

Fullstack Academy of Code · Gabriel Lebec

Promises

Using, generating, and
understanding deferral-style
Javascript promises

What is a promise, anyway?

“A *promise* represents the eventual result of an asynchronous operation.”

–The *Promises/A+* Spec

“The point of promises is to give us back functional composition and error bubbling in the async world.”

–Domenic Denicola, [“You’re Missing the Point of Promises”](#)

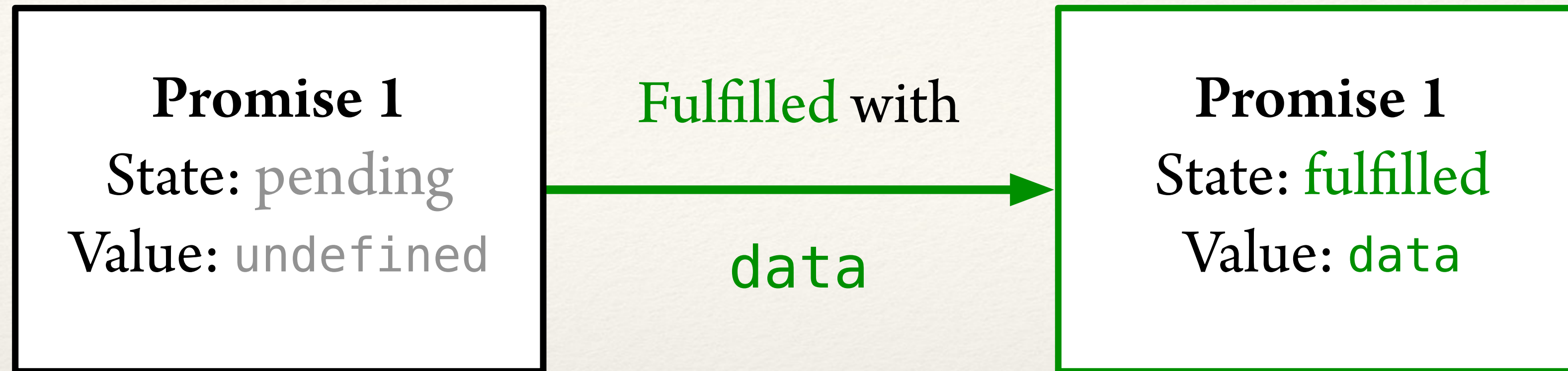
- ❖ Pass into functions
- ❖ Assign to objects
- ❖ Export to modules
- ❖ Return values... from a callback!?! (Yes.)
- ❖ Write just one error handler for a long chain of promises

Promises are Objects

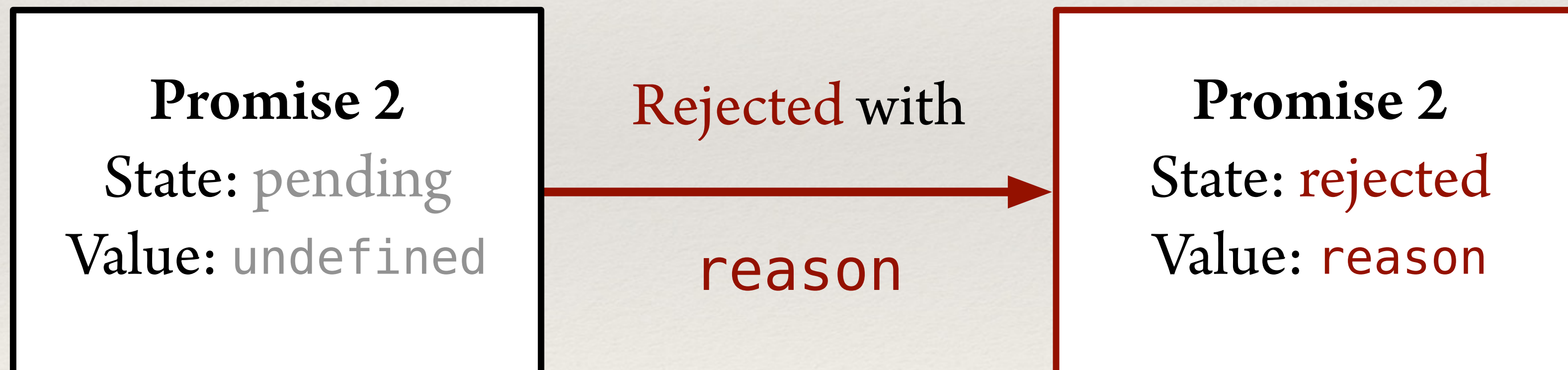
state (pending, fulfilled, or rejected) } (via closure)
value (data or a reason)
.then() (property)

