Introduction au développement de bloc

COMMENT CRÉER VOTRE PREMIER BLOC

Vincent Dubroeucq

DÉVELOPPEUR, FORMATEUR, ORATEUR, AUTEUR DU WPCOOKBOOK

Objectifs

- Créer un bloc ~~bidon~~ simple
- Comprendre le fonctionnement et l'anatomie d'un bloc
- Avoir envie d'essayer

Ressources

- Le dépôt de cette présentation : https://github.com/vincedubroeucq/toto
- La documentation : https://developer.wordpress.org/block-editor

Créer un bloc

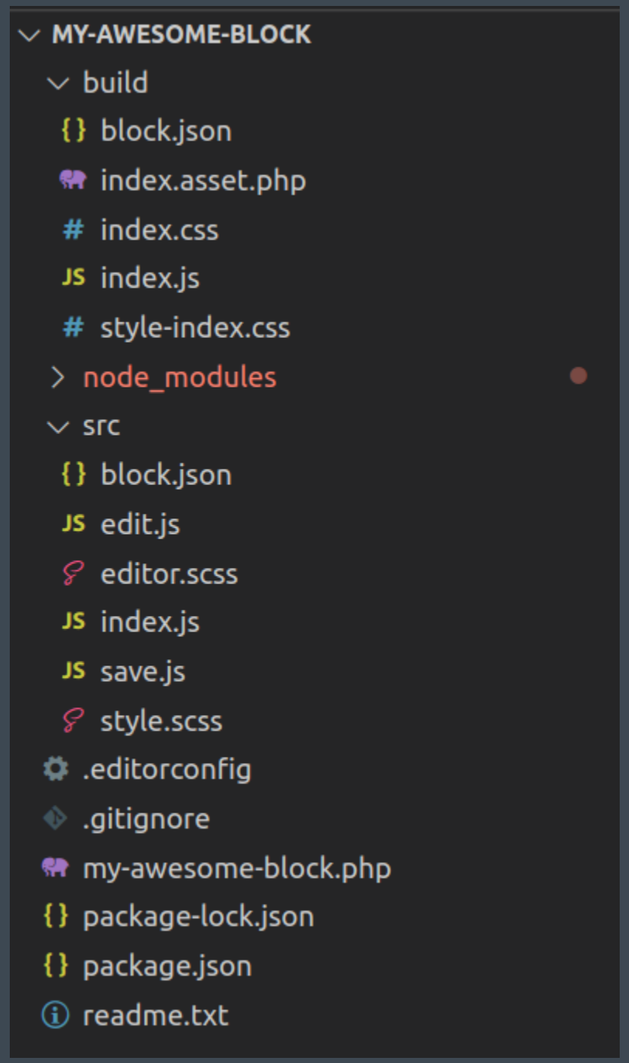
Dans un terminal, dans votre dossier `plugins/`

```
npx @wordpress/create-block my-awesome-block
```

=> génère automatiquement tous les fichiers nécessaires et installe les dépendances

Architecture

- `my-awesome-block.php` : fichier bootstrap de l'extension. Déclare le bloc.
- `package.json` : contient la liste des dépendances JavaScript et les raccourcis.
- `src/` : vos fichiers source
- `build/` : les fichiers de destination chargés par WordPress



my-awesome-block.php

Charge le bloc à partir du fichier `block.json` dans `build/`

```php
/**
 * Registers the block using the metadata loaded from the `block.json` file.
 * Behind the scenes, it registers also all assets so they can be enqueued
 * through the block editor in the corresponding context.
 *
 * @see https://developer.wordpress.org/reference/functions/register_block_type/
 */
function create_block_my_awesome_block_block_init() {
  register_block_type( __DIR__ . '/build' );
}
add_action( 'init', 'create_block_my_awesome_block_block_init' );
```

src/block.json

Contient toutes les metadonnées nécessaires pour déclarer le bloc :

- Nom
- Attributs (= les données du bloc)
- Supports (= les fonctionnalités natives du bloc)
- Styles de bloc
- Variations

```
{
  "$schema":
"https://schemas.wp.org/trunk/block.json",
  "apiVersion": 2,
  "name": "create-block/my-awesome-block",
  "version": "0.1.0",
  "title": "My Awesome Block",
  "category": "widgets",
  "icon": "smiley",
  "description": "...",
  "supports": {},
  "styles": [],
  "variations": [],
  "attributes": {},
  "textdomain": "my-awesome-block",
  "editorScript": "file:./index.js",
  "editorStyle": "file:./index.css",
  "style": "file:./style-index.css"
}
```

