

FUN WITH FOLKELOE

JESSE WARDEN

AVA.JS

AUGUST 1ST, 2017

why?

- ▶ Folktale models functional programming concepts in JS
- ▶ less runtime exceptions, or informative as to why
- ▶ avoid null / undefined (they are ambiguous)
- ▶ “correct” means like Algebra, equations are incorrect or correct

when?

- ▶ Using JUST JavaScript, especially Node, but ES6 Browser too
- ▶ You're not using Elm, Dart, PureScript, or ClojureScript
- ▶ TypeScript in library coming

required foreknowledge

- ▶ pure function
- ▶ immutability
- ▶ Lodash, RxJS, Redux, Folktale

FOLKTALE

FUN / FUNCTIONAL PROGRAMMING

- ▶ Maybe
- ▶ Predicates and Checkers vs Validators
- ▶ Union Types
- ▶ Tasks



WHEN DO YOU USE THEM

- ▶ Maybe: a value is there or it isn't
- ▶ Validator: user or server or file data is legit or not, if not why not
- ▶ Union: another useful data type
- ▶ Task: enhancement on Promises



IN PRACTICE

- ▶ Use Maybe instead of null / undefined
- ▶ use Validators for user input, sanitizing server input, & nice error messages
- ▶ If null/undefined is a legit return value, use a Union type
- ▶ if a bunch of things can be returned, use a Union type



IN PRACTICE: TASKS

- ▶ Tasks are enhanced Promises
- ▶ you can cancel them and have cleanup code for them
- ▶ easily convert back to a Promise



МАУБЕ



MAYBE

A data structure that models the presence or absence of a value.

```
Maybe.Just('I matter');
```

```
Maybe.Nothing();
```

```
const users = [  
  {name: "Jesse", age: 38},  
  {name: "Albus", age: 2},  
  undefined,  
  {name: "Cow"}  
];
```

```
const getUserAtIndex = index => users[index];
```

```
getUserAtIndex(1); // { name: 'Albus', age: 2 }
```

```
getUserAtIndex(2); // undefined
```

```
const getUserAtIndex = index =>
  users[index] ? Maybe.Just(users[index])
  : /* otherwise */ Maybe.Nothing();

getUserAtIndex(1);
// folktale:Maybe.Just({ value: { name: "Albus", age: 2 } })
getUserAtIndex(2);
// folktale:Maybe.Nothing({ })
```



```
const _ = require('lodash');
```

```
const users = [  
  {name: "Jesse", age: 38, id: 1},  
  {name: "Albus", age: 2, id: 2},  
  {name: "Cow", age: 1000, id: 4}  
];
```

```
const findUserByName = name => _.find(users,  
  user => user && user.name === name);
```

```
findUserByName('Jesse'); // { name: 'Jesse', age: 38, id: 1 }
```

```
findUserByName('Bruce'); // undefined
```

```
const findUserByName = name =>
{
  const result = _.find(users,
    user => user && user.name === name);
  if(_.isNil(result))
  {
    return Maybe.Nothing();
  }
  else
  {
    return Maybe.Just(result);
  }
};
```

```
findUserByName('Jesse');
```

```
// folktale:Maybe.Just({ value: { name: "Jesse", age: 38, id: 1 } })
```

```
findUserByName('Bruce');
```

```
// folktale:Maybe.Nothing({ })
```




MAYBE
extracting data

```
const users = [  
  {name: "Jesse", age: 38},  
  {name: "Albus", age: 2},  
  undefined,  
  {name: "Cow"}  
];
```

```
const getUserAtIndex = index =>  
  users[index] ? Maybe.Just(users[index])  
  : /* otherwise */ Maybe.Nothing();
```

```
log(getUserAtIndex(1).getOrNull('No value found at index.'));  
// { name: 'Albus', age: 2 }  
log(getUserAtIndex(2).getOrNull('No value found at index.'));  
// No value found at index.
```

```
const users = {  
  "Jesse": {age: 38, skillz: ['powerlifting', 'parkour']},  
  "Albus": {age: 2, skillz: ['being cute', 'being fluffy']},  
  "Cow": {age: 1000, skillz: ['looking cool']}  
};
```

```
_.get(users, 'Albus', 'Unknown property Albus.');
```

// { age: 2, skillz: ['being cute', 'being fluffy'] }

```
_.get(users, 'Brandy', 'Unknown property Brandy.');
```

// Unknown property Brandy.

```
users.Brandy;  
// undefined  
users.Brandy.age;  
// TypeError: Cannot read property 'age' of undefined  
_.get(users, 'Brandy.age', 'Unknown property Brandy.age.');
```

// Unknown property Brandy.age.



