



Introduction au développement de **blocs** pour l'éditeur

Vincent Dubroeuq

40 min • Développement - Gutenberg

PLANNING



Introduction au développement de blocs

COMMENT CRÉER VOTRE PREMIER BLOC POUR LE NOUVEL ÉDITEUR





Vincent Dubroeuq

DÉVELOPPEUR, FORMATEUR, ORATEUR, AUTEUR DU [WPCOOKBOOK](#)



Objectifs

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



Ressources

- Le dépôt de cette présentation : <https://github.com/vincedubroeucq/WCLyon2022>
- 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 racine 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
  ▼ build
    {} block.json
    🐘 index.asset.php
    # index.css
    JS index.js
    # style-index.css
  > node_modules
  ▼ src
    {} block.json
    JS edit.js
    🐘 editor.scss
    JS index.js
    JS save.js
    🐘 style.scss
    ⚙️ .editorconfig
    🗑️ .gitignore
    🐘 my-awesome-block.php
    {} package-lock.json
    {} package.json
    ⓘ readme.txt
```

my-awesome-block.php

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

```
/**
 * 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' );
```


block.json

Contient toutes les métadonnées nécessaires pour déclarer le bloc :

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

```
{
  "$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"
}
```

Pour commencer

Pour commencer à surveiller les fichiers source :

```
npm start
```

Pour builder pour la production :

```
npm run build
```

index.js

- index.js est le fichier d'entrée, qui va importer les autres.
- Edit() est la fonction responsable de l'affichage dans l'éditeur
- save() est 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

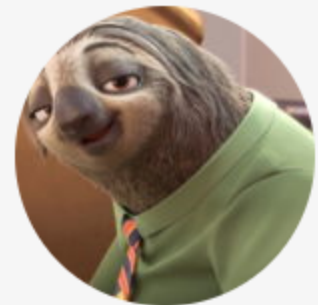
editor.scss

- `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



Hey, Priscilla !

Quel mot qualifie une chamelle à trois bosses ?

Flash Slothmore

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

Les attributs

Les attributs correspondent aux données du bloc : contenus et réglages.

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 éditeur de texte avec des boutons de formatage, comme le bloc paragraphe.

```
import { RichText } from '@wordpress/block-editor';  
  
...  
<RichText  
  value={ attributes.content } // Attribut correspondant au contenu  
  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()

save() doit renvoyer un composant qui correspond à l'HTML à sauvegarder en base de données. On utilise les attributs pour construire l'HTML du bloc.

Attention à utiliser `{ ...useBlockProps.save() }` au lieu de `{ ...useBlockProps() }`

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

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

Le composant `<MediaPlaceholder />`

Affiche une interface pour téléverser des medias, comme le bloc Image.

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

<MediaPlaceholder
  onSelect={onSelectImage} // Fonction appelée quand une image est sélectionnée.
  allowedTypes={ ['image'] } // Type de media autorisés.
  labels = { { title: __( 'Testimonial image', 'my-awesome-block' ) } } // Titre
et instructions
/>
```

