

Applying JavaScript Promises

Asynchronous Honesty

Adam Ranfelt
MidwestJS 2015

github.com/adamRenny/MidwestJS2015

Presenter Notes should be available up on github MidwestJS2015

Bare with my honestly terrible puns throughout :-)

Applying JavaScript Promises



Asynchronous Honesty

Adam Ranfelt
MidwestJS 2015

github.com/adamRenny/MidwestJS2015

Presenter Notes should be available up on github MidwestJS2015

Bare with my honestly terrible puns throughout :-)

Promises

Promises Solve:

- Asynchronous Problems
- Serialize Asynchronity
- Make more readable

Becoming a part of our day-to-day

Popular in today's JavaScript

Now in:

- + ES6/2015/ESNext
- + Promises/A+
- + Other specs

Callbacks

```
var button = document.getElementById('submitBtn');  
  
button.addEventListener('click', function(event) {  
    // Do some work  
}, false);
```

In JS, everyone knows callbacks

Callbacks are a powerful tool and main attributer to the success of promises

If given ample space & time, callbacks run rampant and can produce some of the most difficult to read code

Callback Hell

Code isn't important

Maintenance reveals the truth

Callback hell is the manifestation of overly nested callbacks

Notice the pyramid (gives it away)

Callback Hell

```
$(document).ready(function() {  
  var button = $('#submitBtn');  
  
  $button.on('click', function(event) {  
    $.ajax({  
      url: 'https://someResource.com/auth',  
      success: function(data) {  
        var $successMessage = $('<span />').text(data.message);  
        $('body').append($successMessage);  
  
        $.ajax({  
          url: 'https://someResource.com/data',  
          success: function(data) {  
            var $content = $('<div />');  
            $content.text(data.content);  
            $successMessage.parent().append($content);  
  
            $content.on('click', function(event) {  
              // It just goes on and on...  
            });  
          }  
        });  
      }  
    });  
  });  
});
```

Code isn't important

Maintenance reveals the truth

Callback hell is the manifestation of overly nested callbacks

Notice the pyramid (gives it away)

Enter Promises

Promises are A solution, some may even say A+ solution

This discussion stems from countless hours of myself and my team of developers dealing with common and consistent misconceptions, problems, and hard work

Who's this guy?

Adam Ranfelt

- Creative Coder & Graphics Developer
- Software Engineer at The Nerdery
- Martial Arts Hobbyist
- Console Gamer
- Mega Man Enthusiast



I'm Adam, the presenter. A creative developer, amongst many other hobbies

If you're looking to reach out to me, you can find me on twitter, or check out whatever I may be working on at github. I'm not terribly frequent, but when I am, I hit the work hard.

Who's this guy?

adamRenny

- adamRenny.me
- github.com/adamRenny
- Twitter: @adamRenny



I'm Adam, the presenter. A creative developer, amongst many other hobbies

If you're looking to reach out to me, you can find me on twitter, or check out whatever I may be working on at github. I'm not terribly frequent, but when I am, I hit the work hard.

Discussion

Today's agenda

- Browser-focused
- When To Use
- Libraries
- Patterns

Not for Today

- History
- How it's changing
- The API
- The Specification

One interesting appearance while working on this presentation: <http://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>

Note what we will not be discussing. If you were looking to hear about any of those items in detail, we will not be discussing it.

Using Promises

Using Promises:

- What are they for?
- How do they work?
- What do they solve?
- How are they bad?
- What tools are out there?

Knowing Promises

What makes a Promise

Base level understanding of promises

- The interface
- The lifecycle
- What are they for

Interface

Promise

```
- then(onFulfilled:Promise|Function, onRejected:Promise|Function) : Promise
- success(onFulfilled:Promise|Function) : Promise
- catch(onRejected:Promise|Function) : Promise
- fail(onRejected:Promise|Function) : Promise
- error(onRejected:Promise|Function) : Promise
- done(onComplete:Promise|Function) : Promise
- finally(onComplete:Promise|Function) : Promise

+ all(promises:{Promise|Function}[]) : Promise
+ race(promises:{Promise|Function}[]) : Promise
+ reject(value:{*}) : Promise
+ resolve(value:{*}) : Promise
+ defer() : Deferred<Promise>
```

Note the bold components of the Promise: these are the primary pieces of a promise

The non-bold methods you will learn a bit about, but are effectively non-specification items

Group 1:

then & catch - note that they accept promises or functions as arguments, and if returning a promise will wait for their insides

Note how these also return a promise, that allows for serialized chaining

Group 2:

These static methods are common methods that we'll also be utilizing in our discussion

- all is a way to wait for all async tasks to finish
- race is an existential wait
- reject and resolve are wonderful tools to provide a auto-fulfilled promise

If you don't already, get to know all of these interface methods

Interface

Promise

```
- then(onFulfilled:Promise|Function, onRejected:Promise|Function) : Promise  
- success(onFulfilled:Promise|Function) : Promise  
- catch(onRejected:Promise|Function) : Promise  
- fail(onRejected:Promise|Function) : Promise  
- error(onRejected:Promise|Function) : Promise  
- done(onComplete:Promise|Function) : Promise  
- finally(onComplete:Promise|Function) : Promise  
  
+ all(promises:{Promise|Function}[]) : Promise  
+ race(promises:{Promise|Function}[]) : Promise  
+ reject(value:{*}) : Promise  
+ resolve(value:{*}) : Promise  
+ defer() : Deferred<Promise>
```

Note the bold components of the Promise: these are the primary pieces of a promise

The non-bold methods you will learn a bit about, but are effectively non-specification items

Group 1:

then & catch - note that they accept promises or functions as arguments, and if returning a promise will wait for their insides

Note how these also return a promise, that allows for serialized chaining

Group 2:

These static methods are common methods that we'll also be utilizing in our discussion

- all is a way to wait for all async tasks to finish
- race is an existential wait
- reject and resolve are wonderful tools to provide a auto-fulfilled promise

If you don't already, get to know all of these interface methods

Interface

Promise

```
- then(onFulfilled:Promise|Function, onRejected:Promise|Function) : Promise  
- success(onFulfilled:Promise|Function) : Promise  
- catch(onRejected:Promise|Function) : Promise  
- fail(onRejected:Promise|Function) : Promise  
- error(onRejected:Promise|Function) : Promise  
- done(onComplete:Promise|Function) : Promise  
- finally(onComplete:Promise|Function) : Promise
```

```
+ all(promises:{Promise|Function}[]) : Promise  
+ race(promises:{Promise|Function}[]) : Promise  
+ reject(value:{*}) : Promise  
+ resolve(value:{*}) : Promise  
+ defer() : Deferred<Promise>
```

Note the bold components of the Promise: these are the primary pieces of a promise

The non-bold methods you will learn a bit about, but are effectively non-specification items

Group 1:

then & catch - note that they accept promises or functions as arguments, and if returning a promise will wait for their insides

Note how these also return a promise, that allows for serialized chaining

Group 2:

These static methods are common methods that we'll also be utilizing in our discussion

- all is a way to wait for all async tasks to finish
- race is an existential wait
- reject and resolve are wonderful tools to provide a auto-fulfilled promise

If you don't already, get to know all of these interface methods

Terms

- State
- Fulfilled
- Resolve
- Reject
- Chain

Terms that are important:

State - way the promise exists

Fulfilled - resolved OR rejected

Resolve - fulfill successfully

Reject - fulfill unfortunately

Chain - 1 promise in a string/chain/collection of promises

Lifecycle

What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle

```
var promise = new Promise(fn);
```



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

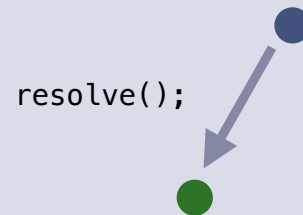
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

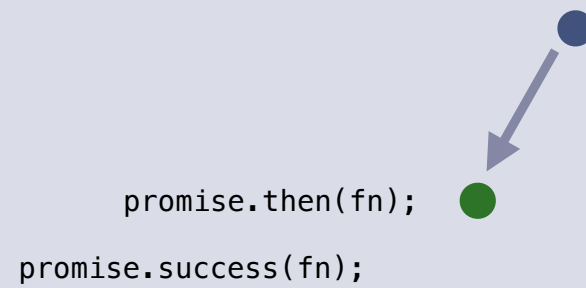
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

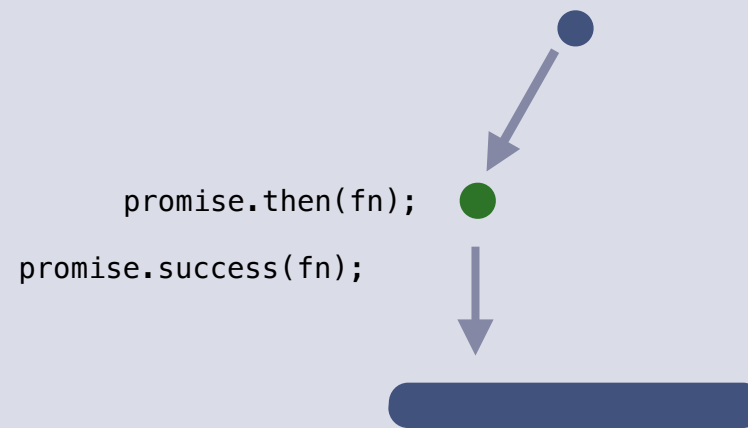
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

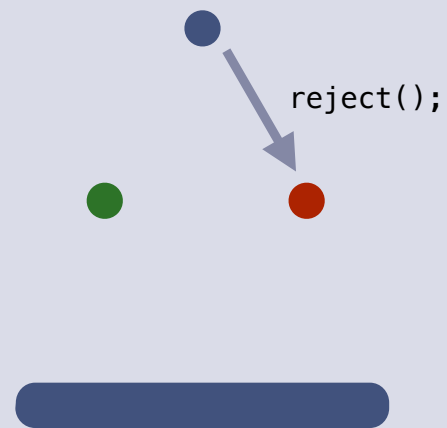
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

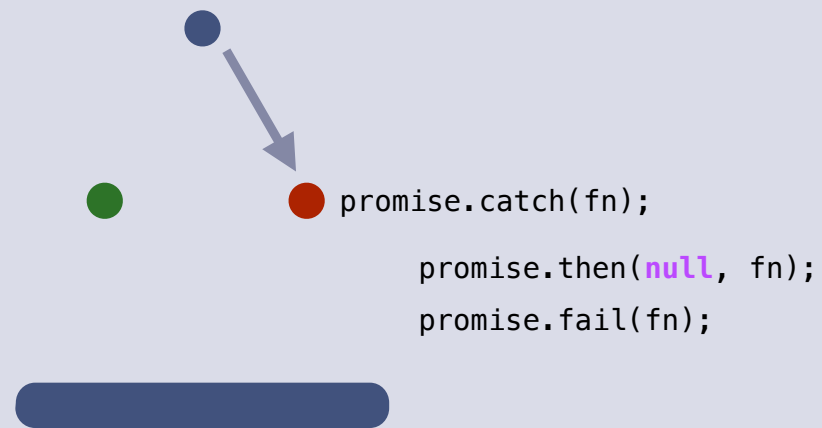
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

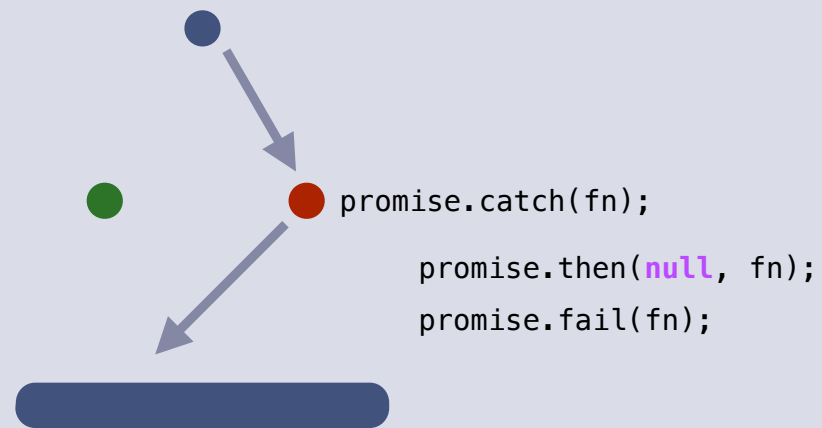
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

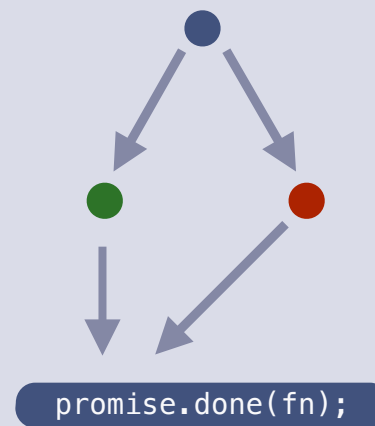
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

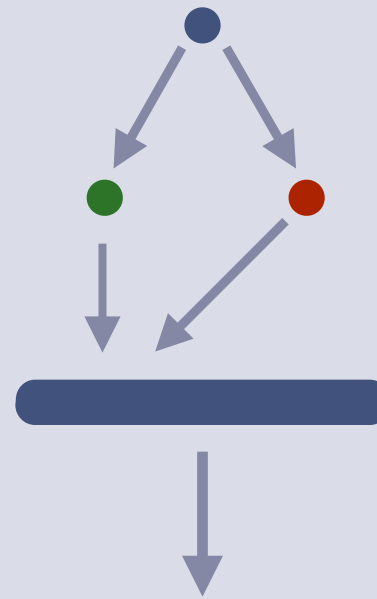
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Lifecycle



What is a lifecycle:

- Google says: the series of changes in the life of an organism, including reproduction
- a lifecycle is a way of seeing the steps that occur within some sort of process, in our case Promises

Red is a rejected state

Green is a resolved state

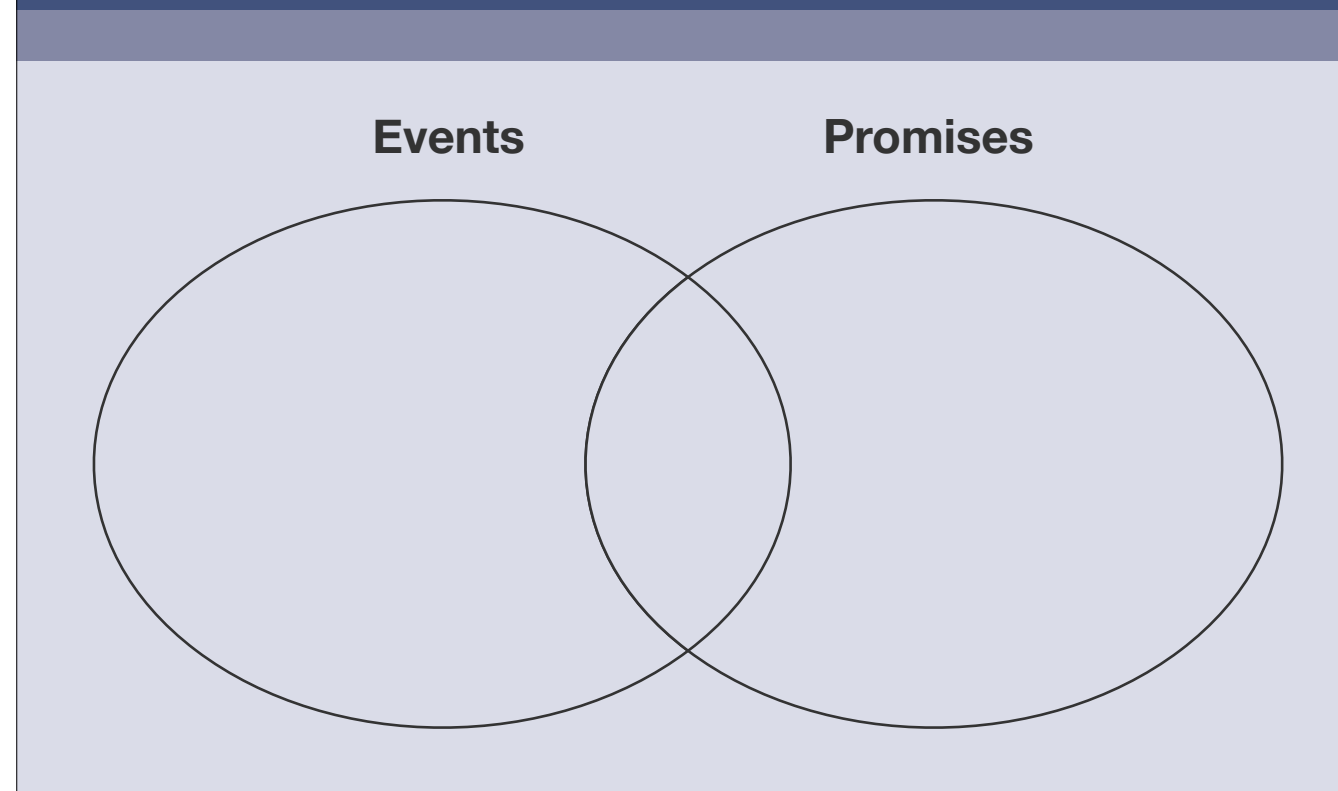
Blue is a neutral state

Promises flow from neutral unfulfilled states to either a resolved or rejected state

These by default flow into a successful state, that we can then treat as a neutral state of our next promise chain

