'intérieur du répertoire foo sauf foo/bar.js
middleware/foo/*.js !middleware/foo/bar.js
```

La propriété ignorePrefix

N'importe quel fichier dans `pages/`, `layout/`, `middleware/` ou `store/` sera ignoré durant le build si son nom de fichier commence par le préfixe spécifié par `ignorePrefix`.

Par défaut, tous les fichiers qui commencent par un `-` seront ignorés, tels que `store/-foo.js` et `pages/-bar.vue`. Cela permet de laisser cohabiter des tests, des utilitaires et des composants avec les appels sans pour autant les convertir en des routes, des stores etc.

La propriété ignore

Plus configurable que `ignorePrefix`: tous les fichiers qui correspondent au schéma (glob pattern) seront ignorés durant le build.

```
export default {
  ignore: 'pages/bar.vue'
}
```

ignoreOptions

Derrière `.nuxtignore` se cache `node-ignore`, ainsi `ignoreOptions` peut être configuré avec les mêmes options que `node-ignore`.

```
export default {
  ignoreOptions: {
    ignorecase: false
  }
}
```

Personnaliser la configuration Webpack

Nous pouvons personnaliser la configuration webpack de Nuxt via l'option `extend` dans notre fichier `nuxt.config.js`, cette option est une méthode qui accepte deux arguments. Le premier argument est l'objet `config` de webpack exporté depuis la configuration webpack de Nuxt. Le second paramètre est un objet `context` contenant les propriétés booléennes suivantes: `{ isDev, isClient, isServer, loaders }`.

```
export default {
  build: {
    extend(config, { isDev, isClient }) {
      // ...
      config.module.rules.push({
        test: /\.(ttf|eot|svg|woff(2)?)(\?|[a-z0-9=&.]*)?$/,
        loader: 'file-loader'
      })
      if (isDev) {
        // asse Webpack en mode développement si `isDev` est vrai.
        config.mode = 'development'
      }
    }
  }
}
```

La méthode `extend` est appelée deux fois, une fois pour le bundle du client et une fois pour celui du serveur.

Personnaliser la configuration des fragments

Nous souhaiterions peut-être customiser un peu la [configuration d'optimisation](#), en évitant de ré-écrire l'objet par défaut.

```
export default {
  build: {
    extend(config, { isClient }) {
      if (isClient) {
        config.optimization.splitChunks.maxSize = 200000
      }
    }
  }
}
```

Exécuter ESLint à chaque build de webpack dans un environnement de développement

Pour être au courant d'erreurs de style de code, nous pourrions avoir envie d'exécuter [ESLint](#) à chaque build dans un environnement de développement.

```
export default {
  build: {
    extend(config, { isDev, isClient }) {
      if (isDev && isClient) {
        config.module.rules.push({
          enforce: 'pre',
          test: /\.js|vue$/,
          loader: 'eslint-loader',
          exclude: /(node_modules)/
        })
      }
    }
}
```

Éditer l'hôte et le port

Par défaut, l'environnement de développement du hôte de Nuxt est `localhost`, qui n'est accessible qu'au sein de la machine hôte. Afin d'admirer notre application sur un autre appareil nous aurons besoin de modifier l'hôte.

Le hôte `'0.0.0.0'` a pour but de dire à Nuxt de résoudre une adresse hôte, afin de lui permettre d'être accessible à d'autres appareils en *dehors* de la machine hôte (ex: LAN). Si le hôte est défini sur la chaîne de caractère `'0'` (pas `0`, qui lui est falsy) ou bien `'0.0.0.0'`, notre adresse IP locale sera assignée à notre application Nuxt.

```
export default {
  server: {
    host: '0' // par défaut: localhost
  }
}
```

Nous pouvons aussi changer le port par défaut.

```
export default {
  server: {
    port: 8000 // par défaut: 3000
  }
}
```

Info

Si le hôte est défini sur la chaîne de caractère `'0'` (pas `0`, qui lui est falsy), un port aléatoire sera assigné à notre application Nuxt.

Même si nous pouvons modifier cela dans le fichier `nuxt.config.js`, ce n'est pas conseillé car cela peut causer des soucis lors de l'hébergement de notre site. Il sera plutôt conseillé de modifier l'hôte et le port directement dans notre commande `dev`.

```
HOST=0 PORT=8000 npm run dev
```

ou créer un script dans notre `package.json`

```
"scripts": {
  "dev:host": "nuxt --hostname '0' --port 8000"
}
```

Configuration Asynchrone

Même s'il est conseillé d'utiliser la configuration normale `export default {}`, nous pouvons utiliser une configuration asynchrone en exportant une fonction asynchrone qui retourne l'objet de configuration.

```
import axios from 'axios'

export default async () => {
  const data = await axios.get('https://api.nuxtjs.dev/posts')
  return {
    head: {
      title: data.title
      //... le reste de la configuration
    }
}
```

Info

Le module `@nuxtjs/axios` ne peut pas être utilisé dans le fichier `nuxt.config.js`. Nous aurons besoin d'importer axios et de le configurer à nouveau.

Configuration avancée

Next

Le fichier `nuxt.config.js` a bien plus d'option de customisation et de configuration ! Se référer aux clés présentes dans le [glossaire de configuration](#).

Chargement

Nuxt fournit sa propre barre de chargement lors de la navigation entre les routes. Nous pouvons la personnaliser, la désactiver ou même créer la notre.

Personnaliser la barre de progression

Parmi les propriétés que nous pouvons modifier, on peut lister la couleur, la taille, la durée et la direction de la barre de progression, cela afin de correspondre aux besoins de notre application. Cela peut être fait en mettant à jour la propriété `loading` du fichier `nuxt.config.js` avec les propriétés correspondantes (`color`, `size`, `duration`, `direction`).

Par exemple, pour définir une barre de progression bleue d'une hauteur de 5 pixels, nous pouvons mettre à jour le fichier `nuxt.config.js` avec:

```
export default {
  loading: {
    color: 'blue',
    height: '5px'
  }
}
```

Voici la liste des propriétés que nous pouvons utiliser pour personnaliser la barre de progression.

Propriété	Type	Défaut	Description
<code>color</code>	String	'black'	Couleur CSS de la barre de progression
<code>failedColor</code>	String	'red'	Couleur CSS de la barre de progression lors d'une erreur lors du render d'une route (ex: de la data ou un fetch ont échoué)
<code>height</code>	String	'2px'	Hauteur de la barre de progression (utilisé dans la propriété <code>style</code> de la barre de progression)
<code>throttle</code>	Number	200	En millisecondes, le temps à attendre avant que la barre de navigation ne soit affichée. Utile pour éviter de créer des flashes.
<code>duration</code>	Number	5000	En millisecondes, la durée maximum de la barre de progression, Nuxt assume que la route sera render en moins de 5 secondes.
<code>continuous</code>	Boolean	false	Continuer d'animer la barre de progression lorsque le chargement prends plus de temps que la valeur de la propriété <code>duration</code> .
<code>css</code>	Boolean	true	Définir à <code>false</code> pour enlever tous les styles appliqués à la barre de progression (y compris les vôtres).
<code>rtl</code>	Boolean	false	Définit la direction de la barre de progression, de droite à gauche.

Désactiver la barre de progression

Si nous ne voulons pas afficher la barre de progression entre les routes, il suffit d'ajouter `loading: false` à notre fichier `nuxt.config.js`:

```
export default {
  loading: false
}
```

La propriété `loading` nous donne l'option de désactiver la barre de progression sur une page spécifique.

```
<template>
  <h1>Ma page</h1>
</template>

<script>
  export default {
    loading: false
  }
</script>
```

Lancer la barre de progression de manière programmatique

La barre de progression peut aussi être lancée de manière programmatique dans nos composants en appelant `this.$nuxt.$loading.start()` pour lancer le chargement et `this.$nuxt.$loading.finish()` pour le finir.

Durant le process de montage du composant page, la propriété `$loading` peut ne pas être immédiatement accessible. Pour remédier à cette problématique et nous permettre de commencer la progression dans la méthode `mounted`, il faut enrouler notre méthode `$loading` dans un `this.$nextTick` comme montré ci-dessous:

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Les rouages de la barre de progression

Malheureusement, il n'est pas possible pour le composant de chargement de connaître en avance le temps que va mettre une page pour charger. Ainsi, il n'est pas possible d'animer la barre de progression de manière adéquate pour correspondre au temps de chargement.

Le composant de chargement de Nuxt résout ce problème en nous laissant la possibilité de définir une durée (`duration`), nous permettant de définir une estimation de la durée de chargement que cela devrait prendre. À moins que nous n'utilisions un composant de chargement personnalisé, la barre de progression ira toujours de 0 à 100% de la durée (`duration`), peu importe la progression actuelle. Lorsque le chargement réel prend plus de temps que la durée définie, la barre de progression va rester à 100% jusqu'à ce que le chargement soit fini.

Nous pouvons changer le comportement par défaut en passant la propriété `continuous` à `true`. Ainsi, la barre de progression reviendra à 0% de nouveau, après avoir atteint les 100%. L'animation va se répéter (de 0 à 100%) jusqu'à ce que le chargement ne soit fini.

```
export default {
  loading: {
    continuous: true
  }
}
```

Exemple d'une barre de progression continue:



.../.../.../static/img/docs/api-continuous-loading.gif

Utiliser un composant de chargement personnalisé

Nous pouvons aussi créer notre propre composant que Nuxt se chargera d'appeler au lieu d'utiliser celui par défaut. Pour ce faire, nous aurons besoin de spécifier le chemin de du composant dans l'option `loading`.

Notre composant devra fournir certaines des méthodes suivantes:

Méthode	Requis	Description
<code>start()</code>	Requis	Appelé lors d'un changement de route, c'est à ce moment là que nous affichons notre composant.
<code>finish()</code>	Requis	Appelé lorsqu'une route est chargée (et la data collectée), c'est à ce moment là que nous cachons notre composant.
<code>fail(error)</code>	Optionnel	Appelé lorsqu'une de nos routes n'a pas pu être chargée (ex: le chargement de la data a échoué)
<code>increase(num)</code>	Optionnel	Appelé durant le chargement du composant de la route, <code>num</code> est un <code>Integer</code> < 100.

Nous pouvons créer notre composant personnalisé dans `components>LoadingBar.vue`:

```
<template>
  <div v-if="loading" class="loading-page">
    <p>Chargement...</p>
  </div>
</template>

<script>
  export default {
    data: () => ({
      loading: false
    }),
    methods: {
      start() {
        this.loading = true
      },
      finish() {
        this.loading = false
      }
    }
  }
</script>

<style scoped>
  .loading-page {
    position: fixed;
    top: 0;
    left: 0;
```

```

width: 100%;
height: 100%;
background: rgba(255, 255, 255, 0.8);
text-align: center;
padding-top: 200px;
font-size: 30px;
font-family: sans-serif;
}
</style>
```

Ensuite, il faut mettre à jour le fichier `nuxt.config.js` pour dire à Nuxt d'utiliser notre composant:

```

export default {
  loading: '~/components>LoadingBar.vue'
}
```

La propriété du tourniquet de chargement

Lorsque nous utilisons Nuxt en mode SPA, il n'y a pas de contenu provenant du serveur lors du premier chargement de la page. Ainsi, au lieu d'afficher une page vide pendant que la page charge, Nuxt nous permet d'utiliser un tourniquet que nous pouvons personnaliser afin d'ajouter nos propres couleurs, notre background et même changer le tourniquet lui-même.

```

export default {
  loadingIndicator: {
    name: 'circle',
    color: '#3B8070',
    background: 'white'
  }
}
```

Tourniquets intégrés

Ces tourniquets sont importés depuis le projet génial qu'est [Spinkit](#). Nous pouvons regarder la page de démonstration pour avoir un aperçu des tourniquets disponibles. Afin d'utiliser l'un des tourniquets, tout ce nous avons à faire est d'ajouter son nom à la propriété `name`. Pas besoin d'importer ou d'installer quoi que ce soit. Voici la liste de tous les tourniquets intégrés que nous pouvons utiliser:

- circle
- cube-grid
- fading-circle
- folding-cube
- chasing-dots
- nuxt
- pulse
- rectangle-bounce
- rotating-plane
- three-bounce
- wandering-cubes

Les tourniquets intégrés supportent les options `color` et `background`.

Tourniquets personnalisés

Si nous souhaitons avoir notre propre tourniquet, une chaîne de caractères ou un nom peuvent aussi être un chemin vers un template de tourniquet. Toutes les options seront ainsi passées au template.

Si nous voulons nous inspirer du tourniquet de base de Nuxt, nous pouvons retrouver le code [ici](#) !

Composants Nuxt

Nuxt arrive avec quelques composants importants, ceux-ci nous seront utiles pour construire notre application. Les composants sont disponibles de manière globale, ce qui veut dire que nous n'avons pas besoin de les importer pour les utiliser.

Dans les paragraphes suivants, chacun de ces composants sera expliqué.

Le composant Nuxt

Le composant `<Nuxt>` nous permet d'afficher les composants page. Basiquement, le composant est remplacé par les pages qui concordent avec la route montrée. Ainsi, il est important d'ajouter le composant `<Nuxt>` à nos layouts.

```
<template>
  <div>
    <div>Ma barre de navigation</div>
    <Nuxt />
    <div>Mon pied de page</div>
  </div>
</template>
```

Info

Le composant `<Nuxt>` ne peut être utilisé que dans les [layouts](#).

Le composant `<Nuxt>` peut prendre la propriété `nuxt-child-key`, cette dernière sera passée à `<RouterView>` afin que les transitions dans les pages dynamiques fonctionnent correctement.

Il y a 2 façons de gérer la propriété `key` de `<RouterView>` :

1. Utiliser la propriété `nuxt-child-key` sur notre composant `<Nuxt>`

```
<template>
  <div>
    <Nuxt :nuxt-child-key="uneKey" />
  </div>
</template>
```

2. Ajouter l'option `key` dans les composants *page* en tant que `string` ou `function`

```
export default {
  key(route) {
    return route fullPath
  }
}
```

Le composant NuxtChild

Ce composant est utilisé pour afficher les composants enfants dans une route imbriquée.

Exemple:

```
-| pages/
---| parent/
  | child.vue
---| parent.vue
```

Cette arborescence de fichiers va générer les chemins suivants :

```
; [
{
  path: '/parent',
  component: '~/pages/parent.vue',
  name: 'parent',
  children: [
    {
      path: 'child',
      component: '~/pages/parent/child.vue',
      name: 'parent-child'
    }
  ]
}]
```

Pour afficher le composant `child.vue`, il faut insérer le composant `<NuxtChild>` à l'intérieur de `pages/parent.vue`:

```
<template>
  <div>
    <h1>Je suis la view parent</h1>
    <NuxtChild :foobar="123" />
  </div>
</template>
```

keep-alive

Les composants `<Nuxt>` et `<NuxtChild/>` acceptent tous les deux `keep-alive` et `keep-alive-props`.

Info

Pour en apprendre davantage sur `keep-alive` et `keep-alive-props`, se référer à la documentation de [Vue](#).

```
<template>
  <div>
    <Nuxt keep-alive :keep-alive-props="{ exclude: ['modal'] }" />
  </div>
</template>

<!-- le code du haut sera converti dans le code suivant --&gt;
&lt;div&gt;</pre>

```

```
<keep-alive :exclude="['modal']">
  <RouterView />
</keep-alive>
</div>
```

```
<template>
  <div>
    <NuxtChild keep-alive :keep-alive-props="{ exclude: ['modal'] }" />
  </div>
</template>

<!-- le code du haut sera converti dans le code suivant --&gt;
&lt;div&gt;
  &lt;keep-alive :exclude="['modal']"&gt;
    &lt;RouterView /&gt;
  &lt;/keep-alive&gt;
&lt;/div&gt;</pre>

```

Les composants `<NuxtChild>` peuvent aussi recevoir des propriétés comme n'importe quel composant standard de Vue.

```
<template>
  <div>
    <NuxtChild :key="$route.params.id" />
  </div>
</template>
```

Pour voir un exemple, on peut regarder celui sur les [routes imbriquées](#).

Le composant NuxtLink

Pour naviguer entre les pages de notre application, il faut utiliser le composant `<NuxtLink>`. Ce composant est inclus dans Nuxt et nous n'avons donc pas besoin de l'importer comme un composant standard. Son fonctionnement est le même qu'une balise HTML `<a>` a l'exception qu'au lieu d'utiliser un `href="/about"` il faut utiliser `to="/about"`. Si nous avons déjà utilisé `vue-router` par le passé, nous pouvons voir `<NuxtLink>` comme un équivalent à `<RouterLink>`.

Un simple lien vers la page `index.vue` dans notre répertoire `pages` :

```
<template>
  <NuxtLink to="/">Page d'accueil</NuxtLink>
</template>
```

Le composant `<NuxtLink>` doit être utilisé pour tous les liens internes. Ce qui veut dire que tous les liens à l'intérieur de notre site doivent utiliser `<NuxtLink>`. La balise `<a>` doit être utilisée pour les liens externes (les sites web qui ne font pas partie de notre application).

```
<template>
  <div>
    <h1>Page d'accueil</h1>
    <nuxt-link to="/about">
      À propos (lien interne à l'application Nuxt)
    </nuxt-link>
  </div>
```

```
<a href="https://nuxtjs.org">Lien externe vers une autre page</a>
</div>
</template>
```

Info

Si nous voulons en savoir plus sur `RouterLink`, se référer à la documentation du Routeur Vue.

Info

`<NuxtLink>` utilise le [préfetching intelligent](#) par défaut.

prefetchLinks

Nuxt fait du prefetching intelligent, il détecte si un lien est visible (soit dans le viewport, soit lors d'un défilement) et il prefetch le JavaScript pour ces pages pour qu'elles soient prêtes quand l'utilisateur clique sur le lien. Nuxt charge la ressource seulement quand le navigateur n'est pas occupé. Le prefetching n'a pas non plus lieu si l'utilisateur est hors-ligne ou qu'il est en connexion 2G.

Désactiver le prefetching pour des liens spécifiques

Cependant, si notre page est lourde en JavaScript ou que nous avons beaucoup de pages différentes qui se retrouvent à charger de nombreux scripts tiers, nous pouvons désactiver le prefetching sur certains de ces liens. Pour cela, nous pouvons utiliser la propriété `no-prefetch`. Depuis Nuxt v2.10.0, nous pouvons aussi définir la propriété `prefetch` à `false`.

```
<NuxtLink to="/about" no-prefetch>Page à propos non pré-fetchée</NuxtLink>
<NuxtLink to="/about" :prefetch="false">Page à propos non pré-fetchée</NuxtLink>
```

Désactiver le prefetching de manière globale

Pour désactiver le prefetching sur tous les liens, il faut passer `prefetchLinks` à `false`:

```
export default {
  router: {
    prefetchLinks: false
  }
}
```

Depuis Nuxt v2.10.0, si nous avons passé `prefetchLinks` à `false` mais que nous souhaitons cependant prefetch un lien particulier, nous pouvons utiliser la propriété `prefetch`:

```
<NuxtLink to="/about" prefetch>Page à propos pré-fetchée</NuxtLink>
```

linkActiveClass

`linkActiveClass` marche de la même façon que la classe sur des liens actifs de `vue-router`. Si nous souhaitons montrer quels liens sont actifs, la seule chose que nous avons besoin de faire est de créer un peu de CSS pour la classe `nuxt-link-active`.

```
.nuxt-link-active {
  color: red;
}
```

Info

Ce CSS peut être ajouté au composant de navigation, dans une page spécifique, dans un layout ou dans le fichier `main.css`.

Si nous souhaitons changer le nom de cette classe par autre chose, nous pouvons modifier `linkActiveClass` dans la propriété `router` à l'intérieur du fichier `nuxt.config.js`.

```
export default {
  router: {
    linkActiveClass: 'mon-super-lien-actif'
  }
}
```

Info

Cette option est donnée directement à la propriété `linkActiveClass` du `vue-router`, plus d'informations [ici](#).

linkExactActiveClass

`linkExactActiveClass` marche exactement de la même façon que `linkActiveClass` dans le cas d'une correspondance exacte de la route.

```
.nuxt-link-exact-active {
  color: green;
}
```

La configuration dans le fichier `nuxt.config.js` se fait de la même façon.

```
export default {
  router: {
    linkExactActiveClass: 'mon-lien-exact-actif'
  }
}
```

linkPrefetchedClass

`linkPrefetchedClass` va nous permettre d'ajouter des styles pour tous les liens qui ont été prefetch. C'est pratique pour tester tous les liens qui ont été prefetch après avoir modifié un comportement. `linkPrefetchedClass` est désactivé par défaut. Si nous souhaitons l'activer, il faut ajouter la propriété dans le fichier `nuxt.config.js`.

```
export default {
  router: {
    linkPrefetchedClass: 'lien-de-nuxt-prefetched'
  }
}
```

Il suffit par la suite d'ajouter les styles pour cette classe.

```
.nuxt-link-prefetched {
  color: orangeRed;
}
```

Info

Comme nous avons pu le voir avec `lien-de-nuxt-prefetched`, le nom de la classe est totalement arbitraire.

Le composant côté-client seulement

L'usage de ce composant force un composant à être render uniquement du côté client. Pour importer un composant seulement sur le client, il faut l'encapsuler dans un `client-only`.

```
<template>
  <div>
    <sidebar />
    <client-only placeholder="Chargement...">
      <!-- ce composant sera render seulement du côté client --&gt;
      &lt;comments&gt;&lt;/comments&gt;
    &lt;/client-only&gt;
  &lt;/div&gt;
&lt;/template&gt;</pre>

```

Utiliser un élément de substitution jusqu'à ce que le contenu du `<client-only />` ne soit monté sur le client.

```
<template>
  <div>
    <sidebar />
    <client-only>
      <!-- ce composant sera render seulement du côté client --&gt;
      &lt;comments&gt;&lt;/comments&gt;

      <!-- indicateur de chargement, render du côté serveur --&gt;
      &lt;comments-placeholder slot="placeholder"&gt;&lt;/comments-placeholder&gt;
    &lt;/client-only&gt;
  &lt;/div&gt;
&lt;/template&gt;</pre>

```

Info

Depuis Nuxt > v2.9.0, il faut utiliser `<client-only>` au lieu de `<no-ssr>`.

[Go to TOC](#)

Activer la découverte automatique (auto-discovery)

Depuis la version `v2.13`, Nuxt est capable d'importer automatiquement les composants utilisés dans un template. Pour activer cette fonctionnalité, ajoutez `components: true` dans la configuration globale de votre projet Nuxt.

```
export default {
  components: true
}
```

Info

Pour en savoir plus, consultez [la documentation de configuration de la découverte automatique \(auto-discovery\)](#).

Utiliser les composants

Une fois les composants créés dans le dossier `/components`, ils seront directement accessibles depuis n'importe quelle partie de l'application. Ils sont utilisables sans avoir à les importer.

```
| components/
--| TheHeader.vue
--| TheFooter.vue
```

```
<template>
<div>
  <TheHeader />
  <Nuxt />
  <TheFooter />
</div>
</template>
```

Info

Plus d'exemples sur la [démonstration CodeSandbox](#) ou sur la [vidéo d'exemple](#).

Noms des composants

Si les composants sont imbriqués dans des dossiers, comme ici :

```
| components/
--| base/
----| foo/
| Button.vue
```

Le nom du composant sera déduit du nom du fichier dans lequel il se trouve et préfixé par le ou les noms des dossiers parents. Avec l'exemple plus haut, le nom du composant sera :

```
<BaseFooButton />
```

Pour plus de clarté, il est recommandé que le composant ait le même nom que le fichier dans lequel il se trouve. Dans l'exemple plus haut, il faut renommer `Button.vue` en `BaseFooButton.vue`.

Si vous voulez utiliser une structure de dossier personnalisée qui ne doit pas se refléter dans le nom du composant, il est possible de le spécifier explicitement dans la configuration globale du projet Nuxt :

```
| components/
--| base/
----| foo/
| Button.vue
```

```
components: {
  dirs: [
    '~/components',
    '~/components/base'
  ]
}
```

Avec cette configuration, le nom du composant sera `FooButton` plutôt que `BaseFooButton`.

```
<FooButton />
```

Info

Pensez à nommer vos composants en vous référant aux [conventions de Vue.js](#).

Imports dynamiques

Pour importer un composant de façon dynamique (chargement différé ou lazy-loading), il suffit d'ajouter le préfixe `Lazy` au nom du composant.

```
<template>
<div>
  <TheHeader />
  <Nuxt />
  <LazyTheFooter />
</div>
</template>
```

En utilisant le préfixe `Lazy`, le chargement du composant est différé en attendant le moment où il devient réellement nécessaire. C'est principalement utilisé pour optimiser la taille du bundle JavaScript.

```
<template>
<div>
  <h1>Mountains</h1>
  <LazyMountainsList v-if="show" />
  <button v-if="!show" @click="show = true">Show List</button>
</div>
</template>

<script>
export default {
  data() {
    return {
      show: false
    }
  }
}</script>
```

```
    }
}
}
</script>
```

```
:modal{src="img" alt="imgAlt"}
```

.nuxt

Le répertoire `.nuxt` est le répertoire de build. Il est dynamiquement généré et caché du système de fichier par défaut. À l'intérieur du répertoire nous pouvons trouver des fichiers générés automatiquement en utilisant la commande `nuxt dev` ou bien nos artefacts de construction lorsque nous utilisons `nuxt build`. La modification de ces fichiers est idéale pour le débogage mais rappelons-nous qu'il s'agit de fichiers générés et qu'une fois que nous exécuterons à nouveau la commande `dev` ou `build`, tout ce qui a été modifié ici sera écrasé.

Warning

Le répertoire `.nuxt` ne doit pas être versionné par le système de contrôle de version (ex: git) et doit être ignoré via le `.gitignore` car il sera généré automatiquement lors de l'exécution de `nuxt dev` ou de `nuxt build`.

La propriété buildDir:

Par défaut, de nombreux outils supposent que `.nuxt` est un répertoire caché car son nom commence par un point. Nous pouvons utiliser l'option `buildDir` dans `nuxt.config.js` pour éviter cela. Si nous changeons le nom n'oubliez pas d'ajouter le nouveau nom au fichier `.gitignore`.

```
export default {
  buildDir: 'nuxt-dist'
}
```

À l'intérieur du répertoire `.nuxt`:

- Le fichier `router.js` est le fichier de routeur généré que Nuxt génère pour nous lorsque nous plaçons les fichiers `.vue` dans le répertoire `pages`. Nous pouvons utiliser ce fichier pour le débogage lorsque nous souhaitons rechercher les routes générées par `vue-router` et connaître les noms d'une route spécifique.
- Le fichier `router.scrollBehavior.js` qui est notre `Router ScrollBehavior`
- Le répertoire `components` contient tous les composants Nuxt tels que `NuxtChild` et `NuxtLink`. Il contient également l'indicateur `nuxt-build-indicator` qui est la page que nous voyons lorsque notre application est en cours de construction et `nuxt-loading` qui est notre composant de chargement qui est vu lorsque nous attendons le chargement de la page. Nous trouverons également la page d'erreur par défaut de Nuxt.
- Le répertoire `mixins` contient les fichiers nécessaires à la méthode Nuxt `$ fetch`.
- Le répertoire `views` contient notre modèle d'application et la page d'erreur du serveur.
- Le fichier `app.js` est le principal fichier de l'application.
- Le fichier `client.js` est le fichier nécessaire pour que tout ce qui se passe bien côté client.
- Le fichier `vide` est volontairement laissé vide pour les *noop aliases*.
- Le fichier `index.js` amorce notre application.
- Le fichier `loading.html` est le fichier qui est utilisé lors du chargement de la page.

- Le fichier `middleware` est l'endroit où notre middleware est conservé.
- Le fichier `server.js` est tout le code exécuté sur le serveur.
- Les *utilities* contiennent les utilitaires dont Nuxt a besoin pour fonctionner.

Déploiement

Le répertoire `.nuxt` fait partie des fichiers nécessaires au déploiement de notre application SSR. Cela n'est pas nécessaire pour déployer notre application statique Nuxt car nous utilisons le répertoire `dist` pour cela.

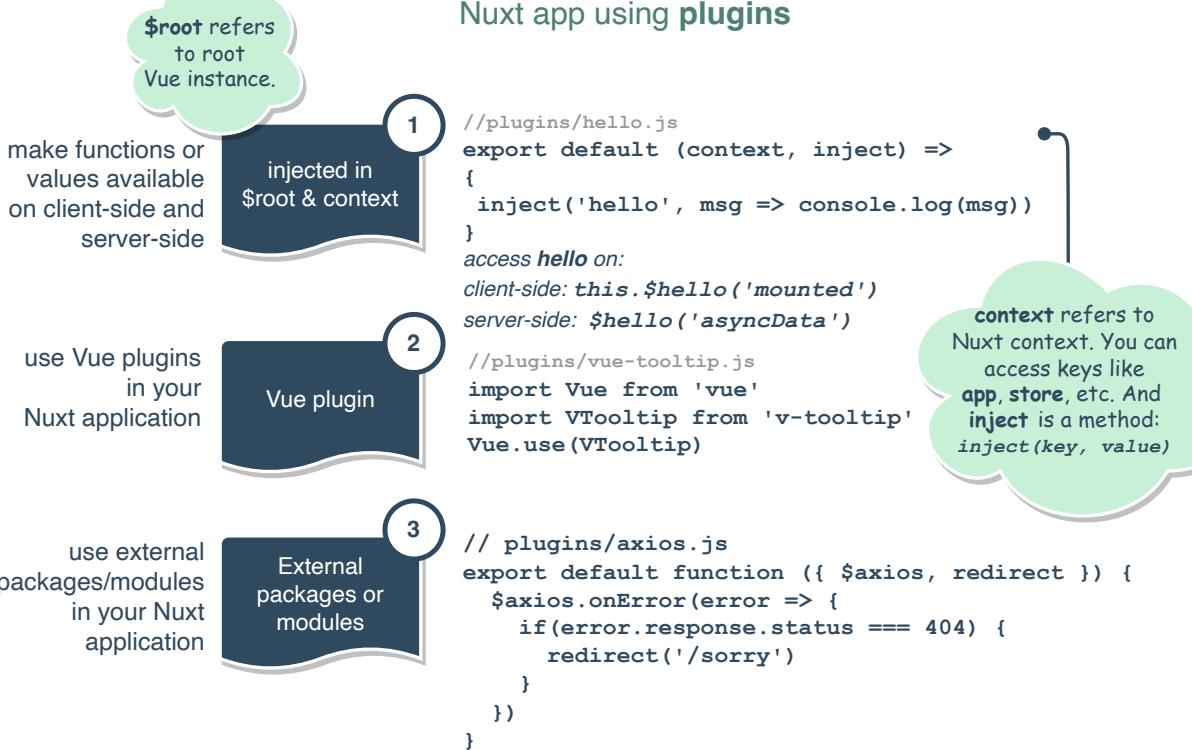
plugins

Le répertoire `plugins` contient les plugins JavaScript que l'on souhaite exécuter avant l'instanciation de l'application principale Vue.js.

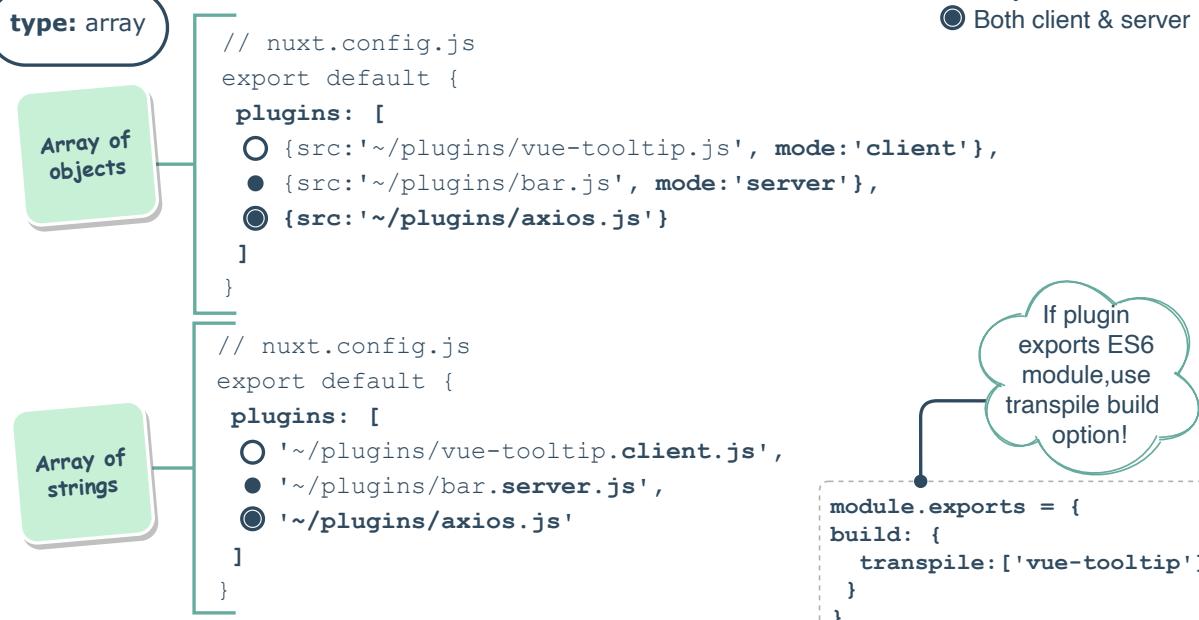
Nuxt Plugins

Nuxt v2.14.1
August 2020

3 ways to extend functionalities of Nuxt app using plugins



Include Nuxt plugin using `plugins` property



Le répertoire `plugins` contient les plugins JavaScript que l'on souhaite exécuter avant l'instanciation de l'application principale Vue.js. C'est ici que l'on peut ajouter des plugins Vue.js et injecter des fonctions/constantes. À chaque fois que l'on a besoin d'utiliser `Vue.use()`, on peut créer un fichier dans `plugins/` et ajouter son chemin à la propriété `plugins` dans le fichier `nuxt.config.js`.

Packages externes

On pourrait avoir envie d'utiliser des packages/modules externes dans notre application (ex: `axios`) pour faire des requêtes HTTP côté client et serveur.

Premièrement, l'installer via npm ou Yarn.

