@wordpress/data

- Installation
- Registering a Store
  - Redux Store Options
    - reducer
    - actions
    - selectors
    - resolvers
      - Object-form resolvers
    - controls (deprecated)
    - initialState
- Generic Stores
- Comparison with Redux
- API
  - AsyncModeProvider
  - combineReducers
  - controls
  - createReduxStore
  - createRegistry
  - createRegistryControl
  - createRegistrySelector
  - createSelector
  - dispatch
  - EnhancedSelector
  - GetDependants
  - keyedReducer
  - plugins
  - register
  - registerGenericStore

    - registerStore
    - RegistryConsumer
    - RegistryProvider
    - resolveSelect
    - select
    - subscribe
    - suspendSelect
    - use
    - useDispatch
    - useRegistry
    - useSelect
    - useSuspenseSelect
    - withDispatch
    - withRegistry
    - withSelect
    - batch
- Selectors
    - hasFinishedResolution
    - hasStartedResolution
    - isResolving
    - Normalizing Selector Arguments
        - Example
    - Sharing a Resolution Between Selector Calls
- Going further
- Contributing to this package

WordPress' data module serves as a hub to manage application state for both plugins and WordPress itself, providing tools to manage data within and between distinct modules. It is designed as a modular pattern for organizing and sharing data: simple enough to satisfy the needs of a small plugin, while scalable to serve the requirements of a complex single-page application.

The data module is built upon and shares many of the same core principles of Redux, but shouldn't be mistaken as merely *Redux for WordPress*, as it includes a few of its own distinguishing characteristics. As you read through this guide, you may find it useful to reference the Redux documentation — particularly its glossary — for more detail on core concepts.

Installation

Install the module

```
npm install @wordpress/data --save
```

*This package assumes that your code will run in an **ES2015+** environment. If you're using an environment that has limited or no support for such language features and APIs, you should include the polyfill shipped in `@wordpress/babel-preset-default` in your code.*

Registering a Store

Use the `register` function to add your own store to the centralized data registry. This function accepts one argument – a store descriptor that can be created with `createReduxStore` factory function. `createReduxStore` accepts two arguments: a name to identify the module, and a configuration object with values describing how your state is represented, modified, and accessed. At a minimum, you must provide a reducer function describing the shape of your state and how it changes in response to actions dispatched to the store.

```
import apiFetch from '@wordpress/api-fetch';
import { createReduxStore, register } from '@wordpress/data';

const DEFAULT_STATE = {
    prices: {},
    discountPercent: 0,
};

const actions = {
    setPrice( item, price ) {
        return {
            type: 'SET_PRICE',
            item,
            price,
        };
    },

    startSale( discountPercent ) {
        return {
```

```
            type: 'START_SALE',
            discountPercent,
        };
    },
};

const store = createReduxStore( 'my-shop', {
    reducer( state = DEFAULT_STATE, action ) {
        switch ( action.type ) {
            case 'SET_PRICE':
                return {
                    ...state,
                    prices: {
                        ...state.prices,
                        [ action.item ]: action.price,
                    },
                };

            case 'START_SALE':
                return {
                    ...state,
                    discountPercent: action.discountPercent,
                };
        }

        return state;
    },

    actions,

    selectors: {
        getPrice( state, item ) {
            const { prices, discountPercent } = state;
            const price = prices[ item ];

            return price * ( 1 - 0.01 * discountPercent );
        },
    },

    resolvers: {
```

```
        getPrice: ( item ) => async ({ dispatch }) => { {
            const path = '/wp/v2/prices/' + item;
            const price = await apiFetch( { path } );
            dispatch.setPrice( item, price );
        },
    },
} );

register( store );
```

The return value of `createReduxStore` is the `StoreDescriptor` object that contains two properties:

- `name` (`string`) – the name of the store
- `instantiate` (`Function`) – it returns a Redux-like store object with the following methods:
  - `getState()`: Returns the state value of the registered reducer
    - *Redux parallel:* `getState`
  - `subscribe( listener: Function )`: Registers a function called any time the value of state changes.
    - *Redux parallel:* `subscribe`
  - `dispatch( action: Object )`: Given an action object, calls the registered reducer and updates the state value.
    - *Redux parallel:* `dispatch`

Redux Store Options

`reducer`

A **reducer** is a function accepting the previous `state` and `action` as arguments and returns an updated `state` value.

`actions`

The **`actions`** object should describe all action creators available for your store. An action creator is a function that optionally accepts arguments and returns an action object to dispatch to the registered reducer. *Dispatching actions is the primary mechanism for making changes to your state.*

`selectors`

The **`selectors`** object includes a set of functions for accessing and deriving state values. A selector is a function which accepts state and optional arguments and returns some value from state. *Calling selectors is the primary mechanism for retrieving data from your state*, and serve as a useful abstraction over the raw data which is typically more susceptible to change and less readily

usable as a normalized object.

