Using TypeScript

- Installing @wordpress/interactivity locally
- Scaffolding a new typed interactive block
- Typing the store
  - 1. Infer the types from your client store definition
  - 2. Manually type the server state, but infer the rest from your client store definition
  - 3. Manually write all the types
- Typing the local context
- Typing the derived state
- Typing asynchronous actions
  - Typing yielded values in asynchronous actions
- Typing stores that are divided into multiple parts
- Importing and exporting typed stores
- Conclusion

The Interactivity API provides robust support for TypeScript, enabling developers to build type-safe stores to enhance the development experience with static type checking, improved code completion, and simplified refactoring. This guide will walk you through the process of using TypeScript with Interactivity API stores, covering everything from basic type definitions to advanced techniques for handling complex store structures.

These are the core principles of TypeScript's interaction with the Interactivity API:

- **Inferred client types**: When you create a store using the `store` function, TypeScript automatically infers the types of the store's properties (`state`, `actions`, etc.). This means that you can often get away with just writing plain JavaScript objects, and TypeScript will figure out the types for you.
- **Explicit server types**: When dealing with data defined on the server, like local context or the initial values of the global state, you can explicitly define its types to ensure that everything is correctly typed.
- **Multiple store parts**: Even if your store is split into multiple parts, you can define or infer the types of each part of the store and then merge them into a single type that represents the entire store.

- **Typed external stores**: You can import typed stores from external namespaces, allowing you to use other plugins' functionality with type safety.

Installing `@wordpress/interactivity` locally

If you haven't done so already, you need to install the package `@wordpress/interactivity` locally so TypeScript can use its types in your IDE. You can do this using the following command:

`npm install @wordpress/interactivity`

It is also a good practice to keep that package updated.

Scaffolding a new typed interactive block

If you want to explore an example of an interactive block using TypeScript in your local environment, you can use the `@wordpress/create-block-interactive-template`.

Start by ensuring you have Node.js and `npm` installed on your computer. Review the Node.js development environment guide if not.

Next, use the `@wordpress/create-block` package and the `@wordpress/create-block-interactive-template` template to scaffold the block.

Choose the folder where you want to create the plugin, execute the following command in the terminal from within that folder, and choose the `typescript` variant when asked.

```
npx @wordpress/create-block@latest --template @wordpress/create-block-interactive-template
```

Important: Do not provide a slug in the terminal. Otherwise, `create-block` will not ask you which variant you want to choose and it will select the default non-TypeScript variant by default.

Finally, you can keep following the instructions in the Getting Started Guide as the rest of the instructions remain the same.

Typing the store

Depending on the structure of your store and your preference, there are three options you can choose from to generate your store's types:

1. Infer the types from your client store definition.
2. Manually type the server state, but infer the rest from your client store definition.
3. Manually write all the types.

1. Infer the types from your client store definition

When you create a store using the `store` function, TypeScript automatically infers the types of the store's properties (`state`, `actions`, `callbacks`, etc.). This means that you can often get away with just writing plain JavaScript objects, and TypeScript will figure out the correct types for you.

Let's start with a basic example of a counter block. We will define the store in the `view.ts` file of the block, which contains the initial global state, an action and a callback.

```
// view.ts
const myStore = store( 'myCounterPlugin', {
    state: {
        counter: 0,
    },
    actions: {
        increment() {
            myStore.state.counter += 1;
        },
    },
    callbacks: {
        log() {
            console.log( `counter: ${ myStore.state.counter }` );
        },
    },
} );
```

If you inspect the types of `myStore` using TypeScript, you will see that TypeScript has been able to infer the types correctly.

```
const myStore: {
    state: {
        counter: number;
    };
    actions: {
        increment(): void;
    };
    callbacks: {
        log(): void;
    };
};
```

You can also destructure the `state`, `actions` and `callbacks` properties, and the types will still work correctly.

```
const { state } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
    },
    actions: {
        increment() {
            state.counter += 1;
        },
    },
    callbacks: {
        log() {
            console.log( `counter: ${ state.counter }` );
        },
    },
} );
```

In conclusion, inferring the types is useful when you have a simple store defined in a single call to the `store` function and you do not need to type any state that has been initialized on the server.

2. Manually type the server state, but infer the rest from your client store definition

The global state that is initialized on the server with the `wp_interactivity_state` function doesn't exist on your client store definition and, therefore, needs to be manually typed. But if you don't want to define all the types of your store, you can infer the types of your client store definition and merge them with the types of your server initialized state.

