Loaning

The elegant way to managing resources

(De-)Motivation

Code all too often looks like this:

```
const doEverything = resourceConfig => {
  try {
    const resource = aquire(resourceConfig);

    const someData = resource.query('SOME QUERY');
    const processedData = process(someData);
    return processedData;
  } finally {
    resource && dispose(resource);
  }
}

const result = doEverything(resourceConfig);
```

What we need to do

- Acquire a **resource**
- Execute a query
- Dispose of the **resource**
- Handle disposal of the **resource** even in exception case!
- Again, and again, and again...

What we want to do

- Acquire a resource
- Execute a query <= This is what we actually **want** to do
- Dispose of the resource
- Handle disposal of the resource even in exception case!
- Once <= And *maybe* this

Motivation

- Adhere to DRY principle
  - Reusable resource management module
- Adhere to Singe Responsibility Principle
  - Separate query logic and resource logic
- Adhere to Dependency Inversion Principle
  - Nicely testable

Motivation (cont'd)

- Go FP style
  - Higher-order functions
  - Encapsuling side-effects
- Go async
  - Because resources usually are

Idea

- Loan resource to a borrower
- Let borrower tell when query is complete
- Let caller trigger creation and passing of functions

Step 1

- Separation of concerns
- Using a higher order function for the borrower

Step 1

```
const loan = (resourceConfig, borrower) => {
  try {
    const resource = aquire(resourceConfig);

    // Call Loanee
    return borrower(resource);

  } finally {
    resource && dispose(resource);
  }
};
```

Step 1

```
const borrower = resource => {
  const someData = resource.query('SOME QUERY');
  const processedData = process(someData);
  return processedData;
};

// Execution
const result = loan(resourceConfig, borrower);
```

Step 2

- Promises
- Using an even higher order function for the loaner

Step 2

```
const loan = resourceConfig => borrower => {
  let resource;
  return Promise
    .resolve(aquire(resourceConfig))
    .then(r => {
      resource = r;
      return borrower(resource);
    })
    .then(result => {
      dispose(resource);
      return result;
    }).catch(error => {
      resource && dispose(resource);
      throw error;
    });
};
```

Step 2

```
// Note: `query` is now async and returns a Promise.
loan(resourceConfig)(
  resource => resource
    .query('SOME QUERY')
    .then(someData => process(someData))
    .then(resolve)
)
.then(result => /* ... */);
```

Bonus Level

Parameterizing the borrower

```
const parameterizeBorrower = params => resource =>
  resource
    .query('SOME QUERY')
    .then(someData => process(someData))
    .then(resolve);

const borrower = parameterizeBorrower(params);

loan(resourceConfig)(borrower)
  .then(result => /* ... */);
```

Application

```
const loanDbConnection = dbConfig => dbLoaner =>
  new Promise((resolve, reject) => {
    let connection;
    db.connect(dbConfig)
      .then(c => {
        connection = c;
        return dbLoaner(connection.query);
      })
      .then(result => {
        connection.close();
        return result;
      })
      .catch(e => {
        connection && connection.close();
        throw e;
      });
  });
```

Application

```
const createMetricsFetcher = desiredMetrics => query => {
  const fields = desiredMetrics.join(', ');
  return query(`SELECT ${fields} FROM metrics;`)
    .then(processResults)
    .then(resolve);
});
```

Application

```
const dbConfig = {
  host: 'some-host',
  database: 'some-db',
  username: 'user',
  password: 'secret'
};
const desiredMetrics = ['metricA', 'metricB', 'metricC'];

loanDbConnection(dbConfig)(createMetricsFetcher(desiredMetrics))
  .then(result => /* ... */);
```