```
yarn add @nuxtjs/axios
```

```
npm install @nuxtjs/axios
```

On peut par exemple configurer les intercepteurs d'axios pour réagir à de possibles erreurs lorsque l'on fait des appels à notre API et ce, à travers toute notre application. Dans l'exemple suivant, on redirige l'utilisateur sur une page d'erreur personnalisée appelée `sorry` lorsque l'on obtient un code d'erreur 500 de la part de notre API.

```
export default function ({ $axios, redirect }) {
  $axios.onError(error => {
    if (error.response.status === 500) {
      redirect('/sorry')
    }
  })
}
```

Il faut par la suite ajouter le module et les plugins fraîchement créés dans la configuration du projet.

```
module.exports = {
  modules: ['@nuxtjs/axios'],
  plugins: ['~/plugins/axios.js']
}
```

Ensuite, on pourra l'utiliser directement dans nos composants page:

```
<template>
  <h1>{{ post.title }}</h1>
</template>

<script>
export default {
  async asyncData ({ $axios, params }) {
    const post = await
$axios.$get(`https://api.nuxtjs.dev/posts/${params.id}`)
    return { post }
  }
}
</script>
```

Info

Si une erreur *Cannot use import statement outside a module* apparaît, ajoutez votre package dans l'option `build > transpile` du fichier `nuxt.config.js` pour rendre votre plugin disponible au loader webpack.

```
build: {
  // Etendez votre configuration webpack ici
  transpile: ['npm-package-name'],
},
```

Plugins Vue

Si on veut utiliser des plugins Vue.js, comme par exemple `v-tooltip` pour afficher des tooltips dans notre application, nous avons besoin de configurer le plugin avant de lancer l'application.

Commençons par l'installer

```
yarn add v-tooltip
```

```
npm install v-tooltip
```

Ensuite, créons le fichier `plugins/vue-tooltip.js`

```
import Vue from 'vue'
import VTooltip from 'v-tooltip'

Vue.use(VTooltip)
```

La propriété `plugins`

Ensuite nous pouvons ajouter le chemin du fichier à l'intérieur de la propriété `plugins` du fichier `nuxt.config.js`. La propriété `plugins` permet d'ajouter facilement des plugins Vue.js à notre application principale. Tous les chemins définis dans la propriété `plugins` seront importés avant d'initialiser l'application principale.

```
export default {
  plugins: ['~/plugins/vue-tooltip.js']
}
```

Plugins ES6

Si le plugin est situé dans les `node_modules` et qu'il exporte un module en ES6, nous aurons besoin de l'ajouter à l'option de build `transpile`:

```
module.exports = {
  build: {
    transpile: ['vue-tooltip']
  }
}
```

Pour davantage d'informations sur les options, veuillez vous référer à la [configuration du build](#).

Côté client ou serveur seulement

Certains plugins ne devraient fonctionner que dans le navigateur car ils n'auront pas de support SSR (Server Side Rendering).

Convention pour le nommage des plugins

Si un plugin est voué à être exécuté seulement du côté client ou seulement du côté serveur, on peut appliquer une extension `.client.js` ou `.server.js` à l'extension du fichier du plugin. Le fichier sera automatiquement inclus du côté client ou serveur seulement (respectivement).

```
export default {
  plugins: [
    '~/plugins/foo.client.js', // seulement du côté client
    '~/plugins/bar.server.js', // seulement du côté serveur
    '~/plugins/baz.js' // sur le serveur et sur le client
  ]
}
```

Syntaxe objet

On peut aussi utiliser la syntaxe objet avec la propriété mode (`'client'` ou `'server'`) dans `plugins`.

```
export default {
  plugins: [
    { src: '~/plugins/both-sides.js' }, // sur le serveur et sur le client
    { src: '~/plugins/client-only.js', mode: 'client' }, // seulement du côté client
    { src: '~/plugins/server-only.js', mode: 'server' } // seulement du côté serveur
  ]
}
```

Injection dans le `$root` et le `context`

Parfois, on souhaiterait rendre des fonctions ou des variables accessibles dans l'intégralité de notre application. On peut injecter ces variables dans notre instance Vue (côté client), dans le `context` (côté serveur) et même dans le store Vuex. La convention pour le préfixe des fonctions que l'on a injecté est `$`.

Nuxt nous fournit une méthode `inject(clé, valeur)` pour faire cela facilement. Le second paramètre donné à `inject` est la fonction que l'on souhaite exporter. Le `$` sera automatiquement préfixé à notre clé.

Info

Il est important de savoir que dans tout `lifecycle` d'une instance Vue, seulement les hooks `beforeCreate` et `created` seront appellés du côté client et serveur. Les hooks restants seront appellés seulement du côté client.

```
export default ({ app }, inject) => {
  // Ceci injecte $hello(msg) dans Vue, dans le context et le store.
  inject('hello', msg => console.log(`Hello ${msg}!`))
}
```

```
export default {
  plugins: ['~/plugins/hello.js']
}
```

Maintenant le service `$hello` est accessible depuis le `context` et `this` dans les pages, les composants, les plugins, et les actions du store.

```
export default {
  mounted() {
    this.$hello('mounted')
    // Va console.log 'Hello mounted!'
  },
  asyncData({ app, $hello }) {
    $hello('asyncData')
    // Si la version de Nuxt <= 2.12, il faut utiliser ↗
    app.$hello('asyncData')
  }
}
```

```
export const state = () => ({
  someValue: ''
})

export const actions = {
  setSomeValueToWhatever({ commit }) {
    this.$hello('store action')
    const newValue = 'whatever'
    commit('changeSomeValue', newValue)
  }
}
```

Warning

Il faut bien faire attention à ne pas utiliser `Vue.use()`, `Vue.component()` ou quoi que ce soit d'autre lié à Vue à l'**intérieur** de cette fonction dédiée à l'injection de Nuxt. Sinon, cela causerait des fuites de mémoire côté serveur.

La propriété `extendPlugins`

On pourrait avoir envie de personnaliser les plugins ou changer l'ordre des plugins créé par Nuxt. Cette fonction accepte un tableau d'objets `plugins` et retourne la même chose, réarrangé.

Un exemple de changement de l'ordre des plugins:

```
export default {
  extendPlugins(plugins) {
    const pluginIndex = plugins.findIndex(
      ({ src }) => src === '~/plugins/shouldBeFirst.js'
    )
    const shouldBeFirstPlugin = plugins[pluginIndex]

    plugins.splice(pluginIndex, 1)
    plugins.unshift(shouldBeFirstPlugin)

    return plugins
  }
}
```

Mixins globaux

Des mixins globaux peuvent être facilement ajoutés aux plugins de Nuxt mais peuvent causer des soucis et des fuites de mémoire s'ils sont mal gérés. À chaque fois que l'on ajoute un mixin global à l'application, il faut bien faire attention à lui passer un flag pour éviter de l'enregistrer plusieurs fois:

```
import Vue from "vue"

// Assurez-vous de choisir un nom unique pour le drapeau
// il n'y aura donc pas de conflit avec aucun autre mixin.
if (!Vue.__my_mixin__) {
  Vue.__my_mixin__ = true
  Vue.mixin({ ... }) // On peut ensuite configurer le mixin
}
```

static

Le répertoire `static` est directement relié à la racine du serveur () et contient des fichiers qui ne seront probablement pas modifiés. Tous les fichiers inclus sont automatiquement servis par Nuxt et accessibles via l'URL racine de votre projet.

`/static/robots.txt` sera disponible à l'adresse `http://localhost:3000/robots.txt`

`/static/favicon.ico` sera disponible à l'adresse `http://localhost:3000/favicon.ico`

Cette option est utile pour les fichiers tels que `robots.txt`, `sitemap.xml` ou `CNAME` (qui est important pour le déploiement des pages sur GitHub).

Warning

Ce répertoire ne peut pas être renommé sans configuration supplémentaire.

Ressources statique

Si vous ne souhaitez pas utiliser les ressources Webpack du répertoire `assets`, vous pouvez ajouter les images dans le répertoire `static`.

Dans votre code, vous pouvez alors référencer ces fichiers par rapport à la racine (`/`) :

```
<!-- Image statique du répertoire `static` -->


<!-- Image webpacked du répertoire `assets` -->

```

Info

Nuxt ne change pas ce chemin, si vous personnalisez votre `router.base` vous devrez alors l'ajouter manuellement à vos chemins. Par exemple:

```

```

Configuration du répertoire `static`

Si vous en avez besoin, vous pouvez configurer le comportement du répertoire `static/` dans le fichier `nuxt.config.js`.

Préfixe des ressources statique

Si nous déployons Nuxt dans un sous-répertoire, par exemple `/blog/`, la base du routeur sera ajoutée par défaut au chemin de la ressource statique. Si nous voulons désactiver ce comportement, vous pouvez mettre `static.prefix` à `false` dans le `nuxt.config.js`.

```
export default {  
  static: {  
    prefix: false  
  }  
}
```

Par défaut : `/blog/mon-image.png`

Avec `static.prefix` désactivé : `/mon-image.png`

[Go to TOC](#)

store

Le répertoire `store` contient vos fichiers Store pour [Vuex](#). Le Store Vuex est livré avec Nuxt mais désactivé par défaut. La création d'un fichier `index.js` dans ce répertoire active le store.

Warning

Ce répertoire ne peut pas être renommé sans configuration supplémentaire.

L'utilisation du store pour gérer l'état est importante pour toute grande application. C'est pourquoi Nuxt implémente Vuex dans son noyau.

Activer le Store

Nuxt recherchera le répertoire `store`, s'il existe, il fera :

1. L'importation de Vuex
2. L'ajout de l'option `store` à la racine de l'instance Vue.

Modules

Chaque fichier `.js` dans le répertoire `store` est transformé en [namespaced module](#) (`index` étant le module racine). L'état `state` doit toujours être une fonction pour éviter les états *partagés* non désirés du côté du serveur.

Pour commencer, exportez l'état `state` en tant que fonction, et les mutations et actions en tant qu'objets.

```
export const state = () => ({
  counter: 0
})

export const mutations = {
  increment(state) {
    state.counter++
  }
}
```

Ensuite, vous pouvez avoir un fichier `store/todos.js` :

```
export const state = () => ({
  list: []
})

export const mutations = {
  add(state, text) {
    state.list.push({
      text,
      done: false
    })
  }
},
```

```

remove(state, { todo }) {
  state.list.splice(state.list.indexOf(todo), 1)
},
toggle(todo) {
  todo.done = !todo.done
}
}

```

Le store sera créé en tant que tel :

```

new Vuex.Store({
  state: () => ({
    counter: 0
  }),
  mutations: {
    increment(state) {
      state.counter++
    }
  },
  modules: {
    todos: {
      namespaced: true,
      state: () => ({
        list: []
      }),
      mutations: {
        add(state, { text }) {
          state.list.push({
            text,
            done: false
          })
        },
        remove(state, { todo }) {
          state.list.splice(state.list.indexOf(todo), 1)
        },
        toggle(state, { todo }) {
          todo.done = !todo.done
        }
      }
    }
  }
})

```

Et dans votre `pages/todos.vue`, en utilisant le module `todos` :

```

<template>
  <ul>
    <li v-for="todo in todos" :key="todo.text">
      <input :checked="todo.done" @change="toggle(todo)" type="checkbox">
      <span :class="{ done: todo.done }>{{ todo.text }}</span>
    </li>
    <li><input @keyup.enter="addTodo" placeholder="Que faut-il faire ?"></li>
  </ul>
</template>

<script>
import { mapMutations } from 'vuex'

export default {
  computed: {

```

```

    todos () {
      return this.$store.state.todos.list
    }
  },
  methods: {
    addTodo (e) {
      this.$store.commit('todos/add', e.target.value)
      e.target.value = ''
    },
    ...mapMutations({
      toggle: 'todos/toggle'
    })
  }
}
</script>

<style>
.done {
  text-decoration: line-through;
}
</style>

```

La méthode des modules fonctionne également pour les définitions de haut niveau sans implémenter un sous-répertoire dans le répertoire `store`.

Exemple pour l'état `state` : nous créons un fichier `store/state.js` et nous ajoutons ce qui suit.

```

export default () => ({
  counter: 0
})

```

Et les mutations correspondantes peuvent se trouver dans le fichier `store/mutations.js`.

```

export default {
  increment(state) {
    state.counter++
  }
}

```

Exemple de structure de dossier

La structure des fichiers/dossiers de configuration du store pourrait ressembler à ceci :

```

store/
--| index.js
--| ui.js
--| shop/
----| cart/
| actions.js
| getters.js
| mutations.js
| state.js
----| products/
| mutations.js
| state.js
| itemsGroup1/
| state.js

```

Plugins dans le Store

Vous pouvez ajouter des plugins supplémentaires au store en les plaçant dans le fichier `store/index.js` :

```
import monPlugin from 'monPlugin'

export const plugins = [monPlugin]

export const state = () => ({
  counter: 0
})

export const mutations = {
  increment(state) {
    state.counter++
  }
}
```

Plus d'informations sur les plugins : [Documentation Vuex](#).

L'action nuxtServerInit

Si l'action `nuxtServerInit` est définie dans le store et que le mode est `universal`, alors Nuxt l'appellera avec le contexte (uniquement du côté du serveur). C'est utile lorsque nous avons des données sur le serveur à passer directement au côté client.

Par exemple, disons que nous avons des sessions côté serveur et que nous pouvons accéder à l'utilisateur connecté par l'intermédiaire de `req.session.user`. Pour donner l'utilisateur authentifié à notre store, nous mettons à jour notre `store/index.js` comme suit :

```
actions: {
  nuxtServerInit ({ commit }, { req }) {
    if (req.session.user) {
      commit('user', req.session.user)
    }
  }
}
```

Warning

Seul le module principal (dans `store/index.js`) recevra cette action. Vous devrez enchaîner vos actions de module à partir de là.

Le `contexte` est donné à `nuxtServerInit` comme 2ème argument dans la méthode `asyncData`.

Si `nuxt generate` est lancé, `nuxtServerInit` sera exécuté pour chaque route dynamique générée.

Info

Les actions asynchrones `nuxtServerInit` doivent retourner une Promesse ou utiliser `async/await` pour permettre au serveur nuxt de les attendre.

```
actions: {  
  async nuxtServerInit({ dispatch }) {  
    await dispatch('core/load')  
  }  
}
```

Vuex Strict Mode

Le mode strict est activé par défaut en mode développement et désactivé en mode production. Pour désactiver le mode strict en dev, suivez l'exemple ci-dessous dans `store/index.js` :

```
export const strict = false
```

nuxt.config

Par défaut, Nuxt est configuré de sorte à pouvoir couvrir la plupart des cas. Cette configuration peut être écrasée grâce au fichier nuxt.config.js.

nuxt.config.js

alias

Cette option vous permet de définir des alias qui seront disponibles dans votre JavaScript et CSS.

```
import { resolve } from 'path'

export default {
  alias: {
    'style': resolve(__dirname, './assets/style')
  }
}
```

Next

See more on the [alias property](#)

build

Cette option permet de configurer différents paramètres lors de l'étape de build, cela inclus les `loaders`, les `noms de fichiers`, la `configuration Webpack` ainsi que la `transpilation`.

```
export default {
  build: {
    /*
      ** On peut personnaliser la configuration de Webpack ici
    */
    extend(config, ctx) {}
  }
}
```

Next

Se référer à la propriété [build](#).

CSS

Cette option permet de définir les fichiers CSS, les modules et les librairies que l'on veut inclure globalement (sur chaque page).

```
export default {
  css: ['~/assets/css/main.css', '~/assets/css/animations.scss']
}
```

On peut omettre l'extension pour les fichiers CSS/SCSS/Postcss/Less/Stylus/... listés dans le tableau css du fichier `nuxt.config.js`.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

En omettant l'extension, cela nous permet de ne pas avoir à renommer le fichier si jamais on décide de le passer du CSS au SASS.

Next

Se référer à la propriété `css`.

dev

Cette option permet de définir le mode `development` ou `production` de Nuxt (utile lorsque l'on veut utiliser Nuxt de façon programmatique).

```
export default {
  dev: process.env.NODE_ENV !== 'production'
}
```

Next

Se référer à la propriété `dev`.

env

Cette option permet de définir des variables d'environnement qui seront disponibles côté client mais aussi serveur.

```
export default {
  env: {
    baseUrl: process.env.BASE_URL
  }
}
```

runtimeConfig

La configuration d'exécution a une prise en charge intégrée de `dotenv` pour une meilleure sécurité et un développement plus rapide. La configuration d'exécution est ajoutée au payload Nuxt, il n'est donc pas nécessaire de reconstruire afin de mettre à jour la configuration d'exécution lorsque vous travaillez en développement ou avec un rendu côté serveur ou des applications côté client uniquement. (Pour les sites statiques, vous devrez toujours régénérer votre site pour voir les changements.)

.env support

Si vous avez un fichier `.env` dans le répertoire racine de votre projet, il sera automatiquement chargé dans `process.env` et accessible dans votre `nuxt.config/serverMiddleware` et tout autre fichier qu'ils importent.

Vous pouvez personnaliser le chemin en utilisant `--dotenv <file>` ou désactiver complètement avec `--dotenv false`. Par exemple, vous pouvez spécifier un fichier ".env" différent dans les environnements de production, de transfert ou de développement.

`publicRuntimeConfig`

- doit contenir toutes les variables env qui sont publiques car elles seront exposées sur le frontend. Cela pourrait inclure une référence à votre URL publique par exemple.
- est disponible en utilisant `$config` à la fois sur le serveur et le client.

```
export default {
  publicRuntimeConfig: {
    baseURL: process.env.BASE_URL || 'https://nuxtjs.org'
  }
}
```

`privateRuntimeConfig`

- doit contenir toutes les variables d'environnement qui sont privées et qui ne doivent pas être exposées sur le frontend. Cela pourrait inclure une référence à vos jetons secrets API par exemple.
- n'est disponible que sur le serveur utilisant le même `$config` (il remplace `publicRuntimeConfig`)

```
export default {
  privateRuntimeConfig: {
    apiSecret: process.env.API_SECRET
  }
}
```

Utilisation de vos valeurs de configuration:

Vous pouvez ensuite accéder à ces valeurs n'importe où en utilisant le contexte dans vos pages, magasin, composants et plugins en utilisant `this.$config` ou `context.$config`.

```
<script>
  asyncData ({ $config: { baseURL } }) {
    const posts = await fetch(`#${baseURL}/posts`)
      .then(res => res.json())
  }
</script>
```

Dans vos modèles, vous pouvez accéder à vos runtimeConfigs directement en utilisant `$config.*`

```
<template>
  <p>Votre url est: {{ $config.baseURL}}</p>
</template>
```

Warning

Votre configuration privée peut être exposée si vous utilisez `$config` en dehors d'un contexte de serveur uniquement (par exemple, si vous utilisez `$config` dans `fetch`, `asyncData` ou directement dans votre modèle).

Next

Voir plus sur le [runtimeConfig](#)

Next

Voir notre article de blog sur [Moving from @nuxtjs/dotenv to runtime config](#)

Next

Voir plus sur le [env property](#)

generate

Cette option permet de définir les paramètres pour chaque route dynamique de l'application, ces derniers seront transformés en fichiers HTML par Nuxt.

```
export default {
  generate: {
    dir: 'gh_pages', // gh_pages/ au lieu de dist/
    subFolders: false // les fichiers HTML sont générés en fonction du chemin de
    la route
  }
}
```

Next

Se référer à la propriété [generate](#).

head

```
export default {
  head: {
    title: 'mon titre',
    meta: [
      { charset: 'utf-8' },
      ....
    ]
  }
}
```

Cette option permet de définir toutes les balises méta par défaut pour l'application.

Next

Se référer à la propriété [head](#).

loading

Cette option permet de personnaliser le composant de chargement que Nuxt utilise par défaut.

```
export default {
  loading: {
    color: '#fff'
  }
}
```

Next

Se référer à l'intégration du [chargement](#).

modules

Cette option permet d'ajouter des modules Nuxt au projet.

```
export default {
  modules: ['@nuxtjs/axios']
}
```

Next

Se référer à la propriété [modules](#).

modulesDir

Cette propriété est utilisée pour définir les répertoires des modules afin de déterminer le chemin de résolution. Par ex: le `resolveLoading`, `nodeExternals` et `postcss` de Webpack. Le chemin de la configuration est relatif par rapport à `options.rootDir` (par défaut: `process.cwd()`).

```
export default {
  modulesDir: ['../../node_modules']
}
```

Adapter cet emplacement peut être nécessaire si le projet est organisé en tant que workspace Yarn avec un mono-repo.

Next

Se référer à la propriété [modulesDir](#)

plugins

Cette option permet de définir les plugins JavaScript qui devraient s'exécuter avant l'instanciation de l'application Vue.js principale.

```
export default {
  plugins: ['~/plugins/url-helpers.js']
}
```

Next

Se référer à la propriété [plugins](#).

router

Avec l'option `router`, on peut écraser la configuration de Nuxt par défaut pour le routeur de Vue.js.

```
export default {
  router: {
    linkExactActiveClass: 'text-primary'
  }
}
```

Next

Se référer à la propriété [router](#).

server

Cette option permet de configurer des variables de connexion pour l'instance du serveur de l'application Nuxt.

```
import path from 'path'
import fs from 'fs'

export default {
  server: {
    https: {
      key: fs.readFileSync(path.resolve(__dirname, 'server.key')),
      cert: fs.readFileSync(path.resolve(__dirname, 'server.crt'))
    }
  }
}
```

Next

Se référer à la propriété [server](#).

srcDir

Cette option permet de définir le répertoire source de l'application Nuxt.

```
export default {
  srcDir: 'client/'
}
```

Exemple de la structure du projet de l'application Nuxt avec le répertoire `client` :

```
**- app/
--- node_modules/
--- nuxt.config.js
--- package.json
--- client/
| assets/
| components/
| layouts/
| middleware/
| pages/
```

```
| plugins/
| static/
| store/**
```

dir

Cette option permet de définir des noms personnalisés pour les répertoires de l'application Nuxt.

```
export default {
  pages: 'views' // Nuxt va chercher les pages dans le répertoire views/ et non
  pages/
}
```

Next

Se référer à la propriété [dir](#)

pageTransition

Cette option permet de définir les propriétés par défaut pour les transitions de page.

```
export default {
  pageTransition: 'page'
}
```

Next

Se référer à la propriété [transition](#)

Autres fichiers de configuration

Outre le fichier `nuxt.config.js`, il peut y avoir d'autres fichiers de configuration à la racine du projet, tels que `.eslintrc`, `prettier.config.json` ou `.gitignore`. Ces derniers sont utilisés pour configurer des outils tels que le linter, le formateur de code ou votre repository Git, ils sont donc décorrélés du fichier `nuxt.config.js`.

.gitignore

Dans le fichier `.gitignore`, il faudra ajouter les entrées qui suivent afin qu'elles soient ignorées du système de contrôle de version. `node_modules` où tous les modules sont installés. Le répertoire `nuxt` qui est créé lorsque l'on lance les commandes `dev` ou `build`. Le répertoire `dist` qui est créé lorsque l'on lance la commande `generate`.

```
node_modules .nuxt dist
```

Et maintenant ?

Next

Se référer au [glossaire de configuration](#)

[Go to TOC](#)

assets

Le répertoire `assets` contient les ressources comme des fichiers Stylus, du SASS, des images ou des polices de caractères.

Images:

À l'intérieur de nos templates `vue`, si nous avons besoin de faire un lien vers notre répertoire `assets` (ressources), il faut utiliser `~/assets/notre_image.png` (avec un slash avant les `assets`).

```
<template>
  
</template>
```

À l'intérieur de nos fichiers `CSS`, si nous avons besoin de faire un lien vers notre répertoire `assets` (ressources), il faut utiliser `~/assets/banniere.svg` (sans le slash).

```
background: url('~/assets/banniere.svg');
```

Si l'on travaille avec des images dynamiques, on aura besoin d'utiliser `require`.

```

```

Next

Voir les [ressources webpack](#).

Styles:

Nuxt nous permet de définir les fichiers/modules/librairies CSS que nous souhaitons utiliser de manière globale (dans chaque page). Dans le fichier de config `nuxt.config.js`, on peut spécifier cela grâce à la propriété `css` :

```
export default {
  css: [
    // charge un module Node.js directement (ici un fichier SASS)
    'bulma',
    // un fichier CSS dans le projet
    '~/assets/css/main.css',
    // un fichier SCSS dans le projet
    '~/assets/css/main.scss'
  ]
}
```

Sass

Info

Dans le cas où l'on veut utiliser `SASS`, il faut bien faire attention à installer les packages `node-sass` et `sass-loader`.

```
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev sass sass-loader@10
```

Nuxt va automatiquement deviner le type de fichier grâce à son extension et utiliser le loader webpack approprié pour le pré-processeur. Nous aurons cependant besoin d'installer le loader requis si nous avons besoin de l'utiliser.

Fonts:

On peut utiliser des polices de caractères locales en les ajoutant dans le répertoire des ressources. Une fois ajoutées, on peut y avoir accès dans notre CSS en utilisant `@font-face`.