Please, visit the Server-side Rendering guide to learn more about `wp_interactivity_state` and how directives are processed on the server.

Following our previous example, let's move our `counter` state initialization to the server.

```
wp_interactivity_state( 'myCounterPlugin', array(
    'counter' => 1,
));
```

Now, let's define the server state types and merge it with the types inferred from the client store definition.

```
// Types the server state.
type ServerState = {
    state: {
        counter: number;
    };
};

// Defines the store in a variable to be able to extract its type later.
const storeDef = {
    actions: {
        increment() {
            state.counter += 1;
        },
    },
    callbacks: {
        log() {
            console.log( `counter: ${ state.counter }` );
        },
    },
};

// Merges the types of the server state and the client store definition.
type Store = ServerState & typeof storeDef;
```

```
// Injects the final types when calling the `store` function.
const { state } = store< Store >( 'myCounterPlugin', storeDef );
```

Alternatively, if you don't mind typing the entire state including both the values defined on the server and the values defined on the client, you can cast the `state` property and let TypeScript infer the rest of the store.

Let's imagine you have an additional property in the client global state called `product`.

```
type State = {
    counter: number; // The server state.
    product: number; // The client state.
};

const { state } = store( 'myCounterPlugin', {
    state: {
        product: 2,
    } as State, // Casts the entire state manually.
    actions: {
        multiply() {
            state.counter *= state.product;
        },
    },
} );
```

That's it. Now, TypeScript will infer the types of the `actions` and `callbacks` properties from the store definition, but it will use the type `State` for the `state` property so it contains the correct types from both the client and server definitions.

In conclusion, this approach is useful when you have a server state that needs to be manually typed, but you still want to infer the types of the rest of the store.

3. Manually write all the types

If you prefer to define all the types of the store manually instead of letting TypeScript infer them from your client store definition, you can do that too. You simply need to pass them to the `store` function.

```
// Defines the store types.
interface Store {
    state: {
```

```
        counter: number; // Initial server state
    };
    actions: {
        increment(): void;
    };
    callbacks: {
        log(): void;
    };
}

// Pass the types when calling the `store` function.
const { state } = store< Store >( 'myCounterPlugin', {
    actions: {
        increment() {
            state.counter += 1;
        },
    },
    callbacks: {
        log() {
            console.log( `counter: ${ state.counter }` );
        },
    },
} );
```

That’s it! In conclusion, this approach is useful when you want to control all the types of your store and you don’t mind writing them by hand.

Typing the local context

The initial local context is defined on the server using the `data-wp-context` directive.

```
<div data-wp-context='{ "counter": 0 }'>...</div>
```

For that reason, you need to define its type manually and pass it to the `getContext` function to ensure the returned properties are correctly typed.

```
// Defines the types of your context.
type MyContext = {
    counter: number;
```

```
};

store( 'myCounterPlugin', {
    actions: {
        increment() {
            // Passes it to the getContext function.
            const context = getContext< MyContext >();
            // Now `context` is properly typed.
            context.counter += 1;
        },
    },
} );
```

To avoid having to pass the context types over and over, you can also define a typed function and use that function instead of `getContext`.

```
// Defines the types of your context.
type MyContext = {
    counter: number;
};

// Defines a typed function. You only have to do this once.
const getMyContext = getContext< MyContext >;

store( 'myCounterPlugin', {
    actions: {
        increment() {
            // Use your typed function.
            const context = getMyContext();
            // Now `context` is properly typed.
            context.counter += 1;
        },
    },
} );
```

That’s it! Now you can access the context properties with the correct types.

Typing the derived state

The derived state is data that is calculated based on the global state or local context. In the client store definition, it is defined using a getter in the `state` object.

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.

Following our previous example, let's create a derived state that is the double of our counter.

```
type MyContext = {
    counter: number;
};

const myStore = store( 'myCounterPlugin', {
    state: {
        get double() {
            const { counter } = getContext< MyContext >();
            return counter * 2;
        },
    },
    actions: {
        increment() {
            state.counter += 1; // This type is number.
        },
    },
} );
```

Normally, when the derived state depends on the local context, TypeScript will be able to infer the correct types:

```
const myStore: {
    state: {
        readonly double: number;
    };
    actions: {
        increment(): void;
    };
};
```

But when the return value of the derived state depends directly on some part of the global state, TypeScript will not be able to infer the types because it will claim that it has a circular reference.