`resolvers`

A **resolver** is a side-effect for a selector. If your selector result may need to be fulfilled from an external source, you can define a resolver such that the first time the selector is called, the fulfillment behavior is effected.

The `resolvers` option should be passed as an object where each key is the name of the selector to act upon, the value a function which receives the same arguments passed to the selector, excluding the state argument. It can then dispatch as necessary to fulfill the requirements of the selector, taking advantage of the fact that most data consumers will subscribe to subsequent state changes (by `subscribe` or `withSelect`).

Resolvers, in combination with thunks, can be used to implement asynchronous data flows for your store.

Object-form resolvers

A resolver can be defined as an object with a `fulfill` method and optional `isFulfilled` and `shouldInvalidate` methods:

```
resolvers: {
    getPage: {
        fulfill: ( id ) => async ( { dispatch } ) => {
            const data = await fetchPage( id );
            dispatch( { type: 'RECEIVE_PAGE', id, data } );
        },
        isFulfilled: ( state, id ) => !! state.pages[ id ],
        shouldInvalidate: ( action, id ) => action.type === 'INVALIDATE_PAGE' && action.id === id,
    },
},
```

A resolver can also be a plain function with `isFulfilled` or `shouldInvalidate` assigned as properties. It will be normalized into the `{ fulfill, isFulfilled, shouldInvalidate }` object form internally.

`isFulfilled( state, ...args )` lets you override the `hasFinishedResolution` meta selector for a given set of args. Normally, a resolver is skipped only when the resolution metadata records that it has already run for those args. With `isFulfilled`, you can look at the actual store state and decide that the data is already there: for example because it was loaded by a different resolver or preloaded server-side. When `isFulfilled` returns `true`, the `fulfill` method is not called and no resolution metadata is written.

`shouldInvalidate( action, ...args )` is called on every dispatched action for each set of args that has already been resolved. If it returns `true`, the resolution for those args is invalidated, causing the resolver to run again on the next selector call. This is useful for automatically re-fetching data when a related action signals that the cached result may be stale.

`controls` (deprecated)

To handle asynchronous data flows, it is recommended to use thunks instead of `controls`.

View *controls* explanation

A *control* defines the execution flow behavior associated with a specific action type. Before thunks, controls were used to implement asynchronous data flows for your store. By defining your action creator or resolvers as a generator which yields specific controlled action types, the execution will proceed as defined by the control handler.

The *controls* option should be passed as an object where each key is the name of the action type to act upon, the value a function which receives the original action object. It should returns either a promise which is to resolve when evaluation of the action should continue, or a value. The value or resolved promise value is assigned on the return value of the yield assignment. If the control handler returns undefined, the execution is not continued.

Refer to the documentation of *@wordpress/redux-routine* for more information.\

`initialState`

An optional preloaded initial state for the store. You may use this to restore some serialized state value or a state generated server-side.

Generic Stores

The `@wordpress/data` module offers a more advanced and generic interface for the purposes of integrating other data systems and situations where more direct control over a data system is needed. In this case, a data store will need to be implemented outside of `@wordpress/data` and then plugged in via three functions:

- `getSelectors()`: Returns an object of selector functions, pre-mapped to the store.
- `getActions()`: Returns an object of action functions, pre-mapped to the store.
- `subscribe( listener: Function )`: Registers a function called any time the value of state changes.
  - Behaves as Redux `subscribe` with the following differences:

- Doesn’t have to implement an unsubscribe, since the registry never uses it.
  – Only has to support one listener (the registry).

By implementing the above interface for your custom store, you gain the benefits of using the registry and the `withSelect` and `withDispatch` higher order components in your application code. This provides seamless integration with existing and alternative data systems.

Integrating an existing redux store with its own reducers, store enhancers and middleware can be accomplished as follows:

Example:

```
import { register } from '@wordpress/data';
import existingSelectors from './existing-app/selectors';
import existingActions from './existing-app/actions';
import createStore from './existing-app/store';

const reduxStore = createStore();

const mapValues = ( obj, callback ) =>
    Object.entries( obj ).reduce(
        ( acc, [ key, value ] ) => ( {
            ...acc,
            [ key ]: callback( value ),
        } ),
        {}
    );

const boundSelectors = mapValues(
    existingSelectors,
    ( selector ) =>
        ( ...args ) =>
            selector( reduxStore.getState(), ...args )
);

const boundActions = mapValues(
    existingActions,
    ( action ) =>
        ( ...args ) =>
            reduxStore.dispatch( action( ...args ) )
);
```

```
const genericStore = {
    name: 'existing-app',
    instantiate: () => ( {
        getSelectors: () => boundSelectors,
        getActions: () => boundActions,
        subscribe: reduxStore.subscribe,
    } ),
};

register( genericStore );
```

It is also possible to implement a completely custom store from scratch:

Example:

```
import { register } from '@wordpress/data';

function customStore() {
    return {
        name: 'custom-data',
        instantiate: () => {
            const listeners = new Set();
            const prices = { hammer: 7.5 };

            function storeChanged() {
                for ( const listener of listeners ) {
                    listener();
                }
            }

            function subscribe( listener ) {
                listeners.add( listener );
                return () => listeners.delete( listener );
            }

            const selectors = {
                getPrice( itemName ) {
                    return prices[ itemName ];
                },
```

```
            };

            const actions = {
                setPrice( itemName, price ) {
                    prices[ itemName ] = price;
                    storeChanged();
                },
            };

            return {
                getSelectors: () => selectors,
                getActions: () => actions,
                subscribe,
            };
        },
    };
}

register( customStore );
```

Comparison with Redux

The data module shares many of the same core principles and API method naming of Redux. In fact, it is implemented atop Redux. Where it differs is in establishing a modularization pattern for creating separate but interdependent stores, and in codifying conventions such as selector functions as the primary entry point for data access.

The higher-order components were created to complement this distinction. The intention with splitting `withSelect` and `withDispatch` — where in React Redux they are combined under `connect` as `mapStateToProps` and `mapDispatchToProps` arguments — is to more accurately reflect that dispatch is not dependent upon a subscription to state changes, and to allow for state-derived values to be used in `withDispatch` (via higher-order component composition).

The data module also has built-in solutions for handling asynchronous side-effects, through resolvers and thunks. These differ slightly from standard redux async solutions like `redux-thunk` or `redux-saga`.

Specific implementation differences from Redux and React Redux:

- In Redux, a `subscribe` listener is called on every dispatch, regardless of whether the value of state has changed.
  - In `@wordpress/data`, a subscriber is only called when state has changed.
- In React Redux, a `mapStateToProps` function must return an object.
  - In `@wordpress/data`, a `withSelect` mapping function can return `undefined` if it has no props to inject.
- In React Redux, the `mapDispatchToProps` argument can be defined as an object or a function.
  - In `@wordpress/data`, the `withDispatch` higher-order component creator must be passed a function.

API

AsyncModeProvider

Context Provider Component used to switch the data module component rerendering between Sync and Async modes.

Usage

```
import { useSelect, AsyncModeProvider } from '@wordpress/data';
import { store as blockEditorStore } from '@wordpress/block-editor';

function BlockCount() {
    const count = useSelect( ( select ) => {
        return select( blockEditorStore ).getBlockCount();
    }, [] );

    return count;
}

function App() {
    return (
        <AsyncModeProvider value={ true }>
            <BlockCount />
        </AsyncModeProvider>
    );
}
```

In this example, the BlockCount component is rerendered asynchronously.
It means if a more critical task is being performed (like typing in an input),
the rerendering is delayed until the browser becomes IDLE.
It is possible to nest multiple levels of AsyncModeProvider to fine-tune the rendering behavior.

Parameters

- *props.value* `boolean`: Enable Async Mode.

combineReducers

The combineReducers helper function turns an object whose values are different reducing functions into a single reducing function you can pass to registerReducer.

Usage

```
import { combineReducers, createReduxStore, register } from '@wordpress/data';

const prices = ( state = {}, action ) => {
    return action.type === 'SET_PRICE'
        ? {
                ...state,
                [ action.item ]: action.price,
            }
        : state;
};

const discountPercent = ( state = 0, action ) => {
    return action.type === 'START_SALE' ? action.discountPercent : state;
};

const store = createReduxStore( 'my-shop', {
    reducer: combineReducers( {
        prices,
        discountPercent,
    } ),
} );
register( store );
```

Type

- `import('./types').combineReducers`

Parameters

- *reducers* `Object`: An object whose values correspond to different reducing functions that need to be combined into one.

Returns

- `Function`: A reducer that invokes every reducer inside the reducers object, and constructs a state object with the same shape.

controls

Undocumented declaration.

createReduxStore

Creates a data store descriptor for the provided Redux store configuration containing properties describing reducer, actions, selectors, controls and resolvers.

Usage

```
import { createReduxStore } from '@wordpress/data';

const store = createReduxStore( 'demo', {
    reducer: ( state = 'OK' ) => state,
    selectors: {
        getValue: ( state ) => state,
    },
} );
```

Parameters

- *key* `string`: Unique namespace identifier.
- *options* `ReduxStoreConfig< State, Actions, Selectors >`: Registered store options, with properties describing reducer, actions, selectors, and resolvers.

Returns

- `StoreDescriptor< ReduxStoreConfig< State, Actions, Selectors > >`: Store Object.

createRegistry

Creates a new store registry, given an optional object of initial store configurations.

Parameters

- *storeConfigs* `Record< string, ReduxStoreConfig< any, any, any > >`: Initial store configurations.
- *parent* `DataRegistry | null`: Parent registry.