```
-l assets
---l fonts
| DMSans-Regular.ttf
| DMSans-Bold.ttf
```

```
@font-face {
  font-family: 'DM Sans';
  font-style: normal;
  font-weight: 400;
  font-display: swap;
  src: url('~assets/fonts/DMSans-Regular.ttf') format('truetype');
}

@font-face {
  font-family: 'DM Sans';
  font-style: normal;
  font-weight: 700;
  font-display: swap;
  src: url('~assets/fonts/DMSans-Bold.ttf') format('truetype');
}
```

Info

Les fichiers CSS ne sont pas chargés automatiquement. Ajoutez-les en utilisant la [propriété css](#).

Next

Pour ajouter des polices de caractères externes telles que des Google Fonts, se référer le chapitre sur le [SEO et les métatags](#).

Ressources de Webpack

Par défaut, Nuxt utilise les `vue-loader`, `file-loader` et `url-loader` de Webpack pour délivrer les ressources. On peut aussi utiliser le répertoire statique pour les ressources, ce dernier ne doit cependant pas passer par Webpack.

Webpack

Le compilateur de templates de Vue s'occupe de processer le style (avec `css-loader`) et les fichiers de templates (grâce à `vue-loader`). Durant ce processus, toutes les URL des ressources telles que ``, `background: url(...)` et les `@import` CSS sont résolues en tant que dépendances de modules.

Par exemple, avec cette arborescence de fichiers:

```
-l assets/
----l image.png
-l pages/
----l index.vue
```

Si dans notre CSS, on utilise `url('~assets/image.png')`, ceci sera traduit en `require('~assets/image.png')`.

Warning

L'alias `~/` ne sera pas résolu correctement dans les fichiers CSS. Il faut bien utiliser `~assets` (**sans le slash**) dans les références CSS `url`, ex: `background: url(~assets/banniere.svg")`.

Si on référence cette image dans notre fichier `pages/index.vue`:

```
<template>
  
</template>
```

Ce sera compilé en:

```
createElement('img', { attrs: { src: require '~/assets/image.png' } })
```

Parce que `.png` n'est pas un fichier JavaScript, Nuxt s'occupe tout seul de demander à Webpack d'utiliser `file-loader` et `url-loader` pour s'occuper de ce format.

Les bénéfices de ces loaders sont les suivants:

`file-loader` s'occupe de définir où copier et placer la ressource, ainsi que comment la nommer en utilisant des hash pour une meilleure mise en cache. En production, on pourra bénéficier d'une mise en cache sur le long terme par défaut !

`url-loader` permet de transformer un fichier en une seule ligne de data URL encodé en base-64 s'il est en dessous d'un certain seuil. Ceci permet de réduire le nombre de requêtes HTTP pour des fichiers anodins. Si la taille du fichier est plus grosse que le seuil fixé, il va utiliser `file-loader` en solution de repli.

Pour ces deux loaders, la configuration par défaut est la suivante:

```
// https://github.com/nuxt/nuxt.js/blob/dev/packages/webpack/src/config/base.js#L382-L411
{
  test: /\.(png|jpe?g|gif|svg|webp|avif)$/i,
  use: [
    {
      loader: 'url-loader',
      options: {
        esModule: false,
        limit: 1000, // 1kB
        name: 'img/[name].[contenthash:7].[ext]'
      }
    }
  ],
  test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/i,
  use: [
    {
      loader: 'url-loader',
      options: {
        esModule: false,
        limit: 1000, // 1kB
        name: 'fonts/[name].[contenthash:7].[ext]'
      }
    }
  ],
  test: /\.(webm|mp4|ogv)$/i,
  use: [
    {
      loader: 'file-loader',
      options: {
        esModule: false,
        name: 'videos/[name].[contenthash:7].[ext]'
      }
    }
  ]
}
```

Ce qui veut dire que n'importe quel fichier en dessous de 1 Ko sera mis sur une seule ligne de data URL encodé en base-64. Autrement, l'image/police de caractères sera copiée dans le répertoire correspondant (à l'intérieur du répertoire `.nuxt`) avec un nom comprenant le hash d'une version pour une meilleure mise en cache.

Lorsqu'on lancera l'application `nuxt` avec un template tel que `pages/index.vue`:

```
<template>
  
</template>
```

Cela sera transformé en:

```

```

Si on souhaite changer la configuration d'un des loaders, on trouvera plus de détails à la page de `build.extend`.

Les Alias

Par défaut le répertoire des sources (`srcDir`) et le répertoire racine (`rootDir`) sont les mêmes. On peut utiliser l'alias `~` pour le répertoire des sources. Au lieu d'écrire des chemins relatifs tels que `../assets/notre_image.png`, on peut écrire `~/assets/notre_image.png` à la place.

Les deux donneront le même résultat.

```
<template>
  <div>
    
    
  </div>
</template>
```

Nous recommandons l'usage de `~` en tant qu'alias. `@` est toujours supporté mais il ne marchera pas dans certains cas tel que les images de background dans le CSS.

On peut utiliser `~~` ou `@@` pour le répertoire racine.

Info

Astuce: Pour avoir accès à `~` sur un clavier espagnol, il faut utiliser `Option + n` sur Mac OS ou (`Alt gr + 4`) sur Windows.

components

Le répertoire des composants contient les composants Vue.js. Les composants sont les éléments qui constituent les différentes parties d'une page et peuvent être réutilisés et importés dans des pages, layouts ou même d'autres composants.

Récupérer de la data

Pour récupérer de la data depuis une API de manière asynchrone dans les composants, on peut utiliser le hook `fetch()` de Nuxt.

En utilisant `$fetchState.pending`, on peut afficher un message pendant le chargement de la data. En utilisant `$fetchState.error`, on peut afficher un message d'erreur s'il y a une erreur lors de la récupération des données. Lorsque l'on utilise `fetch`, il faut définir la data dans la propriété `data`, celle-ci sera par la suite remplie une fois que les données auront été récoltées.

```
<template>
<div>
  <p v-if="$fetchState.pending">Chargement....</p>
  <p v-else-if="$fetchState.error">
    Erreur lors de la récupération des montagnes
  </p>
  <ul v-else>
    <li v-for="(mountain, index) in mountains" :key="index">
      {{ mountain.title }}
    </li>
  </ul>
</div>
</template>
<script>
  export default {
    data() {
      return {
        mountains: []
      }
    },
    async fetch() {
      this.mountains = await fetch(
        'https://api.nuxtjs.dev/mountains'
      ).then(res => res.json())
    }
  }
</script>
```

Next

Se référer au chapitre sur [fetch\(\)](#) pour avoir davantage d'informations sur son fonctionnement.

Découverte des composants

:prose-img{src="../../static/img/docs/components.png"}

À partir de la version `2.13`, Nuxt peut automatiquement importer les composants lorsqu'ils sont utilisés dans les templates. Pour activer cette feature, il suffit de définir `components: true` dans le fichier de configuration:

```
export default {
  components: true
}
```

Ainsi, lors de la création de composants dans le répertoire `components`, ils seront importés et directement disponibles.

```
components/
  TheHeader.vue
  TheFooter.vue
```

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <TheFooter />
  </div>
</template>
```

Imports dynamiques

Pour importer dynamiquement un composant - le "lazy load" (le charger paresseusement) - il suffit juste de le préfixer par un `Lazy` dans les templates.

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <LazyTheFooter />
  </div>
</template>
```

Utiliser le préfixe `Lazy` permet aussi d'importer dynamiquement un composant lorsqu'un évènement est déclenché.

```
<template>
  <div>
    <h1>Mountains</h1>
    <LazyMountainsList v-if="show" />
    <button v-if="!show" @click="show = true">Show List</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      show: false
    }
  }
}</script>
```

Réertoires imbriqués

Si on a des composants dans des réertoires imbriqués comme ceci:

```
components/  
  base/  
    foo/  
      CustomButton.vue
```

Nous recommandons d'utiliser le nom du réertoire dans le nom du fichier pour plus de clarté.

```
<BaseFooCustomButton />
```

Si nous voulons le déclarer comme `<CustomButton />` tout en gardant la structure de réertoire, nous pouvons ajouter le réertoire de `CustomButton.vue` dans `nuxt.config.js`.

```
components: {  
  dirs: [  
    '~/components',  
    '~/components/base/foo'  
  ]  
}
```

Et maintenant nous pouvons utiliser `<CustomButton />` au lieu de `<BaseFooCustomButton />`.

```
<CustomButton />
```

Next

Voir [la propriété components](#) pour d'autres méthodes de contrôle du nom des composants.

Next

Pour en apprendre davantage, se référer au chapitre sur le [module des composants Nuxt](#).

[Go to TOC](#)

content

Augmentez le potentiel de votre application Nuxt avec le module `@nuxt/content` grâce auquel on peut écrire dans un répertoire `content/` et récupérer du Markdown, JSON, YAML et des fichiers CSV à travers une API dans le style de MongoDB, qui remplit donc le rôle d'un **CMS headless basé sur Git**.

Nuxt Content Module

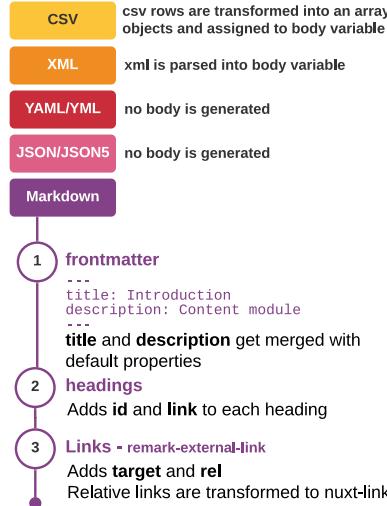
1. WRITE

Getting started:

- Install module `npm install @nuxt/content`
- Create `content/` directory in your Nuxt project.

Module will parse `.md`, `.yaml`, `.yml`, `.csv`, `.json`, `.json5`, `.xml` and generate following default properties:

- dir - extension (`.md`)
- path - `createdAt`
- slug - `updatedAt`



1. Write continues...

- Footnotes** - remark-footnote: Add footnotes like, `[^1]`, then define that footnote like, `[^1]: This is my first footnote`
- Codeblocks**: Add codeblocks
- Syntax highlight** - Prism Js: data line, file name
`...js{1, 3-5}[server.js]`
- HTML**: Add html inside markdown file
- Vue Components**: Add Vue component inside markdown file
✓ `<my-component>` - only kebab case
✗ `<my-component/>` - no self-closing
- Global components**: Put components in `components/global/` and access them directly inside markdown file
- TOC**: - id of titles create links - h2 & h3 are used to create toc

2. FETCH

Fetch content on:

- Client-side using `this.$content`
- Server-side using `context.$content`

```
content(path, options).fetch()
```

Example:

```
$content('articles', params.slug).fetch()
== /articles/${params.slug}
```

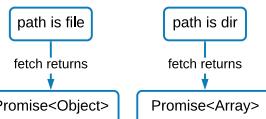
Chainable methods:

- only(keys)**: Select a subset of fields
 - without(keys)**: Remove a subset of fields
 - where(query)**: Filter results by query
 - sortBy(key, direction)**: Sort results by key
 - limit(n)**: Limit number of results
 - skip(n)**: Skip results
 - search(field, value)**: Perform a full-text search on a field
- Default fields:
['title', 'description', 'slug', 'text']

Nuxt Content Module

2. Fetch continues...

- surround(slug, options)**: Get prev and next results around a specific slug
search, limit and skip are ineffective when using this method
- fetch()**: Ends chain sequence and collects data



Example:

```
const articles =
  const articles = await this.$content('articles')
    .only(['title', 'date', 'authors'])
    .sortBy('date', 'asc')
    .limit(5)
    .skip(10)
    .where({tags: 'testing',
      isArchived: false,
      date: { $gt: new Date(2020) },
      rating: { $gte: 3 }
    })
    .search('welcome')
    .fetch()
```

Example with options:

```
Fetch files from subdirectories
const options = {deep:true, text:true}
const options = {deep:true, text:true}
await this.$content('articles', params.slug, options)
```

3. DISPLAY

Use `<nuxt-content>` component directly in template area to display the page body. No import required.

```
<nuxt-content :document="page" />
```

`<nuxt-content>` accepts page object using :document prop.

Live editing (>= v1.4.0): In development, double-click on the `<nuxt-content>` component to live edit the content directly in the browser!

Custom style: `<nuxt-content>` component adds `.nuxt-content` class. `.nuxt-content` class can be used to add custom styles for the content.

```
.nuxt-content h1 {
  // add custom style for h1 here
}
```

Example:

```
<template>
  <article>
    <h1>{{ page.title }}</h1>
    <nuxt-content :document="page" />
  </article>
</template>
<script>
export default {
  async asyncData ({ $content }) {
    const page = await $content('home').fetch()

    return {
      page
    }
  }
}</script>
```

Default configuration

Configure `@nuxt/content` with the `content` property in `nuxt.config.js`.

- Customise api url using `apiPrefix`. By default, `$content` api will be served on `http://localhost:3000/_content/articles`
- Provide different directory for writing content.
- Customise searchable fields.
- Provide custom PrismJS theme.

```
// nuxt.config.js
export default {
  content: {
    1 apiPrefix: '_content',
    2 dir: 'content',
    3 fullTextSearchFields:
      ['title', 'description', 'slug', 'text'],
    nestedProperties: ['categories.slug'],
    markdown: {
      remarkPlugins: [
        'remark-squeeze-paragraphs',
        'remark-slug',
        'remark-autolink-headings',
        'remark-external-links',
        'remark-footnotes'
      ],
      rehypePlugins: [
        'rehype-minify-whitespace',
        'rehype-sort-attribute-values',
        'rehype-sort-attributes',
        'rehype-raw'
      ],
      4 prism: {
        theme: 'prismjs/themes/prism.css'
      }
    },
    yaml: {},
    csv: {},
    xml: {}
  }
}
```

Learn more about custom configuration:
<https://content.nuxtjs.org/configuration>

Rechargement à chaud lors du développement

En développement, le module `content` est super rapide quand il s'agit de rechargement à chaud car il ne doit pas passer par Webpack lorsque l'on fait des changements à nos fichiers markdown. On peut aussi écouter l'événement `content:update` et créer un plugin qui à chaque fois que l'on va mettre à jour un fichier dans le répertoire `content`, va propager une méthode `fetchCategories` par exemple.

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Afficher le contenu

On peut directement utiliser le composant `<nuxt-content>` dans notre template pour afficher le corps de la page.

```
<template>
  <article>
    <nuxt-content :document="article" />
  </article>
</template>
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Styliser votre contenu

En fonction de ce que l'on va utiliser pour styliser notre application, on pourrait avoir besoin d'écrire un peu de style pour afficher notre markdown proprement.

Le composant `<nuxt-content>` va automatiquement ajouter une classe `.nuxt-content`, afin que nous puissions personnaliser nos styles.

```
<style>
  .nuxt-content h2 {
    font-weight: bold;
    font-size: 28px;
  }
  .nuxt-content p {
    margin-bottom: 20px;
  }
</style>
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Gérer les formats Markdown, CSV, YAML, JSON(5)

This module converts your .md files into a JSON AST tree structure, stored in a body variable. You can also add a YAML front matter block to your markdown files or a .yaml file which will be injected into the document. You can also add a json/json5 file which can also be injected into the document. And you can use a .csv file where rows will be assigned to the body variable.

Ce module convertit nos fichiers `.md` en une structure arborescente de type [JSON AST](#), stockée dans une variable `body`.

```
---  
title: Mon premier article de blog  
description: Apprendre à se servir de @nuxt/content pour créer un blog  
---
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Composants Vue dans le markdown

On peut utiliser les composants Vue directement dans les fichiers markdown. Il faudra cependant utiliser les composants sous leur écriture `kebab-case` et non avec des balises auto-fermantes.

```
<template>  
  <div class="p-4 mb-4 text-white bg-blue-500">  
    <p><slot name="info-box">default</slot></p>  
  </div>  
</template>
```

```
<info-box>  
  <template #info-box>  
    This is a vue component inside markdown using slots  
  </template>  
</info-box>
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

API entièrement recherchable

On peut utiliser `$content()` pour lister, filtrer et chercher notre contenu facilement.

```
<script>  
export default {  
  async asyncData({ $content, params }) {  
    const articles = await $content('articles', params.slug)  
      .only(['title', 'description', 'img', 'slug', 'author'])  
      .sortBy('createdAt', 'asc')  
      .fetch()  
  
    return {  
      articles  
    }  
  }  
</script>
```

```

        }
    }
</script>

```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Articles précédents et suivants

Le module `content` inclut un `.surround(slug)` qui nous permet de récupérer facilement les articles précédents et suivants.

```

async asyncData({ $content, params }) {
  const article = await $content('articles', params.slug).fetch()

  const [prev, next] = await $content('articles')
    .only(['title', 'slug'])
    .sortBy('createdAt', 'asc')
    .surround(params.slug)
    .fetch()

  return {
    article,
    prev,
    next
  }
},

```

```
<prev-next :prev="prev" :next="next" />
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Recherche par texte complet

Le module `content` possède une recherche par texte complet afin que l'on puisse rechercher simplement dans nos fichiers markdown sans avoir à installer quoi que ce soit d'autre.

```

<script>
  export default {
    data() {
      return {
        searchQuery: '',
        articles: []
      }
    },
    watch: {
      async searchQuery(searchQuery) {
        if (!searchQuery) {
          this.articles = []
          return
        }
        this.articles = await this.$content('articles')
          .limit(6)
          .search(searchQuery)
      }
    }
  }

```

```

        .fetch()
    }
}
</script>

```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Surbrillance de la syntaxe

Ce module va automatiquement envelopper les blocks de code et y appliquer les classes de [PrismJS](#). On peut bien sûr ajouter un thème PrismJS différent voire même le désactiver totalement.

```
yarn add prism-themes
```

```
npm install prism-themes
```

```

content: {
  markdown: {
    prism: {
      theme: 'prism-themes/themes/prism-material-oceanic.css'
    }
  }
}

```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Personnaliser le parsing du markdown

À l'origine, le markdown ne supporte pas la surbrillance des lignes que ce soit dans les blocks de code ou dans les noms de fichiers. Le module `content` permet cela avec sa propre syntaxe. Les lignes numérotées seront ajoutées à une balise `pre` dans l'attribut `data-line` et le nom du fichier sera converti en un `span` avec une classe `filename`, afin d'être stylisé.

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Génération d'une table des matières

Une propriété `toc` (pour Table of Contents) contenant un tableau, sera injectée dans notre document, lisant tous les entêtes avec leurs titres et identifiants, afin que l'on puisse les lier.

```

<nav>
  <ul>
    <li v-for="link of article.toc" :key="link.id">
      <NuxtLink :to="`#${link.id}`>{{ link.text }}</NuxtLink>
    </li>
  </ul>
</nav>

```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Un QueryBuilder d'API puissant

Le module `content` possède un puissant QueryBuilder d'API (similaire à MongoDB) qui nous permet de facilement voir le JSON de chaque répertoire à l'URL `http://localhost:3000/_content/`. L'extrémité est accessible via des requêtes `GET` et `POST`, on peut donc utiliser des query params.

```
http://localhost:3000/_content/articles?only=title&only=description&limit=10
```

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Personnaliser avec des hooks

On peut utiliser des hooks pour personnaliser le module afin d'ajouter de la data au document avant que ce dernier ne soit stocké.

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Intégration avec `@nuxtjs/feed`

Dans le cas d'articles, le contenu peut être utilisé pour générer un fil d'actualité en utilisant le module [`@nuxtjs/feed`](#).

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Support pour la génération de site statique

Ce module marche avec la génération de site statique en utilisant `nuxt generate`. Toutes les routes seront automatiquement générées grâce à la fonctionnalité du crawler de Nuxt.

Warning

Si on utilise Nuxt avec une version <2.13 et que l'on a besoin de spécifier des routes dynamiques, il faut utiliser la méthode `generate` avec le module `@nuxt/content` de manière programmatique.

Next

Se référer à la [documentation du module content](#) pour davantage de détails.

Prochaine étape

Next

Pour voir le tutoriel sur [comment créer un blog avec du contenu Nuxt](#)

Next

Pour voir la documentation complète du [module content](#)

[Go to TOC](#)

dist

Le répertoire `dist`, abréviation de *distribution*, est généré dynamiquement lors de l'utilisation de la commande `nuxt generate` et contient les fichiers HTML prêts pour la production ainsi que les ressources nécessaires au déploiement et à l'exécution de votre application Nuxt.

Déploiement

C'est le répertoire que vous devez **télécharger pour l'hébergement statique** car il contient vos fichiers HTML prêts à la production et vos ressources

Warning

Le répertoire `dist` ne doit pas être livré à votre système de contrôle de version et doit être ignoré par votre `.gitignore` car il sera généré automatiquement à chaque fois que vous lancez la commande `nuxt generate`.

La propriété 'dir'

Le répertoire dist est nommé dist par défaut mais peut être configuré dans votre fichier `nuxt.config`.

```
generate: {
  dir: 'mon-site'
}
```

Warning

Si vous changez votre répertoire dist, vous devrez l'ajouter à votre contrôle de version pour que git l'ignore.

La propriété subFolders

Nuxt place par défaut toutes nos pages générées dans un répertoire, cependant nous pouvons changer cela si on le souhaite en modifiant le fichier `nuxt.config.js` et en changeant la propriété des sous-répertoires pour qu'ils soient `false`.

```
generate: {
  subFolders: false
}
```

La propriété fallback

Lors du déploiement de votre site, vous devrez vous assurer que le chemin html de repli est correctement défini. Il doit être défini comme page d'erreur afin que les routes inconnues soient rendues via Nuxt. S'il n'est pas défini, Nuxt utilisera la valeur par défaut qui est la page 200.html.

Lors de l'exécution d'une application à page unique, il est plus logique d'utiliser 200.html car c'est le seul fichier nécessaire puisque aucune autre route n'est générée.

Lorsque vous travaillez avec des pages générées statiquement, il est recommandé d'utiliser un fichier 404.html pour les pages d'erreur.

Warning

Selon l'endroit où vous hébergez votre site, vous devez utiliser un fichier 200.html ou 404.html. Veuillez vérifier auprès de votre fournisseur d'hébergement. Netlify, par exemple, utilise 404.html.

```
export default {
  generate: {
    fallback: '404.html'
  }
}
```

La propriété exclude

Vous pouvez exclure des pages de la génération en utilisant la propriété `generate exclude`. Au lieu d'être générée en tant que page statique, elle deviendra une page d'application unique et ne sera rendue que du côté client.

```
generate: {
  exclude: [/admin/]
}
```

Info

Vous pouvez également utiliser une expression régulière ici pour exclure les pages commençant ou se terminant par un mot en particulier.

layouts

Les `layouts` sont d'une grande aide lorsque vous souhaitez modifier l'aspect et la présentation de votre application Nuxt. Vous pouvez inclure une barre latérale ou avoir des pages différentes entre une version mobile et web.

Warning

Ce répertoire ne peut pas être renommé sans configuration supplémentaire.

Layout par défaut

Vous pouvez étendre la présentation principale en ajoutant un fichier `layout/default.vue`. Il sera utilisé pour toutes les pages qui n'ont pas de `layout` défini. Assurez-vous d'ajouter le composant `<Nuxt>` lors de la création d'un `layout` pour inclure le composant de la page.

Tout ce dont vous avez besoin dans votre layout, c'est de ces trois lignes de code qui effectueront le rendu du composant de la page.

```
<template>
  <Nuxt />
</template>
```

Vous pouvez ajouter ici d'autres éléments tels que : Navigation, Header, Footer etc.

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <TheFooter />
  </div>
</template>
```

Info

Si vos `composants sont réglés sur true` alors il n'est pas nécessaire de déclarer l'importation pour vos composants.

Layout personnalisé

Tous les fichiers du répertoire `layouts` créeront un `layout` personnalisé accessible avec la propriété `layout` dans les composants de la page.

Disons que nous voulons créer un layout de blog et le sauvegarder sous `layouts/blog.vue` :

```
<template>
  <div>
    <div>La barre de navigation de mon blog ici</div>
```

```
<Nuxt />
</div>
</template>
```

Ensuite, vous devez indiquer aux pages d'utiliser votre layout personnalisé

```
<script>
export default {
  layout: 'blog',
  // OR
  layout (context) {
    return 'blog'
  }
}</script>
```

Page d'erreur

La page d'erreur est un composant (*page component*) qui est toujours affiché lorsqu'une erreur se produit (qui ne s'est pas lancée du côté du serveur).

Warning

Bien que ce fichier soit placé dans le répertoire `layouts`, il doit être traité comme une page.

Comme mentionné ci-dessus, ce `layout` est spécial et vous ne devez pas inclure `<Nuxt>` dans son modèle. Vous devez voir ce `layout` comme un composant affiché lorsqu'une erreur se produit (404, 500, etc.). Comme pour les autres composants de la page, vous pouvez également définir un `layout` personnalisé pour la page d'erreur de la manière habituelle.

Vous pouvez personnaliser la page d'erreur en ajoutant un fichier `layouts/error.vue` :

```
<template>
<div class="container">
  <h1 v-if="error.statusCode === 404">Page non trouvée</h1>
  <h1 v-else>Une erreur est survenue</h1>
  <NuxtLink to="/">Page d'accueil</NuxtLink>
</div>
</template>

<script>
export default {
  props: ['error'],
  layout: 'blog' // vous pouvez customiser l'affichage pour une page d'erreur
}
</script>
```

Info

Le code source de la page d'erreur par défaut est [disponible sur GitHub](#).

[Go to TOC](#)

middleware

Le répertoire `middleware` contient les middlewares de l'application. Un middleware nous permet de définir des fonctions personnalisées qui peuvent s'exécuter avant de render une page ou un groupe de pages (layout).

Les middlewares partagés devraient être placés dans le répertoire `middleware/`. Le nom du fichiers sera aussi le nom du middleware (un fichier `middleware/auth.js` sera appelé avec middleware `auth`). On peut aussi utiliser des middlewares spécifiques à des pages en utilisant directement une fonction, voir [middlewares anonymes](#).

Un middleware reçoit le `context` en tant que premier argument.

```
export default function (context) {
  // Ajouter la propriété userAgent au contexte
  context.userAgent = process.server
    ? context.req.headers['user-agent']
    : navigator.userAgent
}
```

Dans le mode universel, les middlewares seront appelés une fois côté serveur (à la première requête à l'application Nuxt, ex: lorsque l'on accède directement à l'application ou lors du rafraîchissement de la page) et du côté client lors de la navigation sur des routes supplémentaires. En mode SPA, les middlewares seront appelés côté client dans les deux situations.

Les middlewares seront appelés en série dans cet ordre:

1. `nuxt.config.js` (en suivant l'ordre définit dans le fichier)
2. Les layouts qui match
3. Les pages qui match

Le middleware du routeur

Un middleware peut être asynchrone. Pour cela, il faut renvoyer une `Promesse` ou utiliser `async / await`.

```
import http from 'http'

export default function ({ route }) {
  return http.post('http://mes-stats-api.com', {
    url: route fullPath
  })
}
```

Ensuite, dans le fichier `nuxt.config.js`, il faut utiliser la propriété `router.middleware`.

```
export default {
  router: {
    middleware: 'stats'
  }
}
```

Maintenant, le middleware `stats` sera appelé pour chaque changement de route.

On peut ajouter notre middleware (plusieurs même) pour un layout spécifique mais aussi une page.

```
export default {
  middleware: ['auth', 'stats']
}
```

Middleware nommé

On peut nommer un middleware en créant un fichier à l'intérieur du répertoire `middleware/`, le nom du fichier sera aussi le nom du middleware.

```
export default function ({ store, redirect }) {
  // Si l'utilisateur n'est pas authentifié
  if (!store.state.authenticated) {
    return redirect('/login')
  }
}
```

```
<template>
  <h1>Page secrète</h1>
</template>

<script>
  export default {
    middleware: 'authenticated'
  }
</script>
```

Middleware anonyme

Si l'on a besoin d'un middleware pour seulement une page spécifique, on peut directement utiliser une fonction pour cela (voire un tableau de fonctions):

```
<template>
  <h1>Page secrète</h1>
</template>

<script>
  export default {
    middleware({ store, redirect }) {
      // Si l'utilisateur n'est pas authentifié
      if (!store.state.authenticated) {
        return redirect('/login')
      }
    }
  }
</script>
```


modules

Nuxt fournit un système de modules d'ordre supérieur qui permet de personnaliser son fonctionnement interne. Les modules sont des fonctions qui sont appelées de manière séquentielle lors du démarrage de Nuxt.

Découvrez notre [liste de modules](#) pour enrichir votre projet Nuxt, créés par l'équipe Nuxt et la communauté.

- Plus de 165 Modules
- Plus de 105 Contributeurs

Next

Consultez modules.nuxtjs.org

Nuxt Modules

modules are Nuxt extensions that extend Nuxt's core functionalities

Nuxt v2.14
September 2020

property type:
array

modules

Called once, in sequence,
while booting Nuxt app
i.e. npm run dev

```
// nuxt.config.js
export default {
  modules: [
    // Using package name
    '@nuxtjs/axios',
    // Relative to project srcDir
    '~/modules/simple',
    // Providing options
    ['~/modules/simple', { token: '123' }],
    // Inline definition
    function () {}
  ]
}
```

array item can be
string, array with
options or function

Write your own module

Combine multiple options
sources

Top-level options

```
// nuxt.config.js
export default {
  modules: [
    ['@nuxtjs/axios', { anotherOption: true }],
    // axios module is aware of this by using `this.options.axios`
    axios: { option1, option2 }
  ]
}
```

Access Nuxt options, including
module's options
i.e. add a CSS library or
emit assets

this.options

```
// modules/module.js
export default function (moduleOptions) {
  // `options` will contain option1,option2 & anotherOption
  const options = Object.assign({}, this.options.axios,
    moduleOptions
  )
}
```

Refers to the current Nuxt
instance, allowing us
to register hooks on certain
life-cycle events
i.e. provide plugins or
register custom webpack loaders

this.nuxt

```
// modules/module.js
export default function (moduleOptions) {
  this.nuxt.hook('ready', async nuxt => {})
  this.nuxt.hook('error', async error => {})
  this.nuxt.hook('close', async nuxt => {})
  this.nuxt.hook('listen', async(server, {host, port})) => {}
}
```

this refers to the
moduleContainer
instance

create async modules that
return a promise or
call a callback

async module

```
// modules/mymodule.js
// Use async/await
export default async function asyncModule() {
  const pages = await ...
}
// Return a promise
export default function asyncModule() {
  return fetch().then(response => ...)
}
```

Use modules

If module:

- provides serverMiddleware
- registers Node.js runtime hook
- affects vue-renderer behaviour
- or anything that's outside of webpack scope

```
// nuxt.config.js
export default {
  modules: ['@nuxtjs/sitemap']
}
```

Use buildModules

If module is imported during dev and build time.

simply, add dependency in devDependencies and
use buildModules:

```
// nuxt.config.js
export default {
  buildModules: ['@nuxtjs/eslint-module']
}
```

Benefits:

- Decrease node_module size
- Make production startup faster

Lors du développement d'applications complexes avec Nuxt, on pourrait finir par trouver que les fonctionnalités principales du framework ne sont pas suffisantes. Nuxt peut donc être personnalisé avec des options de configuration et des plugins, mais maintenir ces personnalisations à travers plusieurs projets est pénible, répétitif et chronophage. D'autre part, supporter les besoins de n'importe quel projet par défaut rendrait Nuxt très complexe et difficile à utiliser.

Ces raisons font que Nuxt nous fournit un système de modules d'ordre supérieur qui permet de personnaliser son fonctionnement interne. Les modules sont des fonctions qui sont appelées de manière séquentielle lors du démarrage de Nuxt. Le framework attend que chaque module ait fini avant de procéder. Ainsi, les modules peuvent personnaliser presque n'importe quel aspect du projet. Grâce au design modulaire de Nuxt (basé sur [Tapable](#) de Webpack), les modules peuvent facilement utiliser des hooks pour certains points d'entrée tel que l'initialisation du builder. Les modules peuvent aussi écraser les templates, paramétriser les loaders de Webpack, ajouter des librairies CSS et faire plein d'autres tâches utiles.

Et surtout, les modules de Nuxt peuvent être intégrés dans des packages npm. Cela rend possible leur réutilisation à travers plusieurs projets ainsi que la possibilité de partager avec la communauté, en créant un écosystème d'add-ons de haute qualité.

La propriété modules

Les modules sont des extensions Nuxt qui peuvent personnaliser le fonctionnement interne du framework et ajouter un nombre infini d'intégrations. Une fois le module installé, on peut l'ajouter au fichier `nuxt.-config.js` à la propriété `modules`.