For example, in this case, TypeScript cannot infer the type of `state.double` because it depends on `state.counter`, and the type of `state` is not completed until the type of `state.double` is defined, creating a circular reference.

```
const { state } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
        get double() {
            // TypeScript can't infer this return type because it depends on `state`.
            return state.counter * 2;
        },
    },
    actions: {
        increment() {
            state.counter += 1; // This type is now unknown.
        },
    },
} );
```

In this case, depending on your TypeScript configuration, TypeScript will either warn you about a circular reference or simply add the `any` type to the `state` property.

However, solving this problem is easy; we simply need to manually provide TypeScript with the return type of that getter. Once we do that, the circular reference disappears, and TypeScript can once again infer all the `state` types.

```
const { state } = store( 'myCounterPlugin', {
    state: {
        counter: 1,
        get double(): number {
            return state.counter * 2;
        },
    },
    actions: {
        increment() {
            state.counter += 1; // Correctly inferred!
        },
    },
} );
```

These are now the correct inferred types for the previous store.

```
const myStore: {
    state: {
        counter: number;
        readonly double: number;
    };
    actions: {
        increment(): void;
    };
};
```

When using `wp_interactivity_state` in the server, remember that you also need to define the initial value of your derived state, like this:

```
wp_interactivity_state( 'myCounterPlugin', array(
    'counter' => 1,
    'double'  => 2,
));
```

But if you are inferring the types, you don’t need to manually define the type of the derived state because it already exists in your client’s store definition.

```
// You don't need to type `state.double` here.
type ServerState = {
    state: {
        counter: number;
    };
};

// The `state.double` type is inferred from here.
const storeDef = {
    state: {
        get double(): number {
            return state.counter * 2;
        },
    },
    actions: {
        increment() {
            state.counter += 1;
        },
    },
```

```
};

// Merges the types of the server state and the client store definition.
type Store = ServerState & typeof storeDef;

// Injects the final types when calling the `store` function.
const { state } = store< Store >( 'myCounterPlugin', storeDef );
```

That's it! Now you can access the derived state properties with the correct types.

Typing asynchronous actions

Another thing to keep in mind when using TypeScript with the Interactivity API is that asynchronous actions must be defined with generators instead of async functions.

The reason for using generators in the Interactivity API's asynchronous actions is to be able to restore the scope from the initially triggered action once the asynchronous action continues its execution after yielding. But this is a syntax change only, otherwise, **these functions operate just like regular async functions**, and the inferred types from the `store` function reflect this.

Following our previous example, let's add an asynchronous action to the store.

```
const { state } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
        get double(): number {
            return state.counter * 2;
        },
    },
    actions: {
        increment() {
            state.counter += 1;
        },
        *delayedIncrement() {
            yield new Promise( ( r ) => setTimeout( r, 1000 ) );
            state.counter += 1;
        },
    },
} );
```

The inferred types for this store are:

```
const myStore: {
    state: {
        counter: number;
        readonly double: number;
    };
    actions: {
        increment(): void;
        // This behaves like a regular async function.
        delayedIncrement(): Promise< void >;
    };
};
```

This also means that you can use your async actions in external functions, and TypeScript will correctly use the async function types.

```
const someAsyncFunction = async () => {
    // This works fine and it's correctly typed.
    await actions.delayedIncrement();
};
```

When you are not inferring types but manually writing the types for your entire store, you can use async function types for your async actions.

```
type Store = {
    state: {
        counter: number;
        readonly double: number;
    };
    actions: {
        increment(): void;
        delayedIncrement(): Promise< void >; // You can use async functions here.
    };
};
```

There’s something to keep in mind when using asynchronous actions. Just like with the derived state, if an asynchronous action uses `state` within a `yield` expression (for example, by passing `state` to an async function that is then yielded) or if its return value depends on `state`, TypeScript might not be able to infer the types correctly due to a potential circular reference.

```
const { state, actions } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
    },
    actions: {
        *delayedOperation() {
            // Example: state.counter is used as part of the yielded logic.
            yield fetchCounterData( state.counter );

            // And/or the final return value depends on state.
            return state.counter + 1;
        },
    },
} );
```

In such cases, TypeScript might issue a warning about a circular reference or default to `any`. To solve this, you need to manually type the generator function. The Interactivity API provides a helper type, `AsyncAction<ReturnType>`, for this purpose.

