Server-side rendering: Processing directives on the server

- Processing the directives on the server
- Manipulating the global state and local context in the client
- Initializing the derived state in the server
  - Derived state that can be defined statically
  - Derived state that needs to be defined dynamically
- Serializing other processed values to be consumed on the client
- Processing directives in classic themes
- Conclusion

WordPress has always been built on the foundation of server-side rendering. Traditionally, when a user requests a WordPress page, the server processes the PHP code, queries the database, and generates the HTML markup that is sent to the browser.

In recent years, modern JavaScript frameworks like Vue, React, or Svelte have revolutionized the way we build web applications. These frameworks provide reactive and declarative programming models that enable developers to create dynamic, interactive user interfaces with ease.

However, when it comes to server-side rendering, these frameworks require a JavaScript-based server, such as NodeJS, to execute their code and generate the initial HTML. This means that PHP-based servers, like WordPress, cannot directly utilize these frameworks without sacrificing their native PHP rendering capabilities. This limitation poses a challenge for WordPress developers who want to leverage the power of reactive and declarative programming while still benefiting from WordPress's traditional server-side rendering strengths. The Interactivity API bridges this gap by bringing reactive and declarative programming principles to WordPress without compromising its server-side rendering foundation.

In this guide, we'll explore how the Interactivity API processes directives on the server, enabling WordPress to deliver interactive, state-aware HTML from the initial page load, while setting the stage for seamless client-side interactivity.

Processing the directives on the server

The Interactivity API's Server Directive Processing capabilities enable WordPress to generate the initial HTML with the correct interactive state, providing a faster initial render. After the initial server-side render, the Interactivity API's client-side JavaScript takes over, enabling dynamic updates and interactions without requiring full page reloads. This approach combines the best of both worlds: the SEO and performance benefits of traditional WordPress server-side rendering, and the dynamic, reactive user interfaces offered by modern JavaScript frameworks.

To understand how the Server Directive Processing works, let's start with an example where a list of fruits is rendered using the `data-wp-each` directive.

The following are the necessary steps to ensure that the directives are correctly processed by the Server Directive Processing of the Interactivity API during the server-side rendering of WordPress.

- **1. Mark the block as interactive**
  First, to enable the server processing of the interactive block's directives, you must add `supports.interactivity` to the `block.json`:

```
{
    "supports": {
        "interactivity": true
    }
}
```

- **2. Initialize the global state or local context**
  Then, you must initialize either the global state or the local context that will be used during the server-side rendering of the page.
  If you are using global state, you must use the `wp_interactivity_state` function:

```
wp_interactivity_state( 'myFruitPlugin', array(
  'fruits' => array( 'Apple', 'Banana', 'Cherry' )
));
```

  If you are using local context, the initial values are defined with the `data-wp-context` directive itself, either by:

- Adding it directly to the HTML.

```
<ul data-wp-context='{ "fruits": ["Apple", "Banana", "Cherry"] }'>
    ...
</ul>
```

- Using the `wp_interactivity_data_wp_context` helper.

```
<?php
$context = array( 'fruits' => array( 'Apple', 'Banana', 'Cherry' ) );
?>

<ul <?php echo wp_interactivity_data_wp_context( $context ); ?>>
  ...
</ul>
```

- **3. Define the interactive elements using directives**
  Next, you need to add the necessary directives to the HTML markup.

```
<ul data-wp-interactive="myFruitPlugin">
    <template data-wp-each="state.fruits">
        <li data-wp-text="context.item"></li>
    </template>
</ul>
```

  In this example:
  - The `data-wp-interactive` directive activates the interactivity for the DOM element and its children.
  - The `data-wp-each` directive is used to render a list of elements. The directive can be used in `<template>` tags, with the value being a reference path to an array stored in the global state or the local context.
  - The `data-wp-text` directive sets the inner text of an HTML element. Here, it points to `context.item`, which is where the `data-wp-each` directive stores each item of the array.

  The exact same directives can also be used when using local context instead of global state. The only difference is that `data-wp-each` points to `context.fruits` instead of `state.fruits`:

```
<ul
    data-wp-interactive="myFruitPlugin"
    data-wp-context='{ "fruits": [ "Apple", "Banana", "Cherry" ] }'
>
    <template data-wp-each="context.fruits">
        <li data-wp-text="context.item"></li>
```

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

That’s it! Once you’ve set up your interactive block with `supports.interactivity`, initialized your global state or local context, and added the directives to the HTML markup, the Interactivity API will take care of the rest. There’s no additional code required from the developer to process these directives on the server side.