MAYBE

PATTERN MATCHING


```
Maybe.Just('dat string tho').matchWith({  
  Just: ({value}) => log("value:", value),  
  Nothing: ({value}) => log("error:", value)  
});  
// value: dat string tho
```

```
Maybe.Nothing().matchWith({  
  Just: ({value}) => log("value:", value),  
  Nothing: _ => log("Nothing, brah.")  
});  
// Nothing, brah.
```

```
const result = Maybe.Just('chicken').matchWith({  
  Just: ({value}) => true,  
  Nothing: _ => false  
});  
log("result:", result);  
// result: true
```




MAYBE

- ▶ I/O
- ▶ reading files
- ▶ HTTP calls
- ▶ parsing data

```
const fs = require('fs');
```

```
const readConfig = () => fs.readFileSync('config.json');
```

```
readConfig();
```

```
// throws ENOENT: no such file or directory, open 'config.json'
```

```
const fs = require('fs');
```

```
const readConfig = () => fs.readFileSync('config.json');
```

```
readConfig();
```

```
// <Buffer 7b 22 70 61 74 ...
```

```
const readConfig = () => {  
  try  
  {  
    const result = JSON.parse(fs.readFileSync('config.json').toString('utf8'));  
    return Maybe.Just(result);  
  }  
  catch(err)  
  {  
    return Maybe.Nothing();  
  }  
};
```

```
log(readConfig().getOrElse({path: 'default/path'}));
```

```
// { path: 'chicken/moo/cow' }
```

```
// { path: 'default/path' }
```



VALIDATOR



PREDICATES

- ▶ a function that returns true or false
- ▶ `_.isString`, `_.some`, `validCreditCard`, etc.
- ▶ easiest to make pure / no side effects

```
const nonEmptyString = o => _.isString(o) && o.length > 0;  
log(nonEmptyString('cow')); // true  
log(nonEmptyString('')); // false  
log(nonEmptyString(123)); // false  
log(nonEmptyString(new Date())); // false
```



```
const legitNumber = o => _.isNumber(o) && _.isNaN(o) === false;
```

```
legitNumber(1); // true
```

```
legitNumber(2.34); // true
```

```
legitNumber(10/0); // true
```

```
legitNumber(Number.Infinity / 0); // false
```

```
const legitDate = o => _.isDate(o) && o.toString() !== 'Invalid Date';
```

```
legitDate(new Date()); // true
```

```
legitDate(Date.now()); // false
```

```
legitDate(new Date('cow')); // false
```




VALIDATORS

- ▶ predicates with an error message
- ▶ `validateCreditCard.errorMessage` = “Not a valid credit card.”

```
const nonEmptyString = o => _.isString(o) && o.length > 0;
const validString = o =>
  nonEmptyString(o) ? Success(o)
  : /* otherwise */ Failure(["Not a valid, must be a non-empty String."]);

validString('cow');
// folktale:Validation.Success({ value: "cow" })

validString('');
// folktale:Validation.Failure({ value: ["Not a valid, must be a non-empty String."] })

validString(123);
// folktale:Validation.Failure({ value: ["Not a valid, must be a non-empty String."] })
```

```
const validator = (errorString, predicate) => o =>  
  predicate(o) ? Success(o)  
  : /* otherwise */ Failure([errorString]);
```

```
const legitString      = token      => _.isString(token) && token.length > 0;
const legitNumber      = number     => _.isNumber(number) && _.isNaN(number) === false;
const legitDate        = date       => _.isDate(date) && date.toString() !== 'Invalid Date';
const legitAccessToken = token      => legitString(_.get(token, 'access_token'));
const legitIssuedAt    = token      => legitNumber(_.get(token, 'issued_at')) && legitDate(new Date(_.get(token, 'issued_at')));
const legitExpiresIn   = token      => legitNumber(_.get(token, 'expires_in'));
const legitClientID    = clientID   => _.isString(clientID) && clientID.length > 0;
const legitClientSecret = clientSecret => _.isString(clientSecret) && clientSecret.length > 0;
const legitURL         = url        => _.isString(url) && url.length > 0 && url.indexOf('http') !== -1;

const stringValidator  = validator('Not a string, or an empty string.', legitString);
const accessTokenValidator = validator('Access Token is invalid.', legitAccessToken);
const expiresInValidator = validator('Expires In is invalid.', legitExpiresIn);
const issuedAtValidator = validator('Issued at is not a valid number or not a valid date.', legitIssuedAt);
const clientIDValidator = validator('Invalid clientID, must be a string and length longer than 0.', legitClientID);
const clientSecretValidator = validator('Invalid clientSecret, must be a string and length longer than 0.', legitClientSecret);
const urlValidator     = validator('Invalid URL; must be a string, not empty, and contain http.', legitURL);
```



```
const token = {  
  access_token: "alsdjflkjasdf12u3o4sdf",  
  issued_at: new Date().valueOf(),  
  expires_in: 2  
};
```