Returns

- `DataRegistry`: Data registry.

createRegistryControl

Creates a control function that takes additional curried argument with the `registry` object. While a regular control has signature

```
( action ) => iteratorOrPromise;
```

where the control works with the `action` that it's bound to, a registry control has signature:

```
( registry ) => ( action ) => iteratorOrPromise;
```

A registry control is typically used to select data or dispatch an action to a registered store.

When registering a control created with `createRegistryControl` with a store, the store knows which calling convention to use when executing the control.

Parameters

- *registryControl* `T & { isRegistryControl?: boolean; }`: Function receiving a registry object and returning a control.

Returns

- Registry control that can be registered with a store.

createRegistrySelector

Creates a selector function that takes additional curried argument with the registry `select` function. While a regular selector has signature

```
( state, ...selectorArgs ) => result;
```

that allows to select data from the store's `state`, a registry selector has signature:

```
( select ) =>
    ( state, ...selectorArgs ) =>
        result;
```

that supports also selecting from other registered stores.

Usage

```
import { store as coreStore } from '@wordpress/core-data';
import { store as editorStore } from '@wordpress/editor';

const getCurrentPostId = createRegistrySelector( ( select ) => ( state ) => {
    return select( editorStore ).getCurrentPostId();
} );

const getPostEdits = createRegistrySelector( ( select ) => ( state ) => {
    // calling another registry selector just like any other function
    const postType = getCurrentPostType( state );
    const postId = getCurrentPostId( state );
    return select( coreStore ).getEntityRecordEdits(
        'postType',
        postType,
        postId
    );
} );
```

Note how the `getCurrentPostId` selector can be called just like any other function, (it works even inside a regular non-registry selector) and we don't need to pass the registry as argument. The registry binding happens automatically when registering the selector with a store.

Parameters

- *registrySelector* `( select: ) => Selector`: Function receiving a registry `select` function and returning a state selector.

Returns

- `RegistrySelector< Selector >`: Registry selector that can be registered with a store.

createSelector

Creates a memoized selector that caches the computed values according to the array of “dependants” and the selector parameters, and recomputes the values only when any of them changes.

See The documentation for the `rememo` package from which the `createSelector` function is reexported.

Type

- `( selector: S, getDependants: GetDependants ) => S & EnhancedSelector`

dispatch

Given a store descriptor, returns an object of the store’s action creators. Calling an action creator will cause it to be dispatched, updating the state value accordingly.

Note: Action creators returned by the dispatch will return a promise when they are called.

Warning: This global `dispatch` function only works with the default registry. It will not work with custom `RegistryProvider` or `BlockEditorProvider` contexts. In React components, prefer the `useDispatch` hook instead, which is registry-aware.

Usage

```
import { dispatch } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

dispatch( myCustomStore ).setPrice( 'hammer', 9.75 );
```

Parameters

- *storeNameOrDescriptor* `StoreNameOrDescriptor`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `DispatchReturn< StoreNameOrDescriptor >`: Object containing the action creators.

EnhancedSelector

Undocumented declaration.

GetDependants

Undocumented declaration.

keyedReducer

Higher-order reducer creator which creates a combined reducer object, keyed by a property on the action object.

Usage

```
import { keyedReducer } from '@wordpress/data';

const itemsByContext = keyedReducer( 'context' )( ( state = [], action ) => {
    switch ( action.type ) {
        case 'ADD_ITEM':
            return [ ...state, action.item ];
    }
    return state;
} );
```

Parameters

- *actionProperty* `string`: Action property by which to key object.

Returns

- Higher-order reducer.

plugins

Object of available plugins to use with a registry.

Related

- use

register

Registers a standard `@wordpress/data` store descriptor.

Usage

```
import { createReduxStore, register } from '@wordpress/data';

const store = createReduxStore( 'demo', {
    reducer: ( state = 'OK' ) => state,
    selectors: {
        getValue: ( state ) => state,
    },
} );
register( store );
```

Parameters

- *store* `StoreDescriptor`: Store descriptor.

registerGenericStore

> **Deprecated** Use `register( storeDescriptor )` instead.

Registers a generic store instance.

Parameters

- *name* `string`: Store registry name.
- *store* `Object`: Store instance (`{ getSelectors, getActions, subscribe }`).

registerStore

> **Deprecated** Use `register` instead.

Registers a standard `@wordpress/data` store.

Parameters

- *storeName* `string`: Unique namespace identifier for the store.

- *options* `Object`: Store description (reducer, actions, selectors, resolvers).

Returns

- `Object`: Registered store object.

RegistryConsumer

A custom React context consumer exposing the provided `registry` to children components. Used along with the RegistryProvider.

