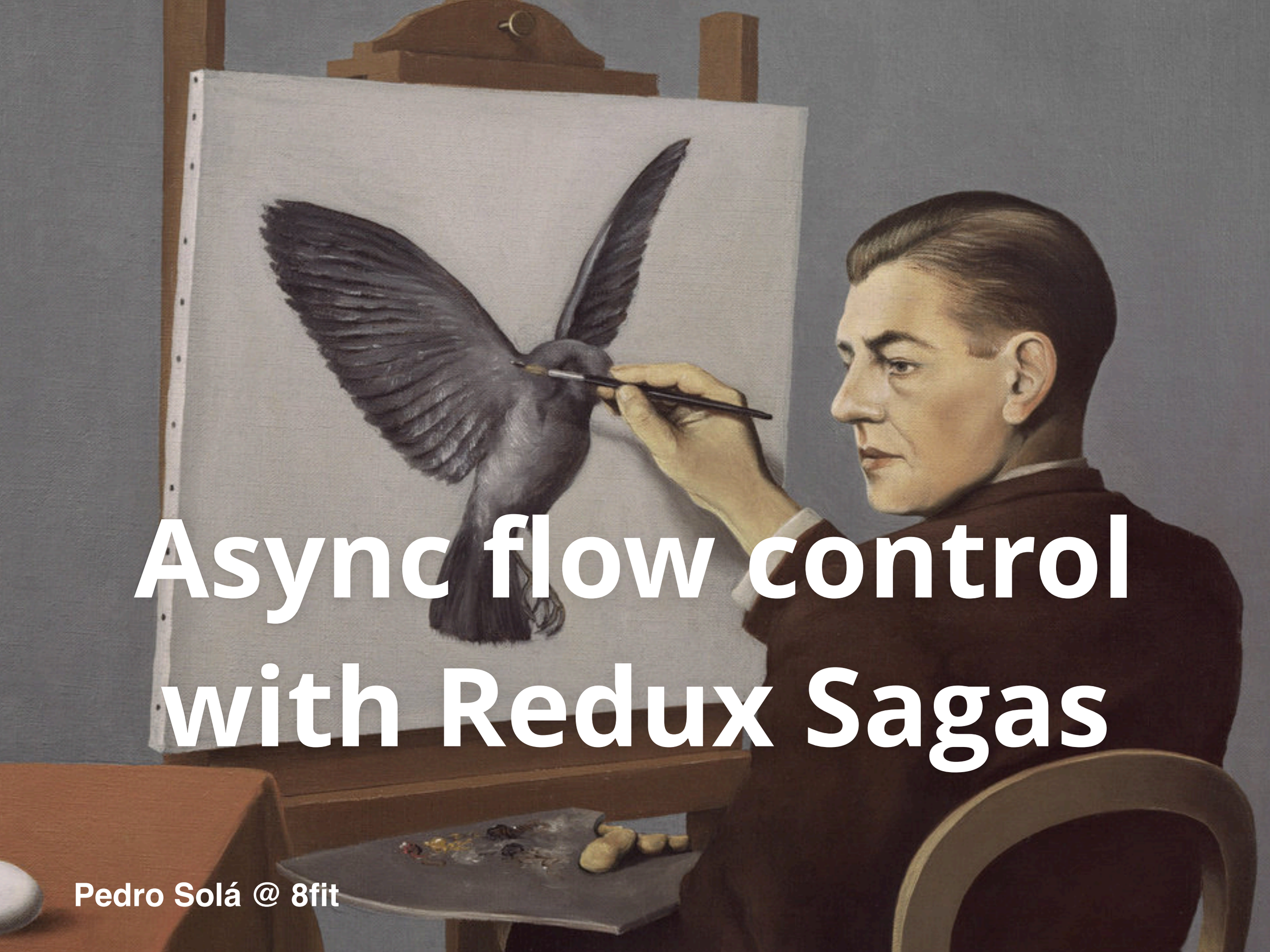


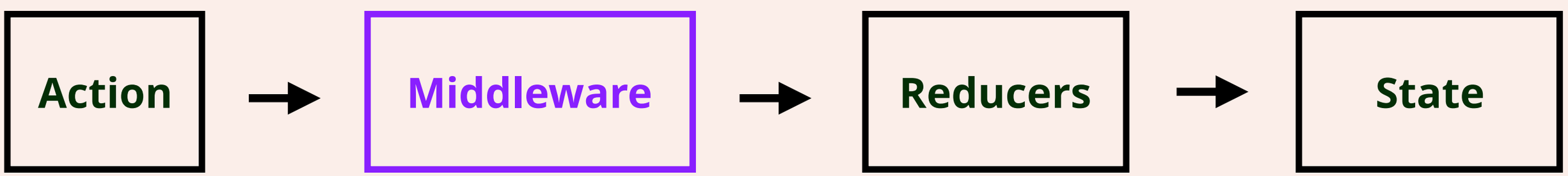
Async flow control with Redux Sagas

Pedro Solá @ 8fit

Sagas

- **Write & reason about asynchronous code as if it was synchronous code**
- **Test like a boss**
- **Complex composition. (fork, cancel, join, race, throttling)**

Side effect management



- redux-thunk
- redux-sagas

Sagas

The mental model is that a saga is like a separate thread in your application that's solely responsible for side effects.