Done is a non standard -> finally fulfilled trigger that some libraries support

Asynchronity

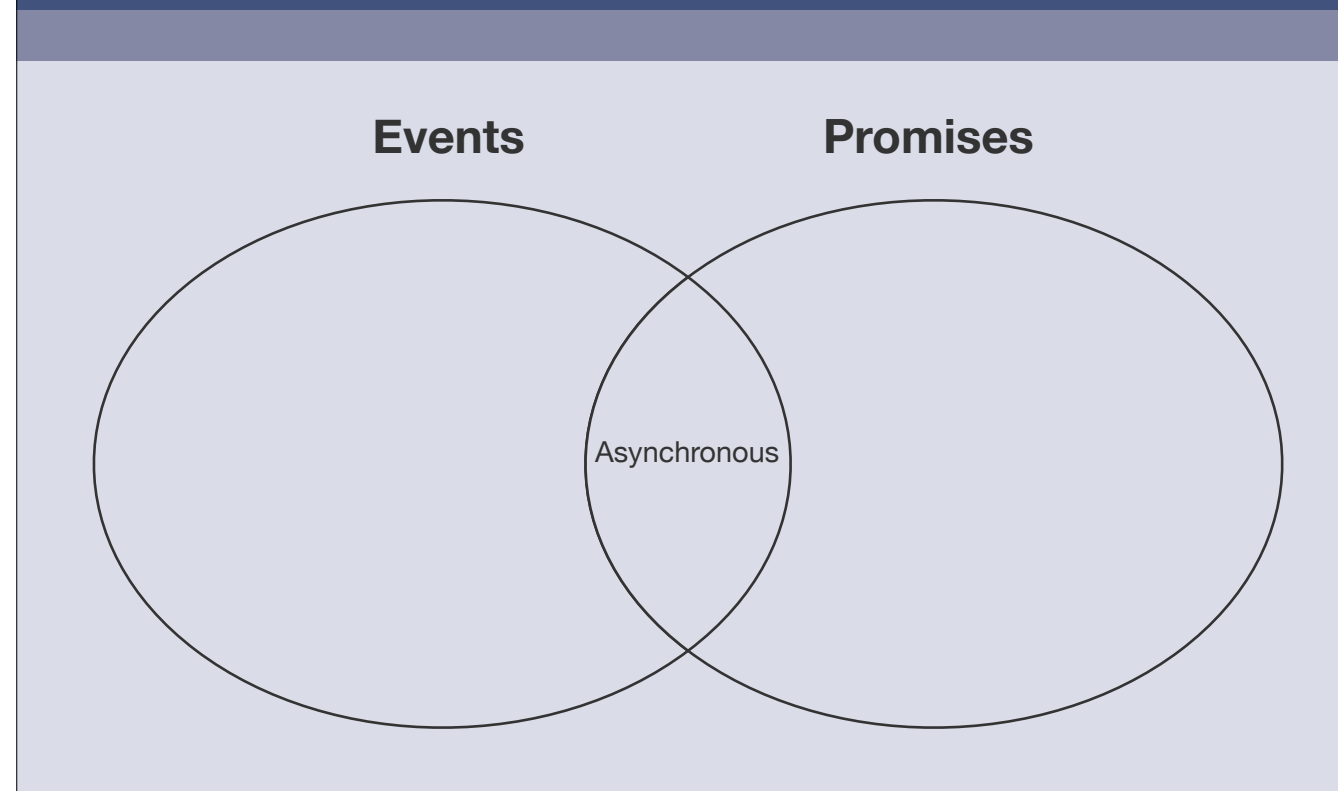


Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Asynchrony

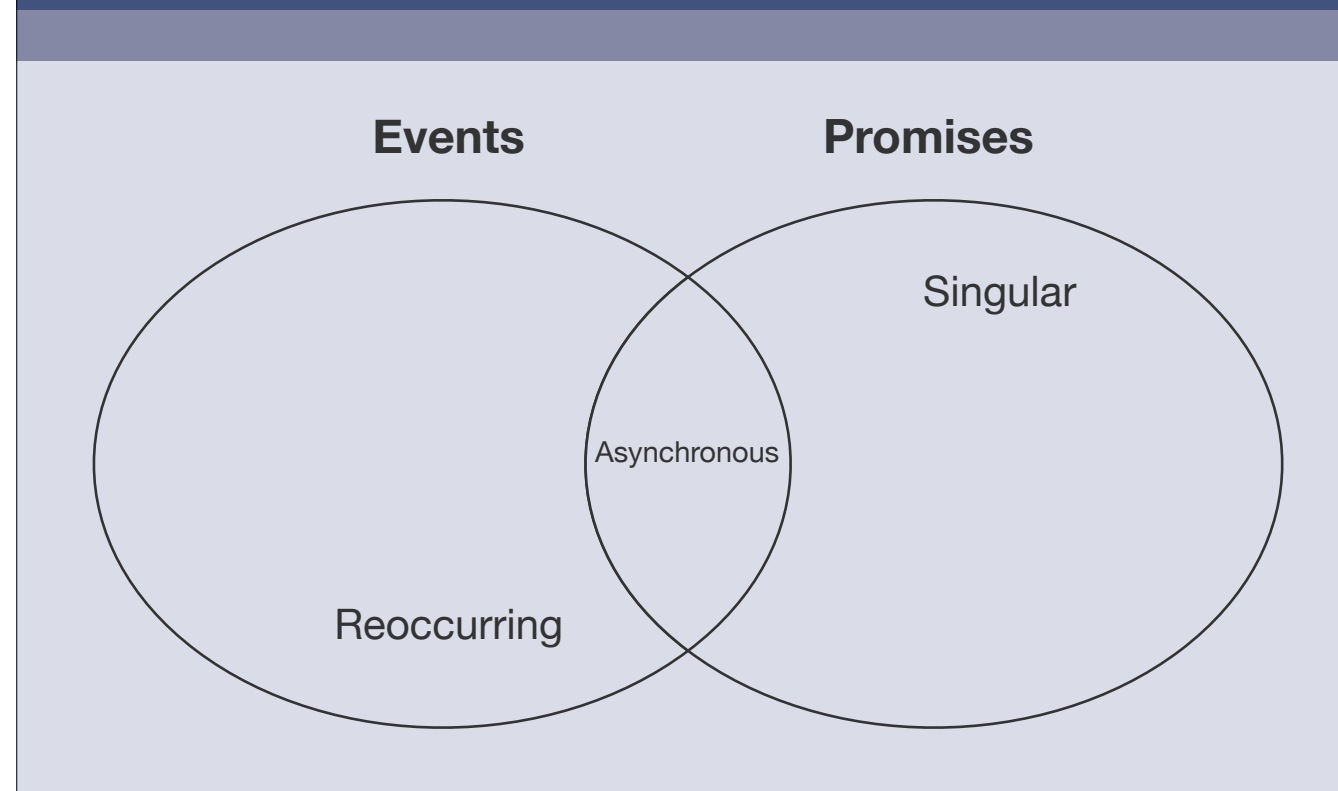


Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Asynchronity

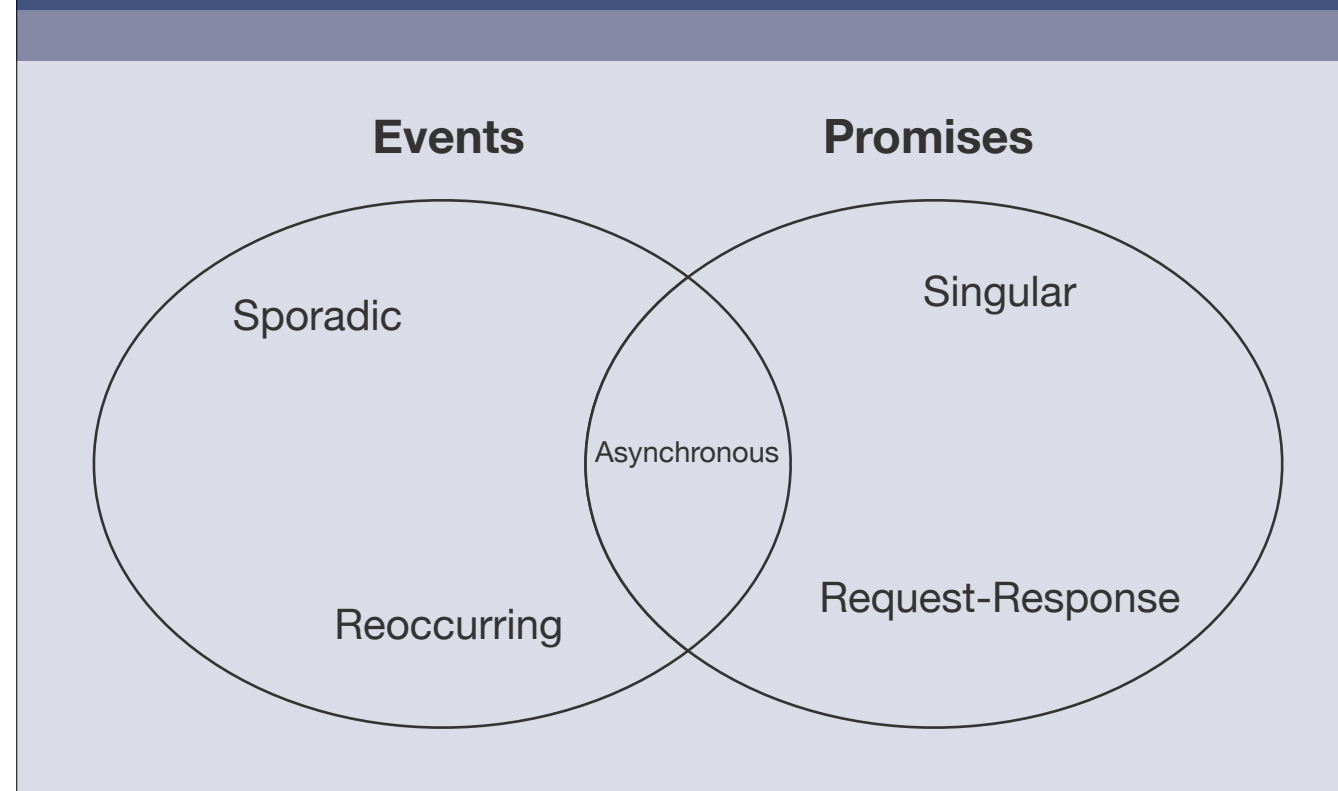


Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Asynchronity

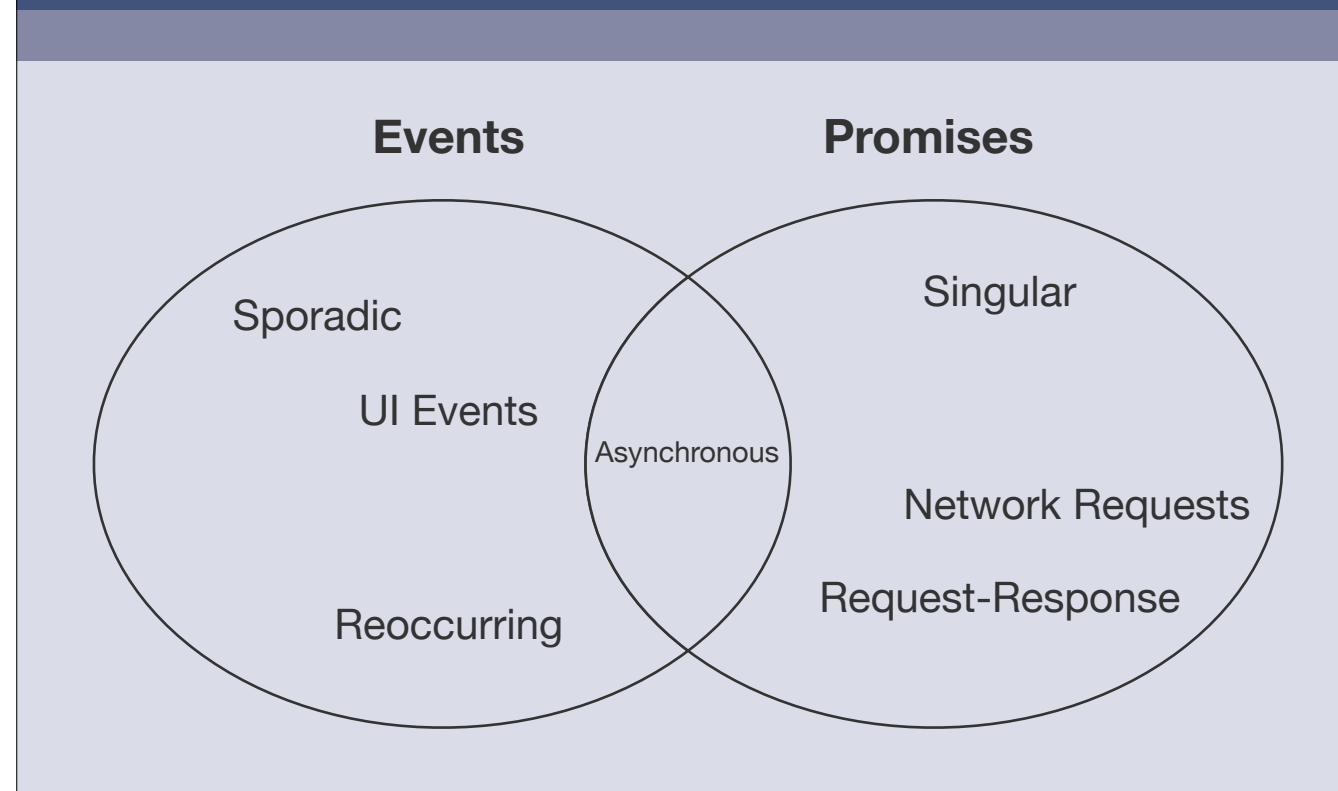


Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Asynchronity

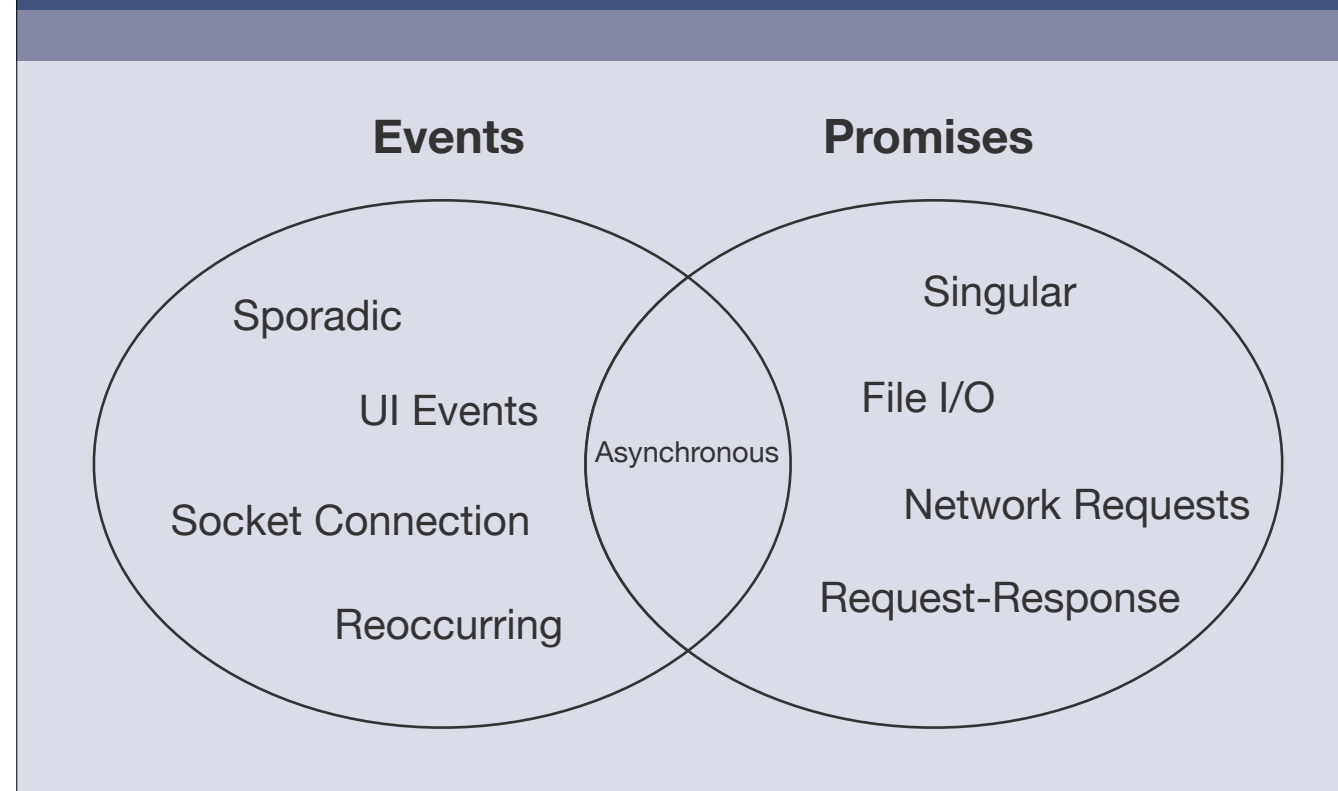


Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Asynchronity



Async as 2 models **Events** vs **Promises**

Events are things like the DOM or Socket Connections - something that happens repeatedly

Promises are things that are requested and may be finished - something that happens once

Events

- Repeated
- Some Frequency
 - Unpredictable
 - Predictable

Events behave in a way that usually will happen multiple times

Show up everywhere in client-side technologies or in embedded systems (interrupts)

Often unpredictable, break logic, and happen over time

Events

- Repeated
- Some Frequency
 - Unpredictable
 - Predictable



Events behave in a way that usually will happen multiple times

Show up everywhere in client-side technologies or in embedded systems (interrupts)

Often unpredictable, break logic, and happen over time

Promise

- Single event
- May have progress
- Entry and Exit

Promises are a single event

Some have progress

Always have an entry and exit

And lack of fulfillment means incompleteness

Promise

- Single event
- May have progress
- Entry and Exit



Promises are a single event

Some have progress

Always have an entry and exit

And lack of fulfillment means incompleteness

Knowing Promises

Why use a Promise

Why are promises even relevant/useful/here?