You can read more about the React context API here: https://react.dev/learn/passing-data-deeply-with-context#step-3-provide-the-context

Usage

```
import {
  RegistryProvider,
  RegistryConsumer,
  createRegistry
} from '@wordpress/data';

const registry = createRegistry( {} );

const App = ( { props } ) => {
  return <RegistryProvider value={ registry }>
    <div>Hello There</div>
    <RegistryConsumer>
      { ( registry ) => (
        <ComponentUsingRegistry
                { ...props }
               registry={ registry }
      ) }
    </RegistryConsumer>
  </RegistryProvider>
}
```

RegistryProvider

A custom Context provider for exposing the provided `registry` to children components via a consumer.

See RegistryConsumer documentation for example.

resolveSelect

Given a store descriptor, returns an object containing the store's selectors pre-bound to state so that you only need to supply additional arguments, and modified so that they return promises that resolve to their eventual values, after any resolvers have ran.

Usage

```
import { resolveSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

resolveSelect( myCustomStore ).getPrice( 'hammer' ).then( console.log );
```

Parameters

- *storeNameOrDescriptor* `StoreDescriptor|string`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `CurriedSelectorsResolveOf< T >`: Object containing the store's promise-wrapped selectors.

select

Given a store descriptor, returns an object of the store's selectors. The selector functions are been pre-bound to pass the current state automatically. As a consumer, you need only pass arguments of the selector, if applicable.

Warning: This global `select` function only works with the default registry. It will not work with custom `RegistryProvider` or `BlockEditorProvider` contexts. In React components, prefer the `useSelect` hook instead, which is registry-aware.

Usage

```
import { select } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';
```

```
select( myCustomStore ).getPrice( 'hammer' );
```

Parameters

- *storeNameOrDescriptor* `string | T`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `CurriedSelectorsOf< T >`: Object containing the store's selectors.

subscribe

Given a listener function, the function will be called any time the state value of one of the registered stores has changed. If you specify the optional `storeNameOrDescriptor` parameter, the listener function will be called only on updates on that one specific registered store.

This function returns an `unsubscribe` function used to stop the subscription.

Usage

```
import { subscribe } from '@wordpress/data';

const unsubscribe = subscribe( () => {
    // You could use this opportunity to test whether the derived result of a
    // selector has subsequently changed as the result of a state update.
} );

// Later, if necessary...
unsubscribe();
```

Parameters

- *listener* `Function`: Callback function.
- *storeNameOrDescriptor* `string|StoreDescriptor?`: Optional store name.

suspendSelect

Given a store descriptor, returns an object containing the store's selectors pre-bound to state so that you only need to supply additional arguments, and modified so that they throw promises in case the selector is not resolved yet.

Parameters

- *storeNameOrDescriptor* `StoreDescriptor|string`: The store descriptor. The legacy calling convention of passing the store name is also supported.

Returns

- `Object`: Object containing the store's suspense-wrapped selectors.

use

Extends a registry to inherit functionality provided by a given plugin. A plugin is an object with properties aligning to that of a registry, merged to extend the default registry behavior.

Parameters

- *plugin* `Object`: Plugin object.

useDispatch

A custom react hook returning the current registry dispatch actions creators.

Note: The component using this hook must be within the context of a RegistryProvider.

Usage

This illustrates a pattern where you may need to retrieve dynamic data from the server via the `useSelect` hook to use in combination with the dispatch action.

```
import { useCallback } from 'react';
import { useDispatch, useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function Button( { onClick, children } ) {
    return (
        <button type="button" onClick={ onClick }>
            { children }
        </button>
    );
}
```

```
const SaleButton = ( { children } ) => {
    const { stockNumber } = useSelect(
        ( select ) => select( myCustomStore ).getStockNumber(),
        []
    );
    const { startSale } = useDispatch( myCustomStore );
    const onClick = useCallback( () => {
        const discountPercent = stockNumber > 50 ? 10 : 20;
        startSale( discountPercent );
    }, [ stockNumber ] );
    return <Button onClick={ onClick }>{ children }</Button>;
};

// Rendered somewhere in the application:
//
// <SaleButton>Start Sale!</SaleButton>
```

Parameters

- *storeNameOrDescriptor* `StoreNameOrDescriptor`: Optionally provide the name of the store or its descriptor from which to retrieve action creators. If not provided, the registry.dispatch function is returned instead.

Returns

- `UseDispatchReturn< StoreNameOrDescriptor >`: The dispatch function or action creators for the store.

useRegistry

A custom React hook exposing the registry context for use.

This exposes the `registry` value provided via the Registry Provider to a component implementing this hook.

It acts similarly to the `useContext` React hook.

Note: Generally speaking, `useRegistry` is a low level hook that in most cases won't be needed for implementation. Most interactions with the `@wordpress/data` API can be performed via the `useSelect` hook, or the `withSelect` and `withDispatch` higher order components.