This thread can be started, paused and cancelled from the main application with normal redux actions.

- **Generators**
- **Declarative effects**

Generators

Specified by the function* declaration

Generators can suspend themselves

A generator will be suspended on the yield keyword, returning control to the callee.

```
4
5   function* A() {
6     yield 1;
7     yield 2;
8     yield 3;
9   }
10
11  function* B() {
12    while (true) Math.random();
13  }
14
```

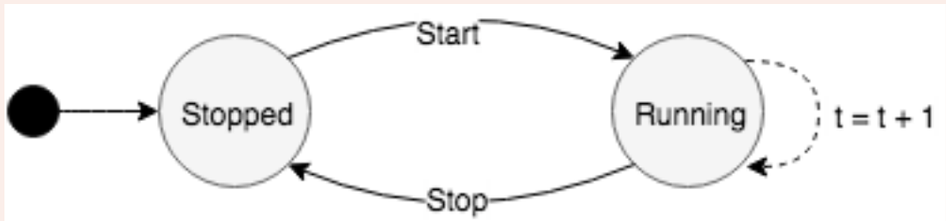
Declarative Effects

You can view Effects like instructions to the middleware to perform some operation (invoke some asynchronous function, dispatch an action to the store).

```
{ '@@redux-saga/IO': true,
  CALL: { context: null, fn: [Function], args: [ 1, 2, 3 ] } }
```

Timer app

Interface



State machine

example from: https://jaysoo.ca/2016/01/03/managing-processes-in-redux-using-sagas/

No middleware

```
class Timer extends Component {

  componentWillReceiveProps(nextProps) {
    const { state: { status: currStatus } } = this.props;
    const { state: { status: nextStatus } } = nextProps;

    if (currState === 'Stopped' && nextState === 'Running') {
      this._startTimer();
    } else if (currState === 'Running' && nextState === 'Stopped') {
      this._stopTimer();
    }
  }

  _startTimer() {
    this._intervalId = setInterval(() => {
        this.props.tick();
    }, 1000);
  }

  _stopTimer() {
    clearInterval(this._intervalId);
  }

  // ...
}
```

Thunk

```
export default {
  start: () => (
    (dispatch, getState) => {
      // This transitions state to Running
      dispatch({ type: 'START' });

      // Check every 1 second if we are still Running.
      // If so, then dispatch a `TICK`, otherwise stop
      // the timer.
      const intervalId = setInterval(() => {
        const { status } = getState();

        if (status === 'Running') {
          dispatch({ type: 'TICK' });
        } else {
          clearInterval(intervalId);
        }
      }, 1000);
    }
  )
  // ...
};
```

Saga

```
function* runTimer() {
  // The sagasMiddleware will start running this generator.

  // Wake up when user starts timer.
  while(yield take('START')) {
    while(true) {

      const { stop, timer } = yield race({
        stop: take('STOP'),
        timer: call(delay, ONE_SECOND),
      });

      // if the stop action has been triggered first,
      // break out of the timer loop
      if (stop) {
        break;
      } else {
        yield put(actions.tick());
      }
    }
  }
}
```

Testing

```
it('should cancel the timer after a STOP action', => {
  const generator = runTimer(); // create the generator object
  let next = generator.next(); // step into

  // the generator is now suspended, waiting for a START
  expect(next).toEqual(take('START'));

  next = generator.next({ type: 'START' });
  const timerRace = race({
    stop: take('STOP'),
    timer: call(delay, ONE_SECOND),
  });
  expect(next).toEqual(timerRace);

  // let's trigger stop before the timer completes
  next = generator.next({ type: 'STOP' });

  // we expect the runTimer to be awaiting a START action now
  expect(next).toEqual(take('START'));
});
```

Common uses

- **API calls**
- **Data sync**
- **Business logic services**
- **Orchestration of components across multiple screens in your applications**

Summary

IMO despite being initially intimidating, expressing asynchronous logic in sagas feels very natural, and is very powerful.

Thanks for listening ❤️

pssssssst! upgrade your life! come work with these beautiful people

pedro@8fit.com