Solving Problems

- Readability
- Maintainability
- Undesired Side Effects

Readability - Software engineering concern with how easily a person can quickly ascertain the state of code

Maintainability - Software engineering concern with how effectively, efficiently, and easily code can be modified

Undesired Side Effects - Things happening because they have to, but aren't in the right place (separation of concerns)

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      });
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      });
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      });
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      })
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      });
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Readability

```
var SubmissionForm = {
  bind: function() {

    $('someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/someOtherResourceName',
            success: function(data) {
              // Yet more updates
            }
          })
        }
      });
    });
  }
};

SubmissionForm.bind();
```

This is hard to look at and understand from first glance

Clearly contrived

Notice the pyramid/callback hell effect

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: Ajax request 2 (async)

This is hard to read and understand.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {  
  bind: function() {  
  
    $('someForm').on('submit', function(e) {  
      e.preventDefault();  
  
      $.ajax({  
        url: 'https://some.resource.com/resourceName',  
        data: $('someForm').serializeArray(),  
        success: function(data) {  
          // Push updates out onto the form  
          $('someForm').html(data);  
  
          $.ajax({  
            url: 'https://some.resource.com/newResourceToRequest',  
            success: function(data) {  
              $.ajax({  
                url: 'https://some.resource.com/someOtherResourceName',  
                success: function(data) {  
                  // Yet more updates  
                }  
              }  
            }  
          });  
        }  
      });  
    }  
  });  
};  
  
SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {  
  bind: function() {  
  
    $('someForm').on('submit', function(e) {  
      e.preventDefault();  
  
      $.ajax({  
        url: 'https://some.resource.com/resourceName',  
        data: $('someForm').serializeArray(),  
        success: function(data) {  
          // Push updates out onto the form  
          $('someForm').html(data);  
  
          $.ajax({  
            url: 'https://some.resource.com/newResourceToRequest',  
            success: function(data) {  
              $.ajax({  
                url: 'https://some.resource.com/someOtherResourceName',  
                success: function(data) {  
                  // Yet more updates  
                }  
              });  
            }  
          });  
        }  
      });  
    }  
  });  
};  
  
SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {  
  bind: function() {  
  
    $('someForm').on('submit', function(e) {  
      e.preventDefault();  
  
      $.ajax({  
        url: 'https://some.resource.com/resourceName',  
        data: $('someForm').serializeArray(),  
        success: function(data) {  
          // Push updates out onto the form  
          $('someForm').html(data);  
  
          $.ajax({  
            url: 'https://some.resource.com/newResourceToRequest',  
            success: function(data) {  
              $.ajax({  
                url: 'https://some.resource.com/someOtherResourceName',  
                success: function(data) {  
                  // Yet more updates  
                }  
              });  
            }  
          });  
        }  
      });  
    }  
  });  
};  
  
SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {
  bind: function() {

    $('.someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('.someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('.someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/newResourceToRequest',
            success: function(data) {
              $.ajax({
                url: 'https://some.resource.com/someOtherResourceName',
                success: function(data) {
                  // Yet more updates
                }
              });
            }
          });
        }
      });
    });
  }
};

SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {
  bind: function() {

    $('.someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('.someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('.someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/newResourceToRequest',
            success: function(data) {
              $.ajax({
                url: 'https://some.resource.com/someOtherResourceName',
                success: function(data) {
                  // Yet more updates
                }
              });
            }
          });
        }
      });
    });
  }
};

SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {
  bind: function() {

    $('.someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('.someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('.someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/newResourceToRequest',
            success: function(data) {
              $.ajax({
                url: 'https://some.resource.com/someOtherResourceName',
                success: function(data) {
                  // Yet more updates
                }
              });
            }
          });
        }
      });
    });
  }
};

SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {
  bind: function() {

    $('.someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('.someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('.someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/newResourceToRequest',
            success: function(data) {
              $.ajax({
                url: 'https://some.resource.com/someOtherResourceName',
                success: function(data) {
                  // Yet more updates
                }
              });
            }
          });
        }
      });
    });
  }
};

SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {
  bind: function() {

    $('.someForm').on('submit', function(e) {
      e.preventDefault();

      $.ajax({
        url: 'https://some.resource.com/resourceName',
        data: $('.someForm').serializeArray(),
        success: function(data) {
          // Push updates out onto the form
          $('.someForm').html(data);

          $.ajax({
            url: 'https://some.resource.com/newResourceToRequest',
            success: function(data) {
              $.ajax({
                url: 'https://some.resource.com/someOtherResourceName',
                success: function(data) {
                  // Yet more updates
                }
              });
            }
          });
        }
      });
    });
  }
};

SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Maintainability

```
var SubmissionForm = {  
  bind: function() {  
  
    $('someForm').on('submit', function(e) {  
      e.preventDefault();  
  
      $.ajax({  
        url: 'https://some.resource.com/resourceName',  
        data: $('someForm').serializeArray(),  
        success: function(data) {  
          // Push updates out onto the form  
          $('someForm').html(data);  
  
          $.ajax({  
            url: 'https://some.resource.com/newResourceToRequest',  
            success: function(data) {  
              $.ajax({  
                url: 'https://some.resource.com/someOtherResourceName',  
                success: function(data) {  
                  // Yet more updates  
                }  
              }  
            }  
          });  
        }  
      });  
    }  
  });  
};  
  
SubmissionForm.bind();
```

Updates: suppose we need to make an additional call before our final request, but after our form is updated.

This needs to seriously modify the innards of our form

Group 1: Submit Callback Event

Group 2: Ajax request (async)

Group 3: Success Callback, post ajax

Group 4: New ajax request (async)

Group 5: Successful handle of the ajax

Group 6: Original Ajax Request 2 (async)

Group 7: Success callback, post everything

This is hard to update without affecting subsequent operations.

Note that callbacks do not necessarily create this problem, but they afford us to do so.

Undesired Side Effects

- Calling asynchronous tasks unexpectedly
- One function calls another

```
function updateForm(content) {  
  $('someForm').html(content);  
  
  $.ajax({  
    url: 'https://some.resource.com/moreContentForMyForm'  
  });  
}
```

Common manifestation of async tasks

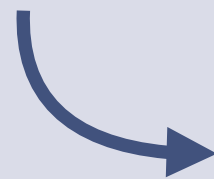
Only way to solve simply is to expose a callback

Further exacerbating what this method does. No longer **just** updateForm

Undesired Side Effects

- Calling asynchronous tasks unexpectedly
- One function calls another

```
function updateForm(content) {  
  $('#someForm').html(content);  
  
  $.ajax({  
    url: 'https://some.resource.com/moreContentForMyForm'  
  });  
}
```



```
function updateForm(content, onSuccess) {  
  $('#someForm').html(content);  
  
  $.ajax({  
    url: 'https://some.resource.com/moreContentForMyForm',  
    success: onSuccess  
  });  
}
```

Common manifestation of async tasks

Only way to solve simply is to expose a callback

Further exacerbating what this method does. No longer **just** updateForm

Making Problems

- Overhead
- Error Trapping
- Hard to Grok

Promises aren't perfect

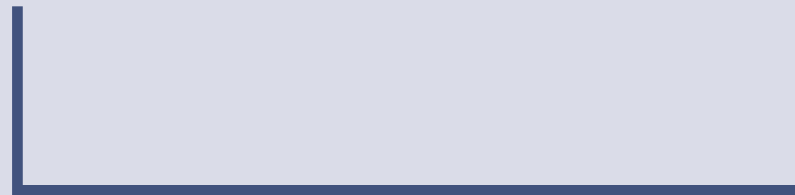
Overhead - Memory, processing, objects, garbage collection

Error Trapping - Common problem during development

Hard to Grok - These are hard to know how to use

Overhead

- What's in a promise?



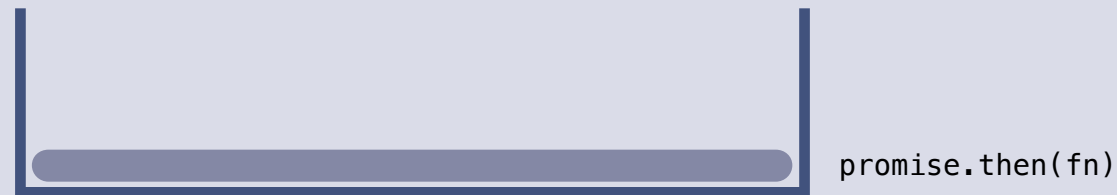
Promises can be heavy (process, memory, etc.)

Compared to callbacks, promises are huge having the capacity to produce at least 2 other promises, and storing at the minimum 1 function.

As an example, each time you call then on a promise, you should remember that you're adding objects to the stack, much like this image. It can very quickly add up without thinking about it.

Overhead

- What's in a promise?



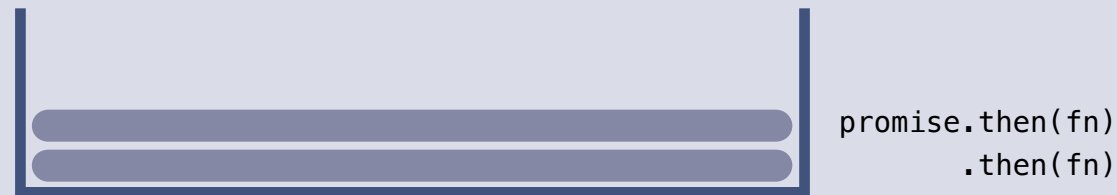
Promises can be heavy (process, memory, etc.)

Compared to callbacks, promises are huge having the capacity to produce at least 2 other promises, and storing at the minimum 1 function.

As an example, each time you call then on a promise, you should remember that you're adding objects to the stack, much like this image. It can very quickly add up without thinking about it.

Overhead

- What's in a promise?



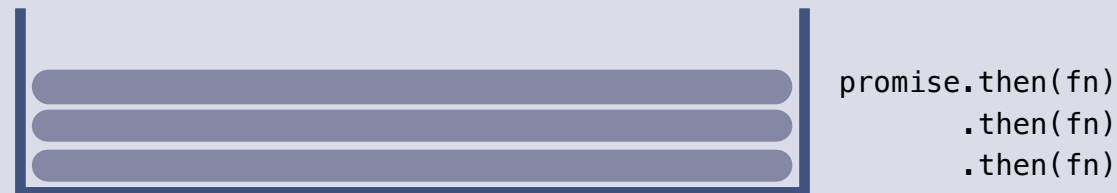
Promises can be heavy (process, memory, etc.)

Compared to callbacks, promises are huge having the capacity to produce at least 2 other promises, and storing at the minimum 1 function.

As an example, each time you call then on a promise, you should remember that you're adding objects to the stack, much like this image. It can very quickly add up without thinking about it.

Overhead

- What's in a promise?



Promises can be heavy (process, memory, etc.)

Compared to callbacks, promises are huge having the capacity to produce at least 2 other promises, and storing at the minimum 1 function.

As an example, each time you call `then` on a promise, you should remember that you're adding objects to the stack, much like this image. It can very quickly add up without thinking about it.

Overhead

- What's in a promise?



Promises can be heavy (process, memory, etc.)

Compared to callbacks, promises are huge having the capacity to produce at least 2 other promises, and storing at the minimum 1 function.

As an example, each time you call then on a promise, you should remember that you're adding objects to the stack, much like this image. It can very quickly add up without thinking about it.

Error Trapping

- Errors miraculously disappear

Error trapping is something that Promises do. Promises can interpret an error thrown as a **rejection**.

There've been some discussion about removing these, but to be honest we're not completely there. Errors are trapped, and show up differently depending on the browser, the library, and almost as if it feels like it that day. Its incredibly frustrating. If you ever are using promises and can't figure out why its failing, just check your catch.

Error Trapping

- Errors miraculously disappear



Error trapping is something that Promises do. Promises can interpret an error thrown as a **rejection**.

There've been some discussion about removing these, but to be honest we're not completely there. Errors are trapped, and show up differently depending on the browser, the library, and almost as if it feels like it that day. Its incredibly frustrating. If you ever are using promises and can't figure out why its failing, just check your catch.

Grok

verb \ˈgräk

to understand profoundly and intuitively

What does grok even mean?

The name itself lends itself well to what our point is

Hard to Grok

- Extremely expansive

Around since 1970s, but only recently brought to JS

A good example of how little we know about our own tool is Forged in Fire Episode 3 with the all-by-hand blacksmith, uses a hardy hole for uncurling his spring

Knowing Promises

Which tools

What tools can we use? What will this presentation cover?

Options

Any of these tools are effective enough
Pick a tool, and stick to it

Options

- Q



Any of these tools are effective enough
Pick a tool, and stick to it

Options

- Q
- Bluebird



Any of these tools are effective enough
Pick a tool, and stick to it

Options

- Q
- Bluebird
- jQuery



Any of these tools are effective enough
Pick a tool, and stick to it

Options

- Q
- Bluebird
- jQuery
- RSVP



Any of these tools are effective enough
Pick a tool, and stick to it

Options

- Q
- Bluebird
- jQuery
- RSVP
- ES6



Any of these tools are effective enough
Pick a tool, and stick to it

Promises/A+

A+

Most common specification

All the libraries mentioned here adhere to it (except jQuery)

We'll be focusing on the A+ spec, and I'd definitely recommend to stay with it, but you can choose to deviate as much as you like

- Why do Promise users bash on jQuery?

Why do they?

- jQuery doesn't implement the A+ spec
 - Errors bubble, and aren't managed within the promise scope
- jQuery versions < 1.8 have thens returning a new promise

Rumors have said 3.0 may start using Promises Spec

- Why do Promise users bash on jQuery?

< 1.8

```
$.ajax()  
  // Chains the original promise  
  .then();
```

Why do they?

- jQuery doesn't implement the A+ spec
 - Errors bubble, and aren't managed within the promise scope
- jQuery versions < 1.8 have thens returning a new promise

Rumors have said 3.0 may start using Promises Spec

- Why do Promise users bash on jQuery?

< 1.8

```
$.ajax()  
  // Chains the original promise  
  .then();
```

>= 1.8

```
$.ajax()  
  // Returns a new promise  
  .then();
```

Why do they?

- jQuery doesn't implement the A+ spec
 - Errors bubble, and aren't managed within the promise scope
- jQuery versions < 1.8 have thens returning a new promise

Rumors have said 3.0 may start using Promises Spec

- Why do Promise users bash on jQuery?

3.0?

Why do they?

- jQuery doesn't implement the A+ spec
 - Errors bubble, and aren't managed within the promise scope
- jQuery versions < 1.8 have then's returning a new promise

Rumors have said 3.0 may start using Promises Spec

For This Presentation

?

For the purposes of this presentation, we won't be focusing on any library, but using various libraries for the examples.

The code in this presentation IS pseudocode, derived from code that was written rather than copied. Since much of that code was heavily integrated with other structures and objects, this hasn't been heavily tested

Patterns

Welcome to the meat of the presentation. We'll be talking about patterns, or some ways to use them. Lets be honest, one of the hardest things a programmer can ever do is name something, so I fully admit that these names may not be the end-all be-all.

- Interface
- Error Bubbling
- Synopsis
- Rejection
- Nesting
- Looping

Starting with interface

Interface

Helping hands

Interface - Most of the names here are used to describe other patterns in our code, interfacing isn't different

Problem

- Not native
- Extra Work

Promises aren't native, and it requires extra work to use them.

We have numerous asynchronous functions and even more asynchronous applications to use them

Asynchronous processes need to use promises. `setTimeout`, `AJAX`, `requestAnimationFrame`, any of these use raw callbacks

Using the pattern means there's going to be extra work. However, if we leverage it, we can go as far as supporting entrance/exit animations, or even as far as a web worker

Solution

- Map Async Functions
- Utility Functions

The solution here is to make our asynchronous functions into promise utility functions

Promisify is a function I see in various places to help “Promisify” our workflows

```
function wait(delay) {  
  return new Promise(function(resolve) {  
    setTimeout(resolve, delay);  
  });  
}
```

wait - setTimeout

```
function request(url) {  
  return new Promise(function(resolve, reject) {  
    function onLoaded(event) {  
      resolve(xhr, event);  
    }  
  
    function onError(event) {  
      reject(xhr, event);  
    }  
  
    var xhr = new XMLHttpRequest();  
    xhr.open('GET', url, true);  
  
    xhr.addEventListener('load', onLoaded, false);  
    xhr.addEventListener('error', onError, false);  
    xhr.addEventListener('abort', onError, false);  
  
    xhr.send();  
  });  
}
```

ajax - see [jQuery](#)/[axios](#)/[reqwest](#)/[Angular.\\$http](#)

This is a very simple version of an XHR request returning a promise. People build libraries to accomplish the task of promise management

```
// CSS

#mainSection {
  transition: opacity 1s;
}

.isActive {
  opacity: 1;
}
```