```
export default {
  modules: [
    // en utilisant le nom du package
    '@nuxtjs/axios',

    // de manière relative par rapport au répertoire source du projet (srcDir)
    '~/modules/awesome.js',

    // on peut lui passer des options
    ['@nuxtjs/google-analytics', { ua: 'X1234567' }],

    // ou le définir directement
    function () {}
  ]
}
```

Info

Les développeurs fournissent la plupart du temps des étapes et détails additionnels pour l'utilisation des modules.

Nuxt essaie de résoudre chaque élément qui est présent dans le tableau des modules en utilisant `require path` de Node (dans les `node_modules`) et résout ensuite en partant du `srcDir` du projet si un alias `@` est utilisé.

Warning

Les modules sont exécutés de manière séquentielle, l'ordre est donc important.

Les modules doivent exporter une fonction pour permettre d'améliorer le build/runtime et peuvent (optionnel) aussi retourner une promesse en attendant que leur job ne soit terminé. À noter qu'ils seront importés au runtime, donc ils doivent déjà être transpilés s'ils utilisent des fonctionnalités modernes (ex: ES6).

Écrire votre propre Module

Les modules sont des fonctions. Ils peuvent être emballés en tant que modules npm ou directement ajoutés dans le code source du projet.

```
export default {
  exampleMsg: 'salut',
  modules: [
    // Usage simple
    '~/modules/example',
    // Ici, on passe des options directement
    ['~/modules/example', { token: '123' }]
  ]
}
```

```
export default function ExampleModule(moduleOptions) {
  console.log(moduleOptions.token) // '123'
  console.log(this.options.exampleMsg) // 'salut'

  this.nuxt.hook('ready', async nuxt => {
    console.log('Nuxt est prêt')
  })
}

// REQUIS si l'on publie le module en tant que package npm
module.exports.meta = require('./package.json')
```

1) ModuleOptions

`moduleOptions` : c'est l'objet passé par l'utilisateur au travers du tableau `modules`. On peut l'utiliser pour personnaliser son comportement.

Options de niveau supérieur

Parfois, il est plus pratique d'utiliser des options de niveau supérieur lors de la déclaration des modules dans le fichier `nuxt.config.js`. Cela nous permet de combiner plusieurs sources d'options.

```
export default {
  modules: [['@nuxtjs/axios', { uneAutreOption: true }]],

  // le module axios est au courant de ceci car il utilise `this.options.axios`
  axios: {
    option1,
    option2
  }
}
```

2) this.options

`this.options` : on peut directement accéder aux options de Nuxt en utilisant cette référence. Cela contient le contenu du fichier `nuxt.config.js` de l'utilisateur avec toutes les options par défaut qui lui sont assignées. Cela peut être utilisé pour partager des options entre les modules.

```
export default function (moduleOptions) {
  // `options` va contenir option1, option2 et uneAutreOption
  const options = Object.assign({}, this.options.axios, moduleOptions)

  // ...
}
```

Ajouter une librairie CSS

Si votre module fournit une librairie CSS, il faut s'assurer de vérifier si l'utilisateur l'a déjà inclus pour éviter des doublons ainsi qu'ajouter une option pour la désactiver.

```
export default function (moduleOptions) {
  if (moduleOptions.fontAwesome !== false) {
    // Ajoute Font Awesome
    this.options.css.push('font-awesome/css/font-awesome.css')
  }
}
```

Émettre des ressources

On peut paramétriser les plugins de Webpack afin qu'ils émettent des ressources durant le build.

```
export default function (moduleOptions) {
  const info = 'Buildé par un super module - 1.3 alpha le ' + Date.now()

  this.options.build.plugins.push({
    apply(compiler) {
      compiler.plugin('emit', (compilation, cb) => {
        // Cela va générer un `.nuxt/dist/info.txt` avec le contenu de la variable
        info.
        // La source peut aussi être un buffer
        compilation.assets['info.txt'] = {
          source: () => info,
          size: () => info.length
        }

        cb()
      })
    }
  })
}
```

3) this.nuxt

`this.nuxt` : ceci est une référence à l'instance actuelle de Nuxt. On peut attacher des hooks à certains événements du lifecycle.

- **Ready** : Nuxt est prêt à être utilisé (ModuleContainer et Renderer sont prêts).

```
nuxt.hook('ready', async nuxt => {
  // Votre code personnalisé ici
})
```

- **Error:** une erreur non générée est survenue durant un hook.

```
nuxt.hook('error', async error => {
  // Votre code personnalisé ici
})
```

- **Close:** l'instance de Nuxt s'arrête proprement.

```
nuxt.hook('close', async nuxt => {
  // Votre code personnalisé ici
})
```

- **Listen:** le serveur interne de Nuxt commence à écouter (lorsque l'on utilise `nuxt start` ou `nuxt dev`).

```
nuxt.hook('listen', async (server, { host, port }) => {
  // Votre code personnalisé ici
})
```

`this`: le contexte des modules. Tous les modules seront appelés au sein du contexte de l'instance de `ModuleContainer`.

Plus d'informations pour les méthodes disponibles sont présents dans la documentation du [ModuleContainer](#).

Exécuter certaines actions lors de hooks spécifiques

Notre module pourrait avoir besoin de faire des choses seulement dans des conditions spécifiques et pas seulement durant l'initialisation de Nuxt. On peut utiliser les puissants hooks de Nuxt grâce à [hookable](#). Nuxt attendra la fin de la fonction si elle retourne une Promesse ou qu'elle est définie en tant que fonction asynchrone.

Voici quelques exemples basiques:

```
export default function myModule() {
  this.nuxt.hook('modules:done', moduleContainer => {
    // Ceci sera appelé lorsque tous les modules auront fini de charger
  })

  this.nuxt.hook('render:before', renderer => {
    // Ceci sera appelé après que le renderer soit créé
  })

  this.nuxt.hook('build:compile', async ({ name, compiler }) => {
    // Ceci sera appelé avant que le compilateur (par défaut: Webpack) ne commence
  })

  this.nuxt.hook('generate:before', async generator => {
    // Ceci sera appelé avant que Nuxt ne génère nos pages
  })
}
```

Ajouter des plugins

Il est commun que des modules arrivent avec un ou plusieurs plugins lorsqu'ils sont ajoutés. Par exemple, le module `bootstrap-vue` demandera à être enregistré dans Vue. Dans ces cas là, on peut utiliser le helper `this.addPlugin`.

```
import Vue from 'vue'
import BootstrapVue from 'bootstrap-vue/dist/bootstrap-vue.esm'

Vue.use(BootstrapVue)

import path from 'path'

export default function nuxtBootstrapVue(moduleOptions) {
  // Enregistrer le template `plugin.js`
  this.addPlugin(path.resolve(__dirname, 'plugin.js'))
}
```

Note: Any plugins injected by modules are added to the *beginning* of the plugins list. Your options are to:

- Manually add your plugin to the end of the list of plugins (`this.nuxt.options.plugins.push(...)`)
- Reverse the order of the modules if it depends on another

Les plugins de template

Les templates et plugins enregistrés peuvent bénéficier des `templates lodash` pour changer conditionnellement les plugins finaux à être enregistrés.

```
// Définition du code UA pour Google Analytics
ga('create', '<%= options.ua %>', 'auto')

<% if (options.debug) { %>
// Code uniquement disponible en développement
<% } %>

import path from 'path'

export default function nuxtGoogleAnalytics(moduleOptions) {
  // Enregistrer le template `plugin.js`
  this.addPlugin({
    src: path.resolve(__dirname, 'plugin.js'),
    options: {
      // Nuxt remplacera `options.ua` avec `123` lors de la copie
      // du plugin dans le projet
      ua: 123,

      // les parties conditionnelles lors du développement seront enlevées
      // du code du plugin lors d'un build en production
      debug: this.options.dev
    }
  })
}
```

Enregistrer des loaders personnalisés pour Webpack

On peut faire pareil que `build.extend` dans le fichier `nuxt.config.js` en utilisant `this.extendBuild`.

```

export default function (moduleOptions) {
  this.extendBuild((config, { isClient, isServer }) => {
    // Loader `bidule`
    config.module.rules.push({
      test: /\.bidule$/,
      use: [...]
    })

    // Personnalisation de loaders existants
    // Se référer au code source du fonctionnement interne de Nuxt:
    //
    https://github.com/nuxt/nuxt.js/blob/dev/packages/webpack/src/config/base.js
    const barLoader = config.module.rules.find(rule => rule.loader === 'bar-loader')
  })
}

```

Modules asynchrones

Tous les modules ne feront pas des choses synchrones. Par exemple, on pourrait avoir envie de développer un module qui a besoin de récupérer de la data depuis une API. C'est pour cela que Nuxt supporte les modules asynchrones qui peuvent retourner une Promesse ou appeler un callback.

Use `async/await`

```

import fse from 'fs-extra'

export default async function asyncModule() {
  // On peut écrire du code asynchrone ici en utilisant `async`/`await`
  const pages = await fse.readJson('./pages.json')
}

```

Retourner une promesse

```

export default function asyncModule($http) {
  return $http
    .get('https://jsonplaceholder.typicode.com/users')
    .then(res => res.data.map(user => '/users/' + user.username))
    .then(routes => {
      // Faire quelques de personnalisé avec les routes de Nuxt
    })
}

```

Info

Il y a beaucoup de hooks et de possibilités pour les modules. Se référer au [fonctionnement interne](#) pour comprendre davantage l'API de Nuxt.

Publication de votre module

`module.exports.meta` : Cette ligne est requise si l'on veut publier le module en tant que package npm. Nuxt utilise `meta` en interne afin de fonctionner au mieux avec votre package.

```
module.exports.meta = require('./package.json')
```

buildModules

Certains modules sont importés seulement lors du développement et du build. Utiliser `buildModules` permet de démarrer un projet en production plus rapidement et réduit aussi significativement la taille des `node_modules` pour les déploiements en production. Se référer à la documentation de chaque module pour voir s'il est recommandé d'utiliser `modules` ou `buildModules`.

La différence à l'utilisation est la suivante:

- Au lieu d'ajouter aux `modules` à l'intérieur du fichier `nuxt.config.js`, il faut utiliser `buildModules`

```
export default {
  buildModules: ['@nuxtjs/eslint-module']
}
```

- Au lieu d'ajouter aux `dependencies` à l'intérieur du fichier `package.json`, il faut utiliser `devDependencies`

```
yarn add --dev @nuxtjs/eslint-module
```

```
npm install --save-dev @nuxtjs/eslint-module
```

Info

Si l'on est l'auteur d'un module, il est hautement recommandé de suggérer à vos utilisateurs d'installer votre package en tant que `devDependency` et d'utiliser `buildModules` au lieu de `modules`.

Un module est un `buildModule` à moins que:

- il fournit un `serverMiddleware`
- il y a besoin d'enregistrer un runtime hook de Node.js (tel que `Sentry`)
- il affecte le comportement de `vue-renderer` ou utilise un hook avec un de ces deux namespaces: `server:` ou `vue-renderer:`.
- tout ce qui est en dehors du scope de Webpack (astuce: les plugins et les templates sont compilés, ils font donc partie du scope de Webpack)

Warning

Si vous vous apprêtez à offrir un `buildModules`, veuillez noter que cette fonctionnalité est disponible depuis seulement la version 2.9. Les versions plus anciennes devront mettre à jour Nuxt ou utiliser la section `modules`.

pages

Le répertoire `pages` contient les vues et les routes de vos applications. Nuxt lit tous les fichiers `.vue` à l'intérieur de ce répertoire et crée automatiquement la configuration du routeur pour vous.

Info

Vous pouvez également créer des routes avec des fichiers `.js` et `.ts`

Chaque composant de Page est un composant Vue mais Nuxt ajoute des attributs et des fonctions spéciales pour rendre le développement de votre application universelle aussi facile que possible.

```
<template>
  <h1 class="red">Bonjour {{ name }}!</h1>
</template>

<script>
  export default {
    // les propriétés de la page vont ici
  }
</script>

<style>
  .red {
    color: red;
  }
</style>
```

Pages dynamiques

Les pages dynamiques peuvent être créées lorsque vous ne connaissez pas le nom de la page parce qu'il provient d'une API ou que vous ne voulez pas avoir à créer la même page encore et encore. Pour créer une page dynamique, vous devez ajouter un underscore (`_`) avant le nom du fichier `.vue` ou avant le nom du répertoire, si vous souhaitez que le répertoire soit dynamique. Vous pouvez nommer le fichier ou le répertoire comme vous le souhaitez, mais vous devez le faire précédé d'un underscore .

Si nous avons défini un fichier nommé `_slug.vue` dans notre répertoire `pages`, nous pouvons accéder à la valeur en utilisant le contexte avec `params.slug`

```
<template>
  <h1>{{ this.slug }}</h1>
</template>

<script>
  export default {
    async asyncData({ params }) {
      const slug = params.slug // En appelant /abc, le slug sera "abc".
      return { slug }
    }
  }
</script>
```

Si nous avons défini un fichier nommé `_slug.vue` dans un répertoire appelé `_book`, nous pouvons accéder à la valeur en utilisant le contexte avec `params.slug` et `params.book`

```
<template>
  <h1>{{ this.book }} / {{ this.slug }}</h1>
</template>

<script>
  export default {
    async asyncData({ params }) {
      const book = params.book
      const slug = params.slug
      return { book, slug }
    }
  }
</script>
```

Propriétés

asyncData

`AsyncData` est appelé à chaque fois avant le chargement du composant. Elle peut être asynchrone et reçoit le contexte en argument. L'objet renvoyé sera fusionné avec votre objet de données.

```
export default {
  asyncData (context) {
    return { name: 'World' }
  }
}
```

Next

Pour en savoir plus sur le fonctionnement d'`asyncData`, consultez notre chapitre [Data Fetching](#)

fetch

Chaque fois que vous avez besoin d'obtenir des données asynchrones, vous pouvez utiliser `fetch()`. `Fetch` est appelé côté serveur lors du rendu de la route, et côté client lors de la navigation.

```
<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    async fetch() {
      this.posts = await fetch('https://api.nuxtjs.dev/posts').then(res =>
        res.json()
      )
    }
  }
</script>
```

Next

Pour en savoir plus sur le fonctionnement de l'extraction, consultez notre chapitre [Data Fetching](#)

head

Définissez des balises pour la page en cours. Nuxt utilise `vue-meta` pour mettre à jour l'en-tête du document et les méta attributs de votre application.

```
export default {
  head() {
    // Définit les balises Méta pour cette page
  }
}
```

Next

Pour en savoir plus, consultez notre chapitre [Balises meta et référencement SEO](#)

layout

Précisez une mise en page définie dans le répertoire des mises en page.

```
export default {
  layout: 'blog'
}
```

Next

Pour en savoir plus sur les mises en page, consultez notre chapitre sur les [Vues](#).

loading

Si ce paramètre est défini sur `false`, il empêche une page d'appeler automatiquement `this.$nuxt.$loading.finish()` lorsque vous l'entrez et `this.$nuxt.$loading.start()` lorsque vous la quittez, ce qui vous permet de contrôler manuellement le comportement, comme le montre [cet exemple](#).

```
export default {
  loading: false
}
```

Info

S'applique uniquement si la propriété `loading` est également défini dans le fichier `nuxt.config.js`.

Next

Pour en savoir plus, consultez notre chapitre sur le [Chargement](#).

transition

Définit une transition spécifique pour la page.

```
export default {
  transition: 'fade'
}
```

Next

Pour en savoir plus, consultez notre chapitre sur les [Transitions](#)

scrollToTop

La propriété `scrollToTop` vous permet de dire à Nuxt de faire défiler la page vers le haut avant de la rendre. Par défaut, Nuxt fait défiler la page vers le haut lorsque vous allez sur une autre page, mais avec les routes enfant, Nuxt garde la position de défilement. Si vous voulez dire à Nuxt de faire défiler la page vers le haut lors du rendu de votre route fille, mettez `scrollToTop` à `true`

```
export default {
  scrollToTop: true
}
```

Inversement, vous pouvez aussi régler manuellement `scrollToTop` à `false` sur les routes parentes.

Si vous souhaitez remplacer le comportement de défilement par défaut de Nuxt, consultez l'option `scrollBehavior`.

middleware

Définit le middleware pour cette page qui sera appelé avant le rendu de la page.

```
export default {
  middleware: 'auth'
}
```

Next

Pour en savoir plus, consultez notre chapitre sur les [Middleware](#)

La propriété watchQuery

Utilisez la propriété `WatchQuery` pour configurer un observateur (watcher) pour les chaînes de requête. Si les chaînes définies changent, toutes les méthodes du composants (`asyncData`, `fetch`, `validate`, `layout`, ...) seront appelées. Le `watcher` est désactivée par défaut pour améliorer les performances.

```
export default {
  watchQuery: ['page']
}
```

Info

Si vous voulez mettre en place un `watcher` pour toutes les chaînes de requête, définissez `watchQuery` sur `true`.

```
export default {
  watchQuery: true
}
```

Vous pouvez également utiliser la fonction `watchQuery(newQuery, oldQuery)` pour avoir des observateurs plus raffinés.

```
export default {
  watchQuery(newQuery, oldQuery) {
    // Exécute les méthodes du composants que si l'ancienne chaîne de requête
    // contenait `bar`.
    // et que la nouvelle chaîne de requête contient `foo`.
    return newQuery.foo && oldQuery.bar
  }
}
```

Next

Pour en savoir plus sur la propriété d'observation, consultez notre chapitre [Data Fetching](#) chapter

key

Comme la propriété `key` qui peut être utilisée sur les composants Vue dans les modèles comme une indication pour le DOM virtuel, cette propriété permet à la valeur de la clé d'être définie à partir de la page elle-même (plutôt que le composant parent).

Par défaut dans Nuxt, cette valeur sera `$route.path`, ce qui signifie que naviguer vers une route différente assurera la création d'un composant de page propre. Logiquement équivalent à :

```
<router-view :key="$route.path" />
```

La propriété peut être une `String` ou une `Function` qui prend la route comme premier argument.

Ignorer des pages

Si vous voulez ignorer des pages afin qu'elles ne soient pas générées dans le fichier `router.js`, alors vous pouvez les ignorer en les préfixant par un `-`.

Par exemple, `pages/-about.vue` sera ignorée.

Next

Pour en savoir plus, consultez notre chapitre sur l'option `ignore`

Configuration

Vous pouvez renommer le répertoire `pages/` en quelque chose de différent en définissant l'option `dir.-pages` :

```
export default {
  dir: {
    // Renommer le répertoire `pages` en `routes`.
```

```
    pages: 'routes'  
  }  
}
```

Next

Pour en savoir plus, consultez notre chapitre sur l'option [dir](#) :

[Go to TOC](#)

La propriété alias

- Type: `Object`
- Default:

```
{
  '~~': `<rootDir>`,
  '@@': `<rootDir>`,
  '~': `<srcDir>`,
  '@': `<srcDir>`,
  'assets': `<srcDir>/assets`, // (inutile si vous avez modifié `dir.assets`)
  'static': `<srcDir>/static`, // (inutile si vous avez modifié `dir.static`)
}
```

Cette option vous permet de définir des alias vers les répertoires de votre projet (en plus de ceux ci-dessus). Ces alias peuvent être utilisés dans votre JavaScript et CSS.

```
import { resolve } from 'path'
export default {
  alias: {
    'images': resolve(__dirname, './assets/images'),
    'style': resolve(__dirname, './assets/style'),
    'data': resolve(__dirname, './assets/other/data')
  }
}
```

```
<template>
  
</template>

<script>
import data from 'data/test.json'

// etc.
</script>

<style>
@import '~style/variables.scss';
@import '~style/utils.scss';
@import '~style/base.scss';

body {
  background-image: url('~images/main-bg.jpg');
}
</style>
```

Warning

Dans un contexte Webpack (sources d'images, CSS - mais *pas* JavaScript), vous devez préfixer votre alias avec `~` (comme dans l'exemple ci-dessus).

Info

Si vous utilisez TypeScript et que vous voulez utiliser l'alias que vous définissez dans vos fichiers TypeScript, vous devrez ajouter les alias à votre objet `paths` dans `tsconfig.json`.

[Go to TOC](#)

La propriété extendPlugins

La propriété `extendPlugins` permet de personnaliser les plugins de Nuxt ([options.plugins](#)).

- Type: `Function`
- Par défaut: `undefined`

On pourrait vouloir personnaliser les plugins ou changer l'ordre créé par Nuxt pour ceux-ci. Cette fonction accepte un tableau d'objets de `plugins` et doit renvoyer un tableau d'objets de plugins.

Exemple de changement de l'ordre des plugins:

```
export default {
  extendPlugins(plugins) {
    const pluginIndex = plugins.findIndex(
      ({ src }) => src === '~/plugins/doitEtreLePremier.js'
    )
    const doitEtreLePremierPlugin = plugins[pluginIndex]

    plugins.splice(pluginIndex, 1)
    plugins.unshift(doitEtreLePremierPlugin)

    return plugins
  }
}
```

La propriété generate

Permet de configurer la génération de votre application web universelle en une application web statique.

- Type: `Object`

Lors de l'appel de `nuxt.generate()`, Nuxt va utiliser la configuration définie dans la propriété `generate`.

```
export default {
  generate: {
    ...
  }
}
```

cache

Introduit avec la v2.14.0

- Type: `Object` ou `false`

Cette option est utilisée par `nuxt generate` avec la `cible statique` pour éviter de re-build lorsque les fichiers traqués n'ont pas été modifiés.

Par défaut:

```
{
  ignore: [
    '.nuxt', // buildDir
    'static', // dir.static
    'dist', // generate.dir
    'node_modules',
    '**/*',
    '*',
    'README.md'
  ]
}
```

Si on veut éviter de re-build lors d'un changement du fichier de configuration, il faut l'ajouter à la liste au travers de l'option `cache.ignore` :

```
export default {
  generate: {
    cache: {
      ignore: ['renovate.json'] // ignore les changements appliqués à ce fichier
    }
  }
}
```

concurrency

- Type: `Number`
- Par défaut: `500`

La génération des routes est concomitance, `generate.concurrency` spécifie le nombre de routes qui peuvent s'exécuter dans un seul thread.

crawler

- Type: `boolean`
- Par défaut: `true`

Depuis Nuxt v2.13, un crawler s'occupera de trouver et de générer toutes les routes dynamiques en passant en revue tous les liens relatifs. Si l'on veut désactiver cette fonctionnalité, il suffit de passer la valeur à `false`.

```
export default {
  generate: {
    crawler: false
  }
}
```

dir

- Type: `String`
- Par défaut: `'dist'`

Le nom du répertoire créé lors du build de l'application web en mode statique avec `nuxt generate` ou en mode SPA avec `nuxt build`.

devtools

- Type: `boolean`
- Par défaut: `false`

Permet de configurer si l'on est autorisé ou pas à inspecter les [vue-devtools](#).

Si c'est déjà activé dans le fichier `nuxt.config.js` ou ailleurs, les devtools seront activés, peu importe le flag.

exclude

- Type: `Array`
 - Items: `String` ou `RegExp`

`exclude` accepte un tableau de chaînes de caractères ou des expressions régulières et s'occupe d'éviter de générer les routes qui match. Les routes seront toujours accessibles lorsque `generate.fallback` est utilisé.

En prenant cet exemple de structure :

```
- | pages/
--- | index.vue
--- | admin/
---- | about.vue
---- | index.vue
```

Par défaut, lors de l'exécution de `nuxt generate`, un fichier sera créé pour chaque route.

```
- | dist/
--- | index.html
--- | admin/
---- | about.html
---- | item.html
```

Lorsque dans `ignore`, on ajoute une expression régulière qui match toutes les routes, cela va empêcher la génération de ces routes.

```
export default {
  generate: {
    exclude: [
      '/^\/admin/' // un chemin qui commence par /admin
    ]
  }
}
```

```
- | dist/
--- | index.html
```

On peut aussi exclure une route spécifique en passant une chaîne de caractères :

```
export default {
  generate: {
    exclude: ['/ma-page-secrete']
  }
}
```

fallback

- Type: `String` ou `Boolean`
- Par défaut: `200.html`

```
export default {
  generate: {
    fallback: '404.html'
  }
}
```

Fournit le chemin du fallback au fichier HTML. Cela devrait être défini en tant que page d'erreur, pour que toutes les routes inconnues soient rendues via Nuxt. Si la valeur n'est pas spécifiée ou est `false`, le fallback sera `200.html`. Si la valeur est à `true`, le fallback sera à `404.html`. Si l'on fournit une chaîne de caractères, elle sera utilisée à la place.

Lorsque notre application est une SPA, il est plus idiomatique d'utiliser une `200.html` vu que ce sera le seul fichier nécessaire car aucune autre route ne sera générée.

`fallback: false`

Lorsque l'on travaille avec des pages générées de manière statique, il est recommandé d'utiliser une `404.html` en tant que page d'erreur et pour celles qui concordent avec les entrées dans `excludes` (autrement dit, les fichiers que l'on ne veut pas voir être générés en tant que pages statiques).

`fallback: true`

Cependant, Nuxt permet de configurer n'importe quelle page si vous ne souhaitez pas utiliser les classiques `200.html` ou `404.html`, il suffit d'ajouter la chaîne de caractères souhaitée et de penser à faire la redirection sur cette page. Ce n'est bien sûr pas nécessaire et le mieux reste encore de rediriger vers des pages `200.html / 404.html`.

`fallback: 'pageDeFallback.html'`

Note: De nombreux services (ex: Netlify) détectent une `404.html` automatiquement. Si on est en charge de la configuration de notre propre serveur web, il faudra consulter la documentation pour trouver comment définir une page d'erreur (et la passer à `404.html`).

interval

- Type: `Number`
- Par défaut: `0`

L'intervalle entre 2 cycles de render, utile pour éviter de spam une API avec les appels de notre application web.

minify

- **Déprécié !**
- Il faut utiliser `build.html.minify` à la place.

routes

- Type: `Array`

Info

Depuis Nuxt v2.13, un crawler est installé. Ce dernier se charge de trouver tous les liens et de générer les routes correspondantes lors d'un `nuxt generate`.

Si on a détaché des pages (comme des pages secrètes) mais que l'on veut que celles-ci soient aussi générées, on peut utiliser la propriété `generate.routes`.

Warning

Les routes dynamiques seront ignorées par la commande `generate` lors de l'utilisation d'une version de Nuxt inférieure à `2.12`.

Exemple :

```
-| pages/
---| index.vue
---| users/
-----| _id.vue
```

Seule la route `/` sera générée par Nuxt.

Si l'on souhaite que Nuxt génère des routes avec des paramètres dynamiques, on a besoin de définir la propriété `generate.routes` à un tableau de routes dynamiques.

On ajoute les routes pour `/users/:id` :

```
export default {
  generate: {
    routes: ['/users/1', '/users/2', '/users/3']
  }
}
```

Ensuite on lance `nuxt generate` :

```
[nuxt] Generating...
[...]
nuxt:render Rendering url / +154ms
nuxt:render Rendering url /users/1 +12ms
nuxt:render Rendering url /users/2 +33ms
nuxt:render Rendering url /users/3 +7ms
nuxt:generate Generate file: /index.html +21ms
nuxt:generate Generate file: /users/1/index.html +31ms
nuxt:generate Generate file: /users/2/index.html +15ms
nuxt:generate Generate file: /users/3/index.html +23ms
nuxt:generate HTML Files generated in 7.6s +6ms
[nuxt] Generate done
```

Bien, mais que se passe-t-il si on a des **paramètres dynamiques** ?

1. On peut utiliser une `fonction` qui nous retournera une `Promesse`.
2. On peut utiliser une `fonction` avec un `callback(erreur, parametres)`.

Fonction qui retourne une Promesse

```
import axios from 'axios'

export default {
  generate: {
    routes() {
      return axios.get('https://mon-api/users').then(res => {
        return res.data.map(user => {
          return '/users/' + user.id
        })
      })
    }
  }
}
```

Fonction avec un callback

```
import axios from 'axios'

export default {
  generate: {
    routes(callback) {
      axios
        .get('https://mon-api/users')
        .then(res => {
          const routes = res.data.map(user => {
            return '/users/' + user.id
          })
          callback(null, routes)
        })
        .catch(callback)
    }
  }
}
```

Accélérer la génération des routes dynamiques avec payload

Dans l'exemple précédent, nous utilisons `user.id` à partir du serveur pour générer les routes mais on jette le reste de la data. En général, on aura besoin de le récupérer de nouveau à l'intérieur de `/users/_id.vue`. Même si on peut faire cela, on aura certainement besoin de passer la valeur de `generate.interval` à quelque chose comme `100` pour éviter de spam le serveur avec nos appels. Parce que cela va augmenter le temps que prend le script de génération, il serait préférable de passer l'objet `user` en entier dans le contexte de `_id.vue`. Nous pouvons faire cela en modifiant le code au dessus par :

```
import axios from 'axios'

export default {
  generate: {
    routes() {
      return axios.get('https://mon-api/users').then(res => {
        return res.data.map(user => {
          return {
            route: '/users/' + user.id,
            payload: user
          }
        })
      })
    }
  }
}
```

```

        })
    }
}

```

Nous pouvons maintenant accéder à `payload` à partir de `/users/_id.vue` comme ceci :

```

async asyncData ({ params, error, payload }) {
  if (payload) return { user: payload }
  else return { user: await backend.fetchUser(params.id) }
}

```

subFolders

- Type: `Boolean`
- Par défaut: `true`

Par défaut, lorsqu'on lance un `nuxt generate`, Nuxt va créer un répertoire pour chaque route et servir un fichier `index.html`.

Exemple :