```
const onSelectImage = media => {
  const image = {
    id: media.id,
    alt: media.alt || '',
    src: media?.sizes?.thumbnail?.url || media.url,
  }
  setAttributes( { image } );
}
```

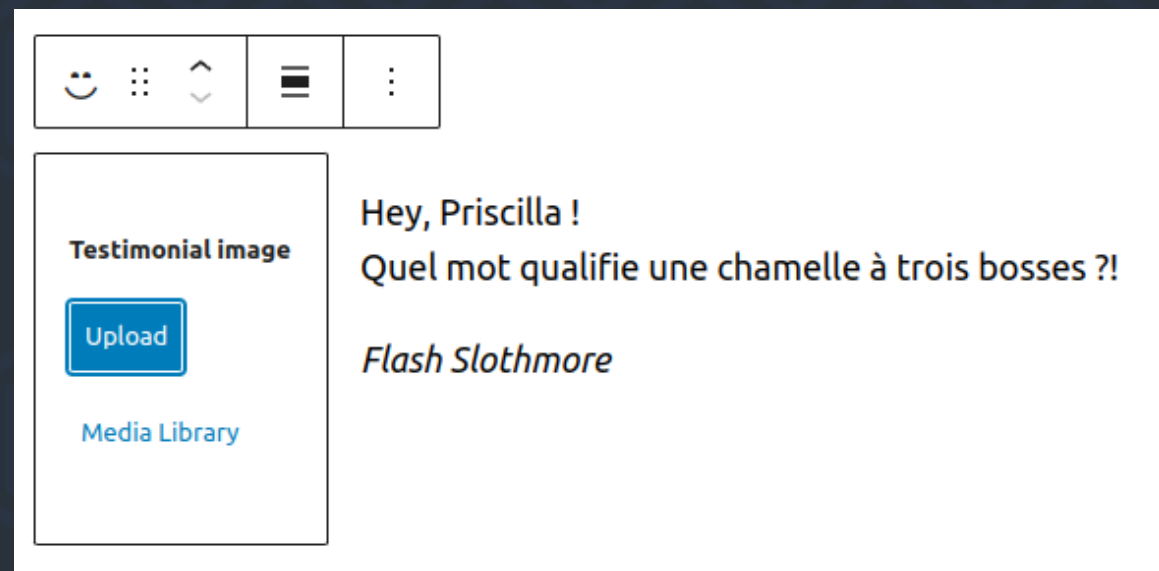
Edit()

```
export default function Edit( { attributes, setAttributes } ) {
  const imageAttributes = {src: attributes.image.src, alt: attributes.image.alt};
  const onDeleteImage = () => { setAttributes( { image: {} } ) };
  const onSelectImage = () => {...};