Lets talk about an animation with CSS

If our CSS looks like this, we know we have a transition and when something's active it has opacity: 1. Assume opacity: 0 is a default.

```
// JS

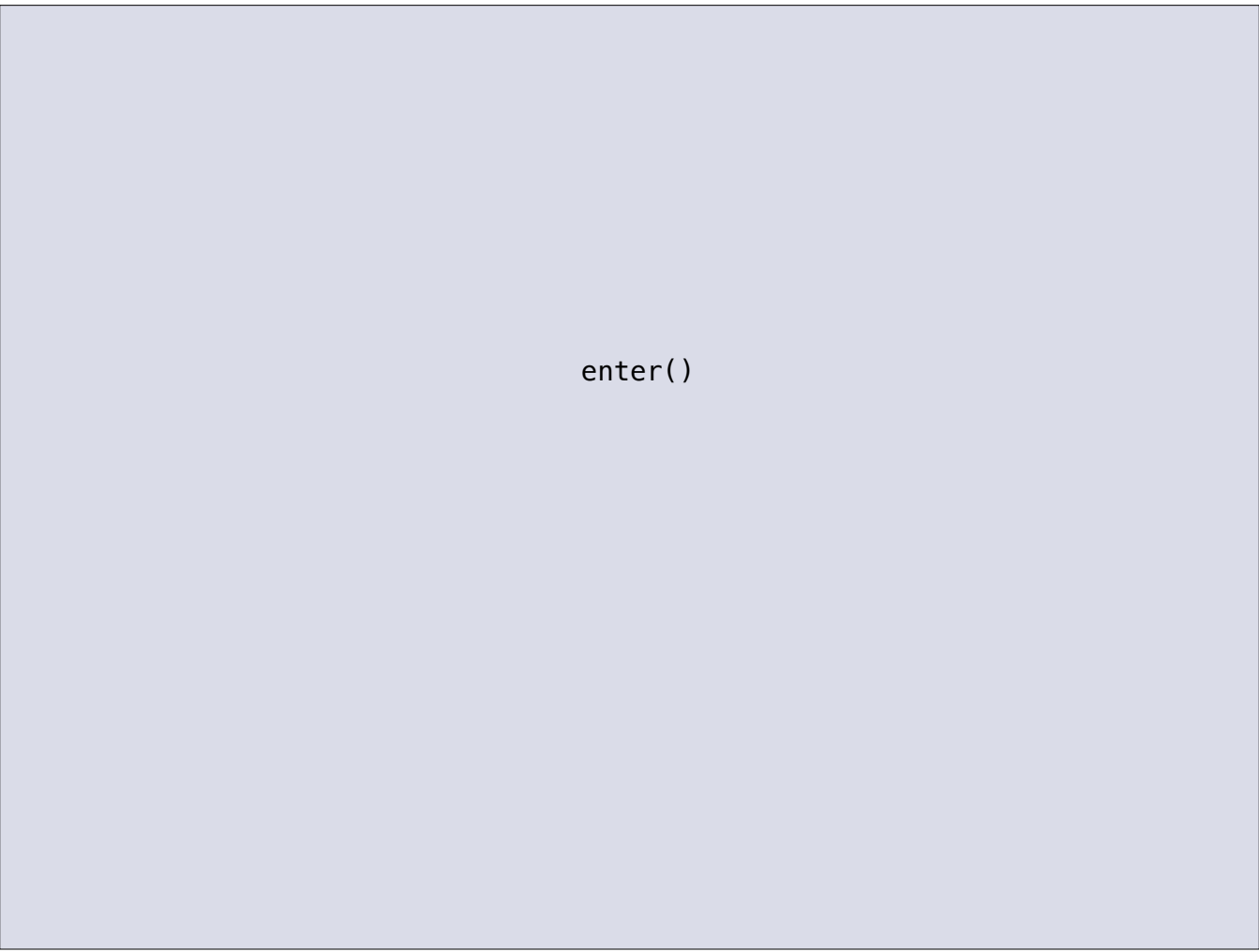
var section = document.getElementById('mainSection');

function addClass(el, className) {
  var classes = el.className.split(/\s+/);
  if (classes.indexOf(className) === -1) {
    classes.push(className);
  }

  el.className = classes.join(' ');
}

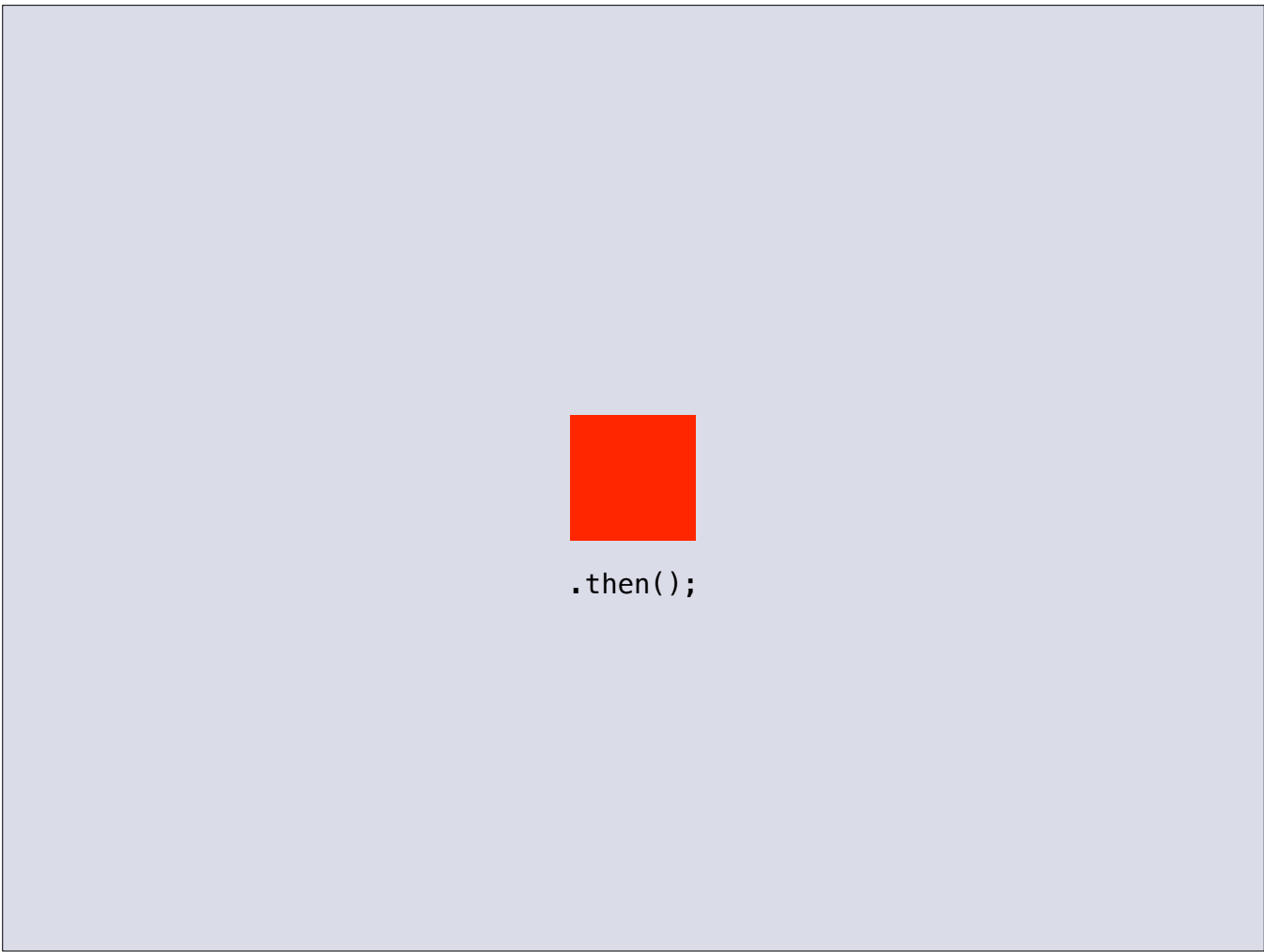
function enter() {
  return new Promise(function(resolve, reject) {
    function onTransitionEnd() {
      section.removeEventListener('transitionend', onTransitionEnd);
      resolve();
    }
    section.addEventListener('transitionend', onTransitionEnd, false);
    addClass(section, 'isActive');
  });
}
```

Notice the enter function here. By handling what is today a event-based model and turn it into what we likely would use it for - a request-response model - we can transform CSS transitions into promisifiable actions.



enter()

The action would occur like the following animation (or slides for you folks reading online :-)



The action would occur like the following animation (or slides for you folks reading online :-)

```
var worker = new Worker('someAsyncTask.js');

function doSomeWork(someJSON) {
  var promise;

  function work(resolve, reject) {
    function onMessage() {
      worker.removeEventListener('message', onMessage);
      resolve.apply(promise, arguments);
    }

    // Only listen once
    worker.addEventListener('message', onMessage);

    worker.postMessage(someJSON);
  }

  promise = new Promise(work);
  return promise;
}
```

One experiment I made was the idea of a web worker to offload work like processing physics engines or complex computations or even color transformations performed on screens (complex image processing) Workers can also be harnessed with promises to do timed chunks of work.

- Interface
- Error Bubbling
- Synopsis
- Rejection
- Nesting
- Looping

Move onto error bubbling

Error Bubbling

Fulfilling lies and
rejection

Error bubbling is a lot like event bubbling, only its intentionally bringing errors back

Problem

- Promises devour errors
- Debugging difficult

Promises devour errors, our errors never make it back to the window
This makes it so in production, if bugs are out there it fails silently, not gracefully
You can't troubleshoot errors you can't see, so its difficult to deal with

Solution

- Catch errors and display/show them

Catch the errors and put them out there. In most situations, you can't throw another error.

Oftentimes you'll want to pipe it over to the `window.onerror` method for a production system, and `console.error` if you're not. So far I've only found `console.error` for actually successfully getting an error to hit the error stack, but I'm sure there'll be more

There should be proper support in the future, but for now this can really kill your development

```
window.onerror = function(error) {  
    // Do something relevant with the error  
    console.error(error);  
};
```

Setup - lets make an onerror function

1. window.onerror
2. Promise code
3. Handle error (all)


```
new Promise(function(resolve) {  
  throw new Error('Irrelevant error.');
```

```
})  
  .then(null, handleError);
```

In the promise, an error is thrown

1. window.onerror
2. Promise code
3. Handle error (all)

```
function handleError() {  
  window.onerror.apply(window, arguments);  
}
```

Now that we have the error, since we're in a promise we can't get the error?

1. window.onerror
2. Promise code
3. Handle error (all)

```
window.onerror = function(error) {  
    // Do something relevant with the error  
    console.error(error);  
};  
  
function handleError() {  
    window.onerror.apply(window, arguments);  
}  
  
new Promise(function(resolve) {  
    throw new Error('Irrelevant error.');})  
    .then(null, handleError);
```

Effectively we want to take all of this together so we can manifest the error ourselves. I'm yet to find a good way to see all of the errors that these things create, but to be honest, its a tough one to solve. Good indicator is if you're using promises, and suddenly things fail without errors, its probably a devoured error

- Interface
- ~~Error Bubbling~~
- Synopsis
- Rejection
- Nesting
- Looping

A more software engineering subject is refactoring and naming

Synopsis

Composing your
narrative

One of my developers once said that controllers are used to compose your narrative. He's right. And promises are used to compose a chain of asynchronous tasks, so we can compose our own narrative.

Problem

- Inline functions
 - Aren't always readable
 - Aren't always maintainable

Inline functions, or your promise chain, if written out right away becomes super hard to read over time

Need to be careful with maintaining them since you can have your code become readable and maintainable if you refactor it, but unbridled promises become quickly unreadable and unmaintainable

Solution

- Refactor
 - Break it apart
 - Naming
- Narrative

Refactor! Break them apart, utilize proper names. The Promise blocks should be the narrative of the code.

```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
    // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  })
  // Link Entries
  .then(function(response) {
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry/link',
        method: 'PATCH',
        data: {
          primaryEntry: entry.id,
          linkedEntry: linkedId
        },
        success: resolve,
        error: reject
      });
    });
  })
  // Delete Errors
  .then(null, function(response) {
    // Display the error
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry',
        method: 'DELETE',
        data: {
          id: entry.id
        },
        success: resolve,
        error: reject
      });
    });
  });
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.


```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
  });
  // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  })
  // Link Entries
  .then(function(response) {
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry/link',
        method: 'PATCH',
        data: {
          primaryEntry: entry.id,
          linkedEntry: linkedId
        },
        success: resolve,
        error: reject
      });
    });
  })
  // Delete Errors
  .then(null, function(response) {
    // Display the error
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry',
        method: 'DELETE',
        data: {
          id: entry.id
        },
        success: resolve,
        error: reject
      });
    });
  });
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.

```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
  });

  // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  });

  // Link Entries
  .then(function(response) {
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry/link',
        method: 'PATCH',
        data: {
          primaryEntry: entry.id,
          linkedEntry: linkedId
        },
        success: resolve,
        error: reject
      });
    });
  });

  // Delete Errors
  .then(null, function(response) {
    // Display the error
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry',
        method: 'DELETE',
        data: {
          id: entry.id
        },
        success: resolve,
        error: reject
      });
    });
  });
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.

```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
    // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  });
  // Link Entries
  .then(function(response) {
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry/link',
        method: 'PATCH',
        data: {
          primaryEntry: entry.id,
          linkedEntry: linkedId
        },
        success: resolve,
        error: reject
      });
    });
  });
  // Delete Errors
  .then(null, function(response) {
    // Display the error
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry',
        method: 'DELETE',
        data: {
          id: entry.id
        },
        success: resolve,
        error: reject
      });
    });
  });
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.

```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
    // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  });
  // Link Entries
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry/link',
      method: 'PATCH',
      data: {
        primaryEntry: entry.id,
        linkedEntry: linkedId
      },
      success: resolve,
      error: reject
    });
  });
}

// Delete Errors
.then(null, function(response) {
  // Display the error
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'DELETE',
      data: {
        id: entry.id
      },
      success: resolve,
      error: reject
    });
  });
});
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.

```

function createAndLinkEntry(data, linkedId) {
  var entry;

  // Create new entry
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry',
      method: 'POST',
      data: data,
      success: resolve,
      error: reject
    });
    // Gather all entries
  }).then(function(response) {
    entry = response;

    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entries',
        method: 'GET',
        success: resolve,
        error: reject
      });
    });
  }, function(error) {
    // Do something about the error
  });
  // Link Entries
  return new Promise(function(resolve, reject) {
    request({
      url: 'https://some.important.resource/entry/link',
      method: 'PATCH',
      data: {
        primaryEntry: entry.id,
        linkedEntry: linkedId
      },
      success: resolve,
      error: reject
    });
  });
  // Delete Errors
  .then(null, function(response) {
    // Display the error
    return new Promise(function(resolve, reject) {
      request({
        url: 'https://some.important.resource/entry',
        method: 'DELETE',
        data: {
          id: entry.id
        },
        success: resolve,
        error: reject
      });
    });
  });
}

```

Picked out a non-jQuery AJAX system to give an example

This is a RESTful API process

Switch over to AJAX with jQuery (admittedly contrived)

This isn't super important as we'll be reviewing it piece by piece

Group 1: This is creating a request it appears

Group 2: This is gathering all the entries, and updating the local entry for usage later in the chain

Group 3: This is linking the entries

Group 4: This is deleting the entries, notice how its called as the second parameter of the then - this is effectively our catch call.

```
function createEntry(entryContent) {  
  return new Promise(function(resolve, reject) {  
    request({  
      url: 'https://some.important.resource/entry',  
      method: 'POST',  
      data: entryContent,  
      success: resolve,  
      error: reject  
    });  
  });  
}
```

Note that we're able to isolate the request and promise to a single method

We now have a reusable createEntry method

```
function fetchAllEntries() {  
  return new Promise(function(resolve, reject) {  
    request({  
      url: 'https://some.important.resource/entries',  
      method: 'GET',  
      success: resolve,  
      error: reject  
    });  
  });  
}
```

We'll likely need to request more entries, so we also move fetchAllEntries into its own request as well

```
function linkEntry(baseEntry, linkId) {  
  return new Promise(function(resolve, reject) {  
    request({  
      url: 'https://some.important.resource/entry/  
link',  
      method: 'PATCH',  
      data: {  
        primaryEntry: baseEntry.id,  
        linkedEntry: linkId  
      },  
      success: resolve,  
      error: reject  
    });  
  });  
}
```

Promise-wrapped request calls start looking the same after a while, but this ones different from the last. When we call a promise mid-chain, the methods context of this is gone, so we have to take special care to make sure we pass in the correct content


```
function removeEntry(entry) {  
  return new Promise(function(resolve, reject) {  
    request({  
      url: 'https://some.important.resource/entry',  
      method: 'DELETE',  
      data: {  
        id: entry.id  
      },  
      success: resolve,  
      error: reject  
    });  
  });  
}
```

Finally, we need to also break out our failure case, albeit a very blind and dumb error case

```
// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}
```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

```
// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}
```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

```

// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}

```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

```
// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}
```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

```
// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}
```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

```
// Higher Scope
var entry = {};

function createAndLinkEntry(data, linkedId) {

    function prepareResponse(response) {
        entry.fromJSON(response);

        return fetchAllEntries();
    }

    // Create new entry
    return createEntry(data)
        .then(prepareResponse, handleError)
        .then(linkEntry.bind(this, entry, linkedId))
        .then(null, removeEntry.bind(this, entry));
}
```

Note that this now explains the creation and linking of the entry

Notice the use of bind. Using bind inline here allows us to make sure we can transfer scope, and if we have any objects that need to transfer from part one to another, we end up getting to gradually modify variables on the way

- Interface
- Error Bubbling
- Synopsis
- Rejection
- Nesting
- Looping

This next one's useful for control. We want to be control freaks.



Rejection

Maintaining rejection

I'm sure everyone's been rejected before. But while we all want to get over it, we also need to accept and maintain that feeling so we don't forget how to evade it next time.

Problem

- Promises fulfill to success normally
- Not always the intended path

When handling errors, sometimes you need to modify state to denote that something had failed.
Its not obvious that after changing that state the workflow

Solution

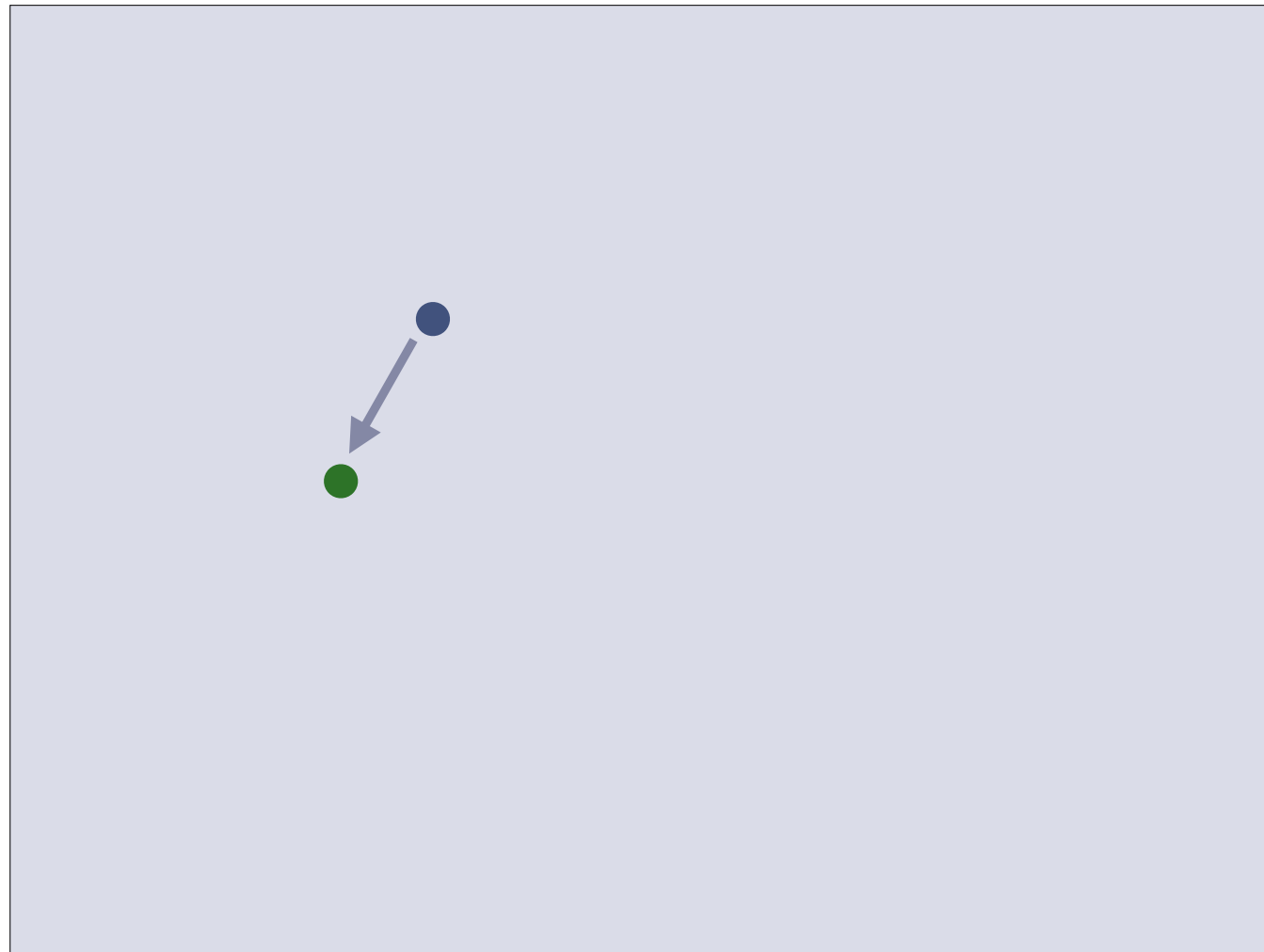
- Force the error state