Behind the scenes, WordPress uses the `wp_interactivity_process_directives` function to find and process the directives in the HTML markup of your block. This function uses the HTML API to make the necessary changes to the markup, based on the found directives and the initial global state and/or local context.

As a result, the HTML markup sent to the browser is already in its final form, with all directives correctly processed. This means that when the page first loads in the browser, it already contains the correct initial state of all interactive elements, without needing any JavaScript to modify it.

This is how the final HTML markup of the fruit list example would look like (directives omitted):

```
<ul>
    <li>Apple</li>
    <li>Banana</li>
    <li>Cherry</li>
</ul>
```

As you can see, the `data-wp-each` directive has generated a `<li>` element for each fruit in the array, and the `data-wp-text` directive has been processed, populating each `<li>` with the correct fruit name.

Manipulating the global state and local context in the client

One of the key strengths of the Interactivity API is how it bridges the gap between server-side rendering and client-side interactivity. To do so, the global state and local context initialized on the server are also serialized and made available to the Interactivity API stores in the client, allowing the application to continue functioning and manipulating the DOM dynamically.

Let’s extend this example to include a button that the user can click to add a new fruit to the list:

```
<button data-wp-on--click="actions.addMango">Add Mango</button>
```

This new button has a `data-wp-on--click` directive that references `actions.addMango`, which is defined in our JavaScript store:

```
const { state } = store( 'myFruitPlugin', {
    actions: {
        addMango() {
            state.fruits.push( 'Mango' );
        },
    },
} );
```

The same example would also work if you were using local context:

```
store( 'myFruitPlugin', {
    actions: {
        addMango() {
            const context = getContext();
            context.fruits.push( 'Mango' );
        },
    },
} );
```

Now, when the user clicks the “Add Mango” button:

1. The `addMango` action is triggered.
2. The `'Mango'` item is added to the `state.fruits` (or `context.fruits`) array.
3. The Interactivity API automatically updates the DOM, adding a new `<li>` element for the new fruit.

```
<ul>
    <li>Apple</li>
    <li>Banana</li>
    <li>Cherry</li>
    <li>Mango</li>
</ul>
```

Remember: initializing the state on the client is not necessary when it has already been done on the server.

```
store( 'myFruitPlugin', {
    state: {
```

```
        fruits: [ 'Apple', 'Banana', 'Cherry' ], // This is not necessary!
    },
} );
```

Initializing the derived state in the server

The derived state, regardless of whether it derives from the global state, local context, or both, can also be processed on the server by the Server Directive Processing.

Please, visit the Understanding global state, local context, derived state and config guide to learn more about how derived state works in the Interactivity API.

Derived state that can be defined statically

Let’s imagine adding a button that can delete all fruits:

```
<button data-wp-on--click="actions.deleteFruits">Delete all fruits</button>
```

```
const { state } = store( 'myFruitPlugin', {
    actions: {
        // ...
        deleteFruits() {
            state.fruits = [];
        },
    },
} );
```

Now, let’s display a special message when there is no fruit. To do this, let’s use a `data-wp-bind--hidden` directive that references a derived state called `state.hasFruits` to show/hide the message.

```
<div data-wp-interactive="myFruitPlugin">
    <ul data-wp-bind--hidden="!state.hasFruits">
        <template data-wp-each="state.fruits">
            <li data-wp-text="context.item"></li>
```

```
        </template>
    </ul>
    <div data-wp-bind--hidden="state.hasFruits">No fruits, sorry!</div>
</div>
```

The derived state `state.hasFruits` is defined on the client using a getter:

```
const { state } = store( 'myFruitPlugin', {
    state: {
        get hasFruits() {
            return state.fruits.length > 0;
        },
    },
    // ...
} );
```

Up to this point, everything is fine in the client, and when we press the “Delete all fruits” button, the “No fruits, sorry!” message will be displayed. The problem is that since `state.hasFruits` is not defined on the server, the `hidden` attribute will not be part of the initial HTML, which means it will also be showing the message until JavaScript loads, causing not only confusion to the visitor, but also a layout shift when JavaScript finally loads and the message is hidden.

To fix this, you must define the initial value of the derived state in the server using `wp_interactivity_state`.

