rangle.io

The Web Inverted

www.rangle.io

@rangleio

150 John St., Suite 501
Toronto, ON Canada
M5V 3E3

1-844-GO-RANGL

PROMISE-BASED ARCHITECTURE

Yuri Takhteyev

CTO, rangle.io

 **@Qaramazov**

 **yuri**

rangle.io
The Web Inverted

http://yto.io/xpromise2

Why are Promises Awesome?

The answer isn’t always obvious even to die-hard fans!

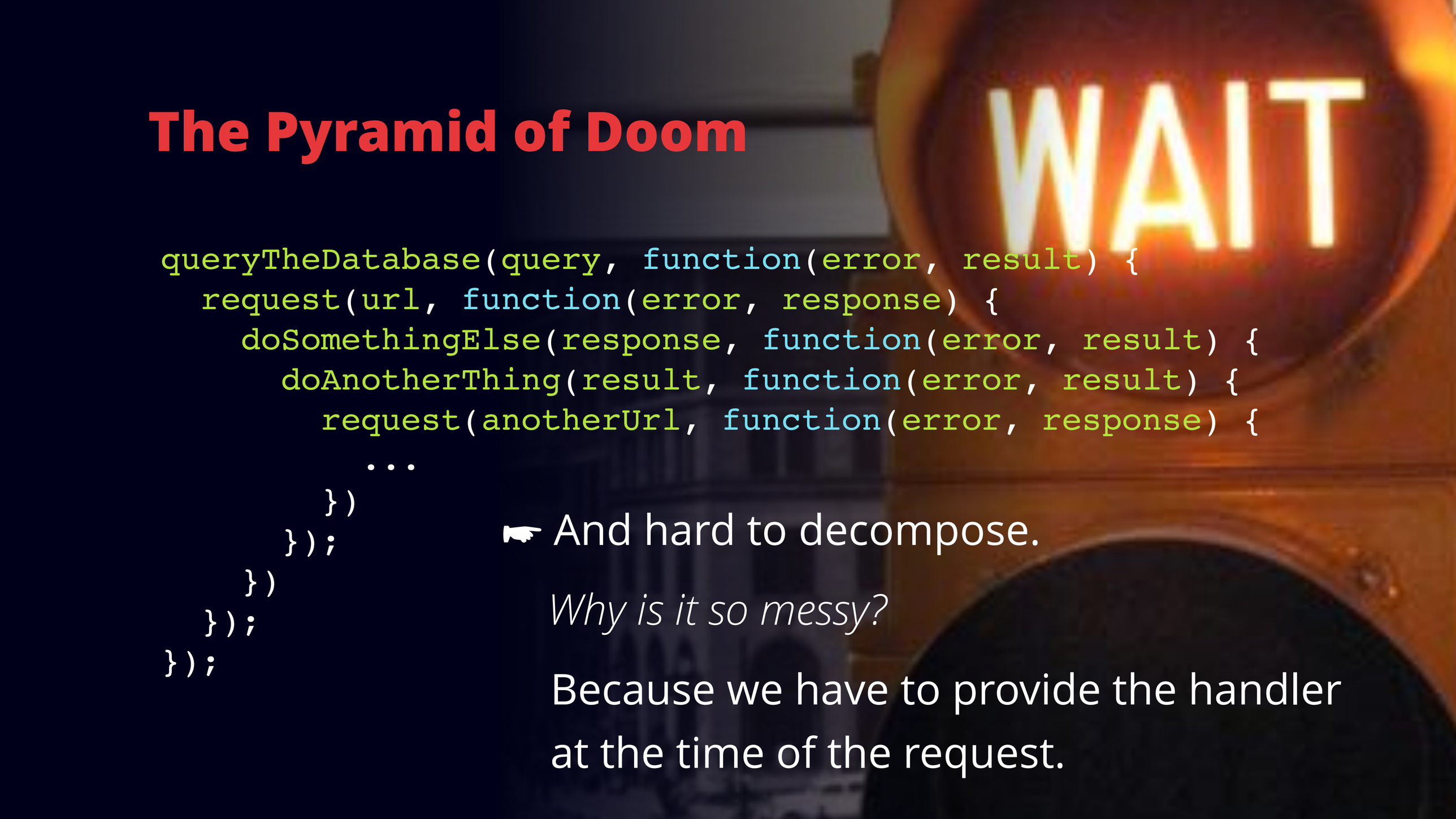
Asynchronicity

- JavaScript can’t wait
- All solutions are variations on callbacks.