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

const { state, actions } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
    },
    actions: {
        *delayedOperation(): AsyncAction< number > {
            // Now, this doesn't cause a circular reference.
            yield fetchCounterData( state.counter );

            // Now, this is correctly typed.
            return state.counter + 1;
        },
    },
} );
```

That’s it! The `AsyncAction<ReturnType>` helper is defined as `Generator<any, ReturnType, unknown>`. By using `any` for the type of values yielded by the generator, it helps break the circular reference, allowing TypeScript to correctly infer the types when `state` is involved in `yield` expressions or in the final return value. You only need to specify the final `ReturnType` of your asynchronous action.

Typing yielded values in asynchronous actions

While `AsyncAction<ReturnType>` types the overall generator and its final return value, the value resolved by an individual `yield` expression within that generator might still be typed as `any`.

If you need to ensure the correct type for a value that a `yield` expression resolves to (e.g., the result of a `fetch` call or another async operation), you can use the `TypeYield<T>` helper. This helper takes the type of the asynchronous function/operation being yielded (`T`) and resolves to the type of the value that the promise fulfills with.

Suppose `fetchCounterData` returns a promise that resolves to an object:

```
import {
    store,
    type AsyncAction,
    type TypeYield,
} from '@wordpress/interactivity';

// Assume this function is defined elsewhere and fetches specific data.
const fetchCounterData = async (
    counterValue: number
): Promise< { current: number; next: number } > => {
    // internal logic...
};

const { state, actions } = store( 'myCounterPlugin', {
    state: {
        counter: 0,
    },
    actions: {
        *loadCounterData(): AsyncAction< void > {
            // Use TypeYield to correctly type the resolved value of the yield.
            const data = ( yield fetchCounterData(
                state.counter
            ) ) as TypeYield< typeof fetchCounterData >;

            // Now, `data` is correctly typed as { current: number, next: number }.
            console.log( data.current, data.next );
```

```
            // Update state based on the fetched data.
            state.counter = data.next;
        },
    },
} );
```

In this example, `( yield fetchCounterData( state.counter ) ) as TypeYield< typeof fetchCounterData >` ensures that the `data` constant is correctly typed as `{ current: number, next: number }`, matching the return type of `fetchCounterData`. This allows you to confidently access properties like `data.current` and `data.next` with type safety.

Typing stores that are divided into multiple parts

Sometimes, stores can be divided into different files. This can happen when different blocks share the same namespace, with each block loading the part of the store it needs.

Let’s look at an example of two blocks:

- `todo-list`: A block that displays a list of todos.
- `add-post-to-todo`: A block that shows a button to add a new todo item to the list with the text “Read {$post_title}”.

First, let’s initialize the global and derived state of the `todo-list` block on the server.

```
<?php
// todo-list-block/render.php
$todos = array( 'Buy milk', 'Walk the dog' );
wp_interactivity_state( 'myTodoPlugin', array(
  'todos'         => $todos,
  'filter'        => 'all',
  'filteredTodos' => $todos,
));
?>

<!-- HTML markup... -->
```

Now, let’s type the server state and add the client store definition. Remember, `filteredTodos` is derived state, so you don’t need to type it manually.

```
// todo-list-block/view.ts
type ServerState = {
    state: {
        todos: string[];
        filter: 'all' | 'completed';
    };
};

const todoList = {
    state: {
        get filteredTodos(): string[] {
            return state.filter === 'completed'
                ? state.todos.filter( ( todo ) => todo.includes( '' ) )
                : state.todos;
        },
    },
    actions: {
        addTodo( todo: string ) {
            state.todos.push( todo );
        },
    },
};

// Merges the inferred types with the server state types.
export type TodoList = ServerState & typeof todoList;

// Injects the final types when calling the `store` function.
const { state } = store< TodoList >( 'myTodoPlugin', todoList );
```

So far, so good. Now let’s create our `add-post-to-todo` block.

First, let’s add the current post title to the server state.

```
<?php
// add-post-to-todo-block/render.php
wp_interactivity_state( 'myTodoPlugin', array(
  'postTitle' => get_the_title(),
));
```

```
?>

<!-- HTML markup... -->
```

Now, let's type that server state and add the client store definition.

```
// add-post-to-todo-block/view.ts
type ServerState = {
    state: {
        postTitle: string;
    };
};

const addPostToTodo = {
    actions: {
        addPostToTodo() {
            const todo = `Read: ${ state.postTitle }`.trim();
            if ( ! state.todos.includes( todo ) ) {
                actions.addTodo( todo );
            }
        },
    },
};

// Merges the inferred types with the server state types.
type Store = ServerState & typeof addPostToTodo;

// Injects the final types when calling the `store` function.
const { state, actions } = store< Store >( 'myTodoPlugin', addPostToTodo );
```

This works fine in the browser, but TypeScript will complain that, in this block, `state` and `actions` do not include `state.todos` and `actions.addtodo`.

To fix this, we need to import the `TodoList` type from the `todo-list` block and merge it with the other types.