```

-| dist/
---| index.html
---| about/
-----| index.html
---| products/
-----| item/
| index.html

```

Lorsque l'on passe la valeur à `false`, les fichiers HTML seront générés d'après le chemin de la route :

```

export default {
  generate: {
    subFolders: false
  }
}

```

```

-| dist/
---| index.html
---| about.html
---| products/
-----| item.html

```

[Go to TOC](#)

La propriété globalName

Nuxt nous permet de personnaliser l'identifiant global utilisé dans le template HTML principal ainsi que le nom de l'instance de Vue et d'autres options encore.

- Type: `String`
- Par défaut: `nuxt`

```
{
  globalName: 'monNomPersonnalisé'
}
```

Warning

Le `globalName` doit être un identifiant JavaScript valide, le changer pourrait casser le support pour certains plugins qui dépendent de certaines fonctions basées sur le nom de Nuxt. Si l'on cherche à changer seulement l'identifiant HTML visible `__nuxt`, alors on peut utiliser la propriété `globals`.

The globals property

Personnalise certains noms globaux spécifiques, basés sur `globalName` par défaut.

- Type: `Object`
- Par défaut:

```
globals: {
  id: globalName => `__${globalName}`,
  nuxt: globalName => `__$${globalName}`,
  context: globalName => `__${globalName.toUpperCase()}__`,
  pluginPrefix: globalName => globalName,
  readyCallback: globalName => `on${_.capitalize(globalName)}Ready`,
  loadedCallback: globalName => `on${_.capitalize(globalName)}Loaded`
},
```

La Propriété head

Nuxt vous permet de définir toutes les métadonnées par défaut pour votre application dans le fichier `nuxt.config.js`.

- Type: `Object` or `Function`

```
export default {
  head: {
    titleTemplate: '%s - Nuxt',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },

      // hid is used as unique identifier. Do not use `vmid` for it as it will not
      work
      { hid: 'description', name: 'description', content: 'Meta description' }
    ]
  }
}
```

Pour connaître la liste des options que vous pouvez donner à `head`, consultez la documentation [vue-meta](#).

Vous pouvez également utiliser `head` comme fonction dans vos composants pour accéder aux données des composants avec `this` ([en savoir plus](#)).

Info

Pour éviter la duplication des métadonnées dans le composant enfant, configurez un identifiant unique avec la clé `hid` pour vos balises méta ([en savoir plus](#)).

[Go to TOC](#)

La propriété hooks

Les hooks sont des écouteurs aux événements de Nuxt qui sont généralement utilisés dans les modules de Nuxt, mais aussi disponibles dans le fichier `nuxt.config.js`.

- Type: `Object`

```
import fs from 'fs'
import path from 'path'

export default {
  hooks: {
    build: {
      done(builder) {
        const extraFilePath = path.join(
          builder.nuxt.options.buildDir,
          'fichier-supplémentaire'
        )
        fs.writeFileSync(extraFilePath, 'Quelque chose de spécial')
      }
    }
  }
}
```

Le fonctionnement des hooks suit un modèle de nommage avec des `:` (ex: `build:done`). Pour des raisons de simplicité de configuration, on peut les structurer de manière hiérarchique pour ses propres hooks en utilisant un objet (cf. code juste au dessus). Se référer au [fonctionnement interne de Nuxt](#) pour davantage d'informations.

Liste des hooks

- [Nuxt hooks](#)
- [Renderer hooks](#)
- [ModulesContainer hooks](#)
- [Builder hooks](#)
- [Generator hooks](#)

Exemples

Rediriger au `router.base` lorsque l'on n'est pas sur `root`

Imaginons que l'on souhaite servir les pages sur `/portal` au lieu de `/`.

C'est peut être un cas particulier, mais le but d'un `router.base` dans `nuxt.config.js` est de permettre à un serveur Web de servir une application Nuxt lorsque son point d'entrée n'est pas le domaine principal.

Ceci dit, lorsque l'on est en développement, aller sur `localhost`, lorsque `router.base` n'est défini à `/` retournera une 404. Pour prévenir cela, on peut configurer un hook.

La redirection n'est peut-être pas le meilleur cas d'usage pour une application Web en production, mais cela permettra de pratiquer les hooks.

Pour commencer, on peut changer le `router.base` dans le fichier `nuxt.config.js` :

```
import hooks from './hooks'
export default {
  router: {
    base: '/portal'
  }
  hooks: hooks(this)
}
```

Ensuite, nous allons créer quelques nouveaux fichiers:

1. `hooks/index.js`, module des hooks

```
import render from './render'

export default nuxtConfig => ({
  render: render(nuxtConfig)
})
```

2. `hooks/render.js`, render du hook

```
import redirectRootToPortal from './route-redirect-portal'

export default nuxtConfig => {
  const router = Reflect.has(nuxtConfig, 'router') ? nuxtConfig.router : {}
  const base = Reflect.has(router, 'base') ? router.base : '/portal'

  return {
    /**
     * 'render:setupMiddleware'
     * {@link node_modules/nuxt/lib/core/renderer.js}
     */
    setupMiddleware(app) {
      app.use('/', redirectRootToPortal(base))
    }
  }
}
```

3. `hooks/route-redirect-portal.js`, le middleware en question

```
/**
 * Hook de middleware de Nuxt pour rediriger de `/` à `/portal` (ou toute autre
 valeur attribuée à `router.base` dans `nuxt.config.js`)
 *
 * Doit avoir la même version que `connect`
 * {@link node_modules/connect/package.json}
 */
import parseurl from 'parseurl'

/**
 * On branche le middleware pour gérer les redirections au contexte principal de
 l'application Web.
 *
 * La documentation de Nuxt manque d'explications sur l'usage des hooks.

```

```

* Ceci est un exemple de router afin d'aider à l'expliquer.
*
* On peut regarder d'autres implémentations en tant qu'inspiration:
* - https://github.com/nuxt/nuxt.js/blob/dev/examples/with-
cookies/plugins/cookies.js
* - https://github.com/yyx990803/launch-editor/blob/master/packages/launch-
editor-middleware/index.js
*
* [http_class_http_clientrequest]:
https://nodejs.org/api/http.html#http_class_http_clientrequest
* [http_class_http_serverresponse]:
https://nodejs.org/api/http.html#http_class_http_serverresponse
*
* @param {http.ClientRequest} req objet de la requête du client interne de
Node.js [http_class_http_clientrequest]
* @param {http.ServerResponse} res Réponse interne de Node.js
[http_class_http_serverresponse]
* @param {Function} next callback du middleware
*/
export default desiredContextRoot =>
  function projectHooksRouteRedirectPortal(req, res, next) {
    const desiredContextRootRegExp = new RegExp(`^${desiredContextRoot}`)
    const _parsedUrl = Reflect.has(req, '_parsedUrl') ? req._parsedUrl : null
    const url = _parsedUrl !== null ? _parsedUrl : parseurl(req)
    const startsWithDesired = desiredContextRootRegExp.test(url.pathname)
    const isNotProperContextRoot = desiredContextRoot !== url.pathname
    if (isNotProperContextRoot && startsWithDesired === false) {
      const pathname = url.pathname === null ? '' : url.pathname
      const search = url.search === null ? '' : url.search
      const Location = desiredContextRoot + pathname + search
      res.writeHead(302, {
        Location
      })
      res.end()
    }
    next()
  }
}

```

Ainsi, lorsqu'un collègue accédera accidentellement à `/` en développement, Nuxt le redirigera automatiquement vers `/portal`.

La propriété ignore

Définit les fichiers à ignorer pour l'application Nuxt

.nuxtignore

On peut utiliser un fichier `.nuxtignore` pour dire à Nuxt d'ignorer des fichiers `layout`, `page`, `store` et `middleware` à la racine du projet (`rootDir`) durant la phase de build. Le fichier `.nuxtignore` est sujet à la même spécification que les fichiers `.gitignore` et `.eslintignore`, dans lesquels chaque ligne est un *glob pattern* indiquant quels fichiers devraient être ignorés.

Par exemple:

```
# ignore le layout foo.vue
layouts/foo.vue
# ignore les fichiers layout dont le nom finit par un `*-ignore.vue`
layouts/*-ignore.vue

# ignore la page bar.vue
pages/bar.vue
# ignore la page à l'intérieur du répertoire ignore
pages/ignore/*.vue

# ignore le store baz.js
store/baz.js
# ignore les fichiers store qui match *.test.*
store/ignore/*.test.*

# ignore les fichiers middleware dans le répertoire foo sauf foo/bar.js
middleware/foo/*.js
!middleware/foo/bar.js
```

Plus de détails peuvent être trouvés dans la documentation du [.gitignore](#).

La propriété ignorePrefix

- Type: `String`
- Par défaut: `'-'`

N'importe quel fichier dans `pages/`, `layouts/`, `middleware/` ou `store/` sera ignoré durant la phase de build si le nom du fichier commence par le préfixe spécifié par `ignorePrefix`.

Par défaut, tous les fichiers qui commencent par un `_` seront ignorés, tels que `store/-foo.js` et `pages/-bar.vue`. Cela permet de pouvoir garder des tests, utilitaires et composants avec leur appelants sans qu'ils ne soient eux même convertis en routes, stores, etc...

La propriété ignore

- Type: `Array`
- Par défaut: `['**/*.test.*']`

Encore plus personnalisable que `ignorePrefix`: tous les fichiers qui match les *glob patterns* spécifiés dans `ignore` seront ignorés lors de la phase de build.

ignoreOptions

`nuxtignore` utilise `node-ignore` de manière implicite, `ignoreOptions` peut donc être configuré comme les `options` dans `node-ignore`.

```
export default {
  ignoreOptions: {
    ignorecase: false
  }
}
```

La propriété loading

Nuxt utilise son propre composant pour montrer une barre de progression lors de la navigation entre les routes. On peut le personnaliser, le désactiver ou même créer le notre.

- Type: `Boolean` ou `Object` ou `String`

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()

      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Désactiver la barre de progression

- Type: `Boolean`

```
export default {
  loading: false
}
```

Personnaliser la barre de progression

- Type: `Object`

```
export default {
  loading: {
    color: 'blue',
    height: '5px'
  }
}
```

Voici la liste des propriétés que vous pouvez utiliser pour personnaliser la barre de progression.

Propriété	Type	Défaut	Description
<code>color</code>	<code>String</code>	<code>'black'</code>	Couleur CSS de la barre de progression
<code>failedColor</code>	<code>String</code>	<code>'red'</code>	Couleur CSS de la barre de progression lors d'une erreur lors du render d'une route (ex: de la <code>data</code> ou un <code>fetch</code> ont échoué).
<code>height</code>	<code>String</code>	<code>'2px'</code>	Hauteur de la barre de progression (utilisé dans la propriété <code>style</code> de la barre de progression)
<code>throttle</code>	<code>Number</code>	<code>200</code>	En millisecondes, le temps à attendre avant que la barre de navigation ne soit affichée. Utile pour éviter de créer des flashes.
<code>duration</code>	<code>Number</code>	<code>5000</code>	En millisecondes, la durée maximum de la barre de progression, Nuxt assume que la route sera render en moins de 5 secondes.
<code>continuous</code>	<code>Boolean</code>	<code>false</code>	Continuer d'animer la barre de progression lorsque le chargement prends plus de temps que la valeur de la propriété <code>duration</code> .
<code>css</code>	<code>Boolean</code>	<code>true</code>	Définir à <code>false</code> pour enlever tous les styles appliqués à la barre de progression (y compris les vôtres).
<code>rtl</code>	<code>Boolean</code>	<code>false</code>	Définit la direction de la barre de progression, de droite à gauche.

Utiliser un composant de chargement personnalisé

- Type: `String`

Votre composant devra fournir certaines des méthodes suivantes:

| Méthode | Requis | Description | | | `start()` | Requis | Appelé lors d'un changement de route, c'est à ce moment là que vous affichez votre composant. | | `finish()` | Requis | Appelé lorsqu'une route est chargée (et la data collectée), c'est à ce moment là que vous cachez votre composant. | | `fail(error)` | Optionnel | Appelé lorsqu'une de vos routes n'a pas pu être chargée (ex: le chargement de la data a échoué) | | `increase(num)` | Optionnel | Appelé durant le chargement du composant de la route, `num` est un `Integer` < 100 . |

```
<template lang="html">
  <div class="loading-page" v-if="loading">
    <p>Chargement...</p>
  </div>
</template>

<script>
  export default {
    data: () => ({
      loading: false
    }),
    methods: {
      start() {
        this.loading = true
      },
      finish() {
        this.loading = false
      }
    }
  }
</script>

<style scoped>
  .loading-page {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(255, 255, 255, 0.8);
    text-align: center;
    padding-top: 200px;
    font-size: 30px;
    font-family: sans-serif;
  }
</style>
```

```
export default {
  loading: '~/components/loading.vue'
}
```

La propriété loading indicator

Affiche des indications de chargement originaux lorsque la page SPA est en train de charger !

Lorsque l'on exécute Nuxt en mode SPA, il n'y a pas de contenu du côté serveur lors du chargement initial de la page. Donc, au lieu d'afficher une page blanche lorsque l'application charge, on pourrait afficher un spinner.

Cette propriété peut prendre 3 types différents: `string` ou `false` ou `object`. Si une chaîne de caractères est fournie, elle sera convertie en objet.

La valeur par défaut est:

```
loadingIndicator: {
  name: 'circle',
  color: '#3B8070',
  background: 'white'
}
```

Indicateurs embarqués

Ces indicateurs sont importés depuis le projet génial qu'est [Spinkit](#). On peut se référer à sa page de démo pour prévisualiser les spinners.

- circle
- cube-grid
- fading-circle
- folding-cube
- chasing-dots
- nuxt
- pulse
- rectangle-bounce
- rotating-plane
- three-bounce
- wandering-cubes

Built-in indicators support `color` and `background` options.

Indicateurs personnalisés

Si vous avez besoin de votre propre indicateur spécial, une chaîne de caractères ou un nom de clé peut aussi être le chemin vers le code source d'un template HTML. Toutes les options seront aussi passées au template.

Le [code source](#) de celui de Nuxt est aussi disponible si l'on souhaite avoir une base !

La propriété mode

Change le mode par défaut de Nuxt

- Type: `string`
 - Par défaut: `universal`
 - Valeurs possibles:
 - `'spa'` : Pas de rendu côté serveur (seulement de la navigation côté client)
 - `'universal'` : Application isomorphique (rendu côté serveur + navigation côté client)

On peut utiliser cette option pour changer le mode par défaut de Nuxt en utilisant le fichier `nuxt.config.js`

Warning

Déprécié: il faut utiliser `ssr: false` au lieu de `mode: spa`.

Next

Pour en savoir davantage sur l'option `SSR`, il faut se référer à la [propriété ssr](#).

Next

Pour en savoir davantage sur l'option `mode`, il faut se référer à la section sur les [modes de render](#).

[Go to TOC](#)

La propriété modern

Build et sert un bundle moderne

Cette fonctionnalité est inspirée du mode `modern` de `vue-cli`.

- Type: `String` ou `Boolean`
 - Par défaut: `false`
 - Valeurs possibles:
 - `'client'` : Sert les deux scripts, le bundle moderne `<script type="module">` et le bundle legacy `<script nomodule>`, fournit aussi un `<link rel="modulepreload">` pour le bundle moderne. Chaque navigateur qui comprend le type `module` chargera le bundle moderne pendant que les navigateurs plus anciens utiliseront le bundle legacy de secours (ce dernier sera transpilé).
 - `'server'` ou `true` : Le serveur Node.js va vérifier la version du navigateur grâce au *user agent* et servira le bundle moderne ou de secours en fonction.
 - `false` : désactive le build moderne.

Les deux bundles possibles sont:

1. Bundle moderne: vise les navigateurs modernes qui supportent les *ES modules*
2. Bundle legacy (de secours): vise les navigateurs plus anciens en fonction de la configuration de Babel (compatible IE9 par défaut).

Info:

- On peut utiliser la commande `--modern | -m=[mode]` pour build/démarrer les bundles modernes:

```
{
  "scripts": {
    "build:modern": "nuxt build --modern=server",
    "start:modern": "nuxt start --modern=server"
  }
}
```

Information à propos de `nuxt generate`: La propriété `modern` marche aussi avec la commande `nuxt generate`, mais dans ce cas là, seulement l'option `client` est honorée et sera sélectionnée lors de l'exécution de la commande `nuxt generate --modern` sans fournir d'autres valeurs.

- Nuxt va automatiquement détecter le build `moderne` dans `nuxt start` lorsque `modern` n'est pas spécifié, le mode d'auto-détection est le suivant:

srr
true

srr

false

- Le mode `moderne` pour `nuxt generate` ne peut être que `client`
- Il faut utiliser `render.crossorigin` pour définir l'attribut `crossorigin` dans `<link>` et `<script>`.

Se référer à l'excellent post de [Phillip Walton](#) sur les builds modernes.

La propriété build

Nuxt nous permet de personnaliser la configuration de webpack afin de build notre application Web comme on l'entend.

analyze

Nuxt utilise [webpack-bundle-analyzer](#) pour nous permettre de visualiser les bundles et trouver comment les optimiser.

- Type: `Boolean` ou `Object`
- Par défaut: `false`

Si c'est un objet, les propriétés disponibles sont visibles [ici](#).

```
export default {
  build: {
    analyze: true,
    // ou
    analyze: {
      analyzerMode: 'static'
    }
  }
}
```

Info

Info: on peut utiliser la commande `yarn nuxt build --analyze` ou bien `yarn nuxt build -a` pour build notre application et lancer l'analyseur des bundles sur <http://localhost:8888>. Si on n'utilise pas `yarn`, on peut exécuter la commande avec `npx`.

corejs

Depuis [Nuxt@2.14](#), Nuxt va automatiquement détecter la version actuelle de `core-js` dans notre projet. On peut bien sûr aussi lui spécifier la version que l'on veut utiliser.

- Type: `number | string` (Les valeurs disponibles sont `'auto'`, `2` et `3`)
- Default: `'auto'`

babel

Permet de personnaliser la configuration Babel pour le JavaScript et les fichiers Vue. `.babelrc` est ignoré par défaut.

- Type: `Object`
- Voir les `options` de `babel-loader` et les `options` de `babel`.
- Par défaut:

```
{
  babelrc: false,
  cacheDirectory: undefined,
  presets: ['@nuxt/babel-preset-app']
}
```

Les cibles par défaut de `@nuxt/babel-preset-app` sont `ie: '9'` dans le build `client`, et `node: 'current'` dans le build `serveur`.

presets

- Type: `Function`
- Argument:
 1. `Object` : `isServer: true`
`false`
 2. `Array` :
 - nom du preset `@nuxt/babel-preset-app`
 - `options` de `@nuxt/babel-preset-app`

Note: Les presets configurés dans `build.babel.presets` seront appliqués tant au build du client que celui du serveur. La cible sera choisie par Nuxt en conséquence (client/serveur). Si on veut configurer le preset différemment, il faut utiliser `presets` en tant que fonction:

Nous recommandons **très chaudement** d'utiliser la configuration par défaut au lieu du code plus bas

```
export default {
  build: {
    babel: {
      presets({ isServer }, [ preset, options ]) {
        // on change directement les options
        options.targets = isServer ? ... : ...
        options.corejs = ...
        // renvoie rien
      }
    }
  }
}
```

On peut aussi écraser les valeurs par défaut en retournant la liste complète des presets:

```
export default {
  build: {
    babel: {
      presets({ isServer }, [preset, options]) {
        return [
          [
            preset,
            {
              buildTarget: isServer ? ... : ...,
              ...options
            }
          ],
          [
            // d'autres presets
          ]
        ]
      }
    }
  }
}
```

cache

- Type: `Boolean`
- Par défaut: `false`
- Δ Expérimental

Active le cache de [terser-webpack-plugin](#) et de [cache-loader](#)

cssSourceMap

- Type: `boolean`
- Par défaut: `true` pour le développement et `false` pour la production.

Active le support du CSS Source Map

devMiddleware

- Type: `Object`

Voir [webpack-dev-middleware](#) pour les options disponibles.

devtools

- Type: `boolean`

- Par défaut: `false`

Définit le droit d'afficher l'inspection des [vue-devtools](#).

Si cela a déjà été activé grâce au fichier `nuxt.config.js` ou autrement, les devtools seront activés, peu importe le flag.

extend

Permet de personnaliser manuellement la configuration de Webpack pour les bundles client et serveur.

- Type: `Function`

`extend` est appelé deux fois, une fois pour le bundle du serveur, et une fois pour celui du client. Les arguments de la méthode sont:

1. L'objet de configuration Webpack
2. L'objet avec les clés suivantes (tous sont des booléens sauf `loaders`): `isDev`, `isClient`, `isServer` et `loaders`.

Warning

Attention: Les clés `isClient` et `isServer` fournies n'ont rien à voir avec celles présentes dans le `context`. Elles ne sont **pas** dépréciées. Il ne faut en outre pas utiliser `process.client` et `process.server` ici car ils seront `undefined` à ce niveau.

```
export default {
  build: {
    extend(config, { isClient }) {
      // personnalisation de la configuration Webpack seulement pour le bundle
      // client
      if (isClient) {
        config.devtool = 'source-map'
      }
    }
  }
}
```

Si on souhaite en savoir davantage sur la configuration Webpack par défaut, il faut aller jeter un coup d'œil au [répertoire de Webpack](#).

Les loaders dans extend

Les `loaders` ont la même structure au niveau de l'objet que `build.loaders`, on peut donc changer les options des `loaders` à l'intérieur d' `extend` .

```
export default {
  build: {
    extend(config, { isClient, loaders: { vue } }) {
      // personnalisation de la configuration Webpack seulement pour le bundle
      // client
    }
  }
}
```

```

client
  if (isClient) {
    vue.transformAssetUrls.video = ['src', 'poster']
  }
}
}
}

```

extractCSS

Permet le Common CSS Extraction en utilisant les [directives](#) du package Vue Server Renderer .

- Type: `Boolean` ou `Object`
- Par défaut: `false`

L'usage de `extract-css-chunks-webpack-plugin` se fait de manière implicite, en séparant notre CSS dans des fichiers séparés, un par composant en général. Cela permet de mettre en cache notre CSS et notre JavaScript séparément et est sans doute quelque chose à essayer dans le cas où on possède beaucoup de CSS global ou partagé.

Exemple (`nuxt.config.js`):

```

export default {
  build: {
    extractCSS: true,
    // ou
    extractCSS: {
      ignoreOrder: true
    }
  }
}

```

Info

Note: Il y avait un bug avant Vue 2.5.18 qui enlevait les imports critiques de CSS lors de l'usage de cette option.

On pourrait souhaiter extraire tout notre CSS dans un seul fichier. Il y a une solution pour cela:

Warning

Ce n'est pas recommandé de tout extraire dans un seul fichier. Extraire dans plusieurs fichiers CSS est meilleur pour la mise en cache et l'isolation dans le cas d'un préchargement. Cela peut aussi améliorer la performance de la page en ne téléchargeant que les ressources dont on a besoin.

```

export default {
  build: {
    extractCSS: true,
    optimization: {
      splitChunks: {
        cacheGroups: {
          styles: {

```

```

        name: 'styles',
        test: /\.(css|vue)$/,
        chunks: 'all',
        enforce: true
    }
}
}
}
}
}
```

filenames

Permet de personnaliser le nom des fichiers des bundles.

- Type: `Object`
- Par défaut:

```
{
  app: ({ isDev, isModern }) => isDev ? `[_name]${isModern ? '.modern' : ''}.js` :
  : `[contenthash:7]${isModern ? '.modern' : ''}.js`,
  chunk: ({ isDev, isModern }) => isDev ? `[_name]${isModern ? '.modern' : ''}.js` :
  : `[contenthash:7]${isModern ? '.modern' : ''}.js`,
  css: ({ isDev }) => isDev ? '[name].css' : 'css/[contenthash:7].css',
  img: ({ isDev }) => isDev ? '[path][name].[ext]' : 'img/[name].
  [contenthash:7].[ext]',
  font: ({ isDev }) => isDev ? '[path][name].[ext]' : 'fonts/[name].
  [contenthash:7].[ext]',
  video: ({ isDev }) => isDev ? '[path][name].[ext]' : 'videos/[name].
  [contenthash:7].[ext]'
}
```

Cet exemple change les noms originaux des fragments en des identifiants numériques:

```

export default {
  build: {
    filenames: {
      chunk: ({ isDev }) => (isDev ? '[name].js' : '[id].[contenthash].js')
    }
  }
}
```

Pour comprendre un peu mieux l'utilisation des `manifests`, il faut se référer à la [documentation de Webpack](#).

Warning

Il faut bien faire attention lorsque l'on utilise des noms de fichiers non hashés en production car la plupart des navigateurs vont mettre en cache la ressource et ne détecteront pas les modifications lors du premier chargement.

friendlyErrors

- Type: `Boolean`
- Par défaut: `true` (Affichage sympa activé)

Active ou désactive les indications claires et précises fournies par [FriendlyErrorsWebpackPlugin](#).

hardSource

- Type: `Boolean`
- Par défaut: `false`
- Δ Expérimental

Active le plugin [HardSourceWebpackPlugin](#) pour une mise en cache améliorée.

hotMiddleware

- Type: `Object`

Se référer à [webpack-hot-middleware](#) pour les options disponibles.

html.minify

- Type: `Object`
- Par défaut:

```
{
  collapseBooleanAttributes: true,
  decodeEntities: true,
  minifyCSS: true,
  minifyJS: true,
  processConditionalComments: true,
  removeEmptyAttributes: true,
  removeRedundantAttributes: true,
  trimCustomFragments: true,
  useShortDoctype: true
}
```

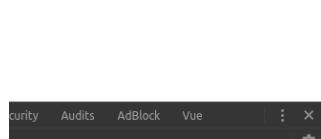
Attention: si l'on fait des changements à `html.minify`, il ne seront pas fusionnés avec les valeurs par défaut !

La configuration pour le plugin [html-minifier](#) est là pour minifier les fichiers HTML créés durant le processus de build (la minification sera appliquée pour *tous les modes*).

indicator

Affiche un indicateur de build pour le Hot Module Replacement (HMR) lors du développement. Disponible depuis la `v2.8.0`.

- Type: Boolean
- Par défaut: true



nuxt-build-indicator

loaders

Permet de personnaliser les options de Nuxt par rapport aux loaders intégrés.

- Type: Object
- Par défaut:

```
{
  file: {},
  fontUrl: { limit: 1000 },
  imgUrl: { limit: 1000 },
  pugPlain: {},
  vue: {
    transformAssetUrls: {
      video: 'src',
      source: 'src',
      object: 'src',
      embed: 'src'
    }
  },
  css: {},
  cssModules: {
    localIdentName: '[local]_[hash:base64:5]'
  },
  less: {},
  sass: {
    indentedSyntax: true
  },
  scss: {},
  stylus: {},
  vueStyle: {}
}
```

Note: en plus de spécifier la configuration dans un fichier `nuxt.config.js`, cela peut aussi être modifié par `build.extend`.

loaders.file

Davantage de détails dans les [options de file-loader](#).

loaders.fontUrl and loaders imgUrl

Davantage de détails dans les [options d'url-loader](#).

loaders.pugPlain

Davantage de détails dans le [pug-plain-loader](#) ou [les options du compilateur Pug](#).

loaders.vue

Davantage de détails dans les [options de vue-loader](#).

loaders.css and loaders.cssModules

Davantage de détails dans les [options du css-loader](#).

Note: les [CSS modules](#) sont des options pour le loader.

loaders.less

On peut passer des options spécifique à Less au `less-loader` via `loaders.less`. Se référer à la documentation de [Less](#) pour toutes les options disponibles.

loaders.sass et loaders.scss

Se référer à la documentation de [Node Sass](#) pour toutes les options disponibles de Sass.

Note: `loaders.sass` est utilisé pour la [Syntaxe Indentée de Sass](#).

loaders.vueStyle

Davantage de détails dans les options de [vue-style-loader](#).

optimization

- Type: `Object`
- Par défaut:

```
{
  minimize: true,
  minimizer: [
    // terser-webpack-plugin
    // optimize-css-assets-webpack-plugin
  ],
  splitChunks: {
    chunks: 'all',
    automaticNameDelimiter: '.',
    name: undefined,
    cacheGroups: {}
  }
}
```

La valeur par défaut de `splitChunks.name` est `true` dans le mode `dev` ou bien `analyze`.

On peut définir `minimizer` à un tableau personnalisé de plugins ou définir `minimize` à `false` pour désactiver tous les minimiseurs. (`minimize` est désactivé lors du développement par défaut)

Se référer à la documentation sur [l'optimisation de Webpack](#).

optimizeCSS

- Type: `Object` ou `Boolean`
- Par défaut:
 - `false`
 - `{}` lorsque `extractCSS` est activé

Les options du plugin `OptimizeCSSAssets`.

Se référer à [NMFR/optimize-css-assets-webpack-plugin](#).

parallel

- Type: `Boolean`
- Par défaut: `false`
- Δ Expérimental

Active le [thread-loader](#) lors du build de Webpack.

plugins

Ajoute des plugins Webpack

- Type: `Array`
- Par défaut: `[]`

```
import webpack from 'webpack'
import { version } from './package.json'
export default {
  build: {
    plugins: [
      new webpack.DefinePlugin({
        'process.VERSION': version
      })
    ]
  }
}
```

postcss

Permet de personnaliser les plugins du [PostCSS Loader](#).

- Type: `Array` (legacy, écrasera les valeurs par défaut), `Object` (recommandé), `Function` ou `Boolean`

Note: Nuxt a appliqué [PostCSS Preset Env](#). Par défaut, cela active les [fonctionnalités du niveau 2](#) et l'[Auto-prefixer](#), on peut utiliser `build.postcss.preset` pour le configurer.

- Par défaut:

```
{
  plugins: {
    'postcss-import': {},
    'postcss-url': {},
    'postcss-preset-env': this.preset,
    'cssnano': { preset: 'default' } // désactivé lors du développement
  },
  order: 'presetEnvAndCssnanoLast',
  preset: {
    stage: 2
  }
}
```

Les paramètres personnalisés du plugin seront fusionnés avec les valeurs par défaut des plugins (à moins qu'on n'utilise un `Array` au lieu d'un `Object`).