  return (
    <blockquote { ...useBlockProps() }>
      <div className="image">
        { attributes.image.id ?
          <div className="wrapper image-wrapper">
            <img {...imageAttributes} />
            <Button onClick={ onDeleteImage }>{ __( 'Remove image' ) }</Button>
          </div>
          :
          <MediaPlaceholder ... />
        }
      </div>
      <div className="content">...</div>
    </blockquote>
  );
}
```

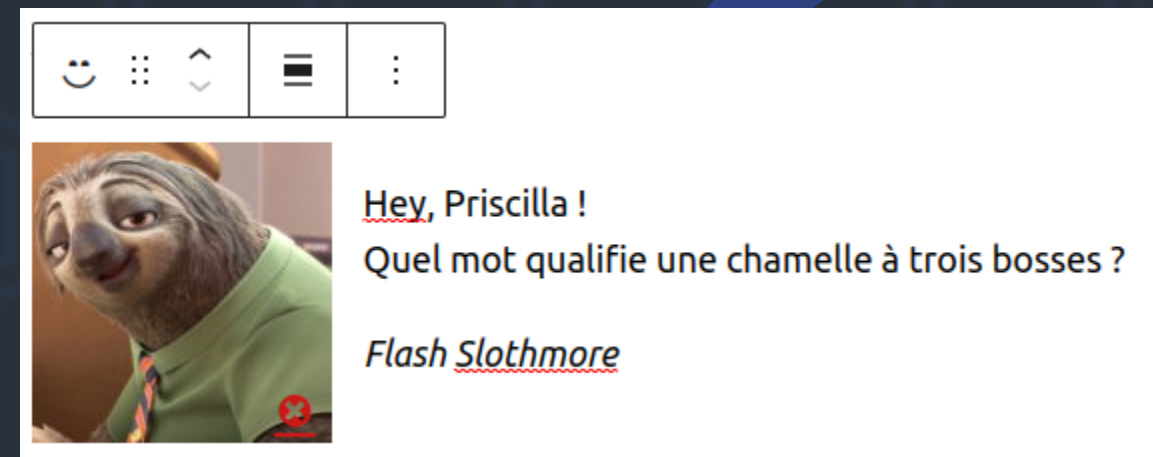
Dans l'éditeur

Sans image :



A screenshot of a text editor interface. At the top, there is a toolbar with icons for smiley face, grid, up arrow, list, and vertical ellipsis. Below the toolbar, on the left, is a sidebar with the heading "Testimonial image", an "Upload" button, and a "Media Library" link. The main text area contains the following text:
Hey, Priscilla !
Quel mot qualifie une chamelle à trois bosses ?!
Flash Slothmore

Avec image :

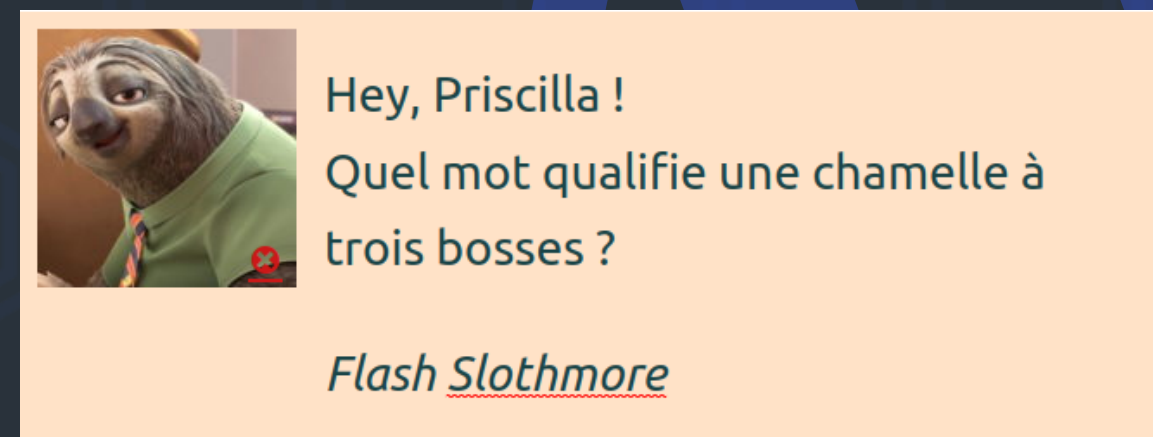
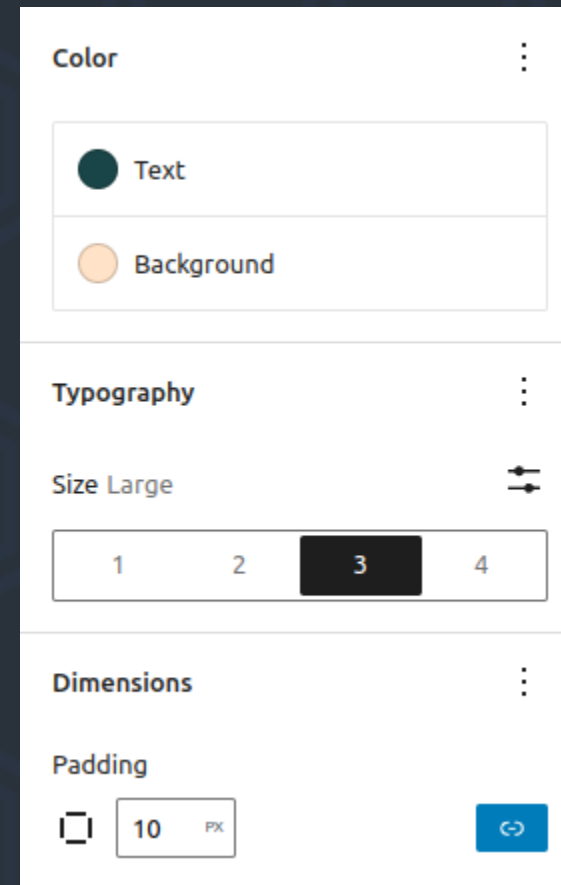


A screenshot of the same text editor interface, but with an image of a sloth character (Flash Slothmore) inserted on the left. The text area now contains the following text:
Hey, Priscilla !
Quel mot qualifie une chamelle à trois bosses ?
Flash Slothmore

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
    }
  },
  ...
}
```