Success()

```
.concat(accessTokenValidator(token))  
.concat(expiresInValidator(token))  
.concat(issuedAtValidator(token));  
// folktale:Validation.Success({ ...
```



NODE VS. ELIXIR

```
const token = {
  access_token: "alsdjflkjasdf12u3o4sdf",
  issued_at: new Date().valueOf(),
  expires_in: '2'
};
log(Success()
.concat(accessTokenValidator(token))
.concat(expiresInValidator(token))
.concat(issuedAtValidator(token)));
// folktale:Validation.Failure({ value: ["Expires In is invalid."] })
```

```
const token = {
  access_token: "alsdjflkjasdf12u3o4sdf",
  issued_at: new Date(),
  expires_in: '2'
};
Success()
.concat(accessTokenValidator(token))
.concat(expiresInValidator(token))
.concat(issuedAtValidator(token));
// ...
// ["Expires In is invalid.",
// "Issued at is not a valid number or not a valid date."
// ...
```



```
const didItValidate = Success()  
  .concat(accessTokenValidator(token))  
  .concat(expiresInValidator(token))  
  .concat(issuedAtValidator(token))  
  .matchWith({  
    Success: _ => true,  
    Failure: _ => false  
  });  
log(didItValidate); // false
```



unionTYPE



SCALAR

- ▶ 1 atomic value
- ▶ “cow”, 1, true



PRODUCT

- ▶ multiple, independent values
- ▶ {name: “Jesse”, age: 38}
- ▶ class Person

- ▶ 1 out of many concepts at any time
- ▶ read a file: Error, Permission Error, file contents
- ▶ access Restify header: it's there, it's there but bad value, no value, here's a default

```
const Maybe = union( 'Maybe',  
{  
  Just(value){ return {value} },  
  Nothing() { return {} }  
});
```

```
const Maybe = union('Maybe',  
{  
  Just(value){ return {value} },  
  Nothing() { return {} }  
});
```

```
log(Maybe.Just('cow'))  
// { value: 'cow' }
```



```
const Maybe = union('Maybe',  
{  
  Just(value){ return value;},  
  Nothing() { return {} }  
});
```



```
const Maybe = union('Maybe',  
{  
  Just(value){ return value },  
  Nothing() { return {} }  
});
```

```
log(Maybe.Just('cow'))
```

```
// { '0': 'c', '1': 'o', '2': 'w' }
```

```
const maybeCow = Maybe.Just('cow');  
const result = maybeCow.matchWith({  
  Just: ({value}) => value,  
  Nothing: _ => false  
});  
log("result:", result);  
// result: cow
```

```
const maybeCow = Maybe.Nothing();  
const result = maybeCow.matchWith({  
  Just: ({value}) => value,  
  Nothing: _ => false  
});  
log("result:", result);  
// result: false
```



```
const HTTPMethod = union('HTTPMethod', {  
  GET() { return {value: 'GET'} },  
  POST() { return {value: 'POST'} },  
  PUT() { return {value: 'PUT'} },  
  DELETE() { return {value: 'DELETE'} },  
  PATCH() { return {value: 'PATCH'} },  
  OPTIONS() { return {value: 'OPTIONS'} }  
});
```

```
HTTPMethod.GET(); // { value: 'GET' }
```

```
HTTPMethod.hasInstance(HTTPMethod.GET()); // true
```

```
HTTPMethod.GET.hasInstance(HTTPMethod.GET()); // true
```

```
HTTPMethod.POST.hasInstance(HTTPMethod.GET()); // false
```

```
const HTTPMethod = union('HTTPMethod', {  
  GET() { return {value: 'GET'} },  
  POST() { return {value: 'POST'} },  
  PUT() { return {value: 'PUT'} },  
  DELETE() { return {value: 'DELETE'} },  
  PATCH() { return {value: 'PATCH'} },  
  OPTIONS() { return {value: 'OPTIONS'} }  
}).derive(Equality);  
  
log(HTTPMethod.GET().equals(HTTPMethod.GET())); // true  
  
const a = HTTPMethod.GET();  
const b = HTTPMethod.GET();  
log(a.equals(b)); // true
```



```
const Attack = union('Attack', {  
  Hit(amount, critical=false) { return {amount, critical}},  
  Miss() { return {value: 'Miss'}}  
}).derive(Equality);  
const { Hit, Miss } = Attack;
```

```
Hit(1, false).hasInstance(Hit(1, false)); // true  
Hit(1, false).hasInstance(Hit(2, true)); // true  
Hit(1, false).equals(Hit(1, false)); // true  
Hit(1, false).equals(Hit(2, true)); // false
```