Node-Style Callbacks


```
request(url, function(error, response) {
  // handle success or error.
});
doSomethingElse();
```

The Pyramid of Doom



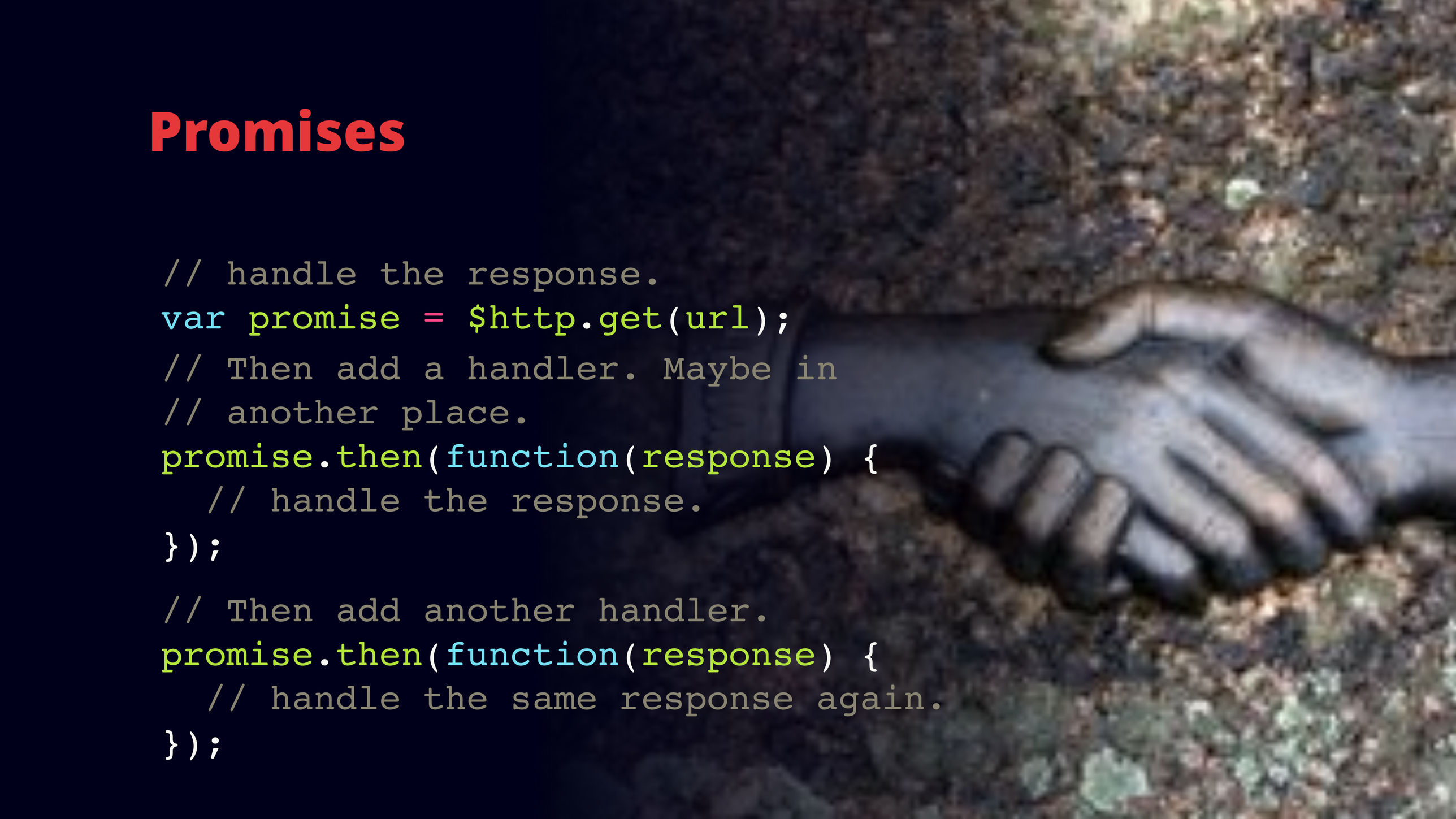

```
queryTheDatabase(query, function(error, result) {
  request(url, function(error, response) {
    doSomethingElse(response, function(error, result) {
      doAnotherThing(result, function(error, result) {
        request(anotherUrl, function(error, response) {
          ...
        })
      });
    })
  });
});
```

☛ And hard to decompose.

Why is it so messy?

Because we have to provide the handler at the time of the request.