```
return Promise.reject();
```

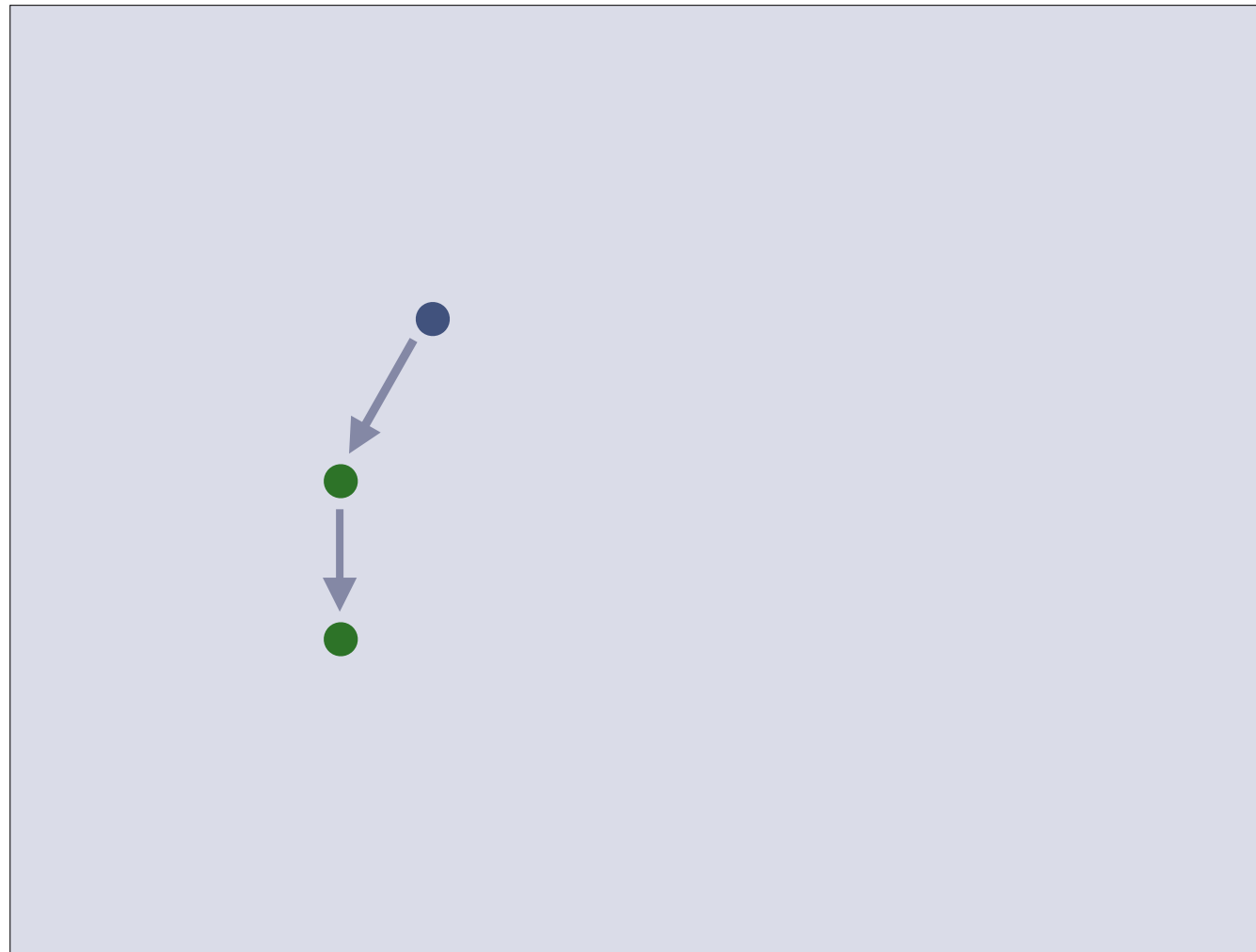
In situations that you want to stay down the error path, its actually quite simple. In most of the libraries, there are instant resolution and rejection methods. If you don't have them, it makes sense to build the interface, or utility function, that you need to make that happen



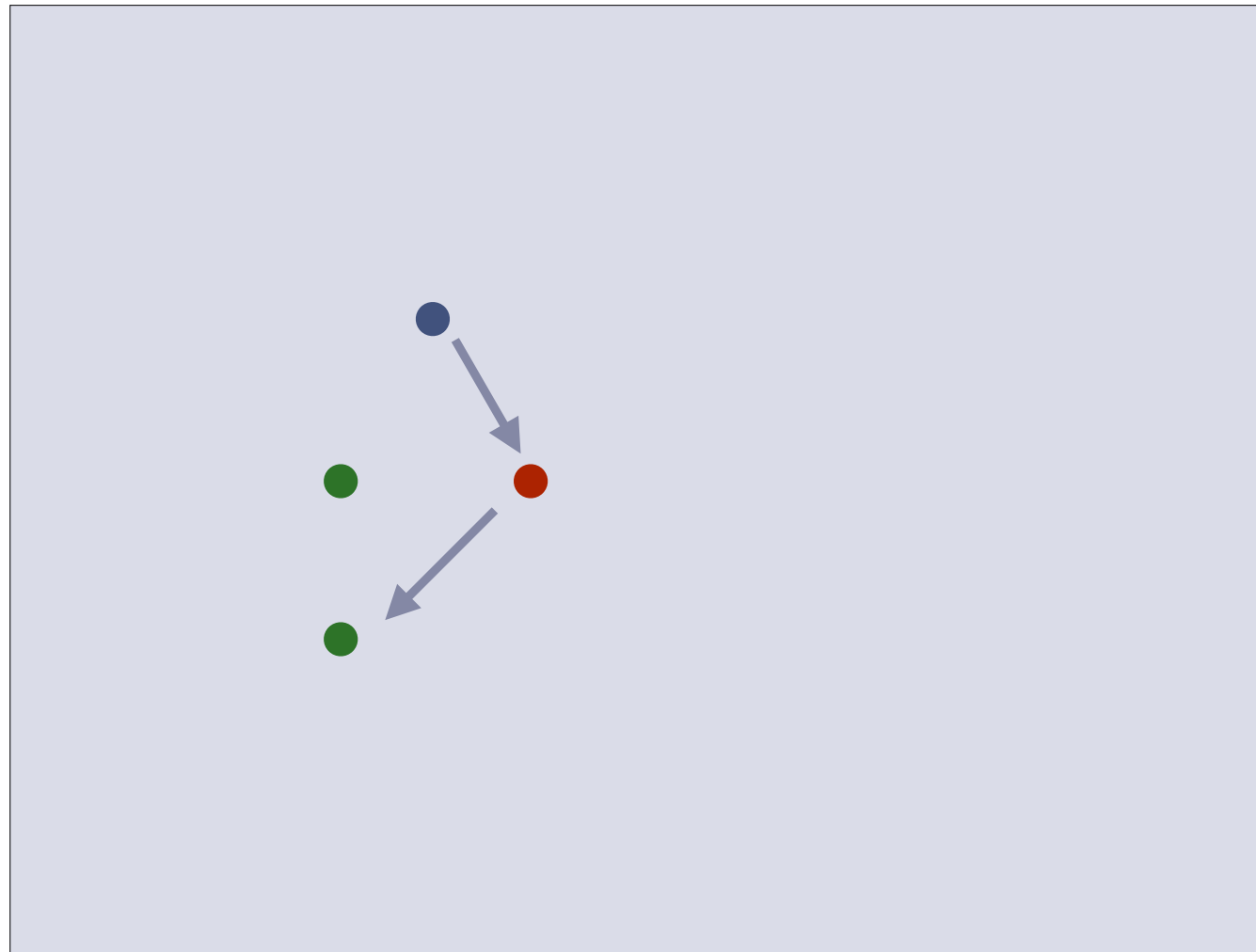
Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.



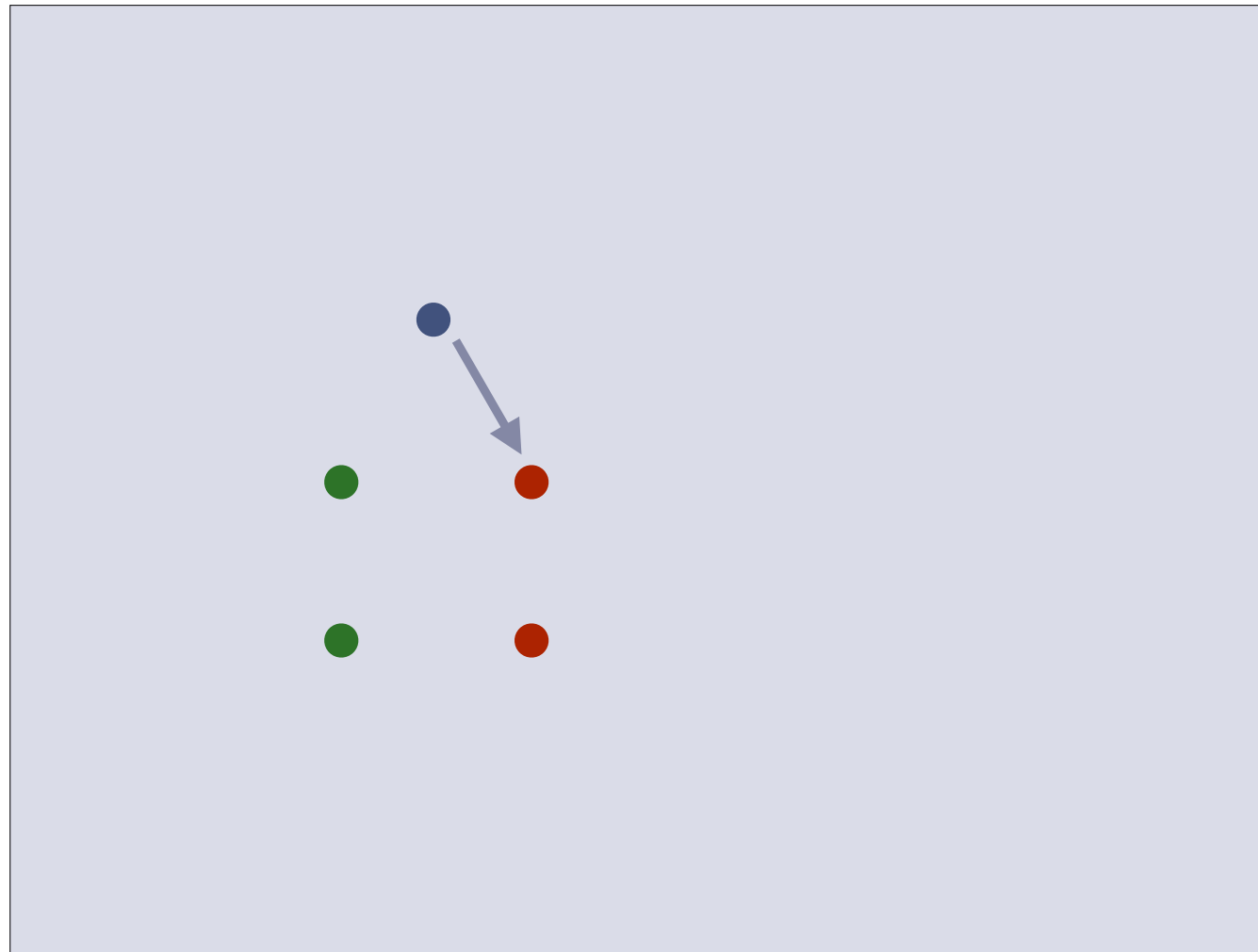
Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.



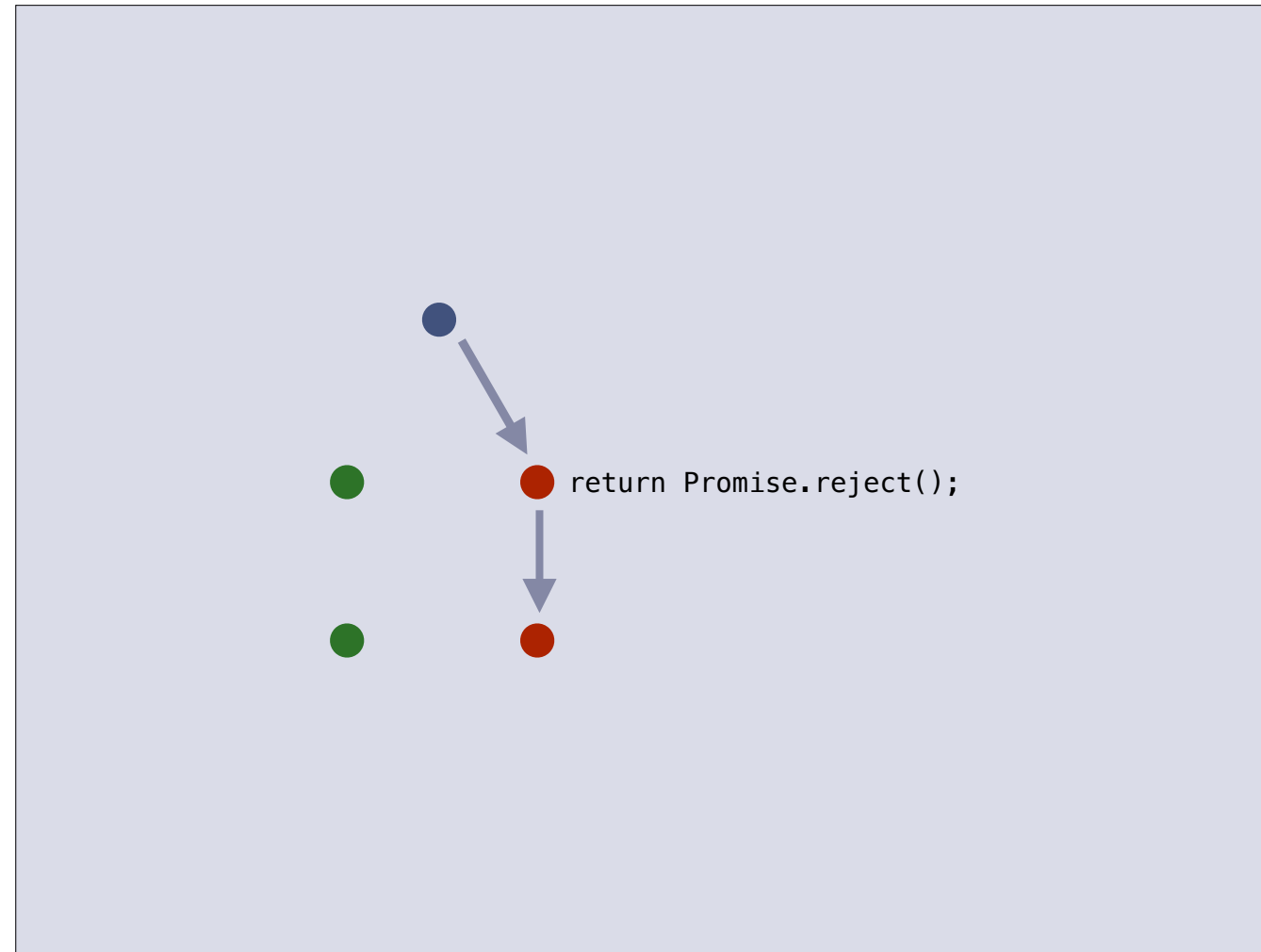
Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.



Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.



Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.



Here we see how to manage our rejection path. By default, these promises are resolved after rejection, meaning that without handling anything a rejection clause will be successful afterwards. Sometimes we want to stay rejected, and in those cases we need to mandate it.