```
export default {
  build: {
    postcss: {
      plugins: {
        // désactive `postcss-url`
        'postcss-url': false,
        // ajouter quelques plugins
        'postcss-nested': {},
        'postcss-responsive-type': {},
        'postcss-hexrgba': {}
      },
      preset: {
        autoprefixer: {
          grid: true
        }
      }
    }
  }
}
```

Si la configuration postCSS est un `Object`, `order` peut être utilisé pour définir l'ordre du plugin:

- Type: `Array` (nom des plugins rangés), `String` (rangé selon le nom du preset), `Function`
- Par défaut: `cssnanoLast` (`cssnano` est en dernier)

```
export default {
  build: {
    postcss: {
      // nom du preset
      order: 'cssnanoLast',
      // nom des plugins rangés
      order: ['postcss-import', 'postcss-preset-env', 'cssnano']
      // fonction pour déterminer l'ordre du plugin
      order: (names, presets) => presets.cssnanoLast(names)
    }
  }
}
```

plugins postcss et nuxt-tailwindcss

Si l'on souhaite appliquer un plugin postCSS (ex: `postcss-pxtorem`) sur une configuration de `nuxt-tailwindcss`, on doit changer l'ordre et charger `tailwindcss` en premier.

Cette configuration n'a pas d'impact sur `nuxt-purgecss`.

```
import { join } from 'path'

export default {
  // ...
  build: {
    postcss: {
      plugins: {
        tailwindcss: join(__dirname, 'tailwind.config.js'),
        'postcss-pxtorem': {
          propList: ['*', '!border*']
        }
      }
    }
  }
}
```

```

        }
    }
}
```

profile

- Type: `Boolean`
- Par défaut: activé avec l'argument `--profile` dans la ligne de commande

Active le profiler dans [WebpackBar](#)

publicPath

Nuxt nous permet de téléverser les fichiers présents dans notre répertoire `dist` jusqu'à notre CDN pour des performances maximales, il suffit de définir l'URL de notre CDN dans `publicPath`.

- Type: `String`
- Default: `'/_nuxt/'`

```

export default {
  build: {
    publicPath: 'https://cdn.nuxtjs.org'
  }
}
```

Ensuite, lorsque on lance `nuxt build`, il suffit de téléverser le contenu du répertoire `.nuxt/dist/client` dans notre CDN et voilà !

quiet

Supprime la majeure partie de la journalisation des builds

- Type: `Boolean`
- Par défaut: Activé lorsque l'environnement détecté par `std-env` est égal à `CI` ou `test`.

splitChunks

- Type: `Object`
- Par défaut:

```
export default {
  build: {
    splitChunks: {
      layouts: false,
      pages: true,
      commons: true
    }
  }
}
```

Si séparation de code pour `layout`, `pages` et `commons` (bibliothèques communes: vue|vue-loader|vue-router|vuex...).

ssr

Crée un bundle Webpack spécial pour le rendu côté serveur.

- Type: `Boolean`
- Par défaut: `true` pour le mode universel et `false` pour notre soirée SPA

Cette option est automatiquement définie en se basant sur la valeur de `mode` si rien n'est fourni.

standalone

Dépendances du groupe de serveurs en ligne (avancé)

- Type: `Boolean`
- Default: `false`

Ce mode regroupe des `node_modules` qui sont normalement conservés en tant qu'externes dans la construction du serveur ([plus d'informations](#)).

Warning

Les dépendances d'exécution (modules, `nuxt.config`, middleware du serveur et répertoire statique) ne sont pas regroupées. Cette fonctionnalité désactive uniquement l'utilisation de `webpack-externes` for server-bundle.

Info

Vous pouvez utiliser la commande `yarn nuxt build --standalone` pour activer ce mode sur la ligne de commande. (Si vous n'utilisez pas `yarn`, vous pouvez exécuter la commande avec `npx .`)

styleResources

- Type: `Object`

- Par défaut: `{}`

Warning

Attention: cette propriété est dépréciée. Il faudrait utiliser le module [style-resources-module](#) à la place, pour avoir des performances accrues et une meilleure DX !

Ceci est utile lorsque on a besoin d'injecter certains variables ou mixins à l'intérieur de nos pages sans avoir besoin de les importer à chaque fois.

Nuxt utilise <https://github.com/yenshih/style-resources-loader> pour réussir ce comportement.

Il y a besoin de spécifier un schéma/un chemin si on veut la partager pour les pre-processors nuls et ajoute des: `less`, `sass`, `scss` ou `stylus`.

On ne peut pas utiliser d'alias de chemin (~ ou @), on aura besoin d'utiliser des chemins relatifs ou absolus.

```
{
  build: {
    styleResources: {
      scss: './assets/variables.scss',
      less: './assets/*.less',
      // sass: ....,
      // scss: ...
      options: {
        // Voir https://github.com/yenshih/style-resources-loader#options
        // Sauf la propriété `patterns`
      }
    }
  }
}
```

templates

Nuxt nous permet d'utiliser nos propres templates qui seront render basés sur les [modules](#).

- Type: `Array`

```
export default {
  build: {
    templates: [
      {
        src: '~/modules/support/plugin.js', // `src` peut être relatif ou absolu
        dst: 'support.js', // `dst` est relative au project `.nuxt` dir
        options: {
          // Les options sont passés aux templates grâce à la clé `options`
          live_chat: false
        }
      }
    ]
  }
}
```

Les templates sont render en utilisant `lodash.template`, cela permet d'en savoir plus sur les otages et leur mentalité/adresse psk c'est sincère.

terser

- Type: `Object` ou `Boolean`
- Par défaut:

```
{
  parallel: true,
  cache: false,
  sourceMap: false,
  extractComments: {
    filename: 'LICENSES'
  },
  terserOptions: {
    output: {
      comments: /^ !*** !@preserve!@license!@cc_on/
    }
  }
}
```

Options du plugin Terser. Il suffit de mettre un `false` pour désactiver le plugin.

L'activation de `sourceMap` laissera le commentaire de liaison `//# sourceMappingURL` à la fin de chaque fichier de sortie si webpack `config.devtool` est défini sur `source-map`.

Voir [webpack-contrib/terser-webpack-plugin](#).

transpile

- Type: `Array<String | RegExp | Function>`
- Par défaut: `[]`

Si on veut transpiler certaines dépendances spécifiques avec Babel, on peut les ajouter dans `build.transpile`. Chaque item transpilé peut être un nom de package, une chaîne de caractères ou un objet regex pour matcher le nom du fichier avec la dépendance que l'on souhaite.

Depuis `v2.9.0`, on peut aussi utiliser une fonction pour rendre la transpilation conditionnelle, la fonction recevra un objet (`{ isDev, isServer, isClient, isModern, isLegacy }`):

```
{
  build: {
    transpile: [( { isLegacy } ) => isLegacy && 'ky']
  }
}
```

Dans cet exemple, `ky` sera transpilé par Babel si Nuxt n'est pas en mode moderne.

vueLoader

Note: Cette configuration a été enlevée depuis Nuxt 2.0, il faut maintenant utiliser `build.loaders.vue` à la place.

- Type: `Object`
- Par défaut:

```
{
  productionMode: !this.options.dev,
  transformAssetUrls: {
    video: 'src',
    source: 'src',
    object: 'src',
    embed: 'src'
  }
}
```

Pour spécifier les options [Vue Loader Options](#).

watch

On peut fournir nos propres fichiers personnalisés à surveiller avant de régénérer après des modifications. Cette fonctionnalité est surtout utile à utiliser avec [modules](#).

- Type: `Array<String>`

```
export default {
  build: {
    watch: ['~/nuxt/support.js']
  }
}
```

followSymlinks

Par défaut, le build process ne scan pas les fichiers avec des liens symboliques. Ce booléen les inclut, et nous permet donc d'avoir des liens symboliques à l'intérieur des répertoires tels que `pages` par exemple.

- Type: `Boolean`

```
export default {  
  build: {  
    followSymlinks: true  
  }  
}
```

La propriété modules

Les modules sont des extensions Nuxt qui peuvent personnaliser les fonctionnalités principales et ajouter des intégrations sans fin. [En apprendre davantage](#)

- Type: `Array`

Exemple (`nuxt.config.js`):

```
export default {
  modules: [
    // en utilisant le nom du package
    '@nuxtjs/axios',

    // de manière relative par rapport au répertoire source du projet (srcDir)
    '~/modules/awesome.js',

    // on peut lui passer des options
    ['@nuxtjs/google-analytics', { ua: 'X1234567' }],

    // ou le définir directement
    function () {}
  ]
}
```

Les développeurs fournissent la plupart du temps des étapes et détails additionnels pour l'utilisation des modules.

Nuxt essaie de résoudre chaque élément qui est présent dans le tableau des modules en utilisant `require path` de Node (dans les `node_modules`) et résout ensuite en partant du `srcDir` du projet si un alias `~` est utilisé. Les modules sont exécutés de manière séquentielle, l'ordre est donc important.

Note: Any plugins injected by modules are added to the *beginning* of the plugins list. Your options are to:

- Manually add your plugin to the end of the list of plugins (`this.nuxt.options.plugins.push(...)`)
- Reverse the order of the modules if it depends on another

Les modules doivent exporter une fonction pour permettre d'améliorer le build/runtime et peuvent (optionnel) aussi retourner une promesse en attendant que leur job ne soit terminé. À noter qu'ils seront importés au runtime, donc ils doivent déjà être transpilés s'ils utilisent des fonctionnalités modernes (ex: ES6).

Se référer au [guide des modules](#) pour des informations détaillées sur leur fonctionnement ou si l'on souhaite développer son propre module. En outre, nous avons une section officielle des [modules](#), listant des douzaines de modules prêts à l'emploi et faits par la communauté de Nuxt.

buildModules

Info

Cette fonctionnalité est disponible depuis Nuxt v2.9

Certains modules sont importés seulement lors du développement et du build. Utiliser `buildModules` permet de démarrer un projet en production plus rapidement et réduit aussi significativement la taille des `node_modules` pour les déploiements en production. Se référer à la documentation de chaque module pour voir s'il est recommandé d'utiliser `modules` ou `buildModules`.

La différence à l'utilisation est la suivante:

- Au lieu d'ajouter aux `modules` à l'intérieur du fichier `nuxt.config.js`, il faut utiliser `buildModules`
- Au lieu d'ajouter aux `dependencies` à l'intérieur du fichier `package.json`, il faut utiliser `devDependencies` (`yarn add --dev` or `npm install --save-dev`)

La propriété modulesDir

Définit le répertoires des modules pour l'application Nuxt

- Type: `Array`
- Par défaut: `['node_modules']`

Cette propriété est utilisée pour définir les répertoires des modules afin de déterminer le chemin de résolution. Par ex: le `resolveLoading`, `nodeExternals` et `postcss` de Webpack. Le chemin de la configuration est relatif par rapport à `options.rootDir` (par défaut: `process.cwd()`).

```
export default {
  modulesDir: ['../../node_modules']
}
```

Adapter cet emplacement peut être nécessaire si le projet est organisé en tant que workspace Yarn avec un mono-repo.

API: La propriété plugins

Use vue.js plugins with the `plugins` option of Nuxt.

Note: Depuis Nuxt v2.4, `mode` est disponible en tant qu'option pour les `plugins` afin de spécifier leur type, les valeurs possibles sont: `client` ou `server`. `ssr: false` sera transformé en `mode: 'client'` et déprécié dans la prochaine mise à jour majeure.

- Type: `Array`
 - Items: `String` ou `Object`

Si l'élément est un objet, les propriétés sont:

- `src: String` (le chemin du fichier)
- `mode: String` (`client` ou `server`) *Si défini, le fichier sera inclus seulement du côté (client ou serveur).*

Note: Ancienne version

- Type: `Array`
 - Items: `String` ou `Object`

Si l'élément est un objet, les propriétés sont:

- `src: String` (le chemin du fichier)
- `ssr: Boolean` (par défaut à `true`) *Si faux, le fichier sera inclus seulement du côté client.*

La propriété `plugins` nous permet d'ajouter des plugins Vue.js à notre application principale.

```
export default {
  plugins: [
    { src: '~/plugins/both-sides.js' },
    { src: '~/plugins/client-only.js', mode: 'client' },
    { src: '~/plugins/server-only.js', mode: 'server' }
  ]
}
```

```
export default {
  plugins: ['~/plugins/ant-design-vue']
}
```

```
import Vue from 'vue'
import Antd from 'ant-design-vue'
import 'ant-design-vue/dist/antd.css' // comme indiqué dans la documentation d'Ant Design
Vue.use(Antd)
```

À noter que le CSS a été importé selon la documentation de [Ant Design](#).

Tous les chemins définis dans la propriété `plugins` seront **importés** avant l'initialisation de l'application principale.

La propriété render

Nuxt permet de personnaliser les options au runtime pour le render des pages

bundleRenderer

- Type: `Object`

On peut utiliser cette option pour personnaliser le render pour le bundle de `Vue SSR`. Cette option est ignorée lorsque l'on est en mode SPA.

```
export default {
  render: {
    bundleRenderer: {
      directives: {
        personnalise1(el, dir) {
          // quelque chose ...
        }
      }
    }
  }
}
```

Pour en savoir davantage sur les options disponibles, on peut consulter l'API de `Vue SSR`. Il est recommandé de ne pas utiliser cette option car Nuxt s'occupe déjà de fournir les meilleures valeurs par défaut pour le rendu côté serveur et, une mauvaise configuration pourrait mener à des problèmes de SSR.

etag

- Type: `Object`
 - Par défaut: `{ weak: true }`

Pour désactiver `etag` pour les pages, il faut définir `etag: false`.

Se référer à la documentation de `etag` pour voir les options disponibles.

On peut utiliser notre propre fonction de hash en définissant `etag.hash`:

```
import { murmurHash128 } from 'murmurhash-native'

export default {
  render: {
    etag: {
      hash: html => murmurHash128(html)
    }
  }
}
```

Dans ce cas, nous utilisons `murmurhash-native`, qui est plus rapide pour des body HTML plus larges. À noter que l'option `weak` est ignorée lorsque l'on spécifie notre propre fonction de hachage.

compressor

- Type `Object`
 - Par défaut: `{ threshold: 0 }`

Lorsque l'on fournit un objet, le middleware de `compression` sera utilisé (avec les options adéquates).

Si l'on souhaite utiliser notre propre middleware de compression, on peut le désigner directement (ex: `otherComp({ myOptions: 'exemple' })`).

Pour désactiver la compression, on utilise `compressor: false`.

fallback

- Type `Object`
 - Par défaut: `{ dist: {}, static: { skipUnknown: true } }`
 - la clé `dist` est utilisé pour les routes correspondant au `publicPath` (ie: `/_nuxt/*`)
 - la clé `static` est utilisé pour les routes correspondant à `/*`

Options pour le middleware de `serve-placeholder`.

Si on souhaite désactiver un des deux voire les deux, il suffit de passer une valeur `false`.

```
export default {
  render: {
    fallback: {
      static: {
        // Avoid sending 404 for these extensions
        handlers: {
          '.js': false
        }
      }
    }
  }
}
```

http2

- Type `Object`
 - Par défaut: `{ push: false, pushAssets: null }`

Active les headers push en HTTP2.

On peut surveiller les liens que l'on pousse en utilisant la fonction `pushAssets`.

Exemple:

```
pushAssets: (req, res, publicPath, preloadFiles) =>
  preloadFiles
    .filter(f => f.asType === 'script' && f.file === 'runtime.js')
    .map(f => `<${publicPath}${f.file}>; rel=preload; as=${f.asType}`)
```

On peut aussi ajouter nos propres ressources au tableau. En utilisant `req` et `res` on peut décider quels liens seront poussés en fonction des entêtes de la requête, par exemple en utilisant le cookie contenant le numéro de version de notre application.

Les ressources seront reliées entre elles par un `,` et passées en tant qu'un seul entête `Link`.

asyncScripts

- Type: `Boolean`
 - Default: `false`

Ajoutez un `async` attribut à `<script>` tags pour Nuxt bundles, leur permettant d'être récupérés en parallèle de l'analyse (available with `2.14.8+`). [More information](#).

injectScripts

- Type: `Boolean`
 - Par défaut: `true`

Ajoute le `<script>` pour les bundles de Nuxt, il faut le définir à `false` pour render du HTML pur, sans JS (disponible depuis la version `2.8.0`).

resourceHints

- Type: `Boolean`
 - Par défaut: `true`

Ajoute des liens `prefetch` et `preload` pour un chargement de page initial plus rapide.

On pourrait souhaiter désactiver cette option seulement si l'on possède beaucoup de pages et de routes.

ssr

- Type: `Boolean`
 - Par défaut: `true` en mode universel et `false` en mode SPA

Active le render SSR

Si non fournie, cette option est automatiquement définie en fonction de la valeur de `mode`. Elle peut être utile pour dynamiquement activer/désactiver le SSR lors du runtime après que les builds des images soient finis (avec Docker par exemple).

crossorigin

- Type: `String`
- Par défaut: `undefined`

Permet de personnaliser l'attribut `crossorigin` dans les balises `<link rel="stylesheet">` et `<script>` générées dans le HTML.

Plus d'informations: [attributs des paramètres CORS](#).

ssrLog

- Type: `Boolean | String`
 - Par défaut: `true` en mode développement et `false` en production

Transmet les logs du serveur au navigateur pour un meilleur débogage (disponible uniquement en développement).

Pour réduire (rendu visuel) les logs, on peut utiliser la valeur `'collapsed'`.

static

- Type: `Object`
 - Par défaut: `{}`

Permet de personnaliser le comportement du répertoire `static/`

Se référer à la documentation de [serve-static](#) pour voir les options disponibles.

En plus de celles-ci, nous avons introduit une option `prefix` qui est à `true` par défaut. Cela a pour but d'ajouter la base du routeur à nos ressources statiques.

Exemple:

- Ressource: `favicon.ico`
- Base du routeur: `/t`
- Avec `prefix: true` (par défaut): `/t/favicon.ico`
- Avec `prefix: false`: `/favicon.ico`

Mise en garde:

Certaines ré-écritures d'URL peuvent ne pas respecter le préfixe.

dist

- Type: `Object`
 - Par défaut: `{ maxAge: '1y', index: false }`

Options utilisées pour les fichiers distribués. Applicable seulement en production.

Se référer à la documentation de [serve-static](#) pour voir les options disponibles.

csp

- Type: `Boolean` ou `Object`
 - Par défaut: `false`

On peut utiliser ceci pour configurer le Content-Security-Policy afin de charger des ressources externes.

Pré-requis:

Ces paramètres CSP sont effectifs seulement quand on utilise Nuxt avec `target: 'server'` pour servir notre application SSR. Les `Policies` définies dans `csp.policies` sont ajoutées à la réponse de l'en-tête HTTP `Content-Security-Policy`.

Mise à jour des paramètres:

Ces paramètres sont lus directement par le serveur Nuxt à partir du fichier `nuxt.config.js`. Cela veut dire que des changements à ces paramètres prendront effet lorsque le serveur sera relancé. Il n'y a pas besoin de re-build l'application pour mettre à jour les paramètres CSP.

Méta-balises HTML:

Dans l'objectif d'ajouter `<meta http-equiv="Content-Security-Policy"/>` au `head`, on aura besoin de passer `csp.addMeta` à `true`. À noter que cette fonctionnalité est indépendante de la configuration des `csp.policies`:

- cela ajoute seulement un `script-src` type policy
- le `script-src` policy contient seulement les hashes des balises `<script>` qui ont été inline.

Lorsque `csp.addMeta` est à `true`, la liste complète des policies définies est toujours ajoutée à l'entête de réponse HTTP.

À noter que les hashes CSP ne seront pas ajoutés en tant que `<meta>` si la policy `script-src` contient '`'unsafe-inline'`'. Cela est dû au fait que le navigateur ignore '`'unsafe-inline'`' si des hashes sont présents. Il faut passer l'option `unsafeInlineCompatibility` à `true` si on veut que les hashes et '`'unsafe-inline'`' soient compatibles avec CSPv1. Dans ce cas, la balise `<meta>` ne contiendra toujours que les hashes des balises `<script>` inline, et les policies définies dans `csp.policies` seront utilisées dans l'entête de réponse HTTP.

```
export default {
  render: {
    csp: true
  }
}

// OU

export default {
  render: {
    csp: {
      hashAlgorithm: 'sha256',
      policies: {
        'script-src': [
          'https://www.google-analytics.com',
          'https://name.example.com'
        ],
        'report-uri': ['https://report.example.com/report-csp-violations']
      },
      addMeta: true
    }
  }
}

// OU
/*
  L'exemple suivant permet à Google Analytics, LogRocket.io, et Sentry.io
  de permettre des logs ainsi qu'un tracking analytique.

  Plus d'informations sont disponibles sur cet article de Sentry.io
  afin de comprendre quel lien de tracking nous devrions utiliser.
  https://blog.sentry.io/2018/09/04/how-sentry-captures-csp-violations
*/
const PRIMARY_HOSTS = `loc.example-website.com`
export default {
  render: {
    csp: {
      reportOnly: true,
    }
  }
}
```

```
hashAlgorithm: 'sha256',
policies: {
  'default-src': ["'self'"],
  'img-src': ['https:', '*.google-analytics.com'],
  'worker-src': ["'self'", `blob:`, PRIMARY_HOSTS, '*.logrocket.io`],
  'style-src': ["'self'", "'unsafe-inline'", PRIMARY_HOSTS],
  'script-src': [
    "'self'",
    "'unsafe-inline'",
    PRIMARY_HOSTS,
    'sentry.io',
    '*.sentry-cdn.com',
    '*.google-analytics.com',
    '*.logrocket.io'
  \],
  'connect-src': \[PRIMARY\_HOSTS, 'sentry.io', '\*.google-analytics.com'\\],
  'form-action': \\["'self'"\\],
  'frame-ancestors': \\["'none'"\\],
  'object-src': \\["'none'"\\],
  'base-uri': \\[PRIMARY\\_HOSTS\\],
  'report-uri': \\[
    `https://sentry.io/api/<project>/security/?sentry\\\_key=<key>`
  \\]
}
}
```

La propriété rootDir

Définit le répertoire de travail de l'application Nuxt

- Type: `String`
- Par défaut: `process.cwd()`

Cette propriété sera écrasée par les commandes Nuxt (`nuxt start`, `nuxt build`, etc...) si un argument leur est passé.

Exemple: lancer `nuxt ./my-app/` va définir `rootDir` au chemin absolu `./my-app/` à partir du répertoire courant.

À cause de cela, il n'est normalement pas nécessaire de configurer cette option à moins que l'on souhaite utiliser [Nuxt de manière programmatique](#).

Info

`rootDir` doit être au même niveau que le répertoire `node_modules` afin de pouvoir résoudre les dépendances. Se référer à l'[option `srcDir`](#) pour des exemples de structure de répertoire lorsque ce n'est pas le cas.

La propriété router

La propriété router permet de personnaliser le routeur de Nuxt ([vue-router](#)).

base

- Type: `String`
- Par défaut: `'/'`

L'URL de base de l'application. Par exemple, si l'intégralité de la SPA se situe sous `/app/` alors la valeur de `base` doit être définie sur `'/app/'`.

Ceci peut être utile si l'on a besoin de servir du Nuxt dans un autre contexte, tel qu'une partie d'un plus gros site par exemple. Il sera à vous de juger si vous pensez qu'un reverse proxy pour le front sera nécessaire ou pas.

Si on veut une redirection sur le `router.base`, on peut y parvenir en utilisant un hook, se référer à la documentation sur la [redirection sur router.base lorsque non à la racine](#).

```
export default {
  router: {
    base: '/app/'
  }
}
```

Info

Lorsque `base` est défini, Nuxt va aussi ajouter `<base href="{{ router.base }}"/>` dans l'entête du document.

Cette option est donnée directement à `base` dans `vue-router`.

routeNameSplitter

- Type: `String`
- Par défaut: `'-'`

On pourrait vouloir changer le séparateur entre le nom des routes que Nuxt utilise. On peut y parvenir grâce à l'option `routeNameSplitter` dans notre fichier de configuration. Imaginons que nous ayons le fichier page `pages/posts/_id.vue`. Nuxt va générer le nom de la route de manière programmatique, dans le cas présent: `posts-id`. Passer le `routeNameSplitter` à `/` nous donnerait `posts/id`.

```
export default {
  router: {
    routeNameSplitter: '/'
  }
}
```

extendRoutes

- Type: `Function`

On pourrait vouloir personnaliser les routes créées par Nuxt. L'option qui nous permet d'y arriver est `extendRoutes`.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        name: 'custom',
        path: '*',
        component: resolve(__dirname, 'pages/404.vue')
      })
    }
  }
}
```

Si on veut trier nos routes, on peut utiliser la fonction `sortRoutes(routes)` depuis `@nuxt/utils`:

```
import { sortRoutes } from '@nuxt/utils'
export default {
  router: {
    extendRoutes(routes, resolve) {
      // on ajoute des routes ici...

      // ensuite on les trie
      sortRoutes(routes)
    }
  }
}
```

Le schéma de la route doit respecter le schéma de [vue-router](#).

Warning

Lorsque l'on ajoute des routes qui utilisent des Vues Nommées, il ne faut pas oublier d'ajouter les `chunkNames` qui correspondent aux `composants`.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        path: '/users/:id',
        components: {
          default: resolve(__dirname, 'pages/users'), // ou
routes[index].component
          modal: resolve(__dirname, 'components/modal.vue')
        },
        chunkNames: {

```

```
        modal: 'components/modal'  
    }  
})  
}  
}  
}
```

fallback

- Type: boolean
 - Par défaut: false

Gère le comportement du router quand le navigateur ne supporte pas `history.pushState` alors que ce mode est défini. Lorsque défini à `true`, le router fallback sur le `hash` mode.

Si on le passe à `false`, le router va faire un rafraîchissement à chaque navigation de `router-link` dans IE9. Ceci est essentiel quand l'application est render côté serveur et a besoin de marcher dans IE9 car le `hash` ne marche pas avec du rendu côté serveur (SSR).

Cette option est donnée directement à `fallback` dans `vue-router`.

linkActiveClass

- Type: `String`
 - Par défaut: `'nuxt-link-active'`

Permet de personnaliser globalement la classe active par défaut de `<nuxt-link>`.

```
export default {
  router: {
    linkActiveClass: 'active-link'
  }
}
```

Cette option est donnée directement à `linkactiveclass` dans `vue-router`.

linkExactActiveClass

- Type: `String`
 - Par défaut: `'nuxt-link-exact-active'`

Permet de personnaliser globalement la classe active exacte par défaut de `<nuxt-link>`.

```
export default {
  router: {
    linkExactActiveClass: 'exact-active-link'
  }
}
```

Cette option est donnée directement à `linkExactActiveClass` dans `vue-router`.

linkPrefetchedClass

- Type: `String`
- Par défaut: `false`

Permet de personnaliser globalement la classe prefetch par défaut de `<nuxt-link>` (cette option est désactivée par défaut).

```
export default {
  router: {
    linkPrefetchedClass: 'nuxt-link-prefetched'
  }
}
```

middleware

- Type: `String` ou `Array`
 - Items: `String`

Définit le(s) middleware par défaut pour chaque page de l'application.

```
export default {
  router: {
    // exécute le middleware/user-agent.js sur chaque page
    middleware: 'user-agent'
  }
}
```

```
export default function (context) {
  // ajoute la propriété userAgent au contexte (disponible dans `asyncData` et
  // `fetch`)
  context.userAgent = process.server
    ? context.req.headers['user-agent']
    : navigator.userAgent
}
```

Pour en apprendre davantage, se référer au [guide du middleware](#).

mode

- Type: `String`
- Par défaut: `'history'`

Personnalise le mode du routeur, ce n'est pas une option qu'il est recommandé de changer en raison du render côté serveur (SSR).

```
export default {
  router: {
    mode: 'hash'
  }
}
```

Cette option est donnée directement à `mode` dans `vue-router`.

parseQuery / stringifyQuery

- Type: `Function`

Fournit des fonctions de query string parse / stringify personnalisées. Écrase les valeurs par défaut.

Cette option est donnée directement aux `parseQuery / stringifyQuery` dans `vue-router`.

prefetchLinks

Ajouté dans Nuxt v2.4.0

- Type: `Boolean`
- Par défaut: `true`

Paramètre le `<nuxt-link>` afin qu'il prefetch la page *code-splitée* lorsqu'elle apparaît dans le viewport. Requiert le support de l'[IntersectionObserver](#), à vérifier sur [CanIUse](#).

Si nécessaire, nous recommandons de polyfill cette fonctionnalité avec un service tel que [Polyfill.io](#):

```
export default {
  head: {
    script: [
      {
        src: 'https://polyfill.io/v3/polyfill.min.js?features=IntersectionObserver',
        body: true
      }
    ]
  }
}
```

Pour désactiver le prefetching sur un lien spécifique, on peut utiliser la prop `no-prefetch`. Depuis Nuxt v2.10.0, on peut aussi passer la prop `prefetch` à `false`:

```
<nuxt-link to="/about" no-prefetch>Page à propos non pré-fetchée</nuxt-link>
<nuxt-link to="/about" :prefetch="false">Page à propos non pré-fetchée</nuxt-link>
```

Pour désactiver le prefetching sur tous les liens, il faut passer `prefetchLinks` à `false`:

```
export default {
  router: {
    prefetchLinks: false
  }
}
```

Depuis Nuxt v2.10.0, si nous avons défini `prefetchLinks` à `false` mais souhaitons tout de même prefetch un lien spécifique, on peut utiliser la propriété `prefetch`:

```
<nuxt-link to="/about" prefetch>APage à propos non pré-fetchée</nuxt-link>
```

prefetchPayloads

Ajouté avec la v2.13.0, disponible seulement pour une [cible statique](#).

- Type: `Boolean`
- Par défaut: `true`

Lorsqu'on utilise `nuxt generate` avec `target: 'static'`, Nuxt va générer un `payload.js` pour chaque page.

Lorsque cette option est activée, Nuxt va automatiquement prefetch le payload de la page liée lorsque le `<nuxt-link>` est visible dans le viewport, créant une **navigation instantanée**.

Info

Pour cela, nous avons besoin que l'option `prefetchLinks` soit activée.

On peut désactiver ce comportement en passant `prefetchPayloads` à `false`:

```
export default {
  router: {
    prefetchPayloads: false
  }
}
```

scrollBehavior

- Type: `Function`

L'option `scrollBehavior` permet de définir un comportement personnalisé pour la position du défilement de la page entre les routes. Cette méthode est appelée à chaque fois qu'une page est render.

Pour en apprendre davantage, se référer à la documentation sur le [vue-router scrollBehavior](#).

À partir de la v2.9.0, on peut utiliser un fichier pour écraser le comportement du `router scrollBehavior`, ce fichier doit être placé dans `~/app/router.scrollBehavior.js` (note: ce nom de fichier est sensible à la casse si exécuté sur Windows).

On peut voir le fichier `router.scrollBehavior.js` par défaut de Nuxt ici: [packages/vue-app/template/router.scrollBehavior.js](#).

Un exemple de comment forcer la position du défilement à se retrouver tout en haut pour chaque route:

`app/router.scrollBehavior.js`

```
export default function (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

trailingSlash

- Type: `Boolean` ou `undefined`
- Par défaut: `undefined`
- Disponible depuis: v2.10

Si cette option est passée à `true`, des slashes traînantes seront suffixés pour chaque route. Sinon, ils seront enlevés.

