Inspecting With Tap

Easily trace pipe and compose calls using tap(). (3 min. read)

One of `pipe`'s potential drawbacks is losing the ability to easily add code for debugging purposes.

Back when we had this

```
const doMath = (num) => {
  const doubled = num * 2;
  const tripled = doubled * 3;
  const squared = tripled * tripled;

  return squared + 1;
}

const result = doMath(2);

console.log({ result });
```

Jamming a `console.log` or little hack was easy. How do we inspect `pipe`d functions?

Introducing `tap`. It takes a function, usually a logger, and supplies it the current value. You can do whatever you want without affecting the pipeline since you're just "tapping" it! :D

```
import { pipe, tap } from 'ramda';

const doMath = pipe(
  tap((num) => {
    console.log('initial number:', num);
  }),

  double,
  tap((num) => {
    console.log('after doubling:', num);
  }),
  triple,
  tap((num) => {
    console.log('after tripling:', num);
  }),

  square,
  tap((num) => {
    console.log('after squaring:', num);
  }),

  increment,
  tap((num) => {
    console.log('after incrementing:', num);
  }),
);

const result = doMath(2);

console.log({ result });
```

You don't even need to use `tap` itself, just insert a function that returns its arguments unchanged.

```
import { pipe, tap } from 'ramda';

const doMath = pipe(
  (num) => {
    console.log('initial number:', num);
    return num;
  },

  double,
  (num) => {
    console.log('after doubling:', num);
    return num;
  },

  triple,
  (num) => {
    console.log('after tripling:', num);
    return num;
  },

  square,
  (num) => {
    console.log('after squaring:', num);
    return num;
  },

  increment,
  (num) => {
    console.log('after incrementing:', num);
    return num;
  }
);

const result = doMath(2);


console.log({ result });
```