- Interface
- Error Bubbling
- Synopsis
- Rejection
- Nesting
- Looping

Stay control freaks, that's the only way to really harness the process flow of a promise (cause they are hard).

Next we'll talk about a common item: nesting



Nesting

Composing Subchains

The power of promises comes from the ability to have promises influence other promises

Problem

- Can be complex
- Need to be broken down

Some startup processes can be complex and difficult.

One of the powers of Promises is its ability to be nearly recursive in composition.

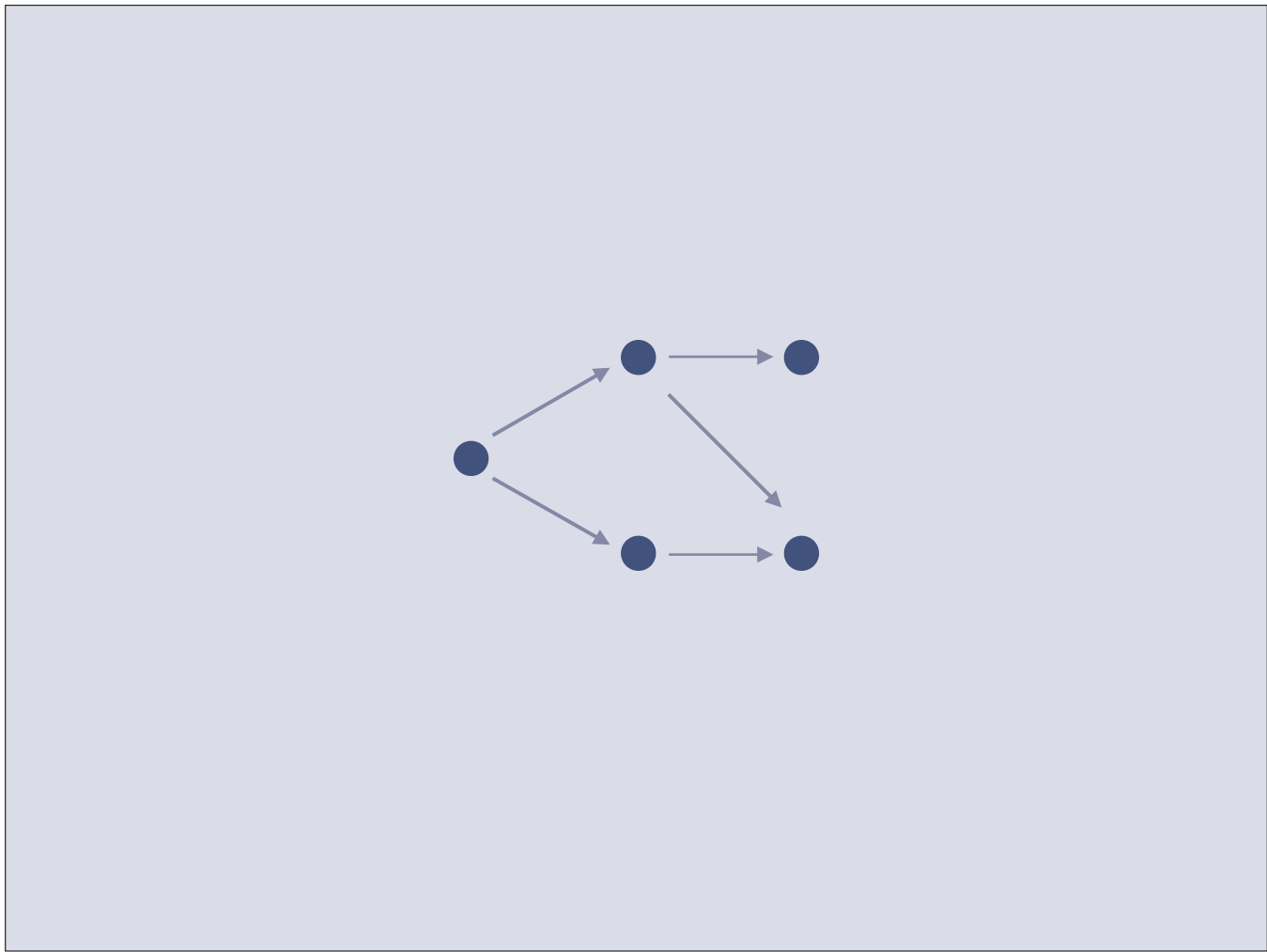
What this means is that you can nest promises within promises to seemingly simplify the workflow.

Solution

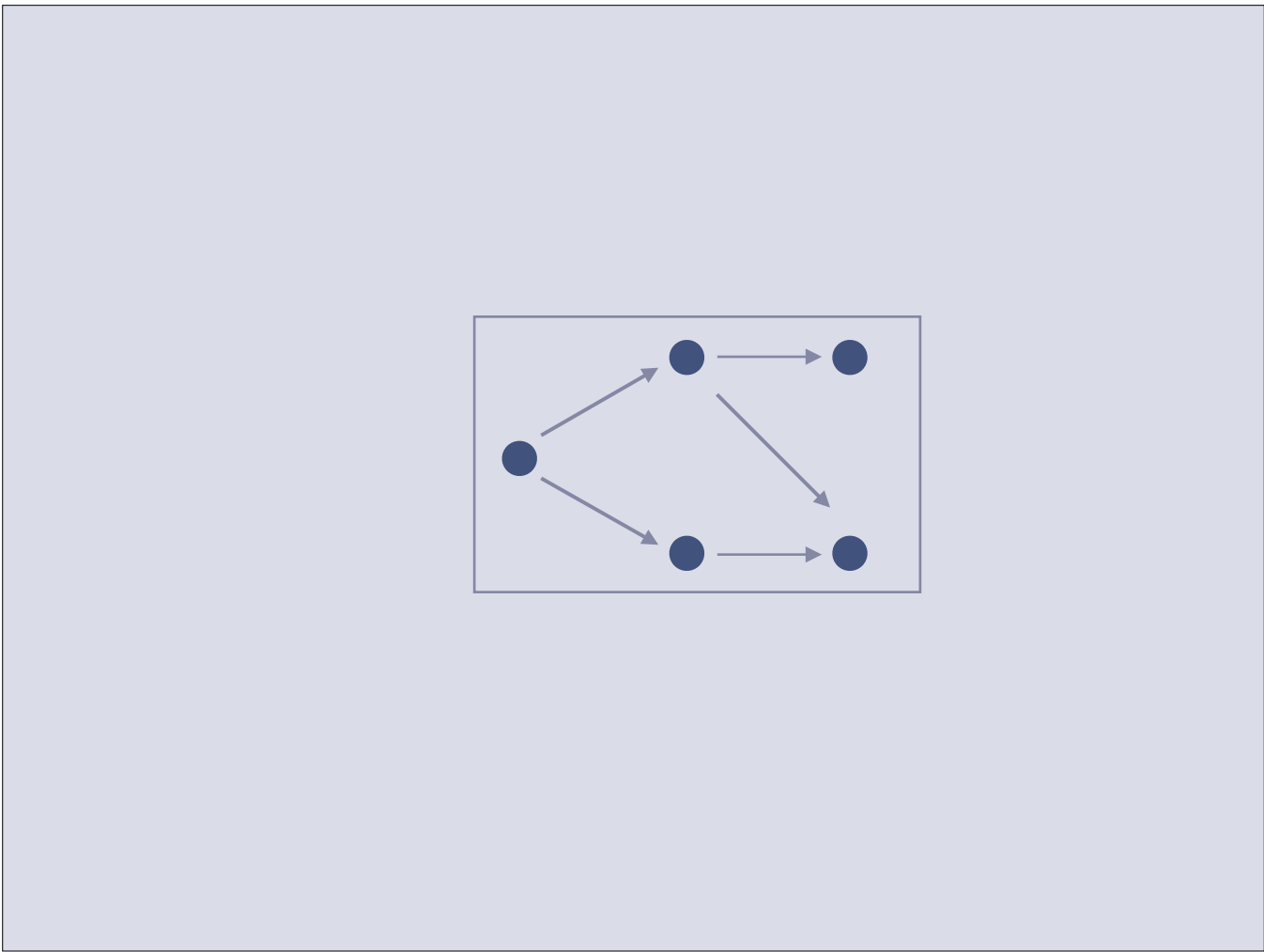
- Define each sub-problem as a process of its own
 - Nest your promises

Defining the subproblems may lead to more complex business logic, but allowing your system to work with specific logical blocks leads to control in composing your process block.