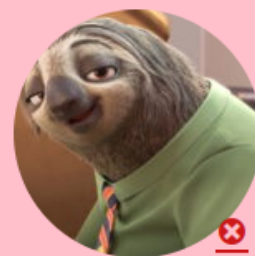
Styles

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%;
    }
  }
}
```



Hey, Priscilla !

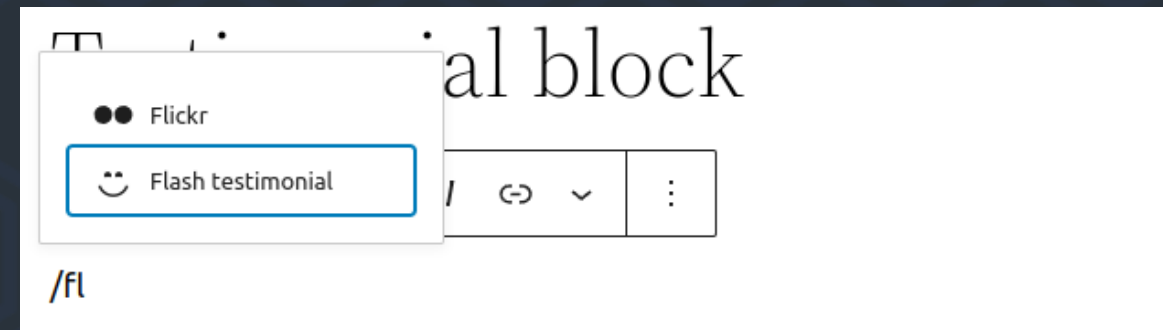
Quel mot qualifie une chamelle à trois bosses ?

Flash Slothmore

Variations

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

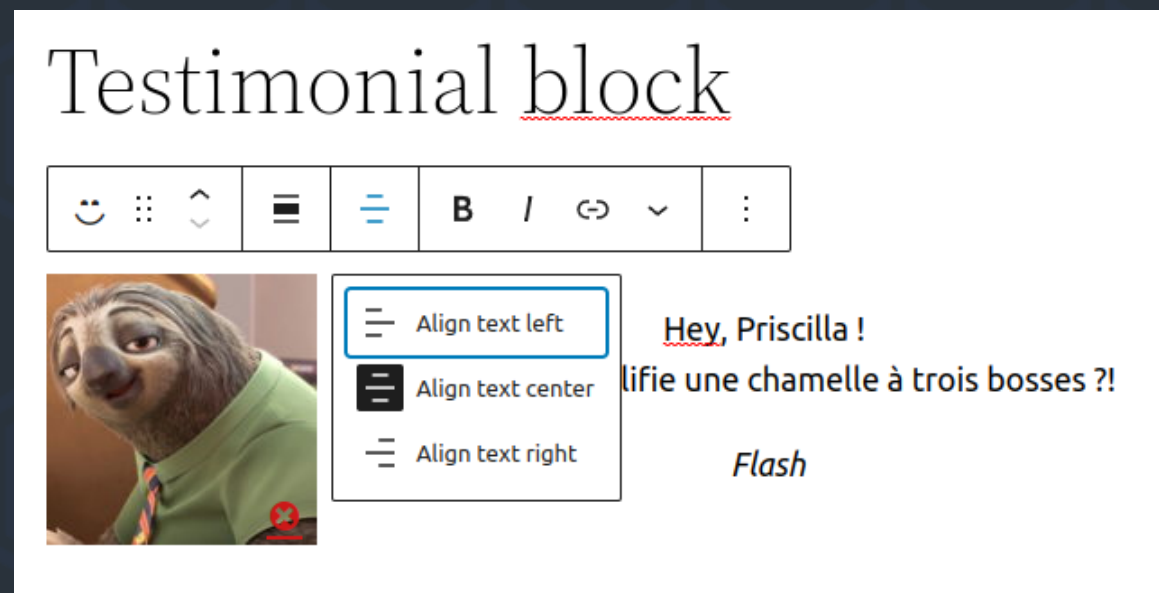
```
{
  "name": "wp-block/my-awesome-block"
  ...
  "variations": [
    {
      "name": "flash",
      "title": "Flash
testimonial",
      "attributes": {
        "source": "Flash",
        "content": "<p>Hey,
Priscilla ! <br>Quel mot qualifie une
chamelle à trois bosses ?!</p>"
      }
    },
    ...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