Commencer

Pour commencer à surveiller les fichiers source :

```
npm start
```

Pour builder pour la production :

```
npm run build
```

index.js
edit.js
save.js

- `index.js` est le fichier d'entrée, qui va importer les autres.
- `edit.js` exporte la fonction responsable de l'affichage dans l'éditeur
- `save.js` exporte la fonction responsable de la sauvegarde en BDD

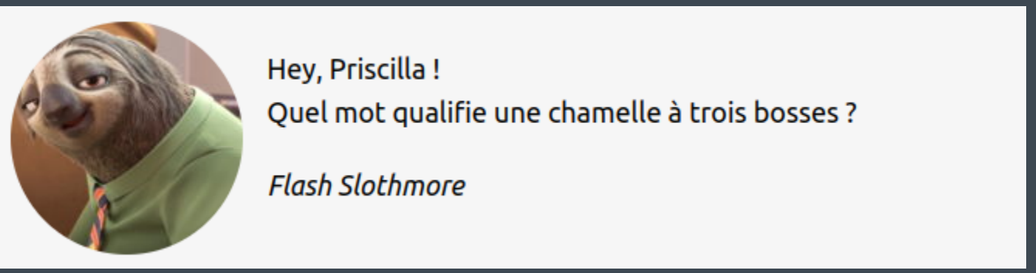
```
/* index.js */
/**
 * Internal dependencies
 */
import Edit from './edit';
import save from './save';
import metadata from './block.json';
/**
 * Every block starts by registering a new block type definition.
 */
registerBlockType( metadata.name, {
  edit: Edit,
  save,
} );
```

`style.scss`
`editor.scss`

- `style.scss` est importé dans le fichier d'entrée `index.js`, et contient les styles appliqués sur le devant du site ET l'éditeur.
- `editor.scss` est importé dans le fichier `edit.js`, et contient les styles appliqués dans l'éditeur uniquement.

`wp-scripts` s'occupe automatiquement de les lire et extraire les styles dans les fichiers `.css` correspondants.

Notre bloc




```
<blockquote class="wp-block-my-awesome-block">
  <div class="image">...</div>
  <div class="content">
    <p>...</p>
    <cite>...</cite>
  </div>
</blockquote>
```

`edit.js` et les attributs

`edit.js` doit renvoyer un composant JSX qui correspond à l'interface d'édition du bloc.

Les attributs correspondent aux données du bloc.
Dans `edit.js` l'objectif est de fournir l'interface pour modifier ces attributs.

```
import { __ } from '@wordpress/i18n';
import { useBlockProps } from '@wordpress/block-editor';
import './editor.scss';

export default function Edit( props ) {
  const { attributes, setAttributes } = props;
  return (
    <blockquote { ...useBlockProps() }>
      <div class="image">
        /* Notre champ image */
      </div>
      <div class="content">
        /* Nos champs RichText */
      </div>
    </blockquote>
  );
}
```

Attributs et `block.json`

Pour fonctionner correctement, les attributs doivent être déclarés dans `block.json`

```
{
    "$schema": "https://schemas.wp.org/trunk/block.json",
    "apiVersion": 2,
    "name": "create-block/my-awesome-block",
    ...
    "attributes":{
      "content": {
        "type" : "string",
        "default" :""
      },
      "source": {
        "type" : "string",
        "default" :""
      },
      "image": {
        "type" : "object",
        "default" : {}
      }
    },
    "textdomain": "my-awesome-block",
    ...
}
```

Le composant `RichText`

Affiche un editeur de texte avec boutons de formattage, comme le bloc paragraphe.

```
import { RichText } from '@wordpress/block-editor';

<RichText
  tagName="div"                 // HTML à appliquer autour du contenu
  value={ attributes.content } // Attribut correspondant au contenu
  allowedFormats={ [ 'core/bold', 'core/italic' ] } // Liste des formats autorisés
  multiline={true} // True pour créer des paragraphes à chaque saut de ligne.
  onChange={ content => setAttributes( { content } ) } // Fonction appelée à chaque changement dans le contenu. Ici, il faut simplement sauvegarder la valeur de l'attribut.
  placeholder={ __( 'This is awesome !', 'my-awesome-block' ) } // Texte de substitution quand le composant est vide.
/>
```

save.js

`save.js` doit renvoyer un composant JSX qui correspond à l'HTML à sauvegarder en base de données.