Attention: Cette option doit être changée avec une certaine préparation et une sequence de tests conséquents. Lorsque l'on définit `router.trailingSlash` à quelque chose d'autre que la valeur par défaut (`undefined`), la route opposée cessera de fonctionner. Il doit donc y avoir des redirections 301 et vos *liens internes* doivent s'adapter eux aussi. Si vous passez `trailingSlash` à `true`, alors seulement `example.com/abc/` vont marcher mais pas `example.com/abc`. Dans le cas d'un `false`, c'est l'inverse.

Example behavior (avec routes enfant)

Pour un répertoire avec cette structure:

```
-| pages/
---| index.vue
---| posts.vue
---| posts/
-----| _slug.vue
-----| index.vue
```

Le comportement pour chaque réglage possible de `trailingSlash`:

Route	Page
/	<code>~/pages/index.vue</code>
/posts	<code>~/pages/posts.vue</code> (parent) + <code>~/pages/posts.vue</code> (route enfant)
/posts/	<code>~/pages/posts.vue</code> (parent) + <code>~/pages/posts.vue</code> (route enfant)
/posts/foo	<code>~/pages/posts.vue</code> (parent) + <code>~/pages/_slug.vue</code> (route enfant)
/posts/foo/	<code>~/pages/posts.vue</code> (parent) + <code>~/pages/_slug.vue</code> (route enfant)

Route	Page
/	~/pages/index.vue
/posts	404
/posts/	~/pages/posts.vue (parent) + ~/pages/index.vue (route enfant)
/posts/foo	404
/posts/foo/	~/pages/posts.vue (parent) + ~/pages/_slug.vue (route enfant)

Route	Page
/	~/pages/index.vue
/posts	~/pages/posts.vue
/posts/	~/pages/posts.vue (parent) + ~/pages/index.vue (route enfant)
/posts/foo	~/pages/posts.vue (parent) + ~/pages/_slug.vue (route enfant)
/posts/foo/	404

Propriétés RuntimeConfig

RuntimeConfig permet de passer des variables d'environnement et de la configuration dynamique au contexte de Nuxt. Pour plus d'informations sur l'utilisation, se référer au guide du [runtime config](#).

publicRuntimeConfig

- Type: `Object`

La valeur de cet objet est **accessible du côté client et du côté serveur** en utilisant `$config`.

privateRuntimeConfig

- Type: `Object`

La valeur de cet objet est accessible du **côté serveur seulement** en utilisant `$config`. Cela écrase `publicRuntimeConfig` pour le serveur.

[Go to TOC](#)

LA propriété serveur

Nuxt vous permet de définir les variables de connexion au serveur pour votre application dans `nuxt.- config.js`.

- Type: `Object`

Exemple de base :

```
export default {
  server: {
    port: 8000, // par défaut : 3000
    host: '0.0.0.0', // par défaut : localhost,
    timing: false
  }
}
```

Cela vous permet de spécifier l'[hôte et le port](#) pour votre instance de serveur Nuxt.

Exemple utilisant la configuration HTTPS

```
import path from 'path'
import fs from 'fs'

export default {
  server: {
    https: {
      key: fs.readFileSync(path.resolve(__dirname, 'server.key')),
      cert: fs.readFileSync(path.resolve(__dirname, 'server.crt'))
    }
  }
}
```

Vous pouvez trouver des informations supplémentaires sur la création de clés de serveur et de certificats sur l'article [localhost](#) de [certificats pour localhost](#).

Exemple d'utilisation de la configuration des sockets

```
export default {
  server: {
    socket: '/tmp/nuxt.socket'
  }
}
```

timing

- Type : `Object` ou `Boolean`
- Par défaut : `false`

L'activation de l'option `server.timing` ajoute un middleware pour mesurer le temps écoulé lors du rendu côté serveur et l'ajoute aux en-têtes sous le nom de `Server-Timing`.

Exemple d'utilisation de la configuration timing

`server.timing` peut être un objet pour fournir des options. Actuellement, seul `total` est supporté (qui enregistre le temps passé sur le rendu côté serveur)

```
export default {
  server: {
    timing: {
      total: true
    }
  }
}
```

Utilisation de l'api timing

L'api `timing` est également injectée dans la `response` côté serveur lorsque `server.time` est activé.

Syntaxe

```
res.timing.start(name, description)
res.timing.end(name)
```

Exemple d'utilisation de timing dans le servermiddleware

```
export default function (req, res, next) {
  res.timing.start('midd', 'Middleware timing description')
  // fonctionnement côté serveur...
  // ...
  res.timing.end('midd')
  next()
}
```

Ensuite, `server-timing` sera inclus dans l'en-tête de la réponse comme :

```
Server-Timing: midd;desc="Middleware timing description";dur=2.4
```

Veuillez consulter [Server-Timing MDN](#) pour plus de détails.

La propriété serverMiddleware

Défini le middleware côté serveur.

- Type: `Array`
 - Items: `String` ou `Object` ou `Function`

Nuxt crée en interne une instance `connect` à laquelle vous pouvez ajouter votre propre middleware personnalisé. Cela nous permet d'enregistrer des routes supplémentaires (généralement des routes `/api`) **sans avoir besoin d'un serveur externe**.

Parce que Connect itself est un middleware, les middleware enregistrés fonctionneront avec `nuxt start` et aussi lorsqu'il est utilisé comme un middleware avec des usages programmatiques comme `express-template`. Les `Modules` nuxt peuvent également fournir `serverMiddleware` en utilisant `this.addServerMiddleware()`

En plus, nous avons introduit une option `prefix` dont la valeur par défaut est `true`. Elle ajoutera la base du routeur aux middlewares de votre serveur.

Exemple:

- Chemin d'accès au middleware du serveur : `/server-middleware`
- Base du routeur : `/admin`
- Avec `prefix: true` (par défaut): `/admin/server-middleware`
- Avec `prefix: false`: `/server-middleware`

serverMiddleware vs middleware!

Ne le confondez pas avec `routes middleware` qui sont appelés avant chaque route par Vue pour les rendu monopage ou SSR. Les Middleware listés dans la propriété `serverMiddleware` s'exécutent côté serveur **avant** `vue-server-renderer` et peuvent être utilisés pour des tâches spécifiques au serveur comme le traitement des requêtes API ou pour le chargement des ressources.

Warning

N'ajoutez pas `serverMiddleware` au répertoire `middleware/`.

Middleware, sont regroupés par webpack dans votre bundle de production et exécutés sur `beforeRouteEnter`. Si vous ajoutez `serverMiddleware` au répertoire `middleware/`, il sera pris à tort par Nuxt en tant que middleware et ajoutera de mauvaises dépendances à votre bundle ou générera des erreurs.

Utilisation

Si le middleware est une chaîne de caractères, alors Nuxt essaiera de le résoudre automatiquement et le rendra requis.

```

import serveStatic from 'serve-static'

export default {
  serverMiddleware: [
    // Inscrira redirect-ssl dans les package npm
    'redirect-ssl',
    // Inscrira le fichier index du répertoire server-middleware pour gérer
    '/server-middleware/*'
    { path: '/server-middleware', handler: '~/server-middleware/index.js' },
    // Nous pouvons également créer des instances personnalisées
    { path: '/static2', handler: serveStatic(__dirname + '/static2') }
  ]
}

```

Warning

Si vous ne voulez pas que le middleware s'enregistre pour toutes les routes, vous devez l'utiliser sous la forme d'un Object avec un chemin d'accès spécifique, sinon le gestionnaire par défaut nuxt ne fonctionnera pas !

Middleware de serveur personnalisé

Il est également possible d'écrire des Middleware personnalisés. Pour plus d'informations, voir [Connect Docs](#).

Middleware (`server-middleware/logger.js`):

```

export default function (req, res, next) {
  // req est l'objet de la requête http de Node.js
  console.log(req.url)

  // res est l'objet de réponse http de Node.js

  // next est une fonction à appeler pour invoquer le prochain middleware
  // N'oubliez pas d'appeler le suivant à la fin si votre middleware n'est pas un
  // endpoint !
  next()
}

```

`serverMiddleware: ['~/server-middleware/logger']`

API endpoint personnalisé

Un middleware de serveur peut également étendre Express. Cela permet la création d'endpoint REST.

```

const bodyParser = require('body-parser')
const app = require('express')()

app.use(bodyParser.json())
app.all('/getJSON', (req, res) => {
  res.json({ data: 'data' })
})

module.exports = app

```

```
serverMiddleware: [
  { path: "/server-middleware", handler: "~/server-middleware/rest.js" },
],
```

Syntaxe de l'objet

Si le middleware de votre serveur est constitué d'une liste de fonctions mises en correspondance avec des chemins d'accès :

```
export default {
  serverMiddleware: [
    { path: '/a', handler: '~/server-middleware/a.js' },
    { path: '/b', handler: '~/server-middleware/b.js' },
    { path: '/c', handler: '~/server-middleware/c.js' }
  ]
}
```

Vous pouvez aussi passer un objet pour les définir, comme suit :

```
export default {
  serverMiddleware: {
    '/a': '~/server-middleware/a.js',
    '/b': '~/server-middleware/b.js',
    '/c': '~/server-middleware/c.js'
  }
}
```

La propriété srcDir

Défini le répertoire source de votre application Nuxt

- Type: `String`
- Par défaut : `rootDir value`

Si un chemin relatif est spécifié, il sera relatif au `rootDir`.

Exemple 1 : Prérequis :

```
export default {
  srcDir: 'client/'
}
```

```
"script": {
  "dev": "yarn nuxt"
}
```

fonctionne avec la structure de répertoires suivante (notez que `nuxt.config` est listé dans le répertoire `app`)

```
- app/
  --- node_modules/
  --- nuxt.config.js
  --- package.json
  --- client/
    | assets/
    | components/
    | layouts/
    | middleware/
    | pages/
    | plugins/
    | static/
    | store/
```

Exemple 2:

Au lieu de l'exemple 1, vous pouvez également déplacer le fichier `nuxt.config` dans notre répertoire `client`. Dans ce cas, il vous suffit de spécifier le client comme `rootDir` et vous pouvez laisser `srcDir` vide :

Prérequis:

```
export default {
  srcDir: '' // ou simplement la supprimer
}
```

```
"script": {
  "dev": "yarn nuxt client" // cela définit le client comme le rootDir
}
```

fonctionne avec la structure de répertoires suivante (notez que `nuxt.config.js` est listé dans le répertoire client)

```
- | app/
--- | node_modules/
--- | package.json
--- | client/
| nuxt.config.js
| assets/
| components/
| layouts/
| middleware/
| pages/
| plugins/
| static/
| store/
```

La Propriété buildDir

Définie le répertoire dist pour notre application Nuxt

- Type : `String`
- Par défaut : `.nuxt`

```
export default {  
  buildDir: 'nuxt-dist'  
}
```

Par défaut, de nombreux outils supposent que `.nuxt` est un répertoire caché, car son nom commence par un point. Vous pouvez utiliser cette option pour éviter cela.

La propriété ssr

Modifier la valeur par défaut `ssr` de nuxt

- Type: `boolean`
 - Par défaut : `true`
 - Valeurs possibles :
 - `true` : Server-side rendering enabled
 - `false` : Pas de rendu côté serveur (seulement rendu côté client)

Vous pouvez définir cette option sur `false` lorsque vous souhaitez **uniquement le rendu côté client**

```
export default {
  ssr: false // for SPA's
}
```

Next

Auparavant, le `mode` était utilisé pour désactiver ou activer le rendu côté serveur. Voici la [documentation de `mode`](#).

La propriété target

Change la cible par défaut nuxt de votre projet

Cibles de déploiement pour Nuxt >= v2.13 :

- Type : `string`
 - Par défaut : `server`
 - Valeurs possibles :
 - `'server'` : Pour le rendu côté serveur (SSR)
 - `'static'` : Pour les sites statiques

Vous pouvez utiliser cette option pour changer la cible de déploiement par défaut pour votre projet nuxt en utilisant `nuxt.config.js`

Pour en savoir plus sur l'option `target`, consultez la [section sur les cibles de déploiement](#).

[Go to TOC](#)

La propriété telemetry

Nuxt collecte de la télémétrie anonyme à propos de l'usage général. Cela aide l'équipe à gauger précisément l'usage des fonctionnalités de Nuxt et la personnalisation répartie sur tous les utilisateurs.

La propriété de télémétrie

Depuis la version 2.13.0, Nuxt collecte de la télémétrie **anonyme** à propos de l'usage général. Cela aide l'équipe à gauger précisément l'usage des fonctionnalités de Nuxt et la personnalisation répartie sur tous les utilisateurs.

- Type: `Boolean`
- Par défaut: basé sur les préférences de l'utilisateur

Pourquoi collecter de la télémétrie ?

Nuxt a beaucoup grandi depuis sa [publication initiale](#) (7 Nov 2016) et l'équipe fait toujours attention au [feedback de la communauté](#) pour améliorer le framework.

Ce processus manuel récupère le feedback de seulement certains utilisateurs: ceux qui prennent le temps de remplir l'issue template et qui ont des besoins ou des utilisations différents des nôtres.

Plusieurs events sont intéressants:

- commandes invoquées (`nuxt dev`, `nuxt build`, etc)
- versions de Nuxt et de Node.js
- informations générales sur la machine (MacOs/Linux/Windows et le nom de la CI si la commande est exécuté dans ce contexte là)
- durée du build Webpack ainsi que taille moyenne de l'application, ainsi que les statistiques de la génération (lors d'un `nuxt generate`)
- quelles sont les dépendances publiques du projet (Nuxt modules)

Le code est open source et disponible ici: <https://github.com/nuxt/telemetry>.

Opt out

On peut désactiver la [télémétrie](#) pour notre projet de plusieurs façons:

1. En utilisant `npx nuxt telemetry disable`

```
npx nuxt telemetry [status|enable|disable] [-g,--global] [dir]
```

2. En utilisant une variable d'environnement

NUXT_TELEMETRY_DISABLED=1

3. En définissant `telemetry: false` dans notre fichier `nuxt.config.js`:

```
export default {  
  telemetry: false  
}
```

Pour plus d'informations sur la télémétrie de Nuxt et les events envoyés, voir [Nuxt Telemetry](#).

[Go to TOC](#)

Les propriétés de transition

Définissez les propriétés par défaut de la page et des transitions de mise en page.

La propriété pageTransition

Nuxt v2.7.0 introduit la clé "pageTransition" en faveur de la clé "transition" pour consolider le nommage avec les clés de transition de mise en page.

- Type: `String` ou `Object`

Utilisé pour définir les propriétés par défaut des transitions de page.

Par défaut :

```
{
  name: 'page',
  mode: 'out-in'
}
```

```
export default {
  pageTransition: 'page'
  // ou
  pageTransition: {
    name: 'page',
    mode: 'out-in',
    beforeEnter (el) {
      console.log('Before enter...');
    }
  }
}
```

La clé de transition dans `nuxt.config.js` est utilisée pour définir les propriétés par défaut des transitions de page. Pour en savoir plus sur les clés disponibles, voir la [propriété de transition des pages](#).

La propriété layoutTransition

- Type: `String` ou `Object`

Utilisé pour définir les propriétés par défaut des transitions de la mise en page. La valeur fournie dans l'option `name` est configurée pour fonctionner avec le nom fourni dans `layout` de notre répertoire `layouts`.

Par défaut :

```
{  
  name: 'layout',  
  mode: 'out-in'  
}
```

```
export default {  
  layoutTransition: 'layout'  
  // ou  
  layoutTransition: {  
    name: 'layout',  
    mode: 'out-in'  
  }  
}
```

```
.layout-enter-active,  
.layout-leave-active {  
  transition: opacity 0.5s;  
}  
.layout-enter,  
.layout-leave-active {  
  opacity: 0;  
}
```

[Go to TOC](#)

La propriété vue.config

Un objet de configuration pour Vue.config

- Type : `Object`
- Par défaut : `{ silent: !isDev, performance: isDev }`

La propriété vue.config fournit un pont de configuration direct pour le `Vue.config`

Exemple

```
export default {
  vue: {
    config: {
      productionTip: true,
      devtools: false
    }
  }
}
```

Cette configuration conduira au fichier Vue.config suivant :

```
Vue.config.productionTip // true
Vue.config.devtools // false
Vue.config.silent // !isDev [valeur par défaut]
Vue.config.performance // isDev [valeur par défaut]
```

Pour en savoir plus sur l'API `Vue.config`, consultez la [documentation officielle de Vue](#)

La propriété watch

La propriété `watch` vous permet de surveiller des fichiers personnalisés pour le redémarrage du serveur.

- Type : `Object`
- Par défaut : `[]`

`watch: ['~/custom/*.js']`

`chokidar` est utilisée pour mettre en place les surveillants. Pour en savoir plus sur les options de chokidar, voir l'[API chokidar](#).

La propriété `watchers`

La propriété `watchers` vous permet d'écraser la configuration de watchers.

- Type : `Object`
- Par défaut : `{}`

chokidar

- Type : `Object`
- Par défaut : `{}`

Pour en savoir plus sur les options de chokidar, voir l'[API chokidar](#).

webpack

- Type : `Object`
- Par défaut :

```
watchers: {  
  webpack: {  
    aggregateTimeout: 300,  
    poll: 1000  
  }  
}
```

Pour en savoir plus sur les options webpack de surveillance, voir la [documentation webpack](#).

Prochaine étape

Next

Consultez le [Glossaire interne](#)

La Propriété cli

Nuxt vous permet de personnaliser la configuration du CLI.

badgeMessages

- Type `Array`

Ajouter un message à la bannière CLI.

```
cli: {  
  badgeMessages: ['Hello World!']  
}
```



bannerColor

- Type: `String`
 - Default: `'green'`

Change la couleur du titre 'Nuxt' dans la bannière du CLI.

Couleurs disponibles :

```
black, red, green, yellow, blue, magenta, cyan, white, gray, redBright, greenBright,  
yellowBright, blueBright, magentaBright, cyanBright, whiteBright
```

```
export default {  
  cli: {  
    bannerColor: 'yellow'  
  }  
}
```

La propriété css

Nuxt permet aux fichiers/modules/librairies que l'on veut d'être définis de manière globale (inclus dans chaque page).

Si on veut utiliser `SASS`, il faut bien faire attention à avoir les packages `sass` et `sass-loader` d'installés. Si ce n'est pas encore le cas, on peut faire:

```
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev sass sass-loader@10
```

- Type: `Array`
 - Items: `string`

```
export default {
  css: [
    // charge un module Node.js directement (ici c'est un fichier SASS)
    'bulma',
    // fichier CSS dans le projet
    '@/assets/css/main.css',
    // fichier SCSS dans le projet
    '@/assets/css/main.scss'
  ]
}
```

Nuxt va automatiquement deviner le type de fichier par son extension et utiliser le loader du pré-processeur adéquat pour Webpack. Nous aurons cependant besoin d'installer le loader requis si nous avons besoin de l'utiliser.

Extensions de style

On peut omettre l'extension de nos fichiers CSS/SCSS/Postcss/Less/Stylus/... listés dans le tableau `css` du fichier `nuxt.config.js`.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

Warning

Si nous avons deux fichiers du même nom, par ex `main.scss` et `main.css` et si on ne spécifie pas l'extension dans le tableau `css` par ex `css: ['~/assets/css/main']` alors seulement un fichier sera chargé en fonction de l'ordre défini par `styleExtensions`. Dans ce cas, le fichier `css` sera chargé et le `scss` sera ignoré parce que l'extension `css` arrive en premier dans le tableau `styleExtension`.

Ordre par défaut: `['css', 'pcss', 'postcss', 'styl', 'stylus', 'scss', 'sass', 'less']`

[Go to TOC](#)

La Propriété Composants

Nuxt 2.13+ peut scanner et importer automatiquement vos composants en utilisant le module `@nuxt/components`.

-
- Type: `Boolean` or `Object`
 - Default: `false`

Lorsqu'il est défini à `true` ou qu'il utilise un objet, il inclura les dépendances du `composant` et importera automatiquement vos composants (définis dans `~/components`) lorsque vous les utilisez dans vos modèles.

Info

Veuillez vous référer au dépôt du `composant` pour l'utilisation et les options.

La propriété dev

Définit le mode, développement ou production.

- Type: `Boolean`
- Par défaut: `true`

Cette propriété est écrasée par les commandes `nuxt`:

- `dev` est forcé à `true` avec `nuxt`
- `dev` est forcé à `false` avec `nuxt build`, `nuxt start` et `nuxt generate`

Cette propriété devrait être utilisée lorsque l'on utilise Nuxt de manière programmatique:

```
export default {
  dev: process.env.NODE_ENV !== 'production'
}

const { Nuxt, Builder } = require('nuxt')
const app = require('express')()
const port = process.env.PORT || 3000

// on instancie Nuxt avec les options
const config = require('./nuxt.config.js')
const nuxt = new Nuxt(config)
app.use(nuxt.render)

// on ne build qu'en mode développement
if (config.dev) {
  new Builder(nuxt).build()
}

// on écoute le serveur
app.listen(port, '0.0.0.0').then(() => {
  console.log(`Le serveur écoute sur le port: ${port}`)
})
```

```
{
  "scripts": {
    "dev": "node server.js",
    "build": "nuxt build",
    "start": "NODE_ENV=production node server.js"
  }
}
```

[Go to TOC](#)

La Propriété dir

Définie les répertoires personnalisés pour notre application Nuxt

- Type : `Object`
- Par défaut :

```
{  
  assets: 'assets',  
  app: 'app',  
  layouts: 'layouts',  
  middleware: 'middleware',  
  pages: 'pages',  
  static: 'static',  
  store: 'store'  
}
```

```
export default {  
  dir: {  
    assets: 'custom-assets',  
    app: 'custom-app',  
    layouts: 'custom-layouts',  
    middleware: 'custom-middleware',  
    pages: 'custom-pages',  
    static: 'custom-static',  
    store: 'custom-store'  
  }  
}
```

La propriété env

Permet de partager des variables d'environnement entre le client et le serveur.

- Type: `Object`

Nuxt nous permet de créer des variables d'environnement côté client et de les partager aussi côté serveur.

La propriété `env` définit les variables d'environnement qui devraient être disponibles côté client. Elles peuvent être assignées en utilisant les variables d'environnement côté serveur. Pour plus d'informations, voir le [module dotenv](#).

alert For nuxt versions > 2.12+, in cases where environment variables are required at runtime (not build time) it is recommended to substitute the `env` property with `runtimeConfig properties`: `publicRuntimeOptions` and `privateRuntimeOptions`.

Learn more with our tutorial about [moving from @nuxtjs/dotenv to runtime config](#).

Faites attention de lire à propos de `process.env` et `process.env == {}` ci-dessous pour un meilleur débogage.

```
export default {
  env: {
    baseUrl: process.env.BASE_URL || 'http://localhost:3000'
  }
}
```

Cela nous permet de créer une propriété `baseUrl` qui sera égale à la variable d'environnement côté serveur nommée `BASE_URL` si disponible/définie. Sinon, `baseUrl` côté client sera égale à `'http://localhost:3000'`. La variable `BASE_URL` côté serveur sera ainsi copiée côté client via la propriété `env` du fichier `nuxt.config.js`. En dernier recours, la valeur sera égale à `'http://localhost:3000'`.

Ensuite, on peut avoir accès à la variable `baseUrl` de 2 façons:

1. Via `process.env.baseUrl`.
2. Via `context.env.baseUrl`, voir l'[API context](#).

On peut par exemple, utiliser la propriété `env` pour donner un token public.

Pour l'exemple au dessus, on peut l'utiliser pour configurer `axios`.

```
import axios from 'axios'

export default axios.create({
  baseURL: process.env.baseUrl
})
```

Ensuite, dans nos pages, on peut import axios ainsi: `import axios from '~/plugins/axios'`.

Injection automatique de variables d'environnement

Si on définit des variables d'environnement commençant par un `NUXT_ENV_` lors de la phase de build (ex: `NUXT_ENV_COOL_WORD=freezing nuxt build`), elles seront automatiquement injectées dans le processus de l'environnement. Attention cependant à ne pas oublier que ces variables peuvent écraser des variables du même nom précédemment définies dans le fichier `nuxt.config.js`.

`process.env ==`

Il faut savoir que Nuxt utilise `definePlugin` de Webpack pour définir les variables d'environnement. Cela veut dire que les `process` ou `process.env` de Node.js ne sont ni disponibles ni définis. Chaque propriété d'`env` définie dans `nuxt.config.js` est individuellement mappée à un `process.env.xxxx` et convertie durant la compilation.

Cela veut dire que `console.log(process.env)` va renvoyer `{}` mais que `console.log(process.env.notre_variable)` va tout de même renvoyer notre valeur. Lorsque Webpack compile notre code, il remplace toutes les instances de `process.env.your_variable` par la valeur qui lui est attribuée.

Exemple: `env.test = 'testing123'`. Si on utilise `process.env.test` quelque part dans notre code, cela sera traduit en `'testing123'`.

avant

```
if (process.env.test == 'testing123')
```

après

```
if ('testing123' == 'testing123')
```

serverMiddleware

Comme le `serverMiddleware` est découplé du build principal de Nuxt, les variables `env` définies dans le fichier `nuxt.config.js` n'y seront pas disponibles.

Le Contexte

Le `context` fournit des objets/paramètres en plus aux composants Vue et est disponible dans des lifecycles Nuxt spéciaux comme par exemple `asyncData`, `fetch`, `plugins`, `middleware` et `nuxtServerInit`.

Note: Le "context" auquel on fait référence ici ne doit pas être confondu avec l'objet `context` disponible dans les actions Vuex. Les deux n'ont rien à voir.

```
function (context) {
  // Paramètres universels
  const {
    app,
    store,
    route,
    params,
    query,
    env,
    isDev,
    isHMR,
    redirect,
    error,
    $config
  } = context
  // Côté serveur
  if (process.server) {
    const { req, res, beforeNuxtRender } = context
  }
  // Côté client
  if (process.client) {
    const { from, nuxtState } = context
  }
}
```

Paramètres universels

Ce sont les paramètres qui sont disponibles côté client et côté serveur.

app

`app` (*NuxtAppOptions*)

L'instance principale de Vue avec les options qui incluent tous vos plugins. Par exemple, lorsque l'on utilise `i18n`, on peut avoir accès à `$i18n` grâce à `context.app.i18n`.

store

`store` (*Store Vuex*)

L'instance du Store Vuex. **Disponible uniquement si le `store vuex` a été défini.**

route

`route` ([Vue Router Route](#))

L'instance du Routeur de Vue.

params

`params` (*Objet*)

Un alias de `route.params`.

query

`query` (*Objet*)

Un alias de `route.query`.

env

`env` (*Objet*)

Variables d'environnement définies dans le fichier `nuxt.config.js`, se référer à l'[API env](#).

isDev

`isDev` (*Booléen*)

Un booléen qui permet de savoir si l'on est en mode développement, cela peut être utile pour mettre en cache de la data en production.

isHMR

`isHMR` (*Booléen*)

Un booléen qui permet de savoir si la méthode/middleware est appellé depuis le module de remplacement à chaud de webpack (*vrai seulement du côté client et en mode développement*).

redirect

`redirect` (*Fonction*)

Utilisez cette méthode pour rediriger l'utilisateur vers une autre route, le status code est utilisé du côté serveur, prend la valeur `302` par défaut. `redirect([status,] path [, query])`.

Exemples:

```
redirect(302, '/login')
redirect({ name: 'slug', params: { slug: mySlug } })
redirect('https://vuejs.org')
```

Voir le [Vue Router docs](#) more info on the Location property.

Info

Il n'est pas possible d'utiliser `redirect` ou `error` dans [client-side Nuxt plugin](#) en raison d'erreurs d'hydratation (le contenu du client serait différent de ce qu'il attend du serveur).

Une solution de contournement valide serait d'utiliser `window.onNuxtReady(() => { window.$nuxt.$router.push('/votre-route') })`

error

`error (Fonction)`

Utilisez cette méthode pour afficher la page d'erreur: `error(paramètres)`. Les `paramètres` doivent avoir les propriétés `statusCode` et `message`.

\$config

`$config (Objet)`

La [configuration de l'exécution](#) même.

Paramètres côté serveur

Ces paramètres ne sont disponibles que du côté serveur.

req

`req (http.Request)`

Requête provenant du serveur Node.js. Si Nuxt est utilisé en tant que middleware, l'objet de la requête risque d'être différent en fonction du framework que l'on utilise.

Non disponible lors d'un `nuxt generate`.

Res

`res (http.Response)`

Réponse du serveur Node.js. Si Nuxt est utilisé en tant que middleware, l'objet de la réponse peut être différent en fonction du framework que l'on utilise.

Non disponible lors d'un `nuxt generate`.

beforeNuxtRender

`beforeNuxtRender(fn) (Fonction)`

Utilisez cette méthode pour mettre à jour la variable `__NUXT__` render du côté client, `fn` (peut être asynchrone) est appelé avec `{ Components, nuxtState }`, voir cet [exemple](#).

Paramètres côté client

Ces paramètres ne sont disponibles que du côté client.

from

`from` (*Vue Router Route*)

La route d'où l'on vient.

nuxtState

`nuxtState` (*Objet*)

Le state de Nuxt, utile pour les plugins qui utilisent `beforeNuxtRender` pour avoir le state de Nuxt avant l'hydratation. **Disponible uniquement dans le mode** `universal`.

nuxt.render(req, res)

Vous pouvez utiliser Nuxt comme middleware pour votre serveur Node.js.