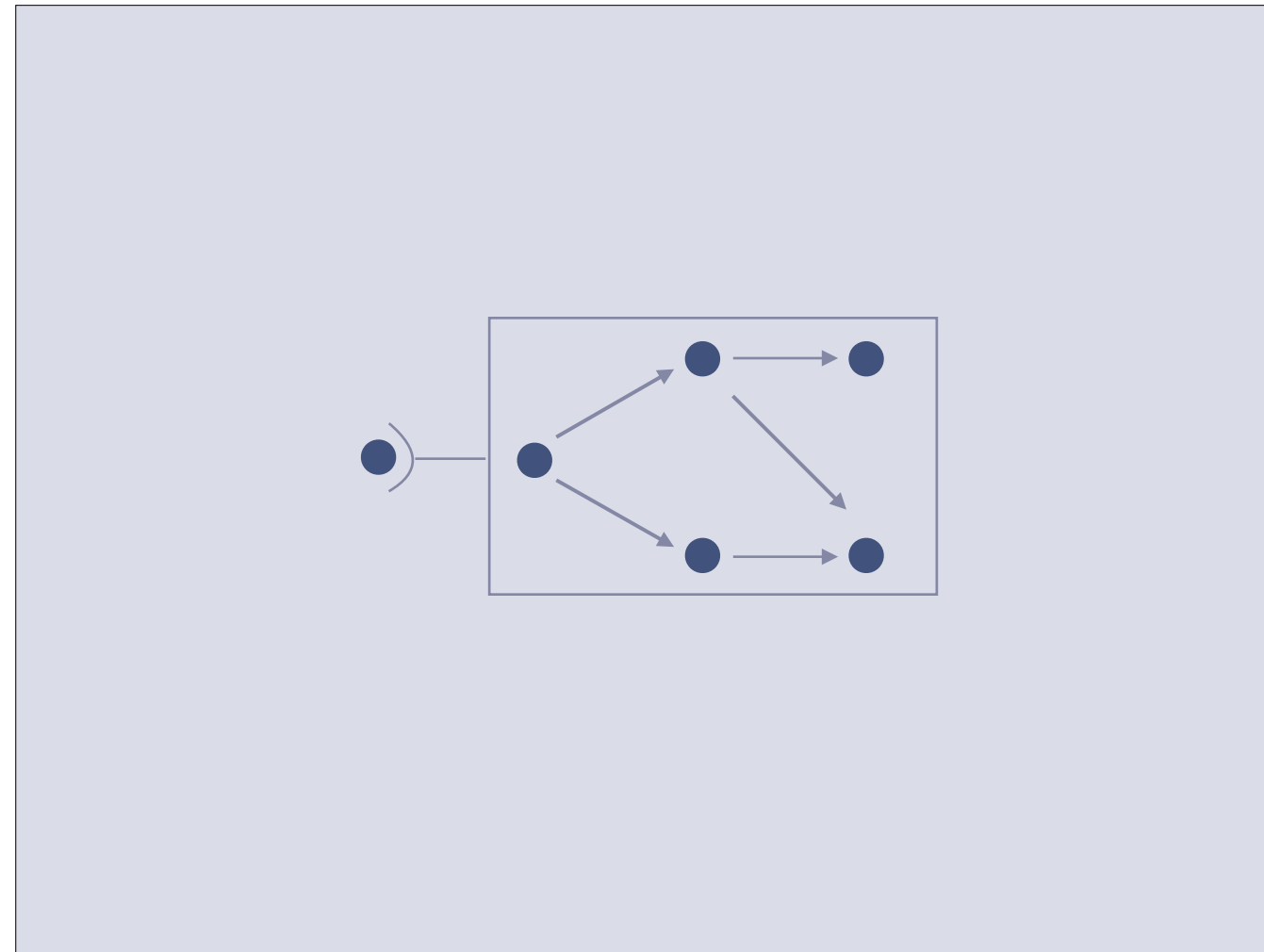
Think about this like a expand/collapse around your logic block.



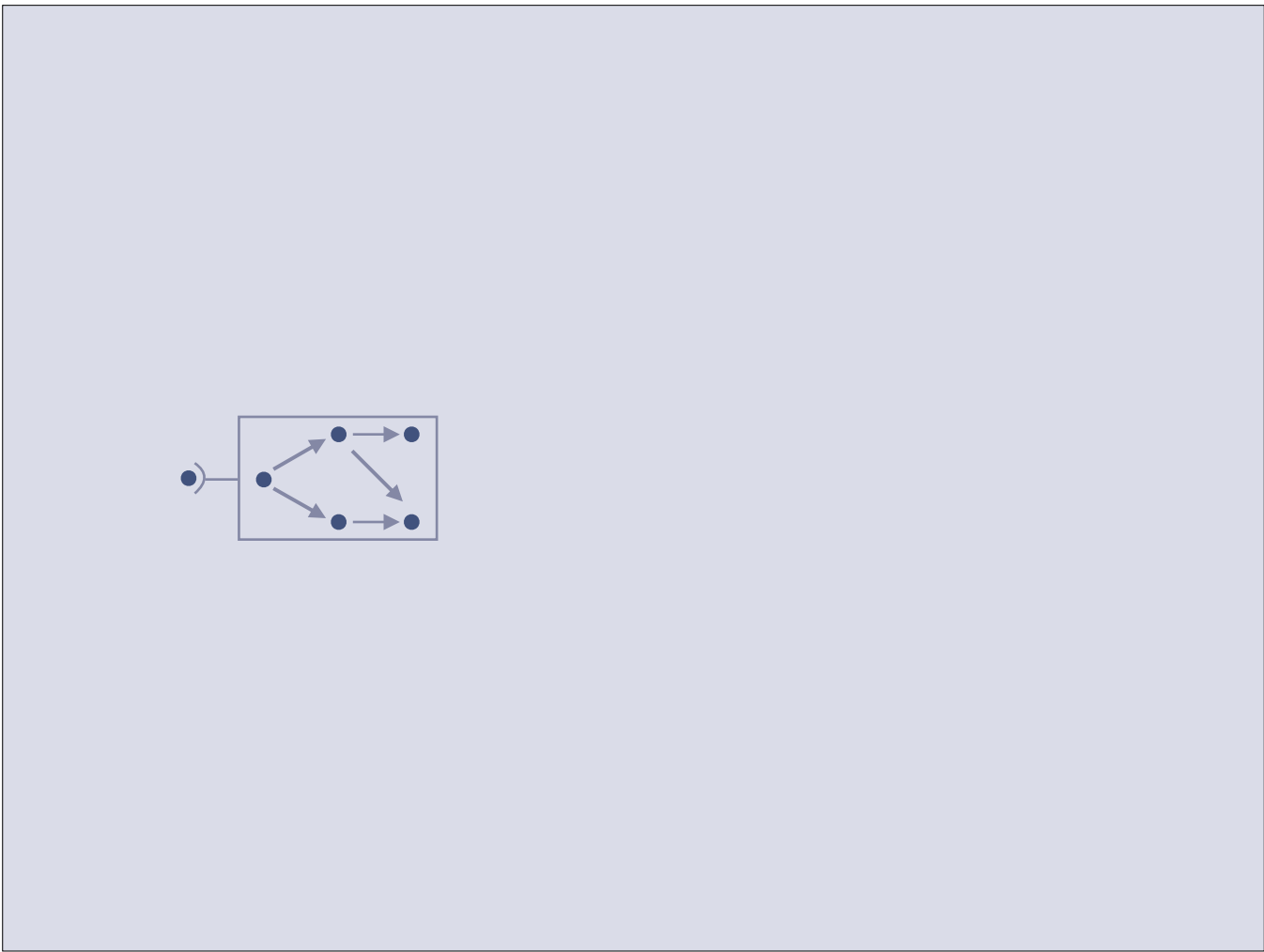
The trick to understanding nesting of your promises is to identify what promise chains will be used collectively, and start to treat minor promises as black boxes.



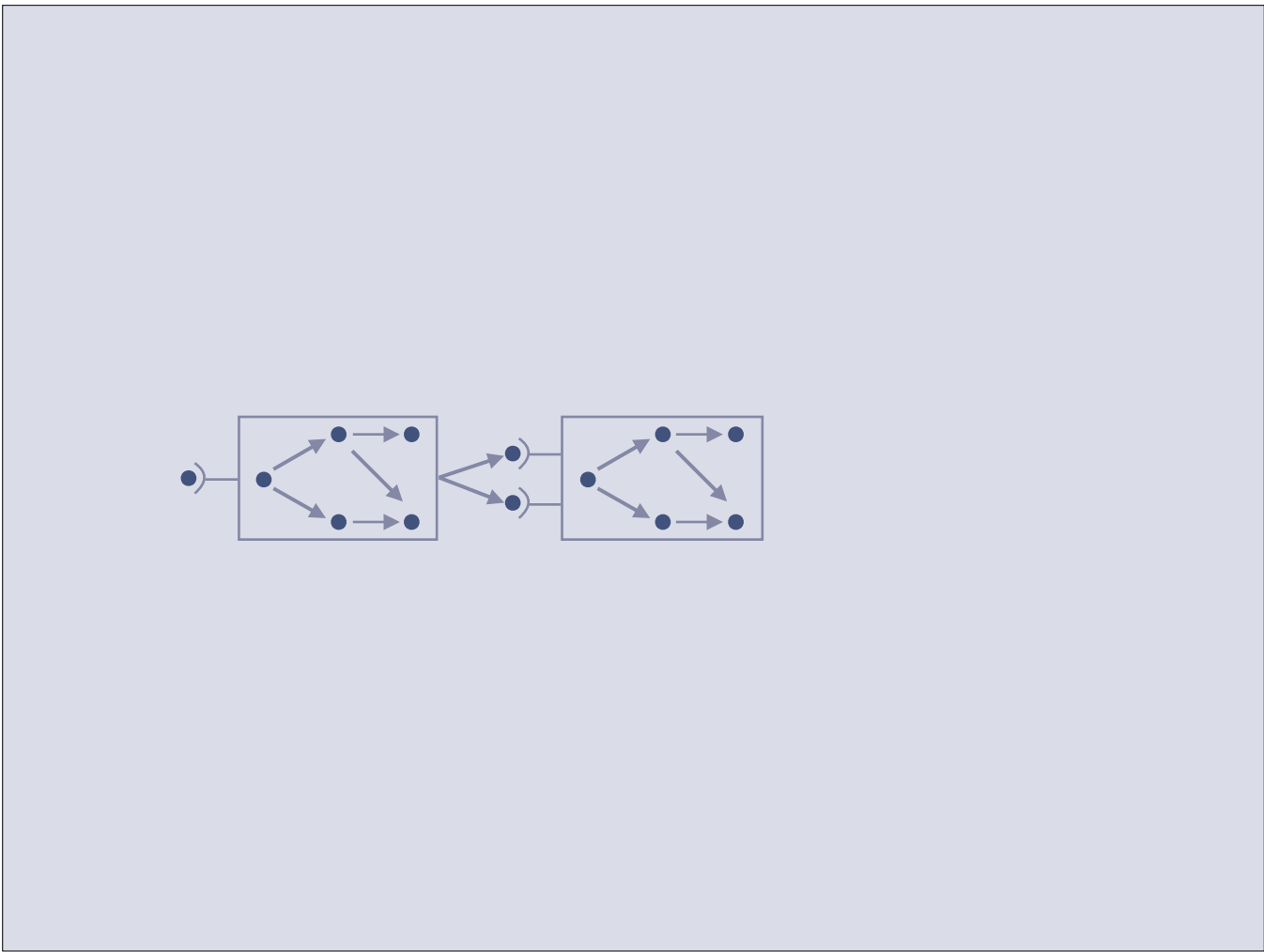
The trick to understanding nesting of your promises is to identify what promise chains will be used collectively, and start to treat minor promises as black boxes.



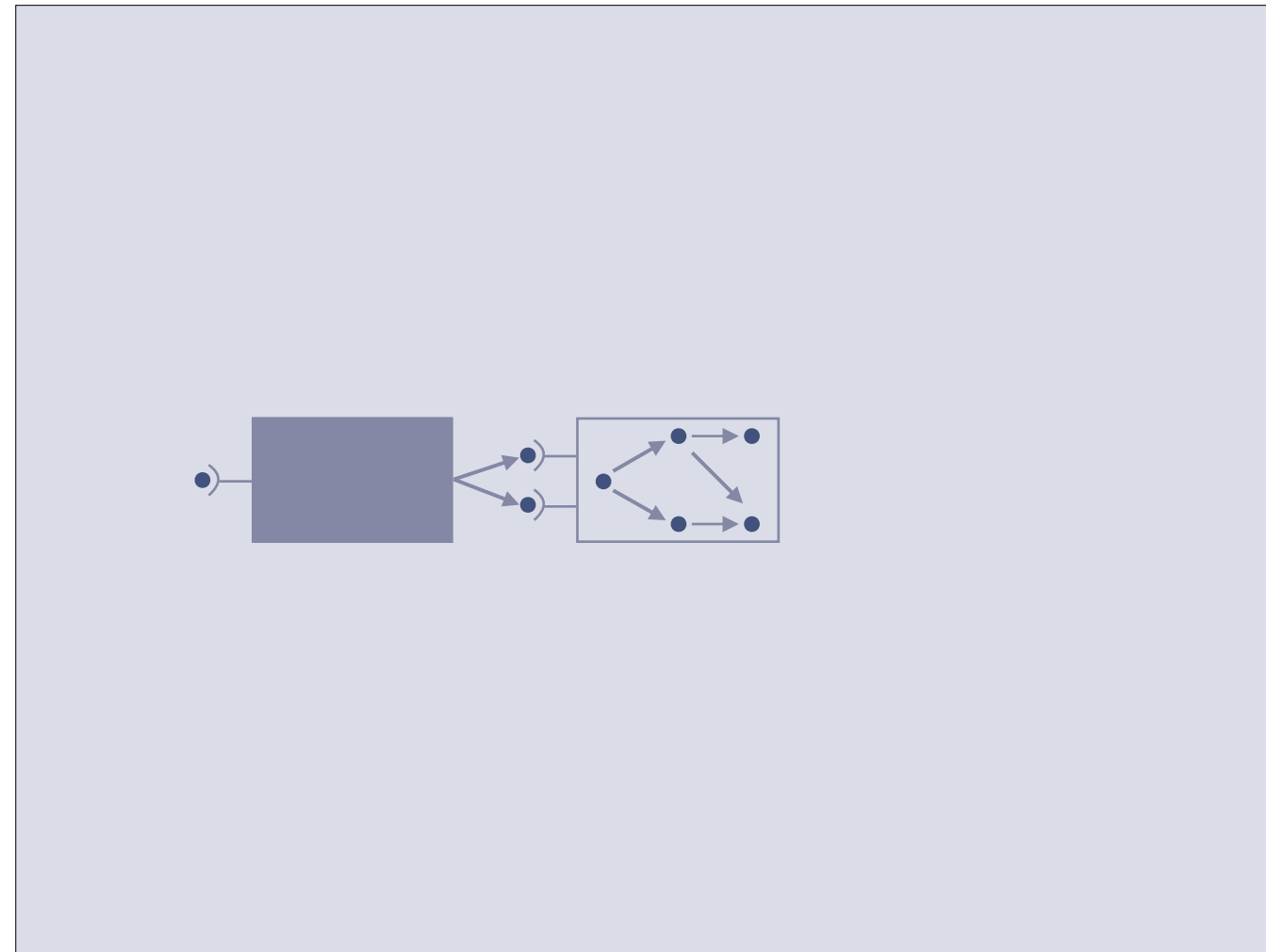
The trick to understanding nesting of your promises is to identify what promise chains will be used collectively, and start to treat minor promises as black boxes.



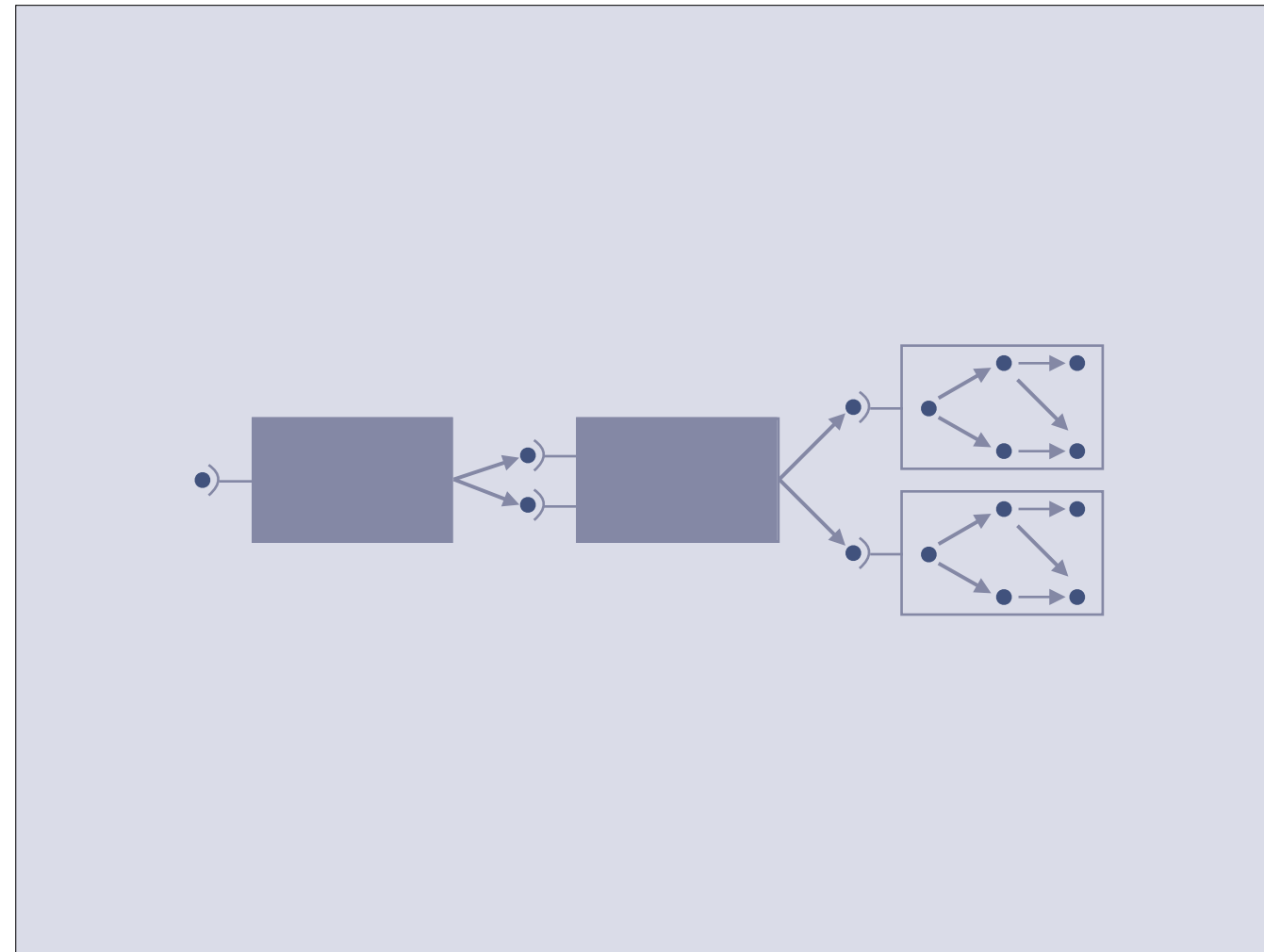
Now that we have the black box figured out, we can compose a collection of subchains into a very concise black box process flow



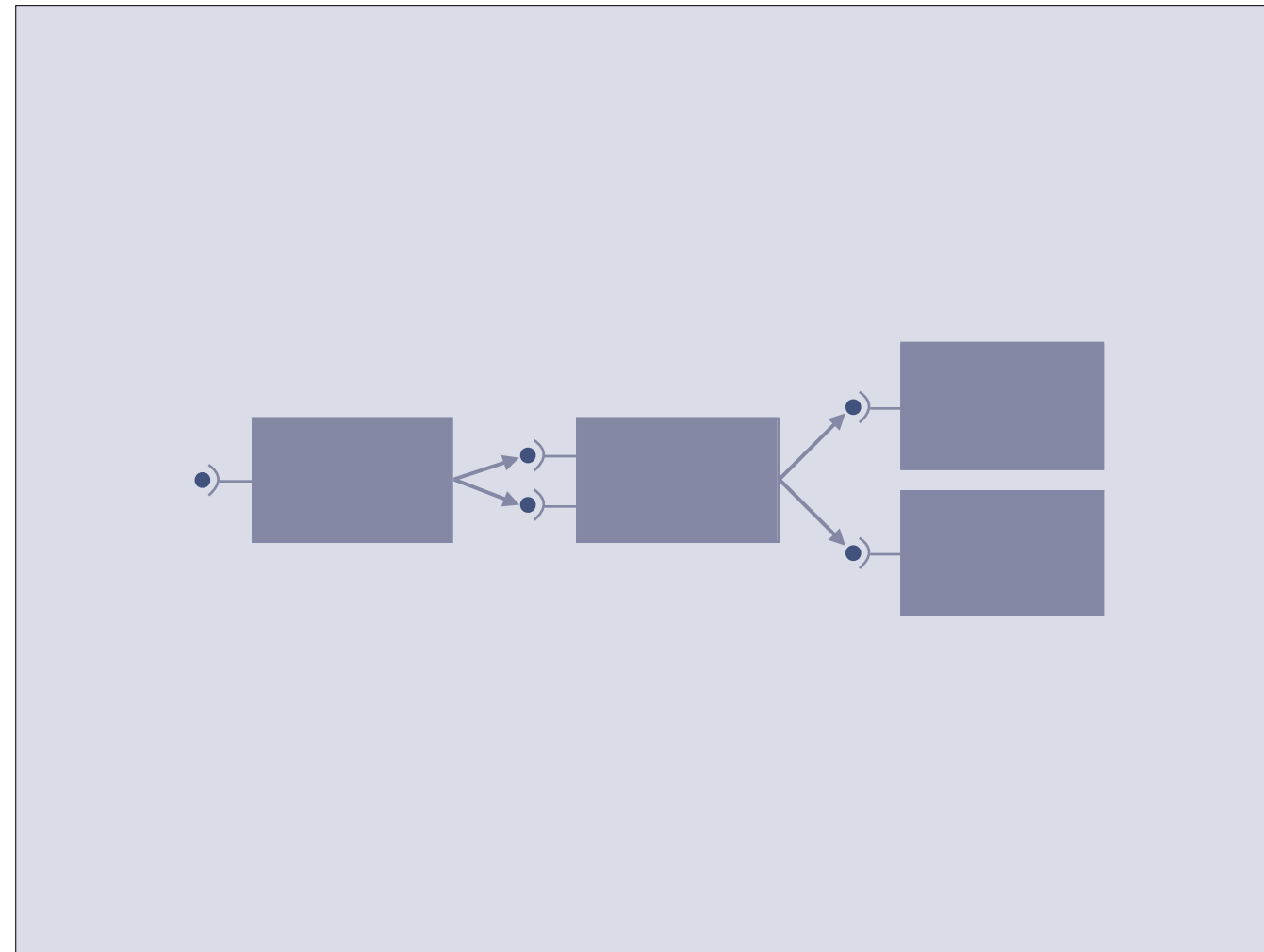
Now that we have the black box figured out, we can compose a collection of subchains into a very concise black box process flow