- When the initial value is known and static, it can be defined directly:

  ```
  wp_interactivity_state( 'myFruitPlugin', array(
    'fruits'    => array( 'Apple', 'Banana', 'Cherry' ),
    'hasFruits' => true
  ));
  ```

- Or it can be defined by doing the necessary computations:

  ```
  $fruits    = array( 'Apple', 'Banana', 'Cherry' );
  $hasFruits = count( $fruits ) > 0;

  wp_interactivity_state( 'myFruitPlugin', array(
    'fruits'    => $fruits,
    'hasFruits' => $hasFruits,
  ));
  ```

Regardless of the approach, the key point is that the initial value of `state.hasFruits` is now defined on the server. This allows the Server Directive Processing to handle the `data-wp-bind--hidden` directive and modify the HTML markup, adding the `hidden` attribute when needed.

Derived state that needs to be defined dynamically

In most cases, the initial derived state can be defined statically, as in the previous example. But sometimes, the value depends on dynamic values that also change in the server, and the derived logic needs to be replicated in PHP.

To see an example of this, let's continue by adding a shopping cart emoji (🛒) for each fruit, depending on whether it is on a shopping list or not.

First, let's add an array that represents the shopping list. *Remember that even though these arrays are static for simplicity sake, usually you will work with dynamic information, for example, information coming from the database.*

```
wp_interactivity_state( 'myFruitPlugin', array(
  'fruits'        => array( 'Apple', 'Banana', 'Cherry' ),
  'shoppingList'  => array( 'Apple', 'Cherry' ),
));
```

Now, let's add a derived state on the client that checks if each fruit is on the shopping list or not and returns the emoji.

```
store( 'myFruitPlugin', {
    state: {
        get onShoppingList() {
            const context = getContext();
            return state.shoppingList.includes( context.item ) ? '🛒' : '';
        },
    },
    // ...
} );
```

And let's use that derived state to show the appropriate emoji for each fruit.

```
<ul data-wp-interactive="myFruitPlugin">
    <template data-wp-each="state.fruits">
        <li>
```

```
            <span data-wp-text="context.item"></span>
            <span data-wp-text="state.onShoppingList"></span>
        </li>
    </template>
</ul>
```

Again, up to this point, everything is fine on the client side and the visitor will see the correct emoji displayed for the fruits they have on their shopping list. However, since `state.onShoppingList` is not defined on the server, the emoji will not be part of the initial HTML, and it will not be shown until JavaScript loads.

Let's fix that by adding the initial derived state using `wp_interactivity_state`. Remember that this time, the value depends on `context.item` that comes from the `data-wp-each` directive, which makes the derived value dynamic, so let's replicate the JavaScript logic in PHP:

```
wp_interactivity_state( 'myFruitPlugin', array(
  // ...
  'onShoppingList' => function() {
    $state   = wp_interactivity_state();
    $context = wp_interactivity_get_context();
    return in_array( $context['item'], $state['shoppingList'] ) ? '□' : '';
  }
));
```

That's it! Now, our server can compute the derived state and know which fruits are on the shopping list and which are not. This allows the Server Directive Processing to populate the initial HTML with the correct values, ensuring that the user sees the correct information immediately, even before the JavaScript runtime loads.

Serializing other processed values to be consumed on the client

The `wp_interactivity_state` function is also valuable for sending processed values from the server to the client so they can be consumed later on. This feature is useful in many situations, such as managing translations.

Let's add translations to our example to see how this would work.

```php
<?php
wp_interactivity_state( 'myFruitPlugin', array(
  'fruits'         => array( __( 'Apple' ), __( 'Banana' ), __( 'Cherry' ) ),
  'shoppingList'   => array( __( 'Apple' ), __( 'Cherry' ) ),
  // ...
) );
?>

<div data-wp-interactive="myFruitPlugin">
  <button data-wp-on--click="actions.deleteFruits">
    <?php echo __( 'Delete all fruits' ); ?>
  </button>
  <button data-wp-on--click="actions.addMango">
    <?php echo __( 'Add Mango' ); ?>
  </button>
  <ul data-wp-bind--hidden="!state.hasFruits">
    <template data-wp-each="state.fruits">
      <li>
        <span data-wp-text="context.item"></span>
        <span data-wp-text="state.onShoppingList"></span>
      </li>
    </template>
  </ul>
  <div data-wp-bind--hidden="state.hasFruits">
    <?php echo __( 'No fruits, sorry!' ); ?>
  </div>
</div>
```

That’s it! Since the Interactivity API works in PHP, you can add translations directly to the global state, the local context and the HTML markup.

