扩展查询循环区块

目录
扩展区块变体
1. Offer sensible defaults
2. 自定义您的变体布局
3. Making Gutenberg recognize your variation
扩展查询功能
Disabling irrelevant or unsupported query controls Understanding the taxQuery structure
Adding additional controls
Making your custom query work on the front-end side Making your custom query work on the editor side

查询循环区块是一个强大的工具，允许用户循环遍历指定的文章列表，并显示一组将继承列表中每篇文章上下
文的区块。例如，可以将其设置为循环遍历特定类别的所有文章，并为每篇文章显示其特色图片。当然，功能
远不止于此！

但正是因为查询循环区块功能如此强大且允许高度自定义，它也可能令人望而生畏。大多数用户不希望面对查
询循环区块的全部功能，因为他们可能不熟悉 “查询 ”
概念及其相关技术术语。相反，大多数用户可能更喜欢预设版本的区块，具有更少的设置项和更清晰的命名。默认提供的文章列表变体就是这种做法的良好示例：用户将在不接触技术细节的情况下使用查询循环区块，并且更有可能发现和理解该区块的用途。

同样，许多扩展开发者可能需要一种方式来呈现定制版本的区块，这些版本具有自己的预设、附加设置，并且
不包含与其用例无关的自定义选项（例如，通常针对其自定义文章类型）。查询循环区块提供了非常强大的方
式来创建此类变体。

扩展区块变体
通过注册具有特定查询循环区块设置的自定义区块变体，您可以更精细地控制其呈现方式，同时仍能充分利用
查询循环区块的全部功能。如果您不熟悉区块变体，请 在此处 了解更多信息。

利用区块变体 API，您可以为特定用例提供最合理的默认设置。

为了使查询循环变体正常工作，我们需要：

- 为 `core/query` 区块注册具有某些默认值的区块变体
- 为区块变体定义布局
- 在 `isActive` 区块变体属性中使用 `namespace` 属性

让我们以注册 `book` 自定义文章类型_____的插件为例，逐步设置变体。

1. Offer sensible defaults

Your first step would be to create a variation which will be set up in such a way to provide a block variation which will display by default a list of books instead of blog posts. The full variation code will look something like this:

```
const MY_VARIATION_NAME = 'my-plugin/books-list';

registerBlockVariation( 'core/query', {
	name: MY_VARIATION_NAME,
	title: 'Books List',
	description: 'Displays a list of books',
	isActive: ( { namespace, query } ) => {
		return (
			namespace === MY_VARIATION_NAME
			&& query.postType === 'book'
		);
	},
	icon: /** An SVG icon can go here*/,
	attributes: {
		namespace: MY_VARIATION_NAME,
		query: {
			perPage: 6,
			pages: 0,
			offset: 0,
			postType: 'book',
			order: 'desc',
			orderBy: 'date',
			author: '',
			search: '',
			exclude: [],
			sticky: '',
```

```
□□□inherit: false,
□□},
□},
□scope: [ 'inserter' ],
□}
);
```

If that sounds like a lot, don't fret, let's go through each of the properties here and see why they are there and what they are doing.

Essentially, you would start with something like this:

```
registerBlockVariation( 'core/query', {
□name: 'my-plugin/books-list',
□attributes: {
□□query: {
□□□/** ...more query settings if needed */
□□□postType: 'book',
□□},
□},
} );
```

In this way, the users won't have to choose the custom `postType` from the dropdown, and be already presented with the correct configuration. However, you might ask, how is a user going to find and insert this variation? Good question! To enable this, you should add:

```
{
□/** ...variation properties */
□scope: [ 'inserter' ],
}
```

In this way, your block will show up just like any other block while the user is in the editor and searching for it. At this point you might also want to add a custom icon, title and description to your variation, just like so:

```
{
□/** ...variation properties */
□title: 'Books List',
□description: 'Displays a list of books',
□icon: /* Your svg icon here */,
}
```

At this point, your custom variation will be virtually indistinguishable from a stand-alone block. Completely branded to your plugin, easy to discover and directly available to the user as a drop in.

However, your query loop variation won't work just yet — we still need to define a layout so that it can render properly.