On va utiliser les attributs pour construire l'HTML du bloc.

```
import { useBlockProps, RichText } from '@wordpress/block-editor';

export default function save( { attributes } ) {
  const { content, source } = attributes;
  return (
    <blockquote { ...useBlockProps.save() }>
      <div className="image">IMG here</div>
      <div className="content"}>
        <RichText.Content value={ content } />
        <RichText.Content tagName="cite" value={ source } />
      </div>
    </blockquote>
  );
}
```

Le composant `<MediaUpload />`

Affiche une interface pour téléverser des medias.

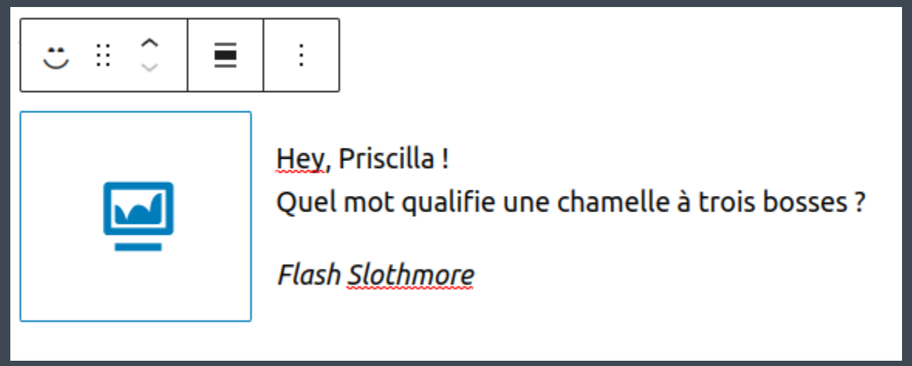
```
import { MediaUploadCheck, MediaUpload } from '@wordpress/block-editor';

<MediaUploadCheck>
  <MediaUpload
    onSelect={onSelectMedia} // Fonction de rappel exécutée quand un media est selectionné
    value={ image.id || 0 }      // Media sélectionné par défaut
    allowedTypes={ [ 'image' ] } // Liste des medias autorisés
    render={ ( { open } ) => (...)} // Fonction affichant l'interface pour ouvrir la bibliothèque
  />
<MediaUploadCheck>
```

edit.js

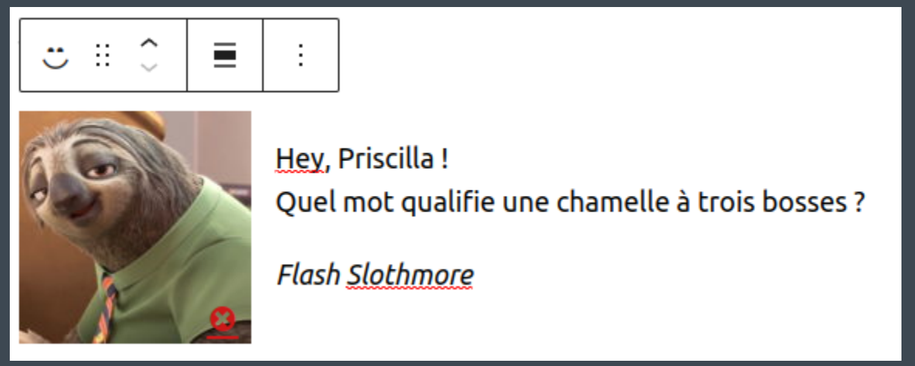
```
export default function Edit( { attributes, setAttributes } ) {
  const imageAttributes = {src: image.src, alt: image.alt};
  const onDeleteImage = () => { setAttributes( { image: {} } ) };
  return (
    <blockquote { ...useBlockProps() }>
      <div className="image">
        {image.id ?
          <div className="wrapper image-wrapper">
            <img {...imageAttributes} />
            <Button onClick={ onDeleteImage }>...</Button>
          </div>
        :
          <MediaUploadCheck>
            <MediaUpload ... />
          </MediaUploadCheck>
        }
      </div>
      <div className="content">
        <RichText .../>
        <RichText .../>
      </div>
    </blockquote>
  );
}
```

Dans l'éditeur

Sans image :