But wait, what happens with our `addMango` action? Remember, this action is defined only on JavaScript:

```
const { state } = store( 'myFruitPlugin', {
    actions: {
        addMango() {
            state.fruits.push( 'Mango' ); // Not translated!
        },
    },
```

```
} );
```

To fix this issue, you can use the `wp_interactivity_state` function to serialize the translated mango string and then access that value in your action.

```
wp_interactivity_state( 'myFruitPlugin', array(
  'fruits' => array( __( 'Apple' ), __( 'Banana' ), __( 'Cherry' ) ),
  'mango'  => __( 'Mango' ),
));
```

```
const { state } = store( 'myFruitPlugin', {
    actions: {
        addMango() {
            // `state.mango` contains the 'Mango' string already translated.
            state.fruits.push( state.mango );
        },
    },
} );
```

Take into account that if your application is more dynamic, you could serialize an array with all the fruit translations and just work with *fruit keywords* in your actions. For example:

```
wp_interactivity_state( 'myFruitPlugin', array(
  'fruits'           => array( 'apple', 'banana', 'cherry' ),
  'translatedFruits' => array(
    'apple'  => __( 'Apple' ),
    'banana' => __( 'Banana' ),
    'cherry' => __( 'Cherry' ),
    'mango'  => __( 'Mango' ),
  ),
  'translatedFruit'  => function() {
    $state   = wp_interactivity_state();
    $context = wp_interactivity_get_context();
    return $state['translatedFruits'][ $context['item'] ];
  }
));
```

```
const { state } = store( 'myFruitPlugin', {
  state: {
    get translatedFruit() {
```

```
      const context = getContext();
      return state.translatedFruits[ context.item ];
    }
  },
  actions: {
    addMango() {
      state.fruits.push( 'mango' );
    },
  },
} );
```

```
<template data-wp-each="state.fruits">
    <li data-wp-text="state.translatedFruit"></li>
</template>
```

Passing non-reactive, static data from the server—such as Ajax/REST-API URLs and nonces—is also a common need. Because these values never change at runtime, they belong in the **config** rather than the global state. Use `wp_interactivity_config()` on the server and `getConfig()` on the client.

```
wp_interactivity_config( 'myPlugin', array(
  'ajaxUrl' => admin_url( 'admin-ajax.php' ),
  'nonce'   => wp_create_nonce( 'myPlugin_nonce' ),
));
```

```
import { store, getConfig } from '@wordpress/interactivity';

store( 'myPlugin', {
    actions: {
        *doSomething() {
            const { ajaxUrl, nonce } = getConfig();

            const formData = new FormData();
            formData.append( 'action', 'do_something' );
            formData.append( '_ajax_nonce', nonce );

            const data = yield fetch( ajaxUrl, {
                method: 'POST',
                body: formData,
            } ).then( ( response ) => response.json() );
```

```
            console.log( 'Server data', data );
        },
    },
} );
```

Processing directives in classic themes

Server Directive Processing happens automatically in your interactive blocks as soon as you add `supports.interactivity` to your `block.json` file. But what about classic themes?

Classic themes can also use the Interactivity API, and if they want to take advantage of the Server Directive Processing (which they should), they can do so through the `wp_interactivity_process_directives` function. This function receives the HTML markup with unprocessed directives and returns the HTML markup modified according to the initial values of the global state, local context, and derived state.

```
// Initializes the global and derived state…
wp_interactivity_state( '...', /* ... */ );

// The interactive HTML markup that contains the directives.
$html = '<div data-wp-...>...</div>';

// Processes the directives so they are ready to be sent to the client.
$processed_html = wp_interactivity_process_directives( $html );
```

That’s it! There’s nothing else you need to do.

If you want to use `wp_interactivity_process_directives` in a template file, you can use `ob_start` and `ob_get_clean` to capture the HTML output and process it before rendering.

```
<?php
wp_interactivity_state( 'myClassicTheme', /* ... */ );
ob_start();
?>

<div data-wp-interactive="myClassicTheme">
```

```
  ...
</div>

<?php
$html = ob_get_clean();
echo wp_interactivity_process_directives( $html );
```

Important: You only need to process the directives once. If you're including an inner template file within another template, ensure that `wp_interactivity_process_directives` is only called in the outermost template file to avoid redundant processing.

Conclusion

The Interactivity API ensures a smooth and transparent transition from server-rendered content to client-side interactivity. The directives you define on the server, the initial global state or local context, and the client-side behavior all form part of the same ecosystem. This unified approach simplifies development, improves maintainability, and provides a better developer experience when creating interactive WordPress blocks.

□□□ #2
□ □□□ □□□ 2026-10-31 18:29:01 UTC
□ □□□ □□□ 2026-10-31 18:29:03 UTC