Promises



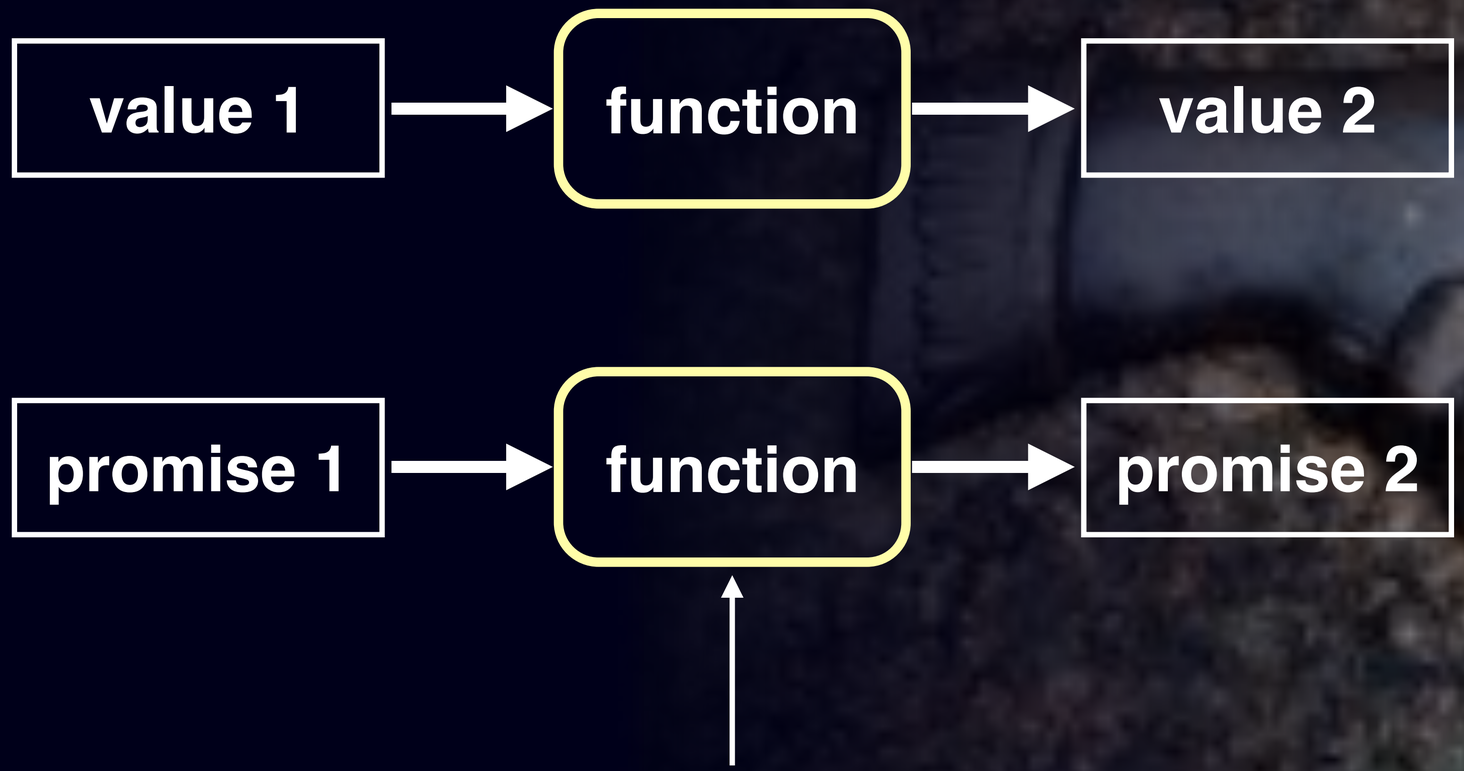
```
// handle the response.
var promise = $http.get(url);
// Then add a handler. Maybe in
// another place.
promise.then(function(response) {
  // handle the response.
});

// Then add another handler.
promise.then(function(response) {
  // handle the same response again.
});
```

Promises as Wrappers

- Pass around promises instead of passing around values.
- In other words, write functions that receive promises and return promises.

General Model



Groking Promises

Promises can be unintuitive, until you grok
some basic axioms.

Chained Promises Can Get Ugly

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks);
  })
  .then(function(tasks) {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, function(error) {
    $log.error(error);
  });
```

Slides are at **http://yto.io/xpromise2**

Let’s Figure It Out

```
var responsePromise = $http.get('http://example.com/api/v1/tasks');

var tasksPromise = responsePromise.then(function(response) {
  return response.data;
});

var filteredTasksPromise = tasksPromise.then(function(tasks) {
  return filterTasksAsynchronously(tasks);
});

var vmUpdatePromise = filteredTasksPromise.then(function(tasks) {
  $log.info(tasks);
  vm.tasks = tasks;
})

var errorHandlerPromise = vmUpdatePromise.then(null, function(error) {
  $log.error(error);
});
```

Axiom: .then() Returns a Promise. Always.