2. 自定义您的变体布局

请注意，查询循环区块支持在 `scope` 属性中使用字符串 `'block'`。理论上，这是为了让变体在插入区块本身后能被识别。更多关于区块变体选择器的信息，请参阅此处。

然而，不建议当前使用此方法，这是因为查询循环与模式和 `scope: [ 'block' ]` 变体的设置方式：除了 `postType` 和 `inherit` 查询属性外，所选模式的所有属性都将被使用，这很可能导致冲突和变体功能失效。

要规避此问题，有两种途径。第一种是添加您的默认 `innerBlocks`，如下所示：

```
innerBlocks: [
□[
□□'core/post-template',
□□{},
□□[ [ 'core/post-title' ], [ 'core/post-excerpt' ] ],
□],
□[ 'core/query-pagination' ],
□[ 'core/query-no-results' ],
],
```

通过在您的变体中包含 `innerBlocks`，您实质上跳过了查询循环区块的建议模式设置阶段，区块将插入这些内部区块作为其起始内容。

另一种方法是为您的变体注册特定模式，这些模式将在设置阶段被建议，并替换区块的流程。

查询循环区块会判断自身是否存在活动变体，以及是否有为此变体可用的特定模式。如果有，这些模式将是唯一建议给用户的模式，不包括原始查询循环区块的默认模式。否则，如果没有此类模式，则将建议默认模式。

要使模式与查询循环变体“关联”，您应将您的变体名称（前缀为查询循环名称，例如 `core/query/$variation_name`）添加到模式的 `blockTypes` 属性中。有关注册模式的更多详细信息，请参阅此处。

如果您未在变体中提供 `innerBlocks`，还有一种方法可以在用户于设置阶段选择 `Start blank` 时建议“关联”变体。这与“关联”模式的处理方式类似，通过检查查询循环是否存在活动变体以及是否有任何关联变体可建议。

□□□□□□□□□□□□□□□□□□□□□□□□□□ `scope` □□□□□□ `['block']`□□□ `namespace` □□ `name` □□□□

□□□□□□□□□□□□ `products` □□□□□□□□□□□□□□□ `scope: ['inserter']` □□□□□□□□□□ `namespace` □□□□□ `['products']` □□□□□□□□ `block` □□□□□□□□□□ `Start blank` □□□□□□□□□□□□□□□□□□□□□□□□□□□□

3. Making Gutenberg recognize your variation

There is one slight problem you might have realized after implementing this variation: while it is transparent to the user as they are inserting it, Gutenberg will still recognize the variation as a Query Loop block at its core and so, after its insertion, it will show up as a Query Loop block in the tree view of the editor, for instance.

We need a way to tell the editor that this block is indeed your specific variation. This is what the `isActive` property is made for: it's a way to determine whether a certain variation is active based on the block's attributes. You could use it like this:

```
{
□/** ...variation properties */
□isActive: ( { namespace, query } ) => {
□□return (
□□□namespace === MY_VARIATION_NAME
□□□&& query.postType === 'book'
□□);
□},
}
```

You might be tempted to only compare the `postType` so that Gutenberg will recognize the block as your variation any time the `postType` matches `book`. This casts a net too wide, however, as other plugins might want to publish variations based on the `book` post type too, or we might just not want the variation to be recognized every time the user sets the type to `book` manually through the editor settings.

That's why the Query Loop block exposes a special attribute called `namespace`. It really doesn't do anything inside the block implementation, and it's used as an easy and consistent way for extenders to recognize and scope their own variation. In addition, `isActive` also accepts just an array of strings with the attributes to compare. Often, `namespace` would be sufficient, so you would use it like so:

```
{
	/** ...variation properties */
	attributes: {
		/** ...variation attributes */
		namespace: 'my-plugin/books-list',
	},
	isActive: [ 'namespace' ],
}
```

Like so, Gutenberg will know that it is your specific variation only in the case it matches your custom namespace! So convenient!







Disabling irrelevant or unsupported query controls

Let's say you don't use at all the `sticky` attribute in your books, so that would be totally irrelevant to the customization of your block. In order to not confuse the users as to what a setting might do, and only exposing a clear UX to them, we want this control to be unavailable. Furthermore, let's say that you don't use the `author` field at all, which generally indicates the person who has added that post to the database, instead you use a custom `bookAuthor` field. As such, not only keeping the `author` filter would be confusing, it would outright “break” your query.

For this reason, the Query Loop block variations support a property called `allowedControls`, which accepts an array of keys of the controls we want to display on the inspector sidebar. By default, we accept all the controls, but as soon as we provide an array to this property, we want to specify only the controls which are going to be relevant for us!

As of Gutenberg version 14.2, the following controls are available:

- `inherit` - Shows the toggle switch for allowing the query to be inherited directly from the template.
- `postType` - Shows a dropdown of available post types.
- `order` - Shows a dropdown to select the order of the query.
- `sticky` - Shows a dropdown to select how to handle sticky posts.

- `taxQuery` - Shows available taxonomies filters for the currently selected post type, including both inclusion and exclusion controls for each taxonomy.
- `author` - Shows an input field to filter the query by author.
- `search` - Shows an input field to filter the query by keywords.
- `format` - Shows an input field to filter the query by array/collection of formats.
- `parents` - Shows an input field to filter the query using parent(s) entity.