```
import type { TodoList } from '../todo-list-block/view';

// ...

// Merges the inferred types inferred the server state types.
```

```
type Store = TodoList & ServerState & typeof addPostToTodo;
```

That's it! Now TypeScript will know that `state.todos` and `actions.addTodo` are available in the `add-post-to-todo` block.

This approach allows the `add-post-to-todo` block to interact with the existing todo list while maintaining type safety and adding its own functionality to the shared store.

If you need to use the `add-post-to-todo` types in the `todo-list` block, you simply have to export its types and import them in the other `view.ts` file.

Finally, if you prefer to define all types manually instead of inferring them, you can define them in a separate file and import that definition into each of your store parts. Here's how you could do that for our todo list example:

```
// types.ts
interface Store {
    state: {
        todos: string[];
        filter: 'all' | 'completed';
        filtered: string[];
        postTitle: string;
    };
    actions: {
        addTodo( todo: string ): void;
        addPostToTodo(): void;
    };
}

export default Store;
```

```
// todo-list-block/view.ts
import type Store from '../types';

const { state } = store< Store >( 'myTodoPlugin', {
    // Everything is correctly typed here
} );
```

```
// add-post-to-todo-block/view.ts
import type Store from '../types';

const { state, actions } = store< Store >( 'myTodoPlugin', {
```

```
    // Everything is correctly typed here
} );
```

This approach allows you to have full control over your types and ensures consistency across all parts of your store. It's particularly useful when you have a complex store structure or when you want to enforce a specific interface across multiple blocks or components.

Importing and exporting typed stores

In the Interactivity API, stores from other namespaces can be accessed using the `store` function.

Let's go back to our `todo-list` block example, but this time, let's imagine that the `add-post-to-todo` block belongs to a different plugin and therefore will use a different namespace.

```
// Import the store of the `todo-list` block.
const myTodoPlugin = store( 'myTodoPlugin' );

store( 'myAddPostToTodoPlugin', {
    actions: {
        addPostToTodo() {
            const todo = `Read: ${ state.postTitle }`.trim();
            if ( ! myTodoPlugin.state.todos.includes( todo ) ) {
                myTodoPlugin.actions.addTodo( todo );
            }
        },
    },
} );
```

This works fine in the browser, but TypeScript will complain that `myTodoPlugin.state` and `myTodoPlugin.actions` are not typed.

To fix that, the `myTodoPlugin` plugin can export the result of calling the `store` function with the correct types, and make that available using a script module.

```
// Export the already typed state and actions.
export const { state, actions } = store< TodoList >( 'myTodoPlugin', {
    // ...
} );
```

Now, the `add-post-to-todo` block can import the typed store from the `myTodoPlugin` script module, and it not only ensures that the store will be loaded, but that it also contains the correct types.

```
import { store } from '@wordpress/interactivity';
import {
    state as todoState,
    actions as todoActions,
} from 'my-todo-plugin-module';

store( 'myAddPostToTodoPlugin', {
    actions: {
        addPostToTodo() {
            const todo = `Read: ${ state.postTitle }`.trim();
            if ( ! todoState.todos.includes( todo ) ) {
                todoActions.addTodo( todo );
            }
        },
    },
} );
```

Remember that you will need to declare the `my-todo-plugin-module` script module as a dependency.

If the other store is optional and you don’t want to load it eagerly, a dynamic import can be used instead of a static import.

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

store( 'myAddPostToTodoPlugin', {
    actions: {
        *addPostToTodo() {
            const todoPlugin = yield import( 'my-todo-plugin-module' );
            const todo = `Read: ${ state.postTitle }`.trim();
            if ( ! todoPlugin.state.todos.includes( todo ) ) {
                todoPlugin.actions.addTodo( todo );
            }
        },
    },
} );
```

Conclusion

In this guide, we explored different approaches to typing the Interactivity API stores, from inferring types automatically to manually defining them. We also covered how to handle server-initialized state, local context, and derived state, as well as how to type asynchronous actions.

Remember that the choice between inferring types and manually defining them depends on your specific needs and the complexity of your store. Whichever approach you choose, TypeScript will help you build better and more reliable interactive blocks.

□□□ #2
□ □□□ □□□ 2026-10-31 18:29:05 UTC
□ □□□ □□□ 2026-10-31 18:29:08 UTC