Toolbar

Pour créer des boutons dans la barre d'outils, on utilise le composant `<BlockControls>`. **Attention!** La barre d'outils 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>
    </>
  );
}
```

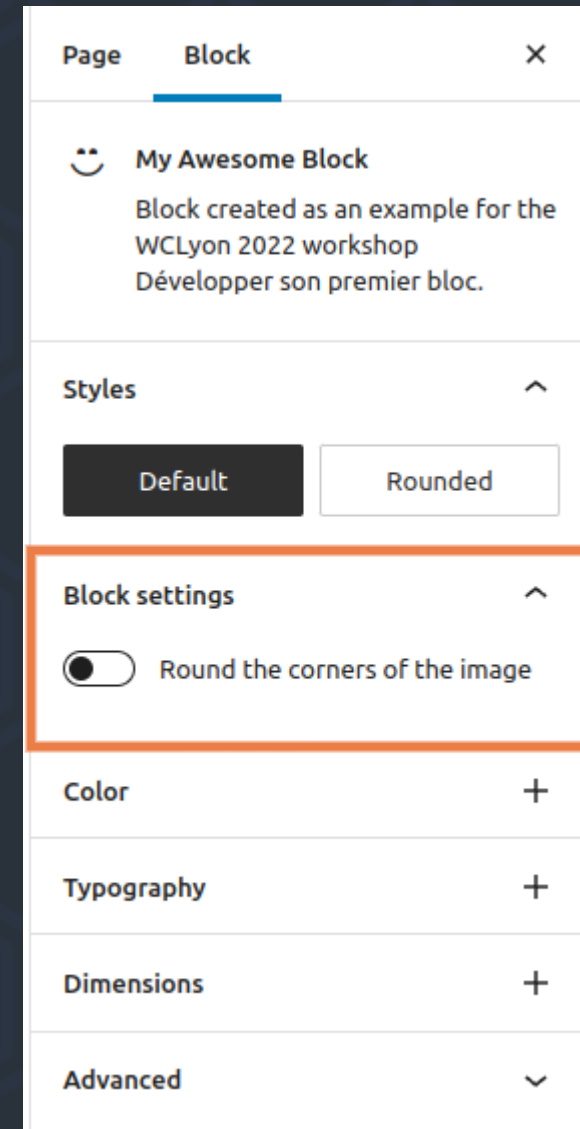
Inspector Controls

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 fonctionnent les attributs
- Utiliser les composants `<RichText>` et `<MediaUpload>`
- Ajouter des Block Supports, des styles de blocs et des variations
- Ajouter des réglages dans la Toolbar et dans l'Inspecteur



Merci ! Des questions ?



<https://github.com/vincedubroeuq/WCLyon2022>



vincent@vincentdubroeuq.com



<https://vincentdubroeuq.com>



[@vincedubroeuq](https://twitter.com/vincedubroeuq)





Merci à toutes et à tous

Des questions ?