Usage

```
import { RegistryProvider, createRegistry, useRegistry } from '@wordpress/data';

const registry = createRegistry( {} );

const SomeChildUsingRegistry = ( props ) => {
    const registry = useRegistry();
    // ...logic implementing the registry in other react hooks.
};

const ParentProvidingRegistry = ( props ) => {
    return (
        <RegistryProvider value={ registry }>
            <SomeChildUsingRegistry { ...props } />
        </RegistryProvider>
    );
};
```

Returns

- `DataRegistry`: A custom React hook exposing the registry context value.

useSelect

Custom react hook for retrieving props from registered selectors.

In general, this custom React hook follows the rules of hooks.

Usage

```
import { useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function HammerPriceDisplay( { currency } ) {
    const price = useSelect(
        ( select ) => {
            return select( myCustomStore ).getPrice( 'hammer', currency );
        },
        [ currency ]
    );
    return new Intl.NumberFormat( 'en-US', {
```

```
        style: 'currency',
        currency,
    } ).format( price );
}

// Rendered in the application:
// <HammerPriceDisplay currency="USD" />
```

In the above example, when `HammerPriceDisplay` is rendered into an application, the price will be retrieved from the store state using the `mapSelect` callback on `useSelect`. If the currency prop changes then any price in the state for that currency is retrieved. If the currency prop doesn't change and other props are passed in that do change, the price will not change because the dependency is just the currency.

When data is only used in an event callback, the data should not be retrieved on render, so it may be useful to get the selectors function instead.

Don't use `useSelect` this way when calling the selectors in the render function because your component won't re-render on a data change.

```
import { useSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function Paste( { children } ) {
    const { getSettings } = useSelect( myCustomStore );
    function onPaste() {
        // Do something with the settings.
        const settings = getSettings();
    }
    return <div onPaste={ onPaste }>{ children }</div>;
}
```

Parameters

- *mapSelect* `T`: Function called on every state change. The returned value is exposed to the component implementing this hook. The function receives the `registry.select` method on the first argument and the `registry` on the second argument. When a store key is passed, all selectors for the store will be returned. This is only meant for usage of these selectors in event callbacks, not for data needed to create the element tree.
- *deps* `unknown[]`: If provided, this memoizes the mapSelect so the same `mapSelect` is invoked on every state change unless the dependencies change.

Returns

- `UseSelectReturn< T >`: The selected data or store selectors.

useSuspenseSelect

A variant of the `useSelect` hook that has the same API, but is a compatible Suspense-enabled data source.

Parameters

- *mapSelect* `T`: Function called on every state change. The returned value is exposed to the component using this hook. The function receives the `registry.suspendSelect` method as the first argument and the `registry` as the second one.
- *deps* `unknown[]`: A dependency array used to memoize the `mapSelect` so that the same `mapSelect` is invoked on every state change unless the dependencies change.

Returns

- `ReturnType< T >`: Data object returned by the `mapSelect` function.

withDispatch

Higher-order component used to add dispatch props using registered action creators.

Usage

```
function Button( { onClick, children } ) {
    return (
        <button type="button" onClick={ onClick }>
            { children }
        </button>
    );
}

import { withDispatch } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

const SaleButton = withDispatch( ( dispatch, ownProps ) => {
    const { startSale } = dispatch( myCustomStore );
    const { discountPercent } = ownProps;
```

```
    return {
        onClick() {
            startSale( discountPercent );
        },
    };
} )( Button );

// Rendered in the application:
//
// <SaleButton discountPercent="20">Start Sale!</SaleButton>
```

In the majority of cases, it will be sufficient to use only two first params passed to `mapDispatchToProps` as illustrated in the previous example. However, there might be some very advanced use cases where using the `registry` object might be used as a tool to optimize the performance of your component. Using `select` function from the registry might be useful when you need to fetch some dynamic data from the store at the time when the event is fired, but at the same time, you never use it to render your component. In such scenario, you can avoid using the `withSelect` higher order component to compute such prop, which might lead to unnecessary re-renders of your component caused by its frequent value change.
Keep in mind, that `mapDispatchToProps` must return an object with functions only.

```
function Button( { onClick, children } ) {
    return (
        <button type="button" onClick={ onClick }>
            { children }
        </button>
    );
}

import { withDispatch } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

const SaleButton = withDispatch( ( dispatch, ownProps, { select } ) => {
    // Stock number changes frequently.
    const { getStockNumber } = select( myCustomStore );
    const { startSale } = dispatch( myCustomStore );
    return {
        onClick() {
```

```
            const discountPercent = getStockNumber() > 50 ? 10 : 20;
            startSale( discountPercent );
        },
    };
} )( Button );

// Rendered in the application:
//
//  <SaleButton>Start Sale!</SaleButton>
```

Note: It is important that the `mapDispatchToProps` function always returns an object with the same keys. For example, it should not contain conditions under which a different value would be returned.

Parameters

- *mapDispatchToProps* `( dispatch: DataRegistry[ 'dispatch' ], ownProps: Record< string, unknown >, registry: DataRegistry ) => Record< string, ( , ...args: never[] ) => unknown >`: A function of returning an object of prop names where value is a dispatch-bound action creator, or a function to be called with the component's props and returning an action creator.