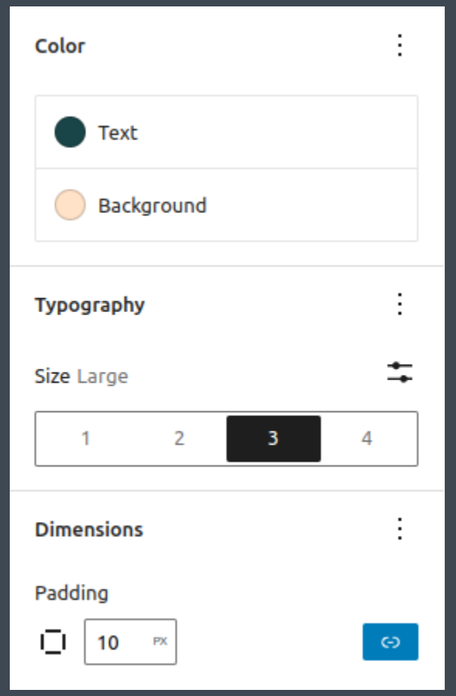
Avec image :



Supports

C'est l'ensemble des fonctionnalités natives de WordPress que le bloc va supporter.
On les déclare dans `block.json`

```
{
      "name": "create-block/my-awesome-block"
      ...
      "supports": {
        "align": true,
        "html": false,
        "color": true,
        "spacing": true,
        "typography": {
          "fontSize": true,
          "lineHeight": true
        }
      },
      ...
    }
```



Hey, Priscilla !
Quel mot qualifie une chamelle à trois bosses ?

Flash Slothmore

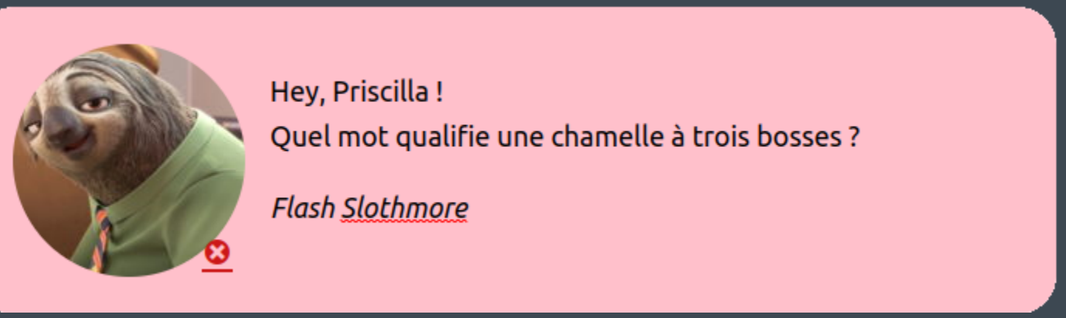
Styles

Les styles différents pour le bloc. Se déclarent aussi dans `block.json`.

Chaque style déclaré ajoute simplement une classe CSS sur le bloc.

```
{
  "name": "wp-block/my-awesome-block"
  ...
  "styles": [
    { "name": "default", "label": "Default",
"isDefault": true },
    { "name": "rounded", "label": "Rounded" }
  ],
  ...
}
```

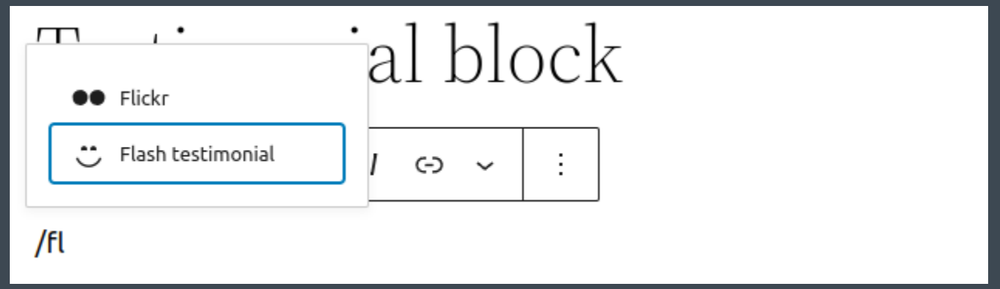
```
/* style.scss */
.wp-block-create-block-my-awesome-block {
  ...
  &.is-style-rounded {
    background: pink;
    border-radius: 25px;
    padding: 1.5rem;
    img {
      border-radius: 50%;
    }
  }
}
```



Variations

Une variation est un bloc pré-rempli, avec des attributs par défaut.