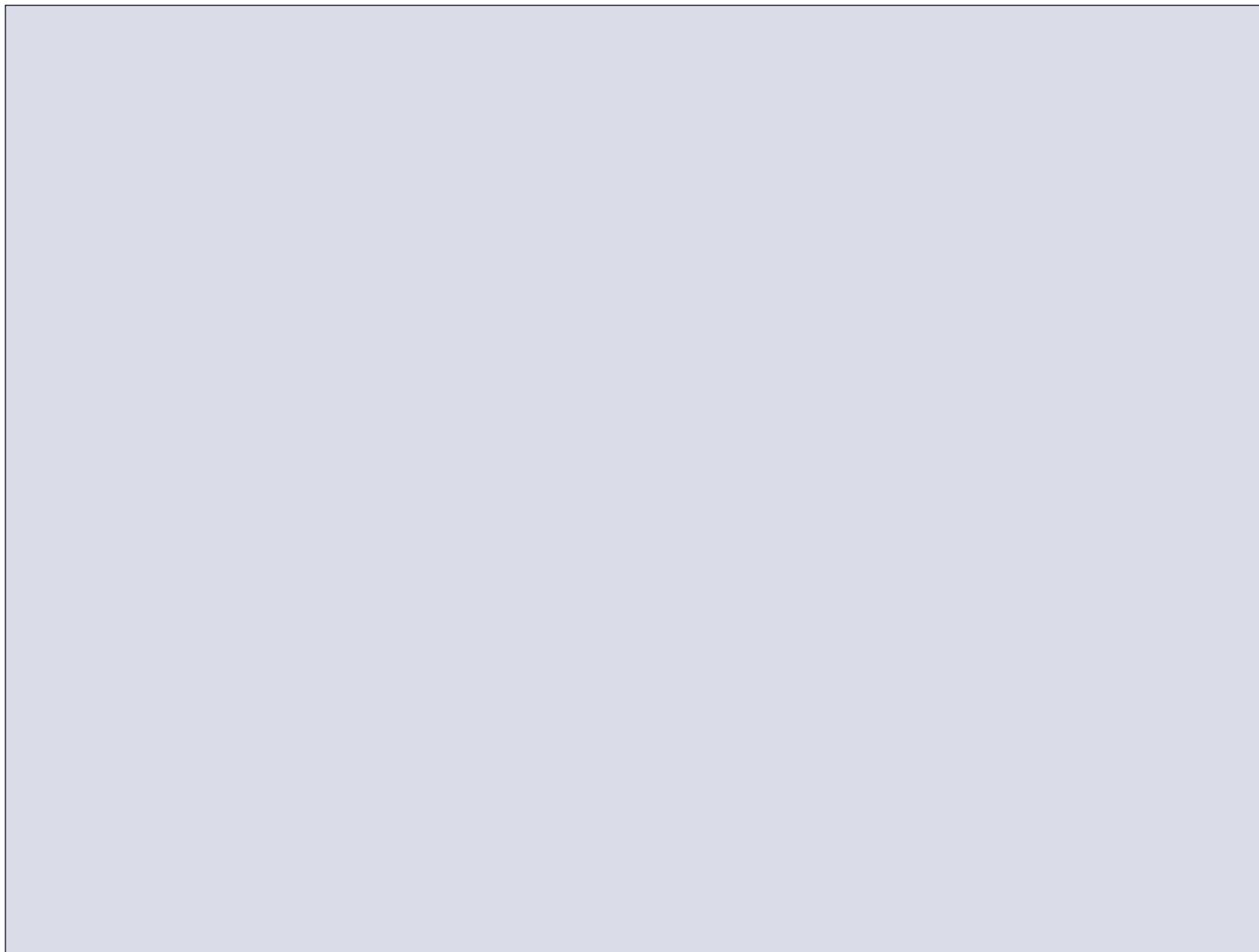
Now that we have the black box figured out, we can compose a collection of subchains into a very concise black box process flow



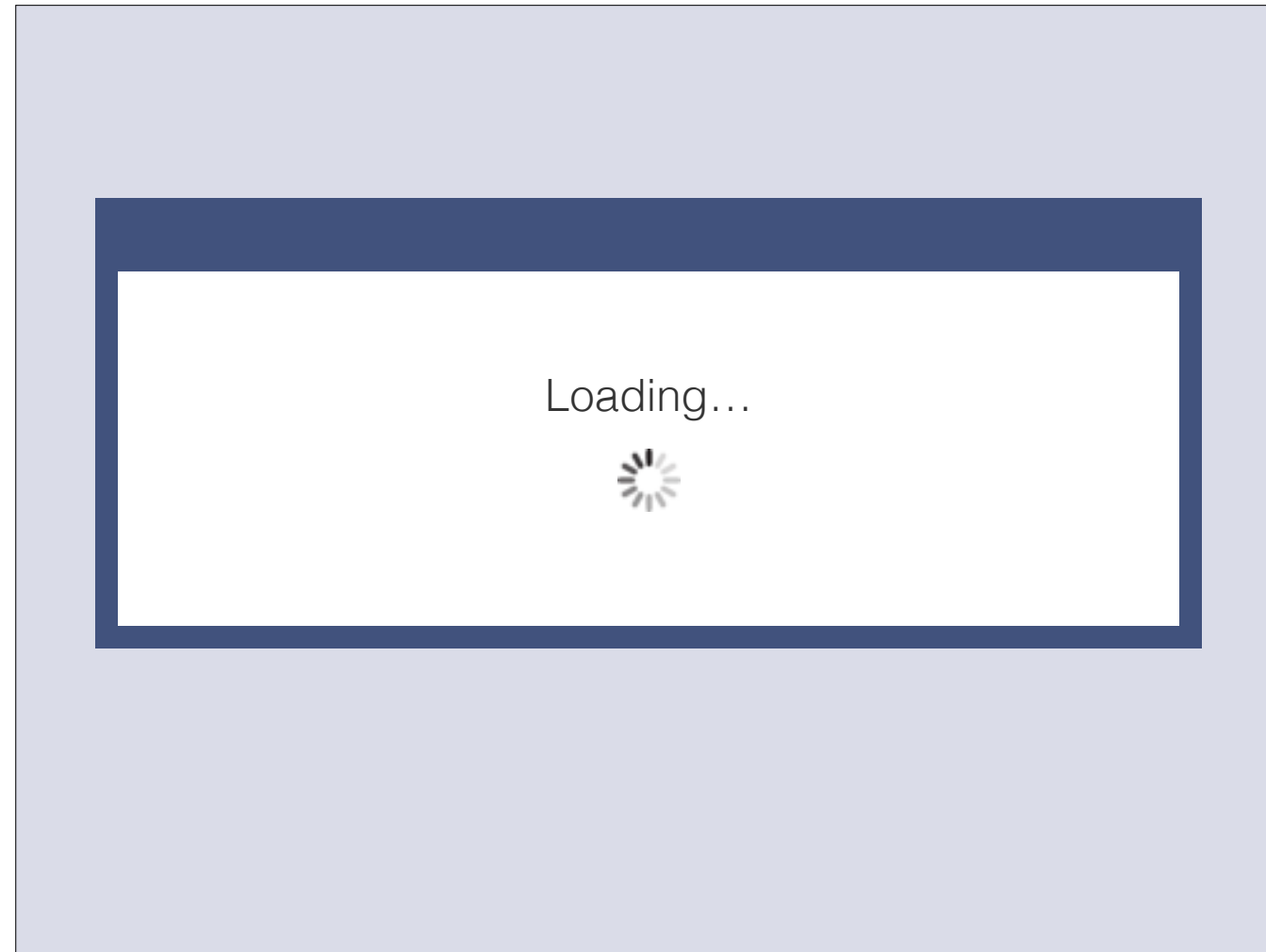
Now that we have the black box figured out, we can compose a collection of subchains into a very concise black box process flow



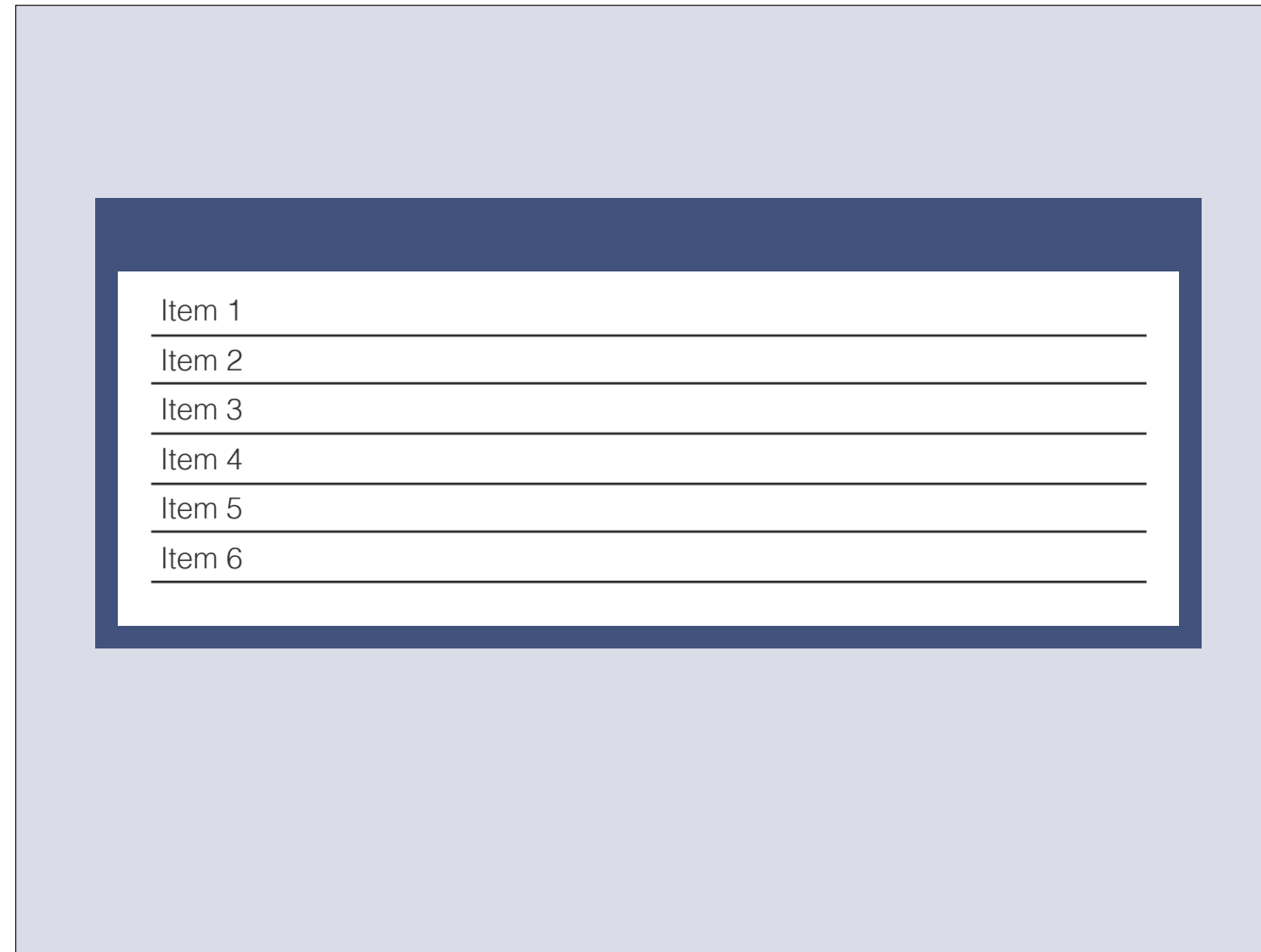
Now that we have the black box figured out, we can compose a collection of subchains into a very concise black box process flow



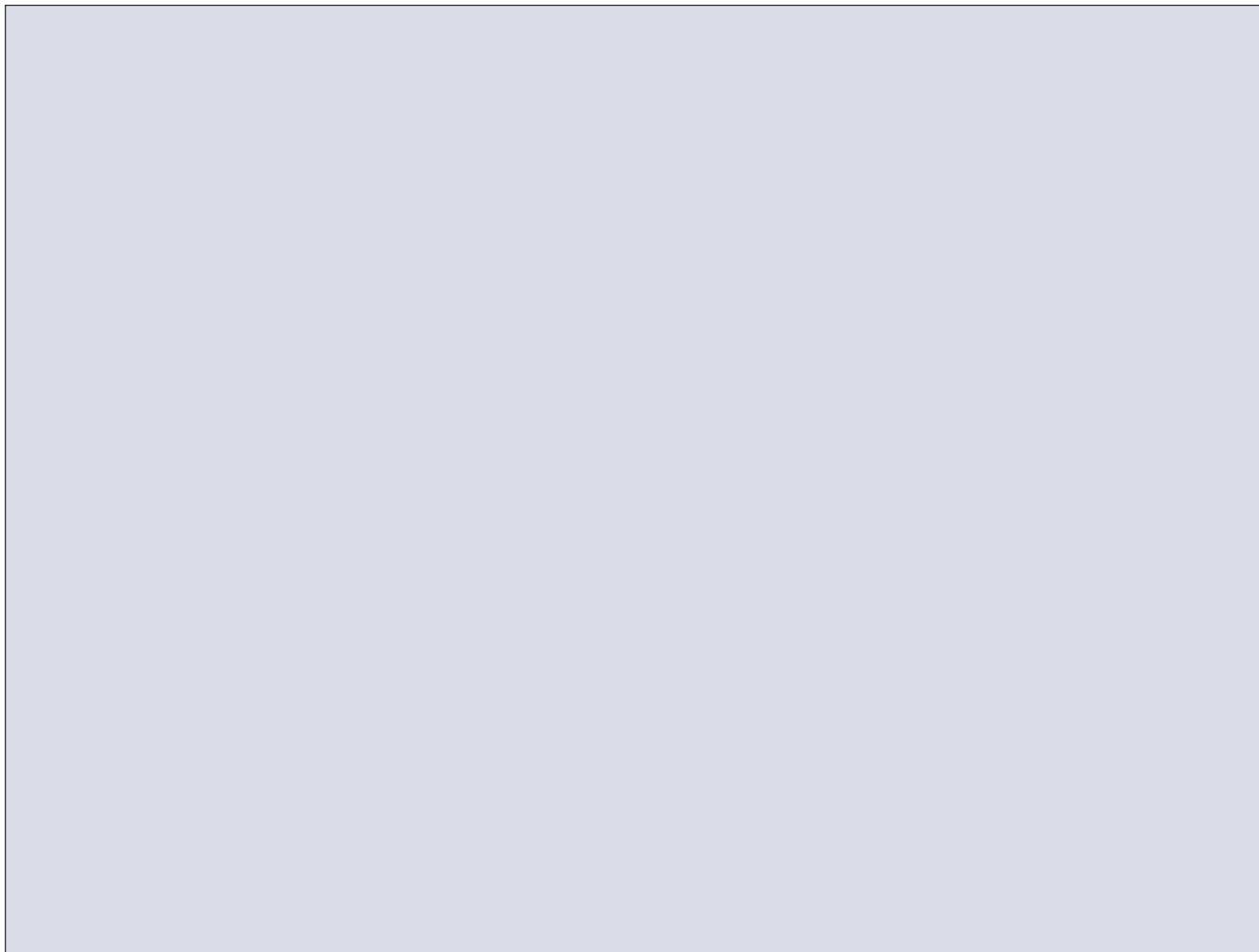
This modal display outlines one common interface problem that our promise approach can help us solve



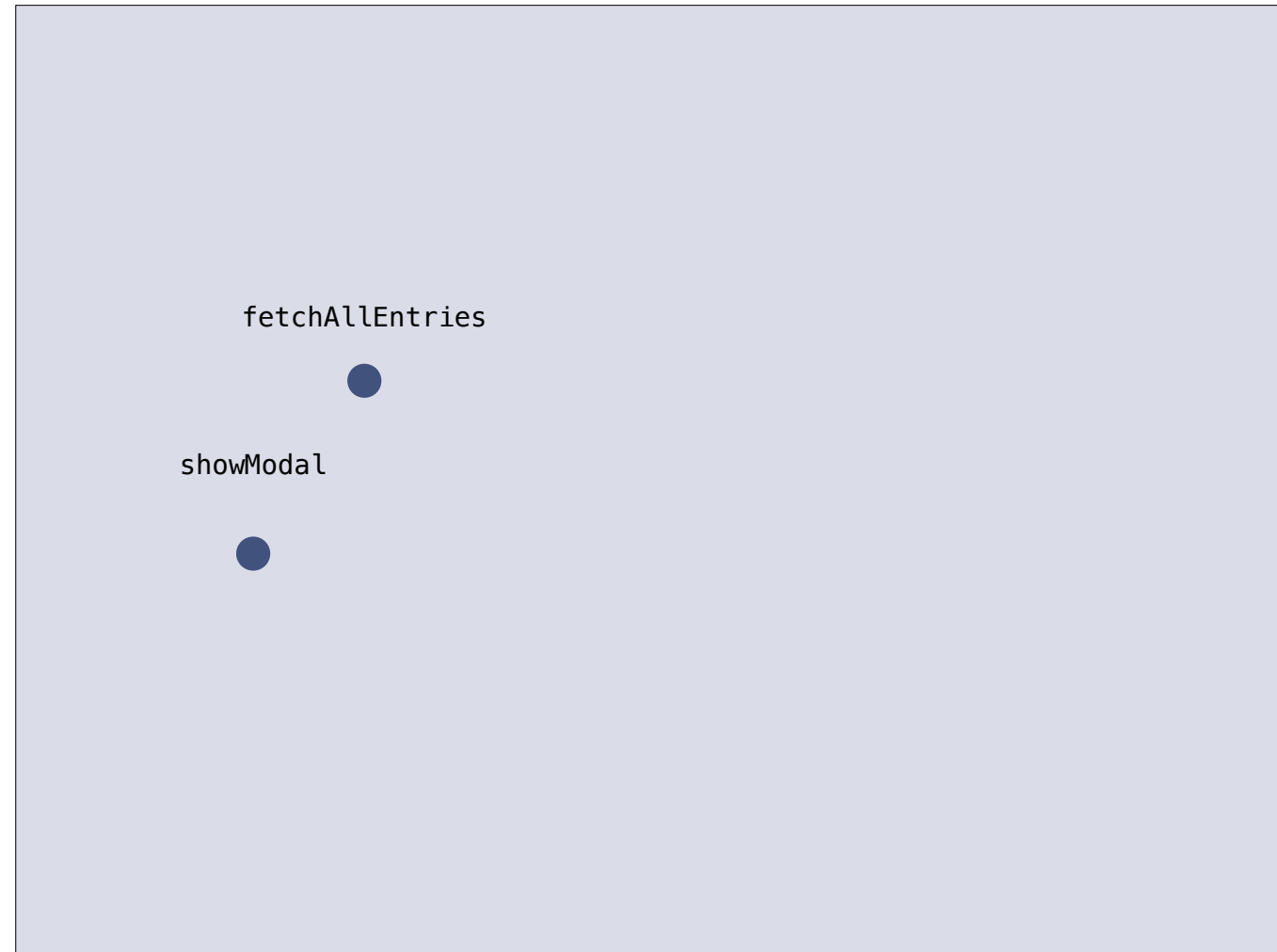
This modal display outlines one common interface problem that our promise approach can help us solve



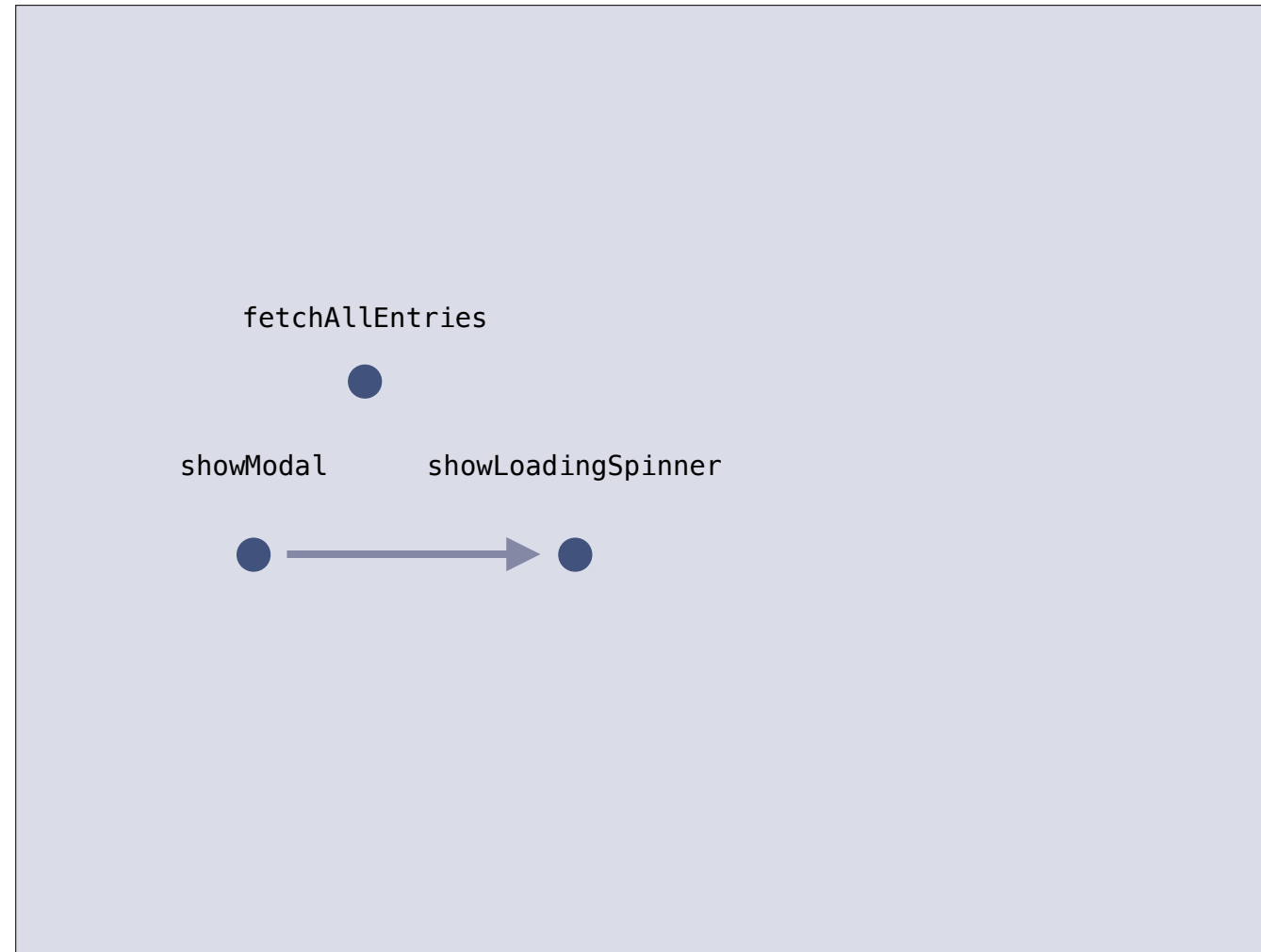
This modal display outlines one common interface problem that our promise approach can help us solve