Returns

- Enhanced component with merged dispatcher props.

withRegistry

Higher-order component which renders the original component with the current registry context passed as its `registry` prop.

withSelect

Higher-order component used to inject state-derived props using registered selectors.

Usage

```
import { withSelect } from '@wordpress/data';
import { store as myCustomStore } from 'my-custom-store';

function PriceDisplay( { price, currency } ) {
    return new Intl.NumberFormat( 'en-US', {
```

```
        style: 'currency',
        currency,
    } ).format( price );
}

const HammerPriceDisplay = withSelect( ( select, ownProps ) => {
    const { getPrice } = select( myCustomStore );
    const { currency } = ownProps;

    return {
        price: getPrice( 'hammer', currency ),
    };
} )( PriceDisplay );

// Rendered in the application:
//
//  <HammerPriceDisplay currency="USD" />
```

In the above example, when `HammerPriceDisplay` is rendered into an application, it will pass the price into the underlying `PriceDisplay` component and update automatically if the price of a hammer ever changes in the store.

Parameters

- *mapSelectToProps* `( select: SelectFunction, ownProps: Record< string, unknown >, registry: DataRegistry ) => Record< string, unknown >`: Function called on every state change, expected to return object of props to merge with the component's own props.

Returns

- Enhanced component with merged state data props.

batch

The `batch` method allows multiple store updates to occur simultaneously, reducing unnecessary executions of selectors and component re-renders during sequential state changes.

In WordPress data applications, dispatching consecutive actions typically triggers store listeners and runs selectors, which can lead to re-renders. The `batch` method pauses these listeners and only activates them once at the end, ensuring selectors run only once with the final state.

This method is particularly effective for optimizing performance with expensive selectors, ensuring atomic operations across multiple stores, and creating single undo/redo entries for several synchronous updates.

Unlike React's built-in batching or React Redux's `batch` function, `registry.batch` operates at the store listener level, completely avoiding unnecessary selector computations.

Usage

```
import { useRegistry } from '@wordpress/data';

function Component() {
    const registry = useRegistry();

    function handleComplexUpdate() {
        // Without batch: listeners are called 3 times, which can result in multiple component re-renders.
        // With batch: notifies listeners once, resulting in a single component re-render as needed.
        registry.batch( () => {
            registry.dispatch( 'someStore' ).someAction();
            registry.dispatch( 'someStore' ).someOtherAction();
            registry.dispatch( 'someStore' ).thirdAction();
        } );
    }

    return <button onClick={ handleComplexUpdate }>Update</button>;
}
```

Selectors

The following selectors are available on the object returned by `wp.data.select( 'core' )`.

Example

```
import { store as coreDataStore } from '@wordpress/core-data';
import { useSelect } from '@wordpress/data';

function Component() {
```

```
const result = useSelect( ( select ) => {
    const query = { per_page: 20 };
    const selectorArgs = [ 'postType', 'page', query ];

    return {
        pages: select( coreDataStore ).getEntityRecords( ...selectorArgs ),
        hasStartedResolution: select( coreDataStore ).hasStartedResolution(
            'getEntityRecords', // _selectorName_
            selectorArgs
        ),
        hasFinishedResolution: select(
            coreDataStore
        ).hasFinishedResolution( 'getEntityRecords', selectorArgs ),
        isResolving: select( coreDataStore ).isResolving(
            'getEntityRecords',
            selectorArgs
        ),
    };
} );

if ( result.hasStartedResolution ) {
    return <>Fetching data...</>;
}

if ( result.isResolving ) {
    return (
        <>
            {
                // show a spinner
             }
        </>
    );
}

if ( result.hasFinishedResolution ) {
    return (
        <>
            {
                // data is ready
```

```
                }
            </>
        );
    }
}
```

hasFinishedResolution

Returns true if resolution has completed for a given selector name, and arguments set.

Parameters

- *state* `State`: Data state.
- *selectorName* `string`: Selector name.
- *args* `unknown[]?`: Arguments passed to selector.

Returns

- `boolean`: Whether resolution has completed.

hasStartedResolution

Returns true if resolution has already been triggered for a given selector name, and arguments set.

Parameters

- *state* `State`: Data state.
- *selectorName* `string`: Selector name.
- *args* `unknown[]?`: Arguments passed to selector.

Returns

- `boolean`: Whether resolution has been triggered.

isResolving

Returns true if resolution has been triggered but has not yet completed for a given selector name, and arguments set.

Parameters

- *state* `State`: Data state.

- *selectorName* `string`: Selector name.
- *args* `unknown[]?`: Arguments passed to selector.

Returns

- `boolean`: Whether resolution is in progress.

Normalizing Selector Arguments

In specific circumstances it may be necessary to normalize the arguments passed to a given *call* of a selector/resolver pairing.