ETHANISED PROMISE
THUG



TASKS

- ▶ enhanced Promise
- ▶ can cancel it
- ▶ resource clean up option

```
const delay = ms => task(
  (resolver) => {
    const timerId = setTimeout(() => resolver.resolve(ms), ms);
    resolver.cleanup(() => {
      clearTimeout(timerId);
    });
    resolver.onCancelled(() => {
      /* does nothing */
    });
  }
);

// waits 100ms
const result = await delay(100).or(delay(2000)).run().promise();
$ASSERT(result == 100);
```

```
const tokenOk = ({res, obj}) => new Promise((resolve, reject)=>
  checkToken(obj)
    .matchWith({
      Success: _ => resolve({res, obj}),
      Failure: err => reject(new Error(err.value))
    })
);
```

```
const tokenOk = ({res, obj}) => task( resolver =>
  checkToken(obj)
    .matchWith({
      Success: _ => resolver.resolve({res, obj}),
      Failure: err => resolver.reject(new Error(err.value))
    })
);
```

```
const sendLoginResponse = () =>
{
    let connection;
    return oracle.getConnection(oracleDefaultConnection(), oracleDefaultConfig())
    .then( conn =>
    {
        connection = conn;
        return user.login(connection, req.body.EID);
    })
    .then( userObject =>
    {
        oracle.release(connection)
        .then(()=>
        {
            res.send(200, {result: true, data: userObject});
        });
    })
    .catch((err)=>
    {
        oracle.release(connection)
        .then(()=>
        {
            res.send(401, {result: false, error: err.toString()});
        });
    });
});
```



```
const sendLoginResponse = task(
  resolver =>
  {
    let connection;
    resolver.cleanup(connection => oracle.release(connection));
    oracle.getConnection(oracleDefaultConnection(), oracleDefaultConfig())
      .then( conn =>
      {
        connection = conn;
        return user.login(connection, req.body.EID);
      })
      .then( userObject => resolver.resolve(res.send(200, {result: true, data: userObject})))
      .catch( err => resolver.reject(res.send(401, {result: false, error: err.toString()}))))
  }
);
```




CONCLUSIONS



МАХБЕ



MAYBE: WHEN DO YOU USE THEM

- ▶ Maybe: a value is there or it isn't
- ▶ instead of returning null or undefined, use Maybe
- ▶ Side effect? I/O? Maybe it'll work... so use a Maybe



VALIDATOR



VALIDATOR: WHEN DO YOU USE THEM

- ▶ Validator: validating user, or server, inputs
- ▶ data is legit or not, if not why not
- ▶ nice error messages



unionTYPE



UNION: WHEN DO YOU USE THEM?

- ▶ Union: another useful data type
- ▶ If null/undefined is a legit return value, use a Union type
- ▶ if a bunch of things can be returned, use a Union type
- ▶ modeling your data (like Objects, Classes)

ETHANISED PROMISE
THQ



TASK: WHEN DO YOU USE THEM

- ▶ Task: enhancement on Promises
- ▶ you can cancel them and have cleanup code for them
- ▶ easily convert back to a Promise

FOLKTALE

FUN / FUNCTIONAL PROGRAMMING

the end

- ▶ Folktale API Docs <http://folktale.origamitower.com/api/v2.0.0/en/folktale.html> (click on API's to get good documentation)
- ▶ Folktale Github <https://github.com/origamitower/folktale>
- ▶ (me) Jesse Warden | @jesterxl | youtube.com/user/jesterxl | jesse@jessewarden.com
- ▶ Functional Programming people to follow on Twitter
 - ▶ @bodil (lots of pizza posts)
 - ▶ @doppioslash
 - ▶ @robotlolita (works on Folktale, lots of Japanese idol posts)
 - ▶ @drboolean (I understand 1% of his tweets)
 - ▶ @swannodette (Clojure luva)
- ▶ Great beginner video on Elm, helps with basic functional concepts <https://www.youtube.com/watch?v=D740qUZVcr4>
- ▶ wonderful weekend exercise, get this working in your project <https://github.com/bodil/eslint-config-cleanjs>