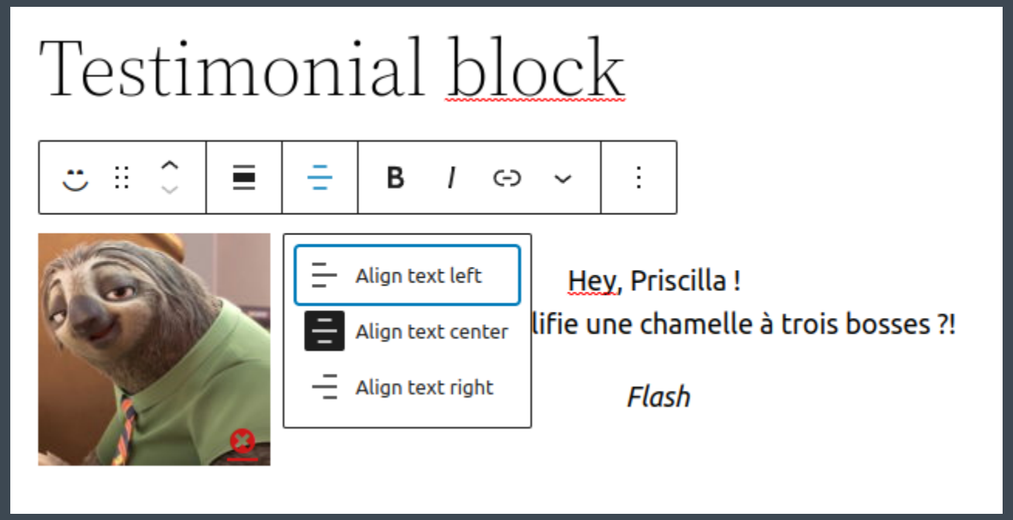
```
{
  "name": "wp-block/my-awesome-block"
  ...
  "variations": [
        {
            "name": "flash",
            "title": "Flash testimonial",
            "attributes": {
                "source": "Flash",
                "content": "<p>Hey, Priscilla !
<br>Quel mot qualifie une chamelle à trois
bosses ?!</p>"
            }
        }
    ],
  ...
}
```



Toolbar

Pour créer des boutons dans la barre d'outils, on utilise le composant `<BlockControls>`.
Attention ! La barre d'outil se déclare **hors du bloc**, dans un `<Fragment>`.

On dispose aussi d'un composant `<AlignmentToolbar>` pour gérer l'alignement du texte.




```
// edit.js
import { BlockControls, AlignmentToolbar } from
'@wordpress/block-editor';

export default function edit( { attributes,
setAttributes } ) {
  return (
    <>
      <BlockControls>
        <AlignmentToolbar
          value={attributes.textAlign}
          onChange={ textAlign =>
setAttributes({ textAlign })}
        />
      </BlockControls>
      <blockquote { ...useBlockProps() }>
        ...
      </blockquote>
    </>
  );
}
```

Toolbar - CONTINUED

Le composant `<ToolbarGroup>` permet de grouper les boutons,
et `<ToolbarButton>` de créer des boutons personnalisés.

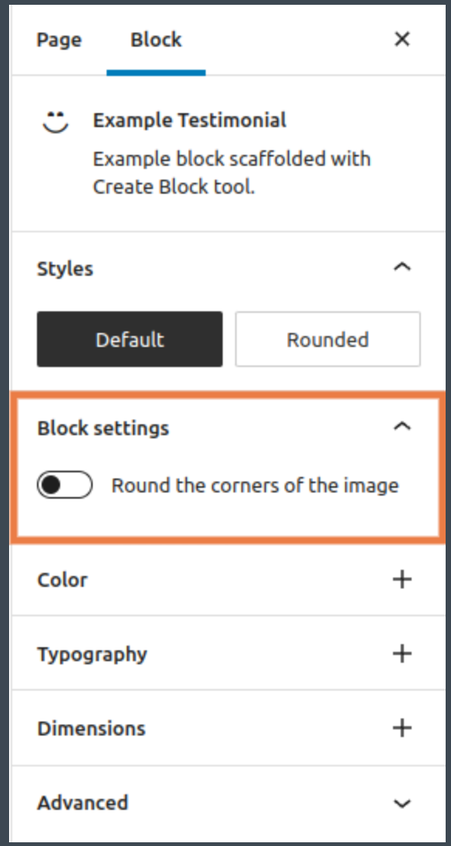
```
// edit.js
import { BlockControls, AlignmentToolbar } from '@wordpress/block-editor';
import { ToolbarButton, ToolbarGroup } from '@wordpress/components';

export default function edit( { attributes, setAttributes } ) {
  return (
    <>
      <BlockControls>
        <AlignmentToolbar ... />
        <ToolbarGroup label={ __( 'Extra options', 'my-awesome-block' ) }>
          <ToolbarButton
            icon="warning"
            label={__( 'Does nothing for now !', 'my-awesome-block' )}
            onClick={ () => alert( __( 'Nothing !', 'my-awesome-block' ) ) }
          />
        </ToolbarGroup>
      </BlockControls>
      <blockquote { ...useBlockProps() }>
        ...
      </blockquote>
    </>
  );
}
```

Inspector Controls

Le composant <InspectorControls> permet de placer des réglages pour le bloc dans l'inspecteur, sur le côté.