- Type: `Function`
- Arguments:
 - `Request`
 - `Response`
- Returns: `Promise`

Exemple avec `Express`:

```
const { loadNuxt, build } = require('nuxt')

const app = require('express')()
const isDev = process.env.NODE_ENV !== 'production'
const port = process.env.PORT || 3000

async function start() {
  // Nous obtenons l'instance Nuxt
  const nuxt = await loadNuxt(isDev ? 'dev' : 'start')

  // On fournit chaque route avec Nuxt
  app.use(nuxt.render)

  // Construit uniquement en mode développement avec rechargement à chaud
  if (isDev) {
    build(nuxt)
  }
  // Écoute le serveur
  app.listen(port, '0.0.0.0')
  console.log(`Server listening on `localhost:${port}`.`)
}

start()
```

Warning

Il est recommandé d'appeler `nuxt.render` à la fin de vos middlewares car il s'occupera du rendu de votre application web et n'appellera pas `next()`.

[Go to TOC](#)

nuxt.renderRoute(route, context)

Rendu d'une route spécifique avec un contexte donné.

- Type: `Function`
- Arguments:
 1. `String` : route to render
 2. *Optional*, `Object`, context given, available keys: `req` & `res`
- Returns: `Promise`
 - `html` : `String`
 - `error` : `null` or `Object`
 - `redirected` : `false` or `Object`

Cette méthode doit être utilisée principalement à des fins d'essai ainsi qu'avec `nuxt.renderAndgetWindow`.

Warning

`nuxt.renderRoute` doit être exécuté après le processus de construction en mode production.

```
const { loadNuxt, build } = require('nuxt')

async function start() {
  // Récupère une instance Nuxt pour commencer (en mode production)
  // Assurez-vous d'avoir exécuté `nuxt build` avant de lancer ce script
  const nuxt = await loadNuxt({ for: 'start' })

  const { html, error, redirected } = await nuxt.renderRoute('/')

  // `html` sera toujours une chaîne de caractères

  // `error` ne sera pas nulle lorsque la mise en page de l'erreur est affichée,
  // le format de l'erreur l'est :
  // { statusCode: 500, message: 'Mon message d\'erreur' }

  // `redirected` n'est pas `false` lorsque `redirect()` a été utilisé dans
  // `asyncData()` ou `fetch()`.

  // { path: '/other-path', query: {}, status: 302 }
}

start()
```

nuxt.renderAndGetWindow(url, options)

Récupère la `window` d'une URL donnée de l'application Nuxt.

- Type : `Function`
- Argument : `String`
 1. `String` : URL à afficher
 2. *Optional*, `Object` : options
 - `virtualConsole`: `Boolean` (par défaut : `true`)
- Renvois : `Promise`
 - Renvois : `window`

Warning

Cette méthode est faite à des fins de test.

Pour utiliser cette fonction, vous devez installer `jsdom` :

```
npm install --save-dev jsdom
```

Exemple :

```
const { loadNuxt } = require('nuxt')

async function init() {
  // En supposant que vous ayez déjà construit votre projet
  const nuxt = await loadNuxt({ for: 'start' })
  await nuxt.listen(3000)
  const window = await nuxt.renderAndGetWindow('http://localhost:3000')
  // Afficher le `<titre>`
  console.log(window.document.title)
  nuxt.close()
}

init()
```

Prochaine étape

Next

Consultez le [Glossaire des composants](#)

[Go to TOC](#)

\$nuxt: Le helper Nuxt helper

`$nuxt` est un helper conçu pour améliorer l'expérience de l'utilisateur. Pour plus d'informations sur les helpers Nuxt, consultez le [chapitre sur le contexte et les aides dans le livre des concepts](#)

Contrôle de connexion

- `isOffline`
 - Type: `Boolean`
 - Description: `true` lorsque la connexion internet de l'utilisateur est hors ligne
- `isOnline`
 - Type: `Boolean`
 - Description: L'opposé de `isOffline`

```
<template>
  <div>
    <div v-if="$nuxt.isOffline">Vous êtes hors ligne</div>
    <nuxt />
  </div>
</template>
```

Rafraîchissement des données de la page

- `refresh()`
 - Lorsque vous souhaitez rafraîchir les données fournies par `asyncData` ou `fetch`

```
<template>
  <div>
    <div>{{ content }}</div>
    <button @click="refresh">Rafraîchir</button>
  </div>
</template>

<script>
  export default {
    asyncData() {
      return { content: 'Créé à : ' + new Date() }
    },
    methods: {
      refresh() {
        this.$nuxt.refresh()
      }
    }
  }
</script>
```

Contrôle de la barre de chargement

- `$loading`
 - Lorsque vous voulez contrôler la barre de chargement de Nuxt

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Introduction aux modules Nuxt

Mieux comprendre les modules internes de Nuxt

Nuxt a une architecture entièrement modulaire qui permet aux développeurs d'étendre n'importe quelle partie de Nuxt Core en utilisant une API flexible.

Pour plus d'informations détaillées, veuillez consulter le [Guide des modules](#) si vous souhaitez développer votre propre module.

Cette section aide à se familiariser avec les modules internes de Nuxt et peut être utilisée comme référence pour mieux comprendre et écrire vos propres modules.

Noyau

Ces classes sont le cœur de Nuxt et doivent exister à la fois sur le runtime et le build time.

Nuxt

- [Classe](#) `Nuxt`
- Source: [core/nuxt.js](#)

Renderer

- [Classe](#) `Renderer`
- Source: [vue-renderer/renderer.js](#)

ModuleContainer

- [Classe](#) `ModuleContainer`
- Source: [core/module.js](#)

Build

Ces classes sont nécessaires que pour le mode build ou dev.

Builder

- [Classe](#) `Builder`
- Source: [builder/builder.js](#)

Generator

- [Classe](#) `Generator`
- Source: [generator/generator.js](#)

Général

Utils

- Source: [utils/src](#)

Options

- Source: [config/options.js](#)

Packaging & Usage

Par défaut, Nuxt exporte toutes les classes. Pour les importer :

```
import { Nuxt, Builder, Utils } from 'nuxt'
```

Patterns courant

Toutes les classes Nuxt font référence à l'instance `nuxt` et aux options, de cette façon nous avons toujours une API cohérente entre les classes pour accéder aux `options` et `nuxt`.

```
class SomeClass {
  constructor(nuxt) {
    super()
    this.nuxt = nuxt
    this.options = nuxt.options
  }

  someFunction() {
    // Nous avons accès à `this.nuxt` et `this.options`
  }
}
```

Les classes sont *plugables* donc elles doivent être enregistré en tant que plugin sur le conteneur principal `nuxt` pour avoir plus de hooks.

```
class FooClass {
  constructor(nuxt) {
    super()
    this.nuxt = nuxt
    this.options = nuxt.options

    this.nuxt.callHook('foo', this)
  }
}
```

On peut avoir un hook au module `foo` comme cela :

```
nuxt.hook('foo', foo => {
  // ...
})
```

[Go to TOC](#)

La classe Nuxt

C'est le conteneur central qui permet à tous les modules et à toutes les classes de communiquer entre eux. Tous les modules ont accès à l'instance Nuxt en utilisant `this.nuxt`.

- Source: [core/nuxt.js](#)

Hooks

Nous pouvons enregistrer des hooks sur certains événements du cycle de vie.

```
nuxt.hook('ready', async nuxt => {
  // Votre code personnalisé ici
})
```

| Plugin | Arguments | Quand | | | `ready` | (nuxt) | Nuxt est prêt à travailler (`ModuleContainer` et `Render`er prêt). | | `error` | (error) | Une erreur non gérée lors de l'appel de hooks. | | `close` | (nuxt) | La dernière instance se termine en douceur. | | `listen` | (server, {host, port}) | Le serveur Nuxt **interne** commence à écouter. (Utilisez `nuxt start` ou `nuxt dev`). |

La classe `Renderer`

Cette classe exporte un middleware de connexion qui traite et sert toutes les requêtes des ressources et SSR.

-
- Source: [vue-renderer/renderer.js](#)

Hooks

Nous pouvons enregistrer des hooks sur certains événements du cycle de vie.

| Hook | Arguments | Quand | | | `render:before` | (renderer, options) | Avant de mettre en place un middleware et des ressources pour la classe Renderer, utile pour surcharger certaines méthodes ou options.
| | `render:setupMiddleware` | (app) *connect instance* | Avant que Nuxt n'ajoute sa pile de middleware. Nous pouvons l'utiliser pour enregistrer le rendu personnalisés côté serveur (SSR). | | `render:errorMiddleware` | (app) *connect instance* | Avant d'ajouter le middleware d'erreur de Nuxt, utile pour ajouter votre propre middleware avant d'utiliser celui de Nuxt. Voir le [module Sentry](#) pour plus d'informations. | | `render:resourcesLoaded` | (resources) | Appelé après le chargement des ressources pour le moteur de rendu (client manifest, server bundle, etc). | | `render:done` | (renderer) | SSR Middleware et toutes les ressources sont prêtes (Renderer ready). | | `render:routeContext` | (context.nuxt) | *Chaque fois qu'une route est rendue par le serveur et avant le hook render:route*. Appelé avant la sérialisation du contexte Nuxt dans `window.__NUXT__`, utile pour ajouter des données que vous pouvez récupérer côté client. | | `render:route` | (url, result, context) | *Chaque fois qu'une route est rendue par le serveur*. Appelé avant de renvoyer la requête au navigateur. | | `render:routeDone` | (url, result, context) | *Chaque fois qu'une route est rendue par le serveur*. Appelé après que la réponse a été envoyée au navigateur. |

[Go to TOC](#)

Conteneur de module

Conteneur de module Nuxt

- Source: [core/module.js](#)

Tous les `modules` seront appelés dans le contexte de l'instance `ModuleContainer`.

Plugins à usage unique

Nous pouvons enregistrer des hooks sur certains événements du cycle de vie.

```
nuxt.moduleContainer.plugin('ready', async moduleContainer => {
  // Faites-le après que tous les modules soient prêts
})
```

Dans le contexte des `modules`, nous pouvons utiliser cette méthode à la place :

```
this.plugin('ready', async moduleContainer => {
  // Faites-le après que tous les modules soient prêts
})
```

| Plugin | Arguments | Quand | | | `ready` | `moduleContainer` | Tous les modules dans `nuxt.config.js` ont été initialisés |

Méthodes

`addVendor (vendor)`

Déprécié car le terme `vendor` n'est plus utilisé

Ajoute à `options.build.vendor` et applique un filtre unique.

`addTemplate (template)`

- `template`: `String` or `Object`
 - `src`
 - `options`
 - `fileName`

Affiche le modèle en utilisant `lodash template` pendant l'intégration au projet `buildDir` (`.nuxt`).

Si `fileName` n'est pas fourni ou si `template` est une chaîne de caractères, le nom du fichier cible est par défaut `[dirName].[fileName].[pathHash].[ext]`.

Cette méthode renvoie l'objet final `{ dst, src, options }`.

addPlugin (template)

- **template:** Object à propriétés (`src`, `options`, `fileName`, `mode`).

Enregistre un plugin en utilisant `addTemplate` et l'ajoute à la première ligne des options `plugins[]`.

Vous pouvez utiliser `template.ssr : false` pour désactiver le plugin y compris dans le bundle SSR.

```
this.addPlugin({
  src: path.resolve(__dirname, 'templates/foo.js'),
  fileName: 'foo.server.js' // [optionnel] inclus seulement dans un serveur bundle
  options: moduleOptions
})
```

Note: Vous pouvez utiliser `mode` ou `.client` et `.server` modificateur avec `fileName` option pour utiliser le plugin uniquement côté client ou côté serveur. (Voir `plugins` pour toutes les options valides)

Si vous choisissez de spécifier un `fileName`, vous pouvez également configurer un chemin personnalisé pour le `fileName`, afin que vous puissiez choisir la structure des dossiers dans le dossier `.nuxt` afin d'éviter les conflits de noms :

```
{
  fileName: path.join('folder', 'foo.client.js'), // will result in
  `._nuxt/folder/foo.client.js`
}
```

addServerMiddleware (middleware)

Pousse le middleware dans `options.serverMiddleware`.

extendBuild (fn)

Permet d'étendre facilement la configuration du webpack par chaînage dans la fonction `options.build.extend`.

extendRoutes (fn)

Permet d'étendre facilement les routes dans la fonction `options.build.extendRoutes`.

addModule (moduleOpts, requireOnce)

Async function

Enregistre un module. Les `moduleOpts` peuvent être une chaîne de caractères ou un tableau (`[src, options]`). Si `requireOnce` est `true` et que le module est résolu dans les `meta`, cela empêchera d'enregistrer deux fois le même module.

requireModule (moduleOpts)

Async function

Est un raccourci pour `addModule(moduleOpts, true)`

Hooks

Nous pouvons enregistrer des hooks sur certains événements du cycle de vie.

| Hook | Arguments | Quand | | | | `modules:before` | (moduleContainer, options) | Appelé avant la création de la classe ModuleContainer, utile pour surcharger les méthodes et les options. | | `modules:done` | (moduleContainer) | Appelé lorsque tous les modules ont été chargés. |

La classe Builder

La classe `Builder` de Nuxt

- Source: [builder/builder.js](#)

Hooks

Nous pouvons enregistrer des hooks sur certains événements du cycle de vie.

```
// Ajout d'hook pour le build
this.nuxt.hook('build:done', (builder) => {
  ...
})
```

| Hook | Arguments | Quand | | | `build:before` | (nuxt, buildOptions) | Avant que le build de Nuxt commence | | `builder:prepared` | (nuxt, buildOptions) | Les répertoires de build ont été créés | | `builder:extendPlugins` | (plugins) | Génération de plugins | | `build:templates` | ({ templatesFiles, templateVars, resolve }) | Génération de fichiers modèles `.nuxt` | | `build:extendRoutes` | (routes, resolve) | Génération des routes | | `webpack:config` | (webpackConfigs) | Avant la configuration des compilateurs | | `build:compile` | ({ name, compiler }) | Avant que webpack compile (le compilateur est une instance du webpack `Compiler`), si mode universel, appelé deux fois avec le nom `'client'` et `'server'` | | `build:compiled` | ({ name, compiler, stats }) | Le build webpack est terminée | | `build:done` | (nuxt) | Le build de Nuxt est terminé |

[Go to TOC](#)

La classe Generator

La classe Generator de Nuxt

-
- Source: [generator/generator.js](#)

Hooks

```
generate: hooks:
```

```
| Hook | Arguments | Quand | | | generate:before | (generator, generateOptions) | Hook avant la génération de la page | | generate:distRemoved | (generator) | Hook sur le répertoire de destination nettoyé | | generate:distCopied | (generator) | Hook sur la copie des fichiers statiques et du build | | generate:route | ({ route, setPayload }) | Hook avant de générer la page, utile pour la charge utile dynamique, voir #7422, disponible pour Nuxt 2.13+ | | generate:page | ({ route, path, html }) | Hook pour permettre à l'utilisateur de mettre à jour le chemin et l'html après la génération | | generate:routeCreated | ({ route, path, errors }) | Hook sur la sauvegarde du succès des pages générées | | generate:extendRoutes | (routes) | Hook pour permettre à l'utilisateur de mettre à jour les routes à générer | | generate:routeFailed | ({ route, errors }) | Hook sur l'enregistrement de l'échec de la page générée | | generate:done | (generator, errors) | Hook sur la génération terminée |
```

Utiliser Nuxt programmatiquement

Vous pouvez utiliser Nuxt par programmation pour l'utiliser comme un middleware vous donnant la liberté de créer votre propre serveur pour le rendu de vos applications web.

Vous pouvez utiliser votre propre serveur avec votre middleware et votre API. C'est pourquoi vous pouvez utiliser Nuxt par programmation.

Le constructeur Nuxt

Pour voir la liste des options à donner à Nuxt, voir la section configuration.

```
const { loadNuxt, build } = require('nuxt')

// Vérifie si Nuxt fonctionne en mode développement
const isDev = process.env.NODE_ENV !== 'production'

// Obtiens une instance Nuxt prête à l'emploi
const nuxt = await loadNuxt(isDev ? 'dev' : 'start')

// Permet la construction en direct et le rechargement en mode développement
if (isDev) {
  build(nuxt)
}

// Nous pouvons utiliser `nuxt.render(req, res)` ou `nuxt.renderRoute(route,
context)`.
```

Vous pouvez consulter les sites [nuxt-express](#) et [adonuxt](#) pour démarrer rapidement.

Journaux de debug

Si vous souhaitez afficher les logs de Nuxt, vous pouvez ajouter ce qui suit en haut de votre fichier :

```
process.env.DEBUG = 'nuxt:*
```

[Go to TOC](#)

La méthode fetch

La méthode fetch est utilisée pour remplir le store avant de render la page, c'est similaire à `asyncData` sauf que cela ne remplit pas le `data` du composant.

Nuxt >= 2.12

Avec l'arrivée de la `v2.12` de Nuxt, nous avons droit à un nouveau hook disponible dans **tous les composants Vue**: `fetch`.

Info

`fetch(context)` a été déprécié, à la place on peut utiliser un **middleware anonyme** dans la page: `middleware(context)`

Quand utiliser fetch?

À chaque fois que l'on a besoin d'avoir de la data **asynchrone**. `fetch` est appelé côté serveur pour render les routes et côté client lors de la navigation.

Il expose `$fetchState` au niveau du composant:

- `$fetchState.pending : Boolean`, permet d'afficher un placeholder pendant que fetch est appelé (côté client).
- `$fetchState.error : null ou Error`, vous permet d'afficher un message d'erreur
- `$fetchState.timestamp : Integer`, l'horodatage du dernier fetch, utile en combinaison avec `keepalive` pour de la mise en cache.

Si vous souhaitez appeler le hook `fetch` depuis votre template, utilisez:

```
<button @click="$fetch">Rafraîchir</button>
```

ou la méthode dans le composant:

```
// méthode dans la section script
export default {
  methods: {
    refresh() {
      this.$fetch()
    }
  }
}
```

On peut accéder au `context` de Nuxt à l'intérieur du hook `fetch` en utilisant `this.$nuxt.context`.

Options

- `fetchOnServer` : `Boolean` ou `Function` (par défaut: `true`), appelle `fetch()` lors d'un render de page côté serveur
- `fetchDelay` : `Integer` (défaut: `200`), définit un temps d'exécution minimal en millisecondes (pour éviter les flashes)

Info

Quand `fetchOnServer` est `false` (`false` ou retourne `false`), `fetch` sera appelé uniquement côté client et `$fetchState.pending` retournera `true` lors d'un render côté serveur du composant.

```
<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    async fetch() {
      this.posts = await this.$http.$get(
        'https://jsonplaceholder.typicode.com/posts'
      )
    },
    fetchOnServer: false
  }
</script>
```

La propriété watchQuery

Surveille les query strings et exécute des méthodes de composant lors d'un changement (asyncData, fetch(context), validate, layout, ...).

- **Type:** Boolean ou Array ou Function (par défaut: [])

Utilisez la propriété `watchQuery` pour surveiller les query strings. Si les strings changent, toutes les méthodes du composant (asyncData, fetch(context), validate, layout, ...) seront appelées. La surveillance est désactivée par défaut pour des raisons de performance.

Si l'on souhaite surveiller toutes les query strings, il faut définir `watchQuery: true`.

```
export default {
  watchQuery: ['page']
}
```

On peut aussi utiliser la fonction `watchQuery(nouvelleRequete, ancienneRequete)` pour avoir une surveillance plus précise.

```
export default {
  watchQuery(nouvelleRequete, ancienneRequete) {
    // Exécute seulement les méthodes du composant dans le cas où l'ancienne query
    // string
    // contenait la chaîne de caractères `bar` et la nouvelle `foo`
    return nouvelleRequete.foo && ancienneRequete.bar
  }
}
```

Warning

Le nouveau hook `fetch` (disponible depuis la version 2.12) n'est pas affecté par `watchQuery`. Pour plus d'informations, se référer aux [changements sur la surveillance des query strings](#).

La méthode head

Nuxt utilise `vue-meta` pour mettre à jour les `entêtes` et `attributs HTML` de votre application.

- **Type:** `Object` ou `Function`

Utilisez la méthode `head` pour définir les balises d'entête HTML de votre page actuelle.

```
<template>
  <h1>{{ titre }}</h1>
</template>

<script>
  export default {
    data() {
      return {
        titre: 'Salut le monde !'
      }
    },
    head() {
      return {
        title: this.titre,
        meta: [
          // `hid` est un identifiant unique. N'utilisez pas `vmid` pour cela car
          // cela ne marchera pas.
          {
            hid: 'description',
            name: 'description',
            content: 'Ma description personnalisée'
          }
        ]
      }
    }
  }
</script>
```

La propriété key

Définit la propriété `key` pour un composant `<router-view>` interne

- **Type:** `String` ou `Function`

La propriété `key` est propagée dans le `<router-view>`, ce qui est utile pour faire des transitions dans une route dynamique et avec une route différente. Des clés différentes résulteront en un re-render des composants de la page.

Il y a plusieurs façons de définir la clé. Pour plus de détails, veuillez vous référer à la propriété `nuxtChildKey` dans le composant `nuxt`.

```
export default {
  key(route) {
    return route fullPath
  }
}
```

La propriété layout

Chaque fichier (du premier niveau) dans le répertoire des `layouts` créera un layout personnalisé accessible depuis la propriété `layout` dans le composant page.

- **Type:** `String` ou `Function` (par défaut: `'default'`)

Utilisez la clé `layout` dans vos composants pages pour définir un layout à utiliser:

```
export default {
  layout: 'blog',
  // OU
  layout(context) {
    return 'blog'
  }
}
```

La propriété loading

La propriété loading vous donne la possibilité de désactiver la barre de progression par défaut sur une page spécifique.

- **Type:** Boolean (par défaut: true)

Par défaut, Nuxt utilise son propre composant pour afficher une barre de progression entre les routes.

Vous pouvez la désactiver ou la personnaliser à travers les [options de configuration du chargement](#), mais aussi la désactiver seulement sur des pages spécifiques en passant la propriété loading à false:

```
<template>
  <h1>Ma page</h1>
</template>

<script>
  export default {
    loading: false
  }
</script>
```

La propriété middleware

Définit le middleware pour une page spécifique de l'application.

- Type: `String` or `Array` or `Function`
 - Items: `String` or `Function`

Middleware nommé

Vous pouvez nommer votre middleware en créant un fichier à l'intérieur du répertoire `middleware/`, le nom du fichier sera le nom du middleware.

```
export default function ({ store, redirect }) {
  // Si l'utilisateur n'est pas authentifié
  if (!store.state.authenticated) {
    return redirect('/login')
  }
}
```

```
<template>
  <h1>Page secrète</h1>
</template>

<script>
  export default {
    middleware: 'authenticated'
  }
</script>
```

Middleware anonyme

Si l'on ressent le besoin d'utiliser un middleware seulement pour une page spécifique, on peut directement utiliser une fonction pour cela (voire même un tableau de fonctions):

```
<template>
  <h1>Page secrète</h1>
</template>

<script>
  export default {
    middleware({ store, redirect }) {
      // Si l'utilisateur n'est pas authentifié
      if (!store.state.authenticated) {
        return redirect('/login')
      }
    }
  }
</script>
```

[Go to TOC](#)

La propriété scrollTop

La propriété scrollTop indique à Nuxt qu'il faut défiler tout en haut de la page avant de la render.

- **Type:** Boolean (par défaut: false)

Par défaut, Nuxt défile tout en haut lorsque l'on va sur une autre page, mais avec les routes imbriquées, Nuxt garde la position du défilement. Si l'on souhaite que Nuxt défile tout en haut lors du render des routes imbriquées, il faut passer scrollTop à true :

```
<template>
  <h1>Mon composant enfant</h1>
</template>

<script>
  export default {
    scrollTop: true
  }
</script>
```

Inversement, on peut passer scrollTop à false sur les routes parent.

Si l'on souhaite écraser le comportement du défilement par défaut de Nuxt, veuillez vous référer à l'[option scrollBehavior](#).

La propriété transition

Nuxt utilise le composant `<transition>` pour nous permettre de créer et appliquer de superbes transitions/animations lors de la navigation entre les pages.

- **Type:** `String` ou `Object` ou `Function`

Pour définir une transition personnalisée pour une route spécifique, il suffit d'ajouter la propriété `transition` au composant page.

```
export default {
  // Peut être une chaîne de caractères
  transition: ''
  // Ou un Objet
  transition: {}
  // Ou une fonction
  transition (to, from) {}
```

Chaîne de caractères

Si le paramètre `transition` est défini en tant que chaîne de caractères, il sera utilisé sous la forme `transition.name`.

```
export default {
  transition: 'test'
}
```

Nuxt va utiliser ces paramètres pour définir le composant comme suit:

```
<transition name="test"></transition>
```

Objet

Si la propriété `transition` est définie en tant qu'objet:

```
export default {
  transition: {
    name: 'test',
    mode: 'out-in'
  }
}
```

Nuxt va utiliser ces paramètres pour définir le composant comme suit:

```
<transition name="test" mode="out-in"></transition>
```

L'objet `transition` peut avoir les propriétés suivantes:

| Propriété | Type | Par défaut | Définition | || | name | String | "page" | Le nom de la transition appliquée à toutes les transitions de routes. || mode | String | "out-in" | Le mode qui s'applique aux transitions sur toutes les routes, se référer à la [documentation de Vue.js](#). || css | Boolean | true | Définit l'application ou non des classes CSS sur les transitions. Si définit à false, seulement les hooks JavaScript définis par des événements seront exécutés. || duration | Integer | non défini | La durée (en millisecondes) appliquée à la transition, voir la [documentation de Vue.js](#). || type | String | non défini | Spécifie le type des événements de transition à attendre pour déterminer la fin d'une transition. Les valeurs disponibles sont "transition" et "animation". Par défaut, le type qui a la plus longue durée sera choisi. || enterClass | String | non défini | Le state du départ d'une classe de transition. Voir la [documentation de Vue.js](#). || enterToClass | String | non défini | Le state de fin d'une classe de transition. Voir la [documentation de Vue.js](#). || enterActiveClass | String | non défini | La classe appliquée pendant l'intégralité de la durée de la transition. Voir la [documentation de Vue.js](#). || leaveClass | String | non défini | Le state du départ d'une classe de transition. Voir la [documentation de Vue.js](#). || leaveToClass | String | non défini | Le state de fin d'une classe de transition. Voir la [documentation de Vue.js](#). || leaveActiveClass | String | non défini | La classe appliquée pendant l'intégralité de la durée de la transition. Voir la [documentation de Vue.js](#). |

On peut aussi définir des méthodes dans la propriété `transition` de la page, celles-ci sont pour des [hooks JavaScript](#):

- `beforeEnter(élément)`
- `enter(élément, terminé)`
- `afterEnter(élément)`
- `enterCancelled(élément)`
- `beforeLeave(élément)`
- `leave(élément, terminé)`
- `afterLeave(élément)`
- `leaveCancelled(élément)`

```
export default {
  transition: {
    afterLeave(élément) {
      console.log('afterLeave', élément)
    }
  }
}
```

Note: une bonne pratique serait d'explicitement déclarer `css:false` pour des transitions uniquement en JavaScript, ainsi Vue pourra ignorer la détection du CSS. Cela prévient de plus de potentiels conflits accidentels avec des règles CSS.

Mode de transition

Le mode de transition par défaut pour les pages est différent de celui de Vue.js. Il est défini à `out-in`. Si l'on souhaite exécuter les transitions entrantes et sortantes simultanément, il faut passer le mode à une chaîne de caractères vide `mode: ''`.

```
export default {
  transition: {
    name: 'test',
    mode: ''
  }
}
```

Fonction

Si la propriété `transition` est définie en tant que fonction:

```
export default {
  transition(to, from) {
    if (!from) {
      return 'slide-left'
    }
    return +to.query.page < +from.query.page ? 'slide-right' : 'slide-left'
  }
}
```

Voici ce que cela va générer comme transitions lors de la navigation:

- `/` to `/posts` => `slide-left`
- `/posts` to `/posts?page=3` => `slide-left`
- `/posts?page=3` to `/posts?page=2` => `slide-right`

[Go to TOC](#)

La méthode validate

Nuxt vous permet de définir une méthode de validation à l'intérieur de votre composant de routes dynamiques.

- **Type:** Function ou Async Function

`validate` est appelé avant chaque navigation sur une nouvelle route. Il sera appelé une fois côté serveur: lors de la première requête à l'application Nuxt et côté client: lors de la navigation sur d'autres routes . Cette méthode prend l'objet `context` en tant qu'argument.

```
validate({ params, query, store }) {
  return true // si les paramètres sont valides
  return false // ceci va arrêter Nuxt dans son render de la route et afficher la
page d'erreur
}
```

```
async validate({ params, query, store }) {
  // opérations asynchrone à attendre
  return true // si les paramètres sont valides
  return false // ceci va arrêter Nuxt dans son render de la route et afficher la
page d'erreur
}
```

On peut aussi renvoyer des promesses:

```
validate({ params, query, store }) {
  return new Promise((resolve) => setTimeout(() => resolve()))
}
```

Nuxt vous permet de définir une méthode de validation à l'intérieur de votre composant de routes dynamiques (dans notre exemple: `pages/users/_id.vue`).

Si la méthode de validation ne retourne pas `true`, Nuxt va automatiquement charger la page d'erreur 404.

```
export default {
  validate({ params }) {
    // Doit être un nombre
    return /^[^d+$/.test(params.id)
  }
}
```

On peut aussi vérifier de la data dans notre `store` (remplie par exemple par `nuxtServerInit` avant l'action):

```
export default {
  validate({ params, store }) {
    // On vérifie si `params.id` est une catégorie existante
    return store.state.categories.some(category => category.id === params.id)
  }
}
```

On peut aussi renvoyer des erreurs attendues (ou pas) durant l'exécution de la fonction de validation:

```
export default {
  async validate({ params, store }) {
    // Renvoie une erreur interne serveur 500 avec un message personnalisé
    throw new Error('En cours de construction !')
  }
}
```

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- Nuxt - Français
- GitHub source: [nuxt/nuxtjs.org](https://nuxtjs.org)
- Created: 2022-12-10
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>