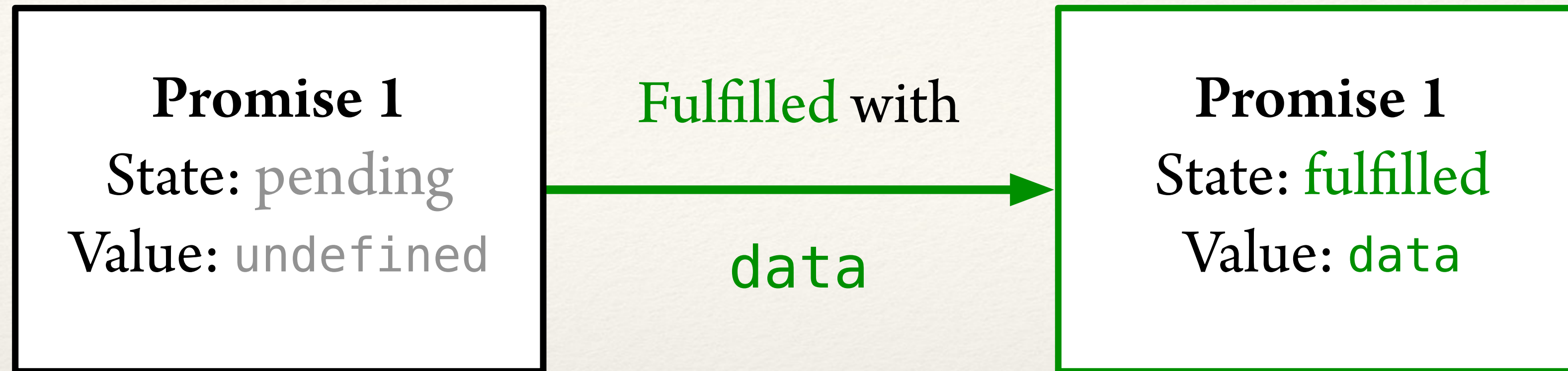
Promises are Objects

- ❖ They have one (very important) property: the function `.then()`
- ❖ They encapsulate (usually via closure) two more crucial variables:
 - ❖ A current *state*, which starts as Pending and ends up as one of:
 - ❖ Fulfilled
 - ❖ Rejected
 - ❖ An eventual *value*, which ends up being one of the following:
 - ❖ fulfilled *data*
 - ❖ a rejection *reason*

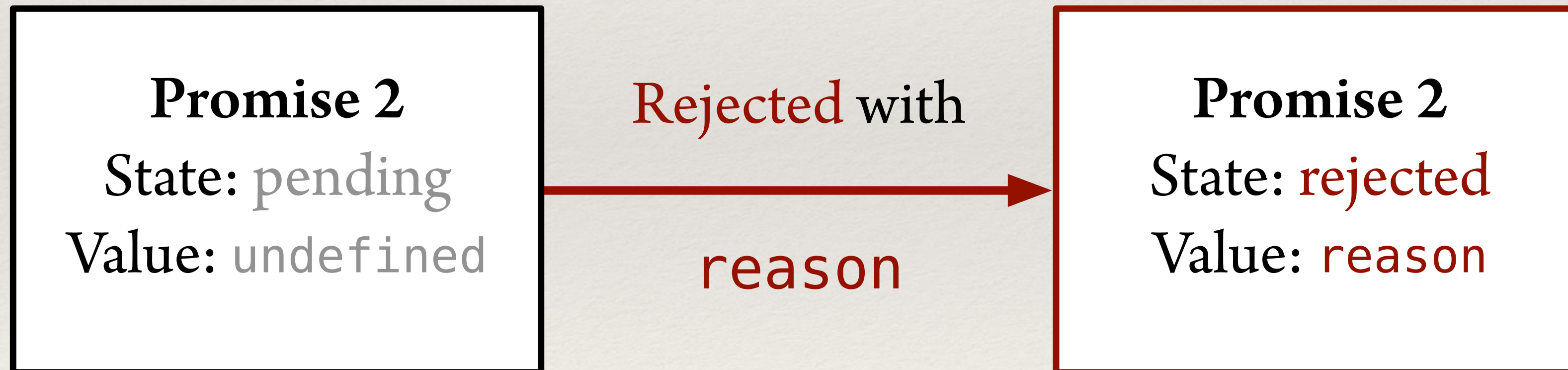


Promises can only change state while pending





`.then(successFunction, errorFunction)`



What does this solve?

// Basic async callback pattern.
// asyncFetchUser asks a server for some data.
// Internally, it gets a response: { name: 'Kim' }.
// That response is then passed to the receiving callback.

```
asyncFetchUser( 123, function received( response ) {  
  console.log( response.name ); // output: Kim  
});
```


// Callback Hell

```
var userID = 'a72jd3720h';
getUserData( userID, function got( userData ) {
  getMessage( userData.messageIDs[0], function got( message ) {
    getComments( message, function got( comments ) {
      console.log( comments[0] );
    });
  });
});
```


// Callback Hell... with error handling, for extra hellishness

```
var userID = 'a72jd3720h';
getUserData( userID, function got( userData, err ) {
  if (err) console.log('user fetch err: ', err);
  else getMessage( userData.messageIDs[0], function got( message, err ) {
    if (err) console.log('message fetch err: ', err);
    else getComments( message, function got( comments, err ) {
      if (err) console.log('comment fetch err: ', err);
      else console.log( comments[0] );
    });
  });
});
```


// Sneak preview: .then chaining and .catch

```
promiseForThing
  .then( function thingSuccess (thing) {
    // run some code
    return promiseForNewStuff;
  })
  .then(function stuffSuccess (newStuff) {
    // run some code
    return promiseForMoreData;
  })
  .then(function moreSuccess (moreData) {
    // etc.
  })
  .catch(function oops (error) {
    // handle the error
  });
```


// This will not work!

```
var person = asyncGetGroup( 123, function got( group ) {  
    return group.users[0];  
});
```



```
// This also will not work
```

```
var person;  
asyncGetGroup( 123, function got( group ) {  
    person = group.users[0];  
});
```

```
// somewhere else
```

```
var headline = person.name; // might be undefined!
```


// Fantasy solution

```
var containerA = new Container();  
asyncGetData( function got( asyncData ) {  
    containerA.save( asyncData ); // once async completes  
});
```

// ...somewhere else...

```
containerA.whenSaved( function use( savedData ) {  
    console.log( savedData ); // once containerA.save() happens  
});
```


// Real version with deferral-style promise

```
myDeferral = $q.defer();
asyncGetData( function got( asyncData ) {
    myDeferral.resolve( asyncData );
});
var myPromise = myDeferral.promise;
```

// ...somewhere else...

```
myPromise.then( function use( resolvedData ) {
    console.log( resolvedData );
});
```

Fragmented Landscape

- ❖ Multiple proposed standards · [CommonJS](#)
- ❖ One leading standard · [Promises/A+](#)
- ❖ Upcoming native [ES6 promises](#) (already working in some browsers)
- ❖ Two different approaches for generating new promises:
 - ❖ CommonJS-style **deferrals** (one extra entity)
 - ❖ ES6-style **constructors** (simplified)
- ❖ jQuery gurus beware! \$.Deferred differs from current standards and is considered flawed. See [Kris Kowal's guide](#).

