## Redux and Typescript

The Redux store shall be fully-typed to provide safety and to enable auto-completion and automated bug-discovery in an IDE.

Please follow these examples when creating new parts of app state and when connecting through `react-redux`.

### Importing Types

Try to keep type imports consistent across different modules.

`action-types` contains the list of possible action types, not surprisingly, as well as the helper types for writing action creators and reducers.

```ts

import \* as A from '../state/action-types';

```

`state` contains the shape of the global Redux state object and the helpers for using `react-redux`'s `connect()` wrapper.

```ts

import \* as S from '../state';

```

`types` contains app-wide type definitions including those for the Simplenote data models, the UI behaviors, and common language/platform helpers.

```ts

import \* as T from '../types';

```

### Defining Component Prop Types

There are multiple legitimate ways to \_type\_ a React component.

In this project we shall explicitly define the prop types by separating out the props we expect to be passed down from a parent component, to come from app state, to dispatch actions, and to be used internally in component state.

The names match the terminology used by `react-redux`

```ts

import \* as T from '../types';

type OwnProps = {

displayMode: 'expanded' | 'concise';

}

type StateProps = {

notes: T.NoteEntity[];

}

type DispatchProps = {

createNote: () => any;

trashNote: (noteId: T.EntityId) => any;

}

type Props = OwnProps & StateProps & DispatchProps;

export class MyComponent extends Component<Props> {

…

}

```

`StateProps` need not mirror the types of elements in the global app state.

These types are specifying the \_component API\_ and so if we have mismatches we are responsible for transforming values inside `mapStateToProps`.

It can be convenient to draw a type from the Redux app state but doing so will weaken the contract for the component.

`DispatchProps` identify what \_actions\_ or side effects a component expects to be able to make.

In almost all cases these will be "fire and forget" functions with no specific demands on their return values.

For example, we might send \_spies\_ in a test file and this is fine according to the component.

Leave the type of `DispatchProps` relaxed to keep this flexibility.

Most components don't utilize any form of local state and so we don't need to define such a type for the sake of having the type.

Similarly, if there are no props being passed down from a parent component there's no need to creaet `OwnProps`.

As a general rule if you find yourself creating an empty type then you should be able to leave it out.

In this example our component only depends on a single value from app state and so the only type we're creating is `StateProps`.

```ts

type StateProps = {

title: string;

};

type Props = StateProps;

```

Should we use `this.state` inside a component then we shall create its type.

```ts

type ComponentState = {

entryText: state;

isComposing: boolean;

};

```

For the special case of `Ref`s create the type in the component and provide the optional node type dependent on what is receiving the `ref`.

```ts

export class Input extends Component {

entry = createRef<HTMLInputElement>();

…

<input ref={this.entry}>

…

}

```

### Connecting React components to Redux

Components shall use the helpers from `state` in combination with the defined component types to provide typing.

```ts

import \* as S from '../state';

…

const mapStateToProps: S.MapState<StateProps, OwnProps> = (state, ownProps) => ({

notes: ownProps.displayMode === 'expanded'

? state.ui.filteredNotes

: state.ui.filteredNotes.slice(0, 10)

})

const mapDispatchToProps: S.DispatchProps<DispatchProps> = {

createNote: () => actionCreators.newNote({content: ''}),

trashNote,

}

export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);

```

Prefer the object form of `mapDispatchToProps` over the function form unless you can't avoid it.

If your prop-mapping functions don't depend on `OwnProps` it is safe to leave that out of the type annotation.

### Creating Reducers

When adding new values into the global Redux app state we need to create a reducer to manage that data type.

Use the `Reducer` helper type to define the type of the property being added to state.

```ts

import \* as A from '../action-types';

import \* as T from '../../types';

const listMode = A.Reducer<T.ListDisplayMode> = (state = 'compact', action) => {

switch (action.type) {

case 'SET\_LIST\_DISPLAY\_MODE':

return action.displayMode;

default:

return state;

}

}

const loginAttempts: A.Reducer<number> = (state = 0, action) => {

switch (action.type) {

case 'LOGIN\_SUBMIT':

return state + 1;

case 'SET\_AUTH':

return action.status === 'authorized' ? 0 : state;

default:

return state;

}

}

const selectedNote: A.Reducer<T.NoteEntity[] | null> = (state = null, action) => {

switch (action.type) {

case 'CREATE\_NOTE':

return makeNote();

case 'TRASH\_NOTE':

return null;

case 'FILTER\_NOTES':

return action.filteredNotes.has(state) ? state : null;

default:

return state;

}

}

```

### Creating Action Types

Each state transition in the app should follow a semantically-oriented Redux ction.

When creating new actions add the \_type\_ specifying that action in `action-types`.

```ts

import \* as T from '../types';

export type CreateNote = Action<'CREATE\_NOTE'>;

export type TrashNote = Action<'TRASH\_NOTE', { noteId: T.EntityId }>;

export type FilterNotes = Action<'FILTER\_NOTES', { notes: T.NoteEntity[] }>;

export type SetListMode = Action<

'SET\_LIST\_DISPLAY\_MODE',

{ displayMode: T.ListDisplayMode; irrelevantProp: boolean }

>;

```

After defining the named action type add it to the \_type\_ of "any possible action."

```ts

export type ActionType = CreateNote | TrashNote | FilterNotes | SetListMode;

```

### Creating Action Creators

Action creators should return dispatch-able actions.

We should not create new thunk-returning actions.

We shall annotate our action-creators with the `ActionCreator` helper type and specify the types of actions it might return for dispatch.

```ts

import \* as A from '../action-types';

import \* as T from '../../types';

const setListMode: A.ActionCreator<A.SetListMode> = (

mode: T.ListDisplaymode

) => ({

type: 'SET\_LIST\_DISPLAY\_MODE',

displayMode: mode,

});

const jumbleApp: A.ActionCreator<A.CreateNote | A.TrashNote> = () =>

Math.random() > 0.5

? { type: 'CREATE\_NOTE' }

: { type: 'TRASH\_NOTE', noteId: randomNoteId() };

```