Redux middleware

The middleware pattern is an expression of the open-closed principle

It allows the developer to insert as many interim functions into the stream of processes without making changes to the processes themselves.

Middleware has a "next()" function

- In any middleware function, you can call next() to pass control to the *next* process.
- We can string any number of middlewares along, each doing it's own specialized functionality until we have a truly beautiful and powerful machine.

Redux middleware are functions that give you more control over what is happening

- Redux allows the insertion of functions into the flow between the dispatch method and the reducer invocation.
- Each can read state and the current action
- Each can introduce additional steps to the process and can alter the process based on logic ...
  - stop the current action or pass it along the chain
  - dispatch new actions -- as many as you like
  - alter the action before it is passed along

What kinds of things will you do with middleware?

- Saving state to local storage and restoring it on startup
- Logging actions and pre/post state for each dispatch
- All asynchronous processing
  - Ajax calls
  - setTimeout() and/or setInterval()
  - Anything involving promises or async/await functions
- Complicated, multi-step processes
- Long-running processes
- Processes that need to re-dispatch or dispatch further actions
- Processes that may have side-effects (non-pure functions)

Shape of middleware

Redux expects your middleware to conform to a strange interface

Warning: While the concept of middleware isn't that tough to understand, the code can be. To smooth that out, just view the code as a recipe and don't waste brainpower on understanding why it is the way it is.

All middleware must be shaped like this:

```
function (store) {
  return function (next) {
    return function (action) {
      // Do stuff in here
    }
  }
}
```

Or written differently ...

```
store => next => action => {
  // Do stuff in here
}
```

But most accurately ...

```
({getState, dispatch}) => next => action => {
  // Do stuff in here
}
```

Therefore the middleware layer has access to four things:

1. the action object being dispatched
2. the dispatch() method
3. the getState() method
4. the next() method

Example: Do nothing but pass control along

```
{getState, dispatch} => next => action => {
  next(action);
}
```

Example: Conditionally dispatch a new action

```
{getState, dispatch} => next => action => {
  if (action.type = types.FOO)
    dispatch(actions.doThing());
  next(action);
}
```

Example: Alter the action in progress

```
{getState, dispatch} => next => action => {
  if (someCondition)
    action.prop = "Some new value";
  next(action);
}
```

Example: Run some code AFTER the next step returns is run

```
{getState, dispatch} => next => action => {
  next(action);
  runAFunction();
}
```

Example: Under certain conditions, abort the dispatch

```
{getState, dispatch} => next => action => {
  if (getState().someCondition)
    // Do nothing
  else
    next(action);
}
```

How to register middleware

The store must be made aware of middleware

- It turns out that createStore actually has this shape:

```
createStore(
  reducerFunction, // The main reducer
  initialState,    // The default state
  ...enhancers     // A list of store "enhancers"
)
```

- Enhancers make the store better. Middleware can be converted to an enhancer using the applyMiddle:

```
const enhancerList = applyMiddleware(...functions);
```