Deferral-style

```
var myDeferral = $q.defer();
someAsyncCall( function (data, err) {
    if (err) myDeferral.reject( err );
    else myDeferral.resolve( data );
});
var myPromise = myDeferral.promise;

// elsewhere
myPromise.then( mySuccessHandler, myErrorHandler );
```

Constructor-style

```
var myPromise = new Promise( function (resolve, reject) {  
    someAsyncCall( function (data, err) {  
        if (err) reject( err );  
        else resolve( data );  
    });  
});
```

```
// elsewhere
```

```
myPromise.then( mySuccessHandler, myErrorHandler );
```


So Why Deferrals ... ?

AngularJS: \$q (has both styles, but...)

Server-side: Q (only deferrals)

`.notify()`

Advanced: makes the resolver portable.

Meta: if you get deferrals, you can get ES6-style.

A deferral controls its promise's state. Promises attach callbacks to an eventual value.

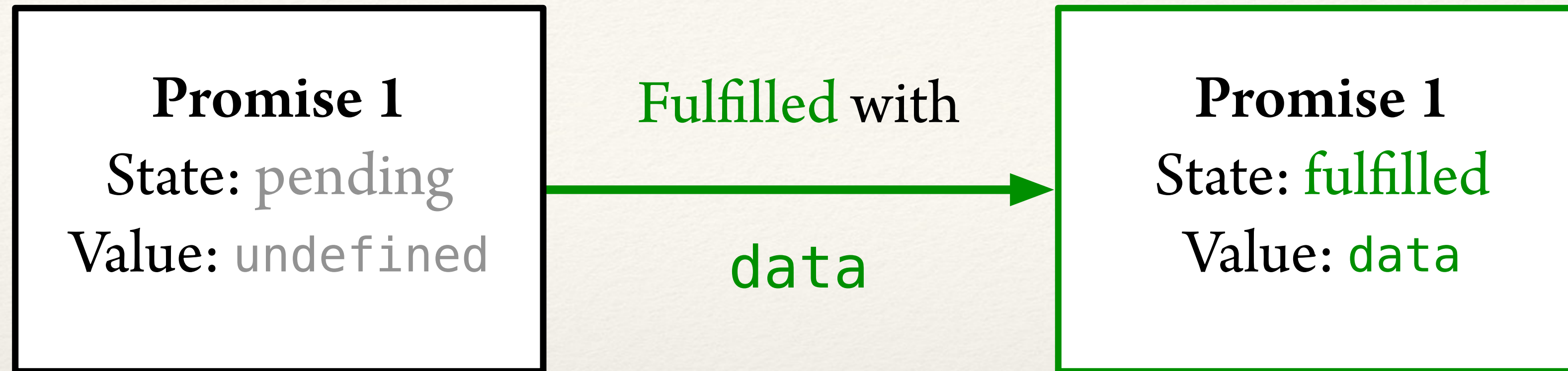
```
var myDeferral = $q.defer();  
  
// async code that produces myData...  
myDeferral.resolve( myData );  
  
// ...or instead produces myReason...  
myDeferral.reject( myReason );  
  
// ...and meanwhile produces myInfo  
myDeferral.notify( myInfo );  
  
return myDeferral.promise
```

promiseForData.then(

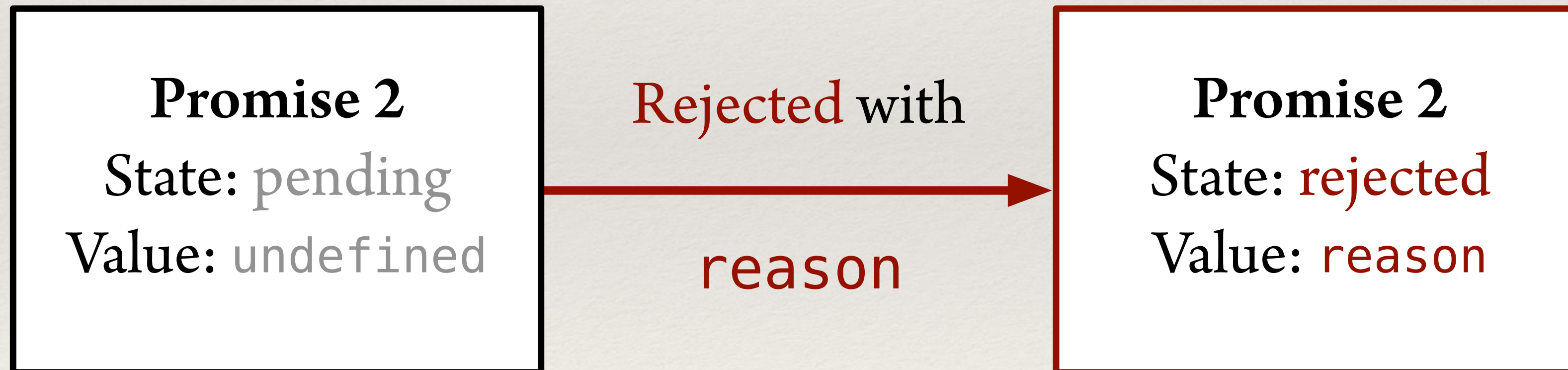
```
function handleSuccess( someData ) {  
    // do stuff with someData  
},  
  
function handleError( someReason ) {  
    // do stuff with someReason  
},  
  
function handleUpdate( someInfo ) {  
    // do stuff with someInfo  
}  
  
);
```

The async code can exist in a service ...

...and return a promise for myData to a controller!



`.then(successFunction, errorFunction)`



.then() always returns a new promise.

```
promiseB = promiseA.then( successHandler, errorHandler );
```


.then() always returns a new promise.

```
promiseB = promiseA.then( successHandler, errorHandler );
```

```
    promiseForThing  
        .then( doStuff )  
    —————→ .then( doOtherStuff )  
    —————→ .then( doMoreStuff )  
    —————→ .catch( handleError );
```

This is why we can chain .then

.then() always returns a new promise.

```
promiseB = promiseA.then( successHandler, errorHandler );
```

```
promiseForThing  
  .then( doStuff )  
  .then( doOtherStuff )  
  .then( doMoreStuff )  
  .catch( handleErr );
```

```
return promiseForThing  
  .then( function thingSuccess (thing) {  
    // run some code  
    return someOtherThing;  
  })
```

And why we can return from a handler

.then() always returns a new promise.

```
promiseB = promiseA.then( successHandler, errorHandler );
```

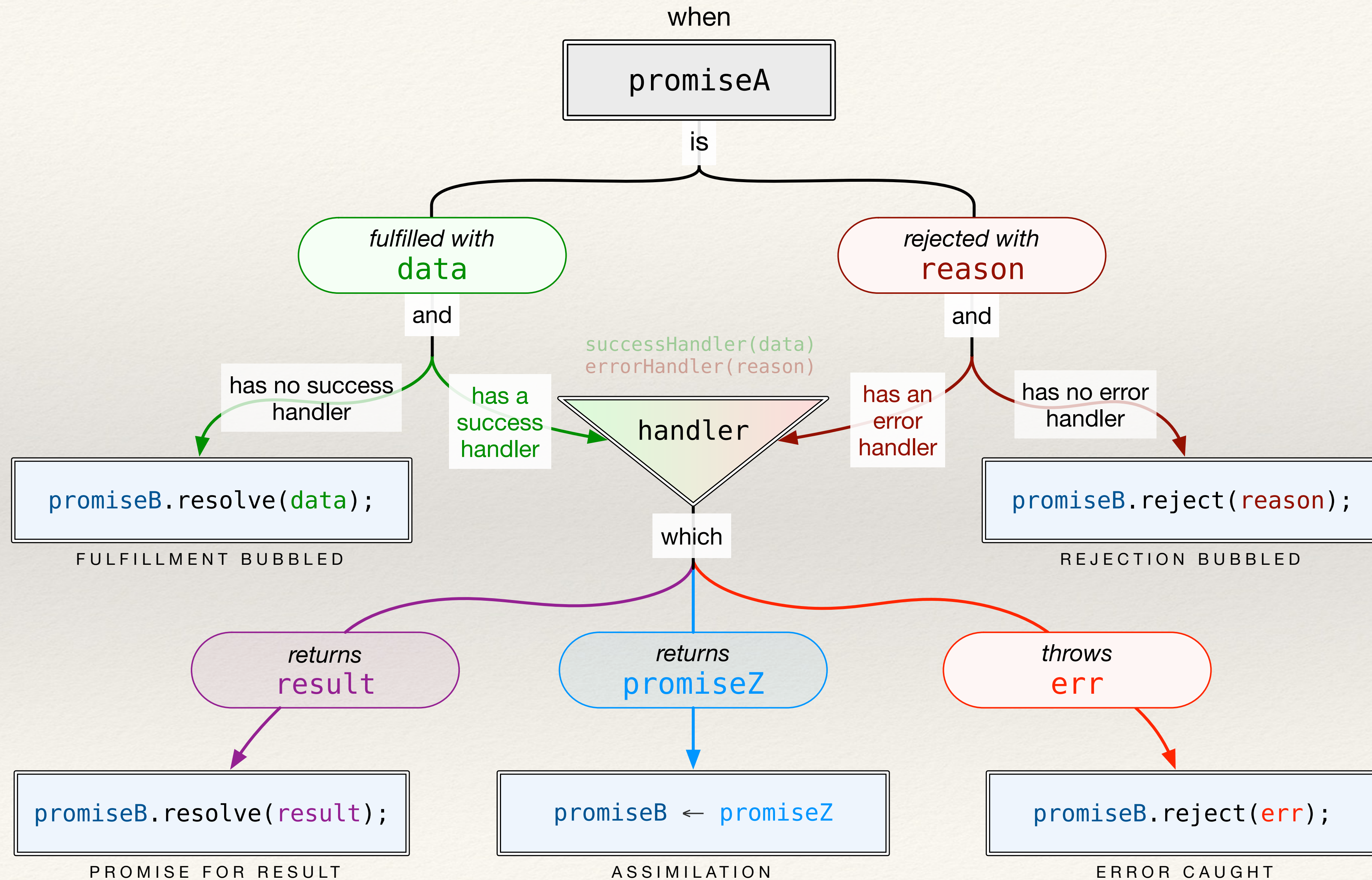
```
promiseForThing  
  .then( doStuff )  
  .then( doOtherStuff )  
  .then( doMoreStuff )  
  .catch( handleError );
```

```
return promiseForThing  
  .then( function thingSuccess (thing) {  
    // run some code  
    return someOtherThing;  
  })
```

So what actually happens to promiseB?

.then() always returns a new promise. What happens to that promise?

```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```

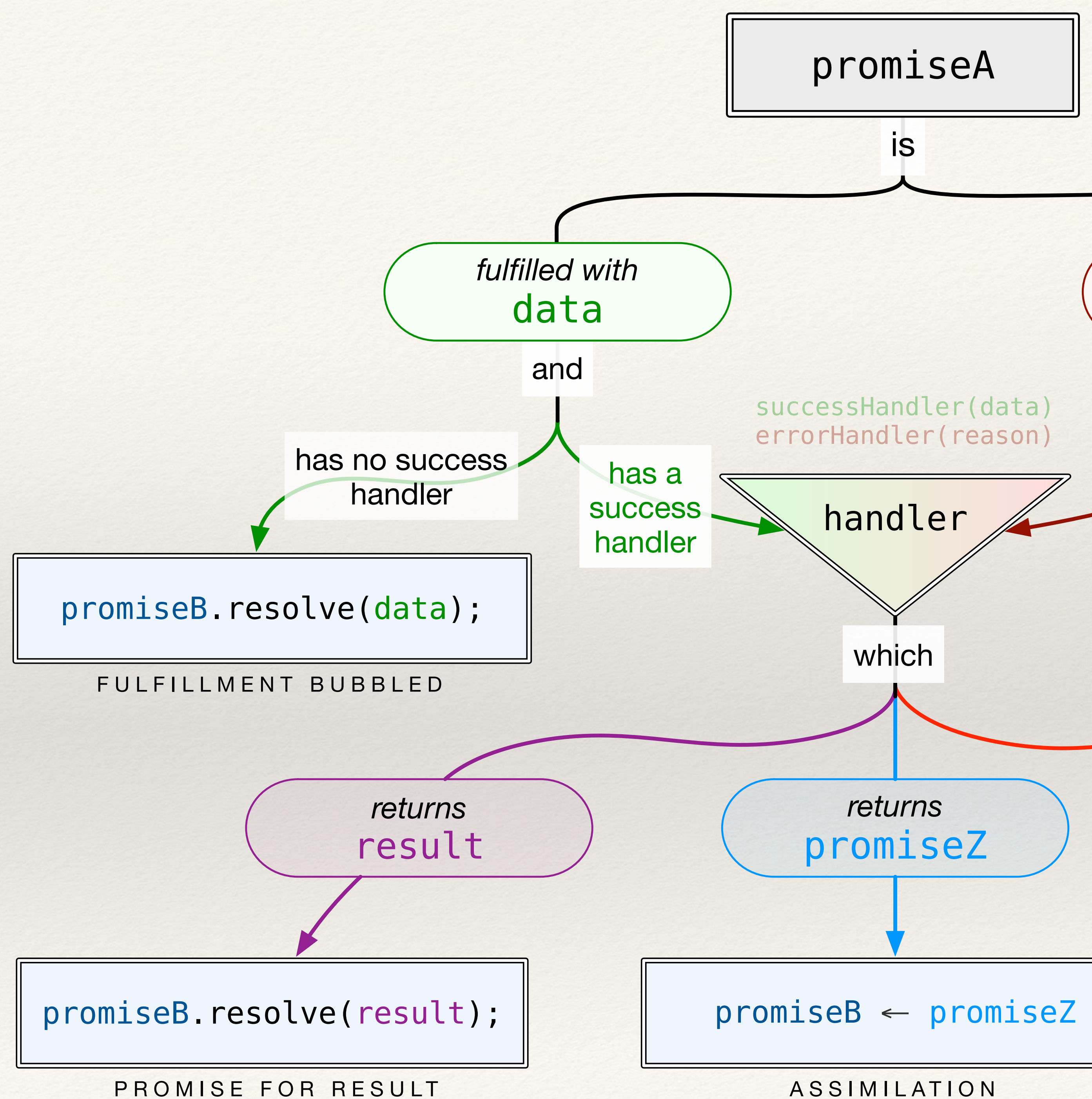



```
// promise0 is fulfilled with  
'Hey there.'
```

```
promise0  
  .then() // -> p1  
  .then() // -> p2  
  .then() // -> p3  
  .then() // -> p4  
  .then() // -> p5  
  .then(console.log);
```

Fulfillment bubbled down to first available
success handler:

Console log reads “Hey there.”

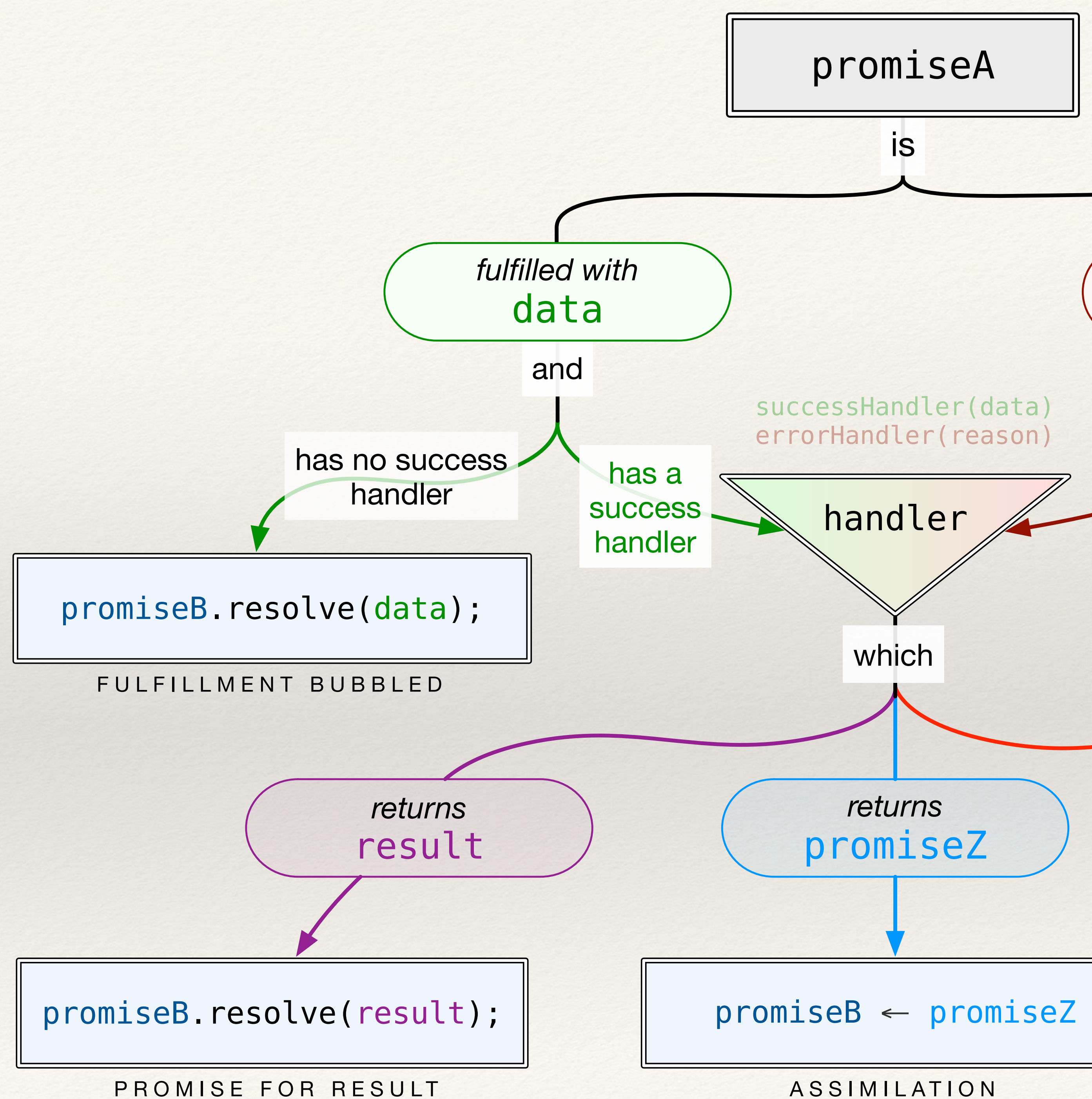


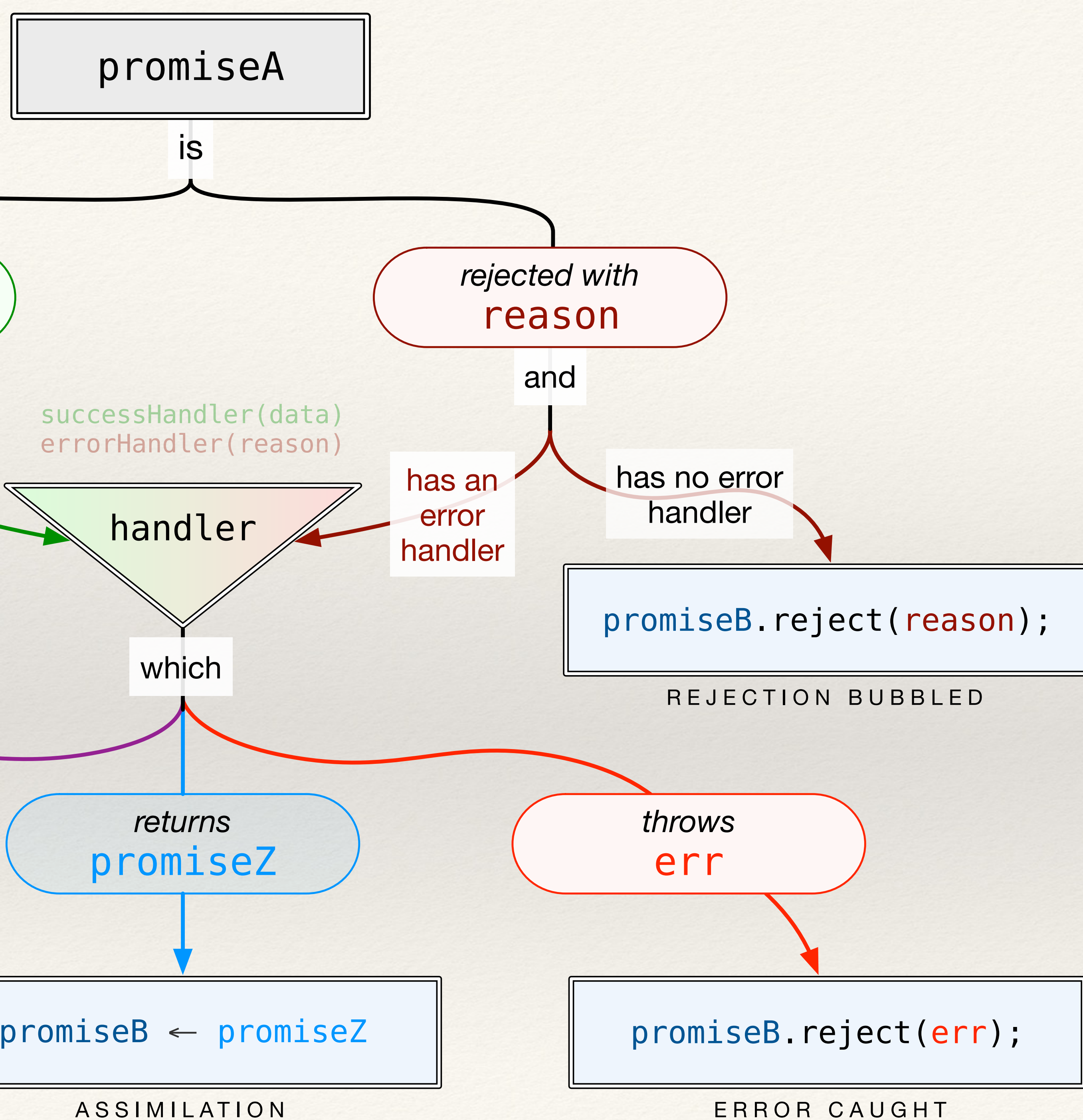

```
// promise0 is fulfilled with  
'Hey there.'
```

```
promise0  
  .then(null, warnUser) // -> p1  
  .then() // -> p2  
  .then() // -> p3 etc.  
  .then(null, null, updateDom)  
  .then()  
  .then(console.log);
```

Same thing! Fulfillment bubbles down to first available success handler.

Console log reads “Hey there.”

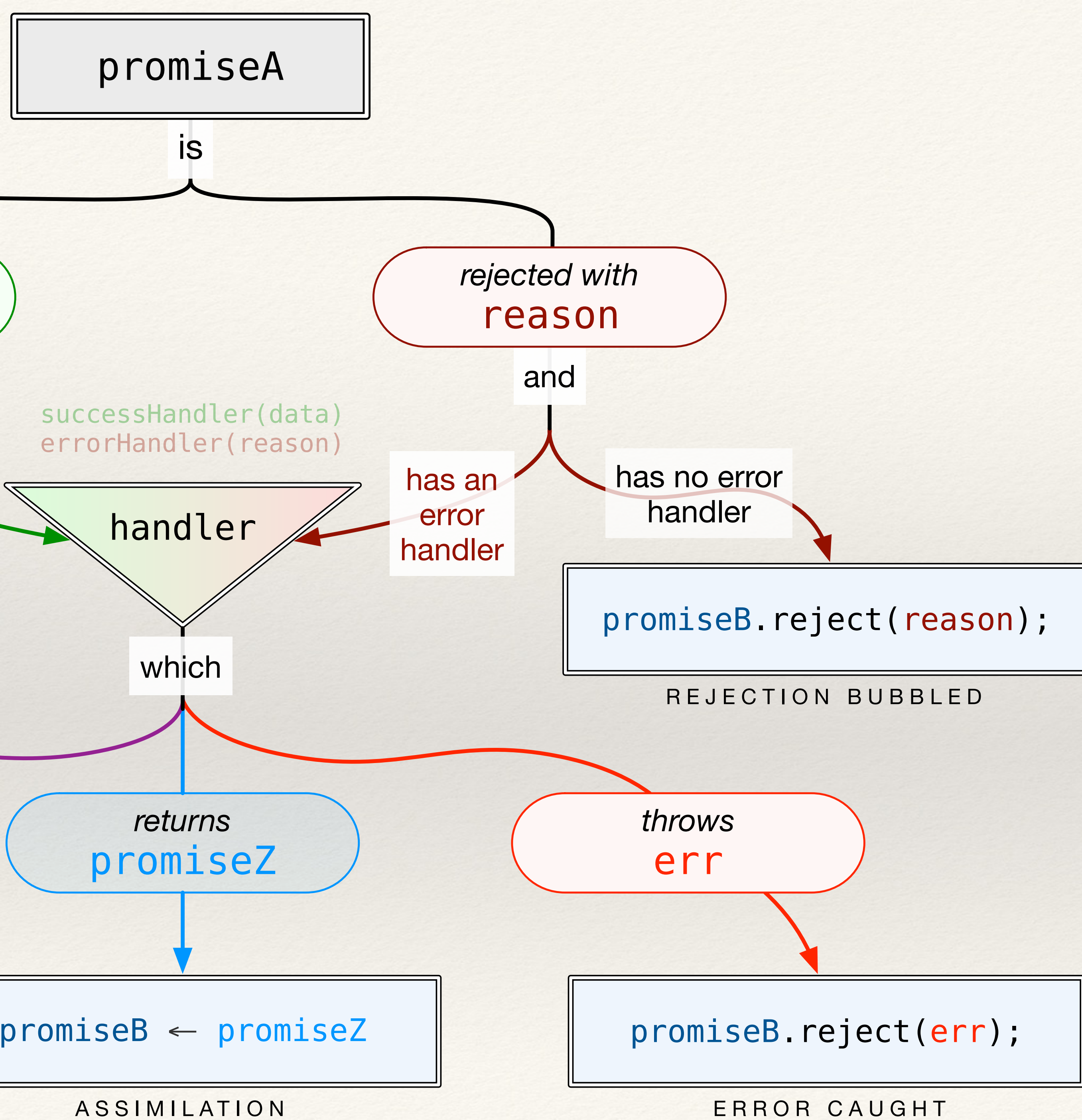




```
function logNormal (input) {  
  console.log(input);  
}  
function logYell (input) {  
  console.log(input+'!');  
}  
  
// promise0 rejected with 'Sorry'  
  
promise0  
  .then() // -> p1  
  .then() // -> p2 and so on  
  .then()  
  .then(null, logNormal);
```

Rejection bubbles down to the first available error handler.

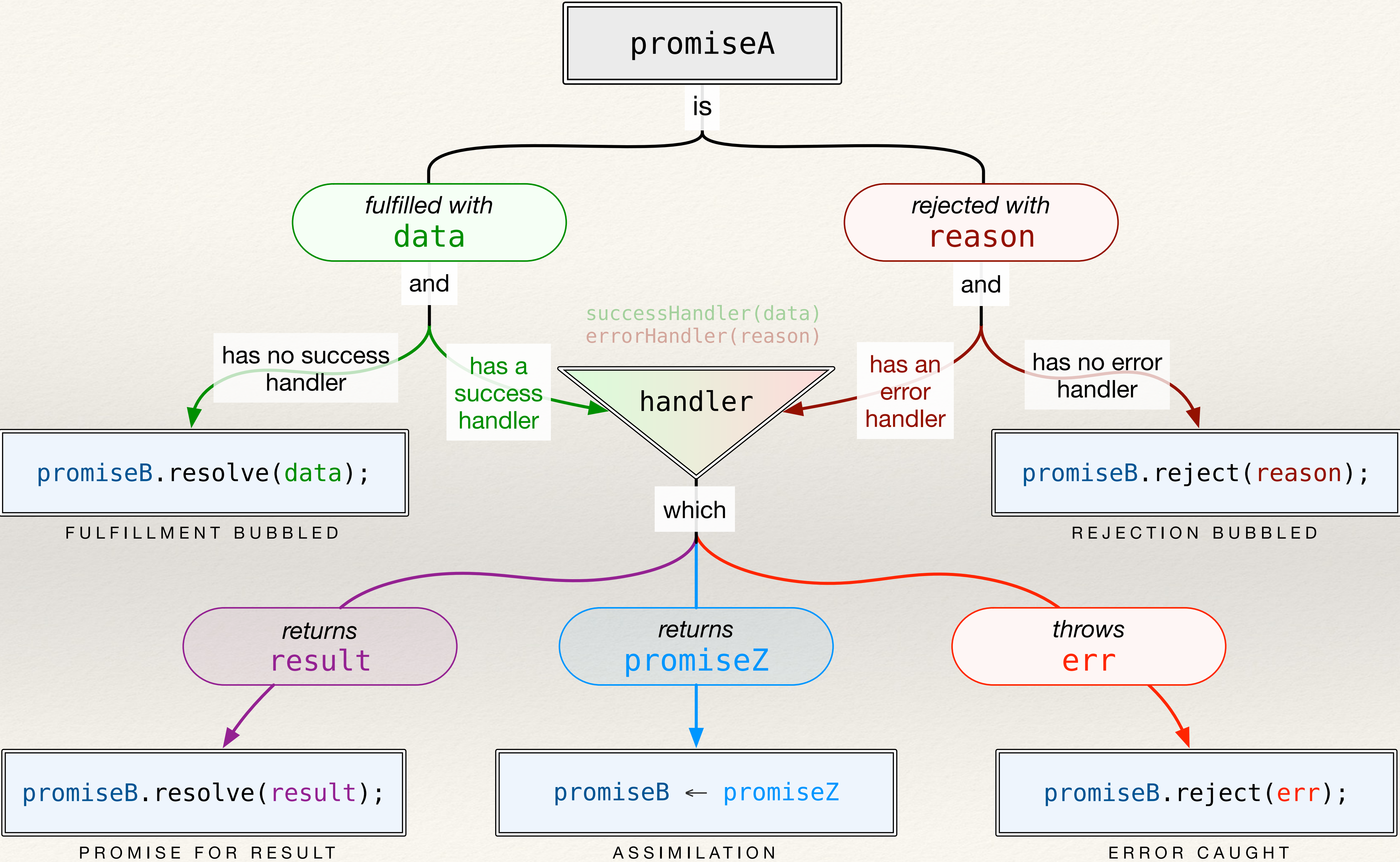
Console log is “Sorry”.

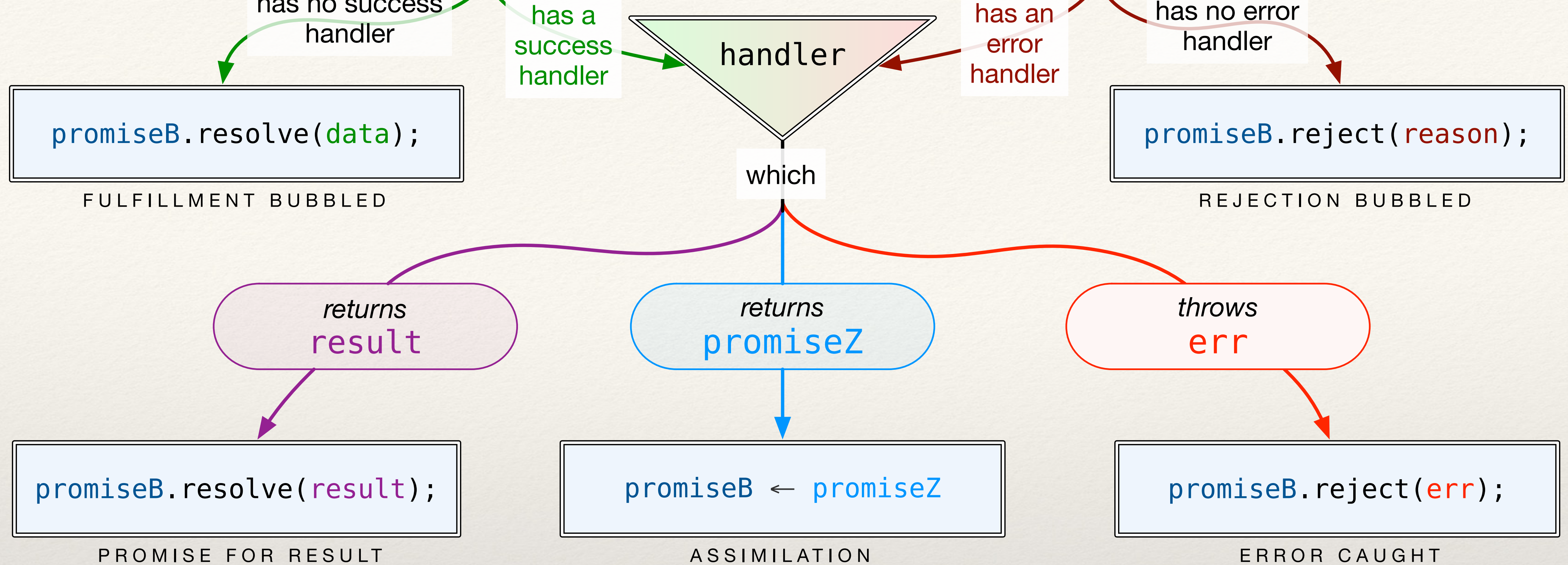


```
function logNormal (input) {  
  console.log(input);  
}  
function logYell (input) {  
  console.log(input+'!');  
}  
  
// promise0 rejected with 'Sorry'  
  
promise0  
  .then(logNormal)  
  .then(null, null, logNormal)  
  .then()  
  .then(null, logYell);
```

Again, rejection bubbles down to the first available **error** handler.

Console log is “Sorry!”





Live examples ...

MEAN stack:

- ❖ Some Angular methods already generate promises for you (\$http, \$timeout)

```
var promiseForUser = $http.get('users/1').then(function(resp) {return resp.data});
```

- ❖ Dependency-inject the \$q service if you need to generate your own promise:

```
angular.module('myApp').service('myService', function ($q) { ... });
```

- ❖ Install the Q library and require q in modules to generate promises on the server:

```
npm install q --save
```

```
var Q = require('q');
```

- ❖ Use .exec() to turn Mongoose queries into promises:

```
var promiseForCentenarians = User.find({age: 100}).exec();
```



```
// array of API calls to make
var apiCalls = [
    '/api1/',
    '/api2/',
    '/api3/'
];
// map each url to a promise for its call result
apiCallPromises = apiCalls.map( function makeCall (url) {
    return $http.get(url).then( function got (response) {
        return response.data;
    });
});
// make a promise for an array of results once all arrive:
var thingsPromise = $q.all( apiCallPromises );
// use it:
thingsPromise.then( function got (results) {
    results.forEach( function print (result) {
        console.log(result);
    });
});
```


External Resources for Further Reading

- [**AngularJS documentation for \\$q**](#)
- [Kris Kowal & Domenic Denicola: Q](#) (the library \$q mimics; great examples & resources)
- [The Promises/A+ Standard](#) (with use patterns and an example implementation)
- [HTML5 Rocks: Promises](#) (deep walkthrough with use patterns)
- [Xebia: Promises and Design Patterns in AngularJS](#)
- [AngularJS Corner: Using promises and \\$q to handle asynchronous calls](#)
- [DailyJS: Javascript Promises in Wicked Detail](#) (build an ES6-style implementation)
- [MDN: ES6 Promises](#) (upcoming native functions)
- [Promise Nuggets](#) (use patterns)