Each resolver has its resolution status cached in an internal state where the key is the arguments supplied to the selector at *call* time.

For example for a selector with a single argument, the related resolver would generate a cache key of: `[ 123 ]`.

This cache is used to determine the resolution status of a given resolver which is used to avoid unwanted additional invocations of resolvers (which often undertake “expensive” operations such as network requests).

As a result it’s important that arguments remain *consistent* when calling the selector. For example, by *default* these two calls will not be cached using the same key, even though they are likely identical:

```
// Arg as number
getSomeDataById( 123 );

// Arg as string
getSomeDataById( '123' );
```

This is an opportunity to utilize the `normalizeArgs` property to guarantee consistency by protecting callers from passing incorrect types.

`normalizeArgs` was previously named `__unstableNormalizeArgs`, and the old name still works.

Example

The *3rd* argument of the following selector is intended to be a `Number`:

```
const getItemsSelector = ( state, name, type, id ) => {
    return state.items[ name ][ type ][ id ] || null;
```

```
};
```

However, it is possible that the `id` parameter will be passed as a `String`. In this case, the `normalizeArgs` method (property) can be defined on the *selector* to coerce the arguments to the desired type even if they are provided “incorrectly”:

```
// Define normalization method.
getItemsSelector.normalizeArgs = ( args ) => {
    const newArgs = [ ...args ];
    // "id" argument at the 2nd index
    if ( newArgs[ 2 ] && typeof newArgs[ 2 ] === 'string' ) {
        newArgs[ 2 ] = Number( newArgs[ 2 ] );
    }

    return newArgs;
};
```

With this in place the following code will behave consistently:

```
const getItemsSelector = ( state, name, type, id ) => {
    // here 'id' is now guaranteed to be a number.
    return state.items[ name ][ type ][ id ] || null;
};

const getItemsResolver = ( name, type, id ) => {
    // 'id' is also guaranteed to be a number in the resolver.
    return {};
};

registry.registerStore( 'store', {
    // ...
    selectors: {
        getItems: getItemsSelector,
    },
    resolvers: {
        getItems: getItemsResolver,
    },
} );

// Call with correct number type.
registry.select( 'store' ).getItems( 'foo', 'bar', 54 );
```

```
// Call with the wrong string type, **but** here we have avoided an
// unwanted resolver call because '54' is guaranteed to have been
// coerced to a number by the `normalizeArgs` method.
registry.select( 'store' ).getItems( 'foo', 'bar', '54' );
```

Ensuring consistency of arguments for a given selector call is an important optimization to help improve performance in the data layer. However, this type of problem can be usually be avoided by ensuring selectors don't use variable types for their arguments.

Sharing a Resolution Between Selector Calls

`normalizeArgs` changes the arguments for the selector and the resolver together, so it can only coerce values the selector still needs. Sometimes a selector takes an argument the resolver has no use for, and every call that differs only in that argument should share one resolution.

Define `getResolutionArgs` on the *resolver* to say which arguments the resolution is keyed by. It runs after `normalizeArgs`. Its result is the resolution cache key, and it is what `fulfill`, `isFulfilled` and `shouldInvalidate` are called with. The selector keeps its own arguments.

For example, `canUser( action, resource )` answers one action at a time, but a single request returns the permissions for every action on the resource. Leaving `action` out of the resolution arguments makes all the calls for a resource share one request:

```
const canUserResolver =
    ( resource ) =>
    async ( { dispatch } ) => {
        // Fetches the permissions for every action at once.
        dispatch.receivePermissions( await fetchPermissions( resource ) );
    };

canUserResolver.getResolutionArgs = ( action, resource ) => [ resource ];

registry.registerStore( 'store', {
    // ...
    selectors: {
        canUser: ( state, action, resource ) =>
            state.permissions[ resource ]?.[ action ],
    },
```

```
    resolvers: {
        canUser: canUserResolver,
    },
} );

// One request, and both calls wait for it.
await registry.resolveSelect( 'store' ).canUser( 'create', 'media' );
await registry.resolveSelect( 'store' ).canUser( 'read', 'media' );
```

Resolution metadata is keyed by these arguments too, so code calling `startResolution`, `finishResolution(s)` or `invalidateResolution` for such a selector has to pass them. Above that means `dispatch.finishResolution( 'canUser', [ 'media' ] )`, not `[ 'create', 'media' ]`.

Going further

- What is WordPress Data?

Contributing to this package

This is an individual package that's part of the Gutenberg project. The project is organized as a monorepo. It's made up of multiple self-contained software packages, each with a specific purpose.

The packages in this monorepo are published to npm and used by WordPress as well as other software projects.

To find out more about contributing to this package or Gutenberg as a whole, please read the project's main contributor guide.

□□□ #2
□ □□□ □□□ 2026-10-31 18:18:18 UTC
□ □□□ □□□ 2026-10-31 18:18:22 UTC