Redux Saga lab

Creating the sagas structure

1. First, install Redux Saga

```
npm install redux-saga.
```

2. Create a new folder called "sagas" under the "stores" folder.
3. Create a sagas.js file and put this in it:

```
export function* rootSaga() {
  console.log("redux-saga ran successfully.");
}
```

This is the world's simplest saga. It will do nothing but console.log() a message but it won't run until we register it. Let's do so in the file where we're creating our Redux store.

4. Edit store.js. Import your saga and createSagaMiddleware at the top:

```
import createSagaMiddleware from 'redux-saga';
import { rootSaga } from './sagas/sagas';
```

5. At the bottom, change the last few lines to look kind of like this:

```
const sagaMiddleware = createSagaMiddleware();  // a
const middleware = applyMiddleware(...middlewares, sagaMiddleware);  // b
export const store = createStore(combinedReducers, initialState, middleware);
sagaMiddleware.run(rootSaga);  // c
```

   Notes:
   a. Calling createSagaMiddleware() and saving it in a const variable called sagaMiddleware. You'll need that variable later.
   b. You're changing the applyMiddleware() line, adding sagaMiddleware. This is registering the hook that your sagas will plug into soon.
   c. Lastly you're plugging your rootSaga into the middleware socket.
6. Run and test. Look at the console for your message. If you can see the message, it means your saga has run.

Deferring it to run on dispatch

We don't usually want our sagas to all run immediately. We usually want them to run when a Redux action is being dispatched. We'll do that using a redux-saga effect called 'take'.

7. Change your rootSaga:

```
export function* rootSaga() {
  yield take('SAY_HELLO');  // <-- Add this line
  console.log("redux-saga ran successfully.");
}
```

8. Run and test. You will not see the message because the take() effect tells redux-saga to only continue when it sees a dispatched action with type of 'SAY_HELLO'.
9. Now add this somewhere at the start of your app:

```
store.dispatch({type:'SAY_HELLO'})
```

10. Run and test again and you'll see your message.

Deferring it to run on every dispatch

11. Copy and paste that store.dispatch() line three or four times. Run the app again. You'll see that it only dispatches the one time.

That's okay if you only want something to fire once, but if you want it to fire every time an action is dispatched, you need the *takeEvery* effect.

12. Remove the "yield take()" line from rootSaga. Replace it with this:

```
yield takeEvery('SAY_HELLO', helloSaga);
```

13. This says to run another saga called *helloSaga* every time 'SAY_HELLO' is dispatched. So obviously we have to write helloSaga. Make it look like this:

```
function* helloSaga() {
  yield console.log("redux-saga ran successfully.");
}
```

14. Run and test again. You should see multiple messages now.

Making an Ajax call

This is all well and good but we're hoping to do more with redux-saga than just writing to the console! Let's make an Ajax call.

15. To prep for testing, run your React application and make sure you can see the list of films that are coming from the initial dispatch to 'FETCH_FILMS'. Do you see them all?
16. Next, open middleware.js. Find the fetchFilmsMiddleware function. Comment out everything in it except for the next(action) line. This will cause it to run and pass control to the reducer but not actually do anything when 'FETCH_FILMS' is dispatched.
17. Run and test. Your application should not show any films because none have been fetched.

Now let's go get them using redux-saga.

18. Create a new saga that looks like this:

```
function* watchFetchFilms() {
  yield takeEvery('FETCH_FILMS', fetchFilms);
}
```

19. This will run fetchFilms when 'FETCH_FILMS' is dispatched, so we need to create fetchFilms:

```
function* fetchFilms(action) {
  const films = yield fetch('/api/films').then(res => res.json());
  yield put(actions.setFilms(films));
}
```

20. Let's register watchFetchFilms. Change rootSaga to do this:

```
yield watchFetchFilms();
```

21. Run and test. The films should all be back.

Of course this means that the only saga running is watchFetchFilms. That's fine for now but if you ever want to run more than one, what do you do? The *all* effect is what you need.

Registering multiple sagas

The all effect allows you to return a saga that combines other sagas. You simply provide an array of generator functions (or sagas) to it.

22. Create a watchSayHelloSaga like so:

```
function* watchSayHelloSaga() {
```

```
  yield takeEvery('SAY_HELLO', helloSaga);
}
```

23. Edit rootSaga and make it look like this:

```
export function* rootSaga() {
  yield all([
    watchSayHelloSaga(),
    watchFetchFilms(),
  ]);
}
```

24. Run and test. You should see your console.log and your films, proving that both are running. When you can see them both, you can be finished.