In our case, the property would look like this:

```
{
	/** ...variation properties */
	allowedControls: [ 'inherit', 'order', 'taxQuery', 'search' ],
}
```

If you want to hide all the above available controls, you can set an empty array as a value of `allowedControls`.

Notice that we have also disabled the `postType` control. When the user selects our variation, why show them a confusing dropdown to change the post type? On top of that it might break the block as we can implement custom controls, as we'll see shortly.

Understanding the `taxQuery` structure

The `taxQuery` attribute supports both inclusion and exclusion of taxonomy terms. The structure looks like this:

```
{
	query: {
		taxQuery: {
			include: {
				category: [1, 2, 3], // Include posts with these category IDs.
				post_tag: [10, 20] // Include posts with these tag IDs.
			},
			exclude: {
				category: [5, 6], // Exclude posts with these category IDs.
				post_tag: [15] // Exclude posts with these tag IDs.
			}
		}
	}
}
```

When you use the `taxQuery` control in your variation, users will see both "[Taxonomy]" (inclusion) and "Exclude: [Taxonomy]" controls for each applicable taxonomy. The inclusion and exclusion are mutually exclusive in the UI - terms selected in one won't appear as suggestions in the other.

Adding additional controls

Because our plugin uses custom attributes that we need to query, we want to add our own controls to allow the users to select those instead of the ones we have just disabled from the core inspector controls. We can do this via a React HOC hooked into a block filter, like so:

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

export const withBookQueryControls = ( BlockEdit ) => ( props ) => {
	// We only want to add these controls if it is our variation,
	// so here we can implement a custom logic to check for that, similar
	// to the `isActive` function described above.
	// The following assumes that you wrote a custom `isMyBooksVariation`
	// function to handle that.
	return isMyBooksVariation( props ) ? (
		<>
			<BlockEdit key="edit" { ...props } />
			<InspectorControls>
				<BookAuthorSelector /> { /** Our custom component */ }
			</InspectorControls>
		</>
	) : (
		<BlockEdit key="edit" { ...props } />
	);
};

addFilter( 'editor.BlockEdit', 'core/query', withBookQueryControls );
```

Of course, you'll be responsible for implementing the logic of your control (you might want to take a look at `@wordpress/components` to make your controls fit seamlessly within the Gutenberg UI). Any extra parameter you assign within the `query` object inside the blocks attributes can be used to create a custom query according to your needs, with a little extra effort.

Currently, you'll likely have to implement slightly different paths to make the query behave correctly both on the front-end side (i.e. on the end user's side) and to show the correct preview on the editor side.

```
{
	/** ...variation properties */
	attributes: {
		/** ...variation attributes */
		query: {
			/** ...more query settings if needed */
			postType: 'book',
			/** Our custom query parameter */
			bookAuthor: 'J. R. R. Tolkien'
		}
	}
}
```

Making your custom query work on the front-end side

The Query Loop block functions mainly through the Post Template block which receives the attributes and builds the query from there. Other first-class children of the Query Loop block (such as the Pagination block) behave in the same way. They build their query and then expose the result via the filter `query_loop_block_query_vars`.

You can hook into that filter and modify your query accordingly. Just make sure you don't cause side-effects to other Query Loop blocks by at least checking that you apply the filter only to your variation!

```
if( 'my-plugin/books-list' === $block[ 'attrs' ][ 'namespace' ] ) {
	add_filter(
		'query_loop_block_query_vars',
		function( $query ) {
			/** You can read your block custom query parameters here and build your query */
		},
	);
}
```

(In the code above, we assume you have some way to access the block, for example within a `pre_render_block` filter, but the specific solution can be different depending on the use-case, so this is not a firm recommendation).

Making your custom query work on the editor side

To finish up our custom variation, we might want the editor to react to changes in our custom query and display an appropriate preview accordingly. This is not required for a functioning block, but it enables a fully integrated user experience for the consumers of your block.

The Query Loop block fetches its posts to show the preview using the WordPress REST API. Any extra parameter added to the `query` object will be passed as a query argument to the API. This means that these extra parameters should be either supported by the REST API, or be handled by custom filters such as the `rest_{$this->post_type}_query` filter which allows you to hook into any API request for your custom post type. Like so:

```
add_filter(
	'rest_book_query',
	function( $args, $request ) {
		/** We can access our custom parameters from here */
		$book_author = $request->get_param( 'bookAuthor' );
		/** ...your custom query logic */
	}
);
```

And, just like that, you'll have created a fully functional variation of the Query Loop block!

□□□ #2
□ □□□ □□□ 2026-03-26 17:08:43 UTC
□ □□□ □□□ 2026-03-26 17:08:43 UTC