Comme le composant `<BlockControls>`, il faut le déclarer hors du bloc.

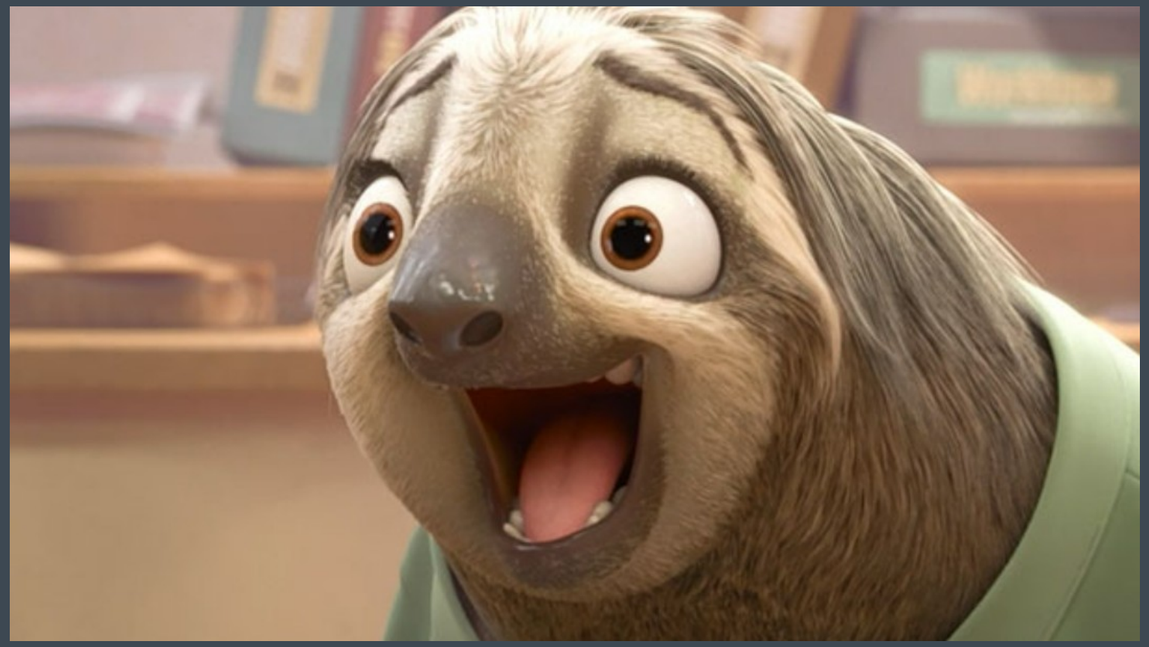
```
import { InspectorControls } from '@wordpress/block-editor';
import { Panel, PanelBody, ToggleControl } from '@wordpress/components';

export default function edit( { attributes, setAttributes } ) {
  return (
    <>
      <BlockControls>...</BlockControls>
      <InspectorControls>
        <Panel>
          <PanelBody title={__( 'Block settings', 'my-awesome-block' )}>
            <ToggleControl
              label={__( 'Round the corners of the image', 'my-awesome-block' )}
              checked={ attributes.rounded }
              onChange={ rounded => setAttributes( { rounded } ) }
            />
          </PanelBody>
        </Panel>
      </InspectorControls>
      <blockquote { ...useBlockProps() }>...</blockquote>
    </>
  );
}
```



Wow ! That's a lot !

Maintenant, vous savez :

- Créer la structure du bloc rapidement avec `create-block`
- Comment sont organisés les fichiers
- Comment fonctionnent les attributs des blocs
- Utiliser les composants `<RichText>` et `<MediaUpload>`
- Ajouter le support pour les options de base des blocs (Block Supports)
- Ajouter des styles de blocs et des variations
- Ajouter des réglages dans la Toolbar et dans l'inspecteur



Merci ! Des questions ?

 @vincedubroeucq

 vincent@vincentdubroeucq.com

 https://vincentdubroeucq.com

 https://github.com/vincedubroeucq/toto