```
var dataPromise = getDataAsync(query);

var transformedDataPromise = dataPromise
  .then(function (results) {
    return transformData(results);
  });
```

`transformedDataPromise` will be a promise

regardless of what ***`transformedData`*** *does.*

When the callback…	then .then() returns a promise that…
returns a regular value	resolves to that value.
returns a promise	resolves to the same value
throws an exception	rejects with the exception

Catching Rejections

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data; 🚩
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks);
  })
  .then(function(tasks) {
    $log.info(tasks);
    vm.tasks = tasks;
  }, function(error) {
    $log.error(error); ☮
  });
```

Catching Rejections

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks); 🚩
  })
  .then(function(tasks) {
    $log.info(tasks);
    vm.tasks = tasks;
  }, function(error) {
    $log.error(error); ☮
  });
```

Catching Rejections

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks);
  })
  .then(function(tasks) {
    $log.info(tasks); 🚩
    vm.tasks = tasks;
  }, function(error) {
    $log.error(error);
  });
```

Nobody hears you scream…

Catching Rejections

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks);
  })
  .then(function(tasks) {
    $log.info(tasks); 🚩
    vm.tasks = tasks;
  }, function(error) {
    $log.error(error);
  });
```

Nobody hears you scream…

Better

```
$http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  })
  .then(function(tasks) {
    return filterTasksAsynchronously(tasks);
  })
  .then(function(tasks) {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(null, function(error) {
    $log.error(error);
  });
```

Or Pass the Buck

```
return $http.get('http://example.com/api/v1/tasks')
  .then(function(response) {
    return response.data;
  });
```

Making Promises

You rarely need to do this. But it’s best to know how.

Avoid $q.defer. Denodeify.

```
var getFooPromise = denodeify(getFooWithCallbacks);

return getFooPromise()
  .then(function(result) {
    // do something with the result.
  });
```

https://github.com/rangle/angular-promisify

Trivial Promises

- **$q.when(x):** Returns a promise that resolves to x.
- **$q.reject(e):** Returns a promise that rejects with e.

Do’s and Don’ts

Things to do and things not to do.

Promise Chains Considered Harmful

```
function getTasks() {
  return $http.get('http://example.com/api/v1/tasks')
    .then(function(response) {
      return response.data;
    });
}

function getMyTasks() {
  return getTasks()
    .then(function(tasks) {
      return filterTasks(tasks, {
        owner: user.username
      });
    });
}
```

Stay Consistent

- **A function that returns a promise should *always* return a promise and should never throw.**
  - Return $q.reject(error) instead of throwing.
  - Wrap non-promise return values in $q.when().

When in Doubt, Return a Promise

- If you are not sure whether your operation will eventually be synchronous or not, assume asynchronous and return a promise.

Pass the Buck, But Don’t Drop it

- **A function that receives a promise, should normally return a promise, only handling those errors that it is equipped to handle.** Your caller can figure out how to handle rejections.
- **If you do not return a promise, though, then you gotta handle the errors.**

Avoid Optimistic Code

- **Avoid code that assumes that something has already happen.**
    - Instead, ask for a promise, return a promise.
- **If you do write such code, name your functions to communicate this.**

Neat Things We Can Do with Promises

Promises can be unintuitive, until you grok some basic axioms.

Promise Caching

```
var tasksPromise;
function getTasks() {
  taskPromise = taskPromise || getTasksFromTheServer();
  return taskPromise;
}
```

Prefetching

```
var tasksPromise = getTasksFromTheServer();
function getTasks() {
 return taskPromise;
}
```

Postponing Requests

```
function get(path) {
  return user.waitForAuthentication()
    .then(function() {
      return $http.get(path);
    })
    .then(function(response) {
      return response.data;
    });
};
```

Functional Composition with Promises

```
var R = require('ramda');
var getExifDate = R.pPipe(
  getExifData, // returns a promise
  R.prop('exif'), // a synchronous function
  R.prop('DateTimeOriginal') // same here
);

getExifDate('/path/to/file.jpg')
  .then(function(date) {
    // Do something with the date.
  })
  .then(null, $log.error);
```

Koa-Style Generator Magic

```
app.use(function* () {
  var data  = yield getPromiseForData();
  // Proceed to use data
  console.log(data.items);
});
```

THANK YOU!

Yuri Takhteyev
CTO, rangle.io

@qaramazov

yuri

rangle.io
The Web Inverted

Image Credits

by jbrazito

by Quinn Dombrowski

Promises vs Events

Occasionally promises are not the answer.

Promises	Events (aka “Publish – Subscribe”)
Things that happen ONCE	Things that happen MANY TIMES
Same treatment for past and future	Only care about the future*
Easily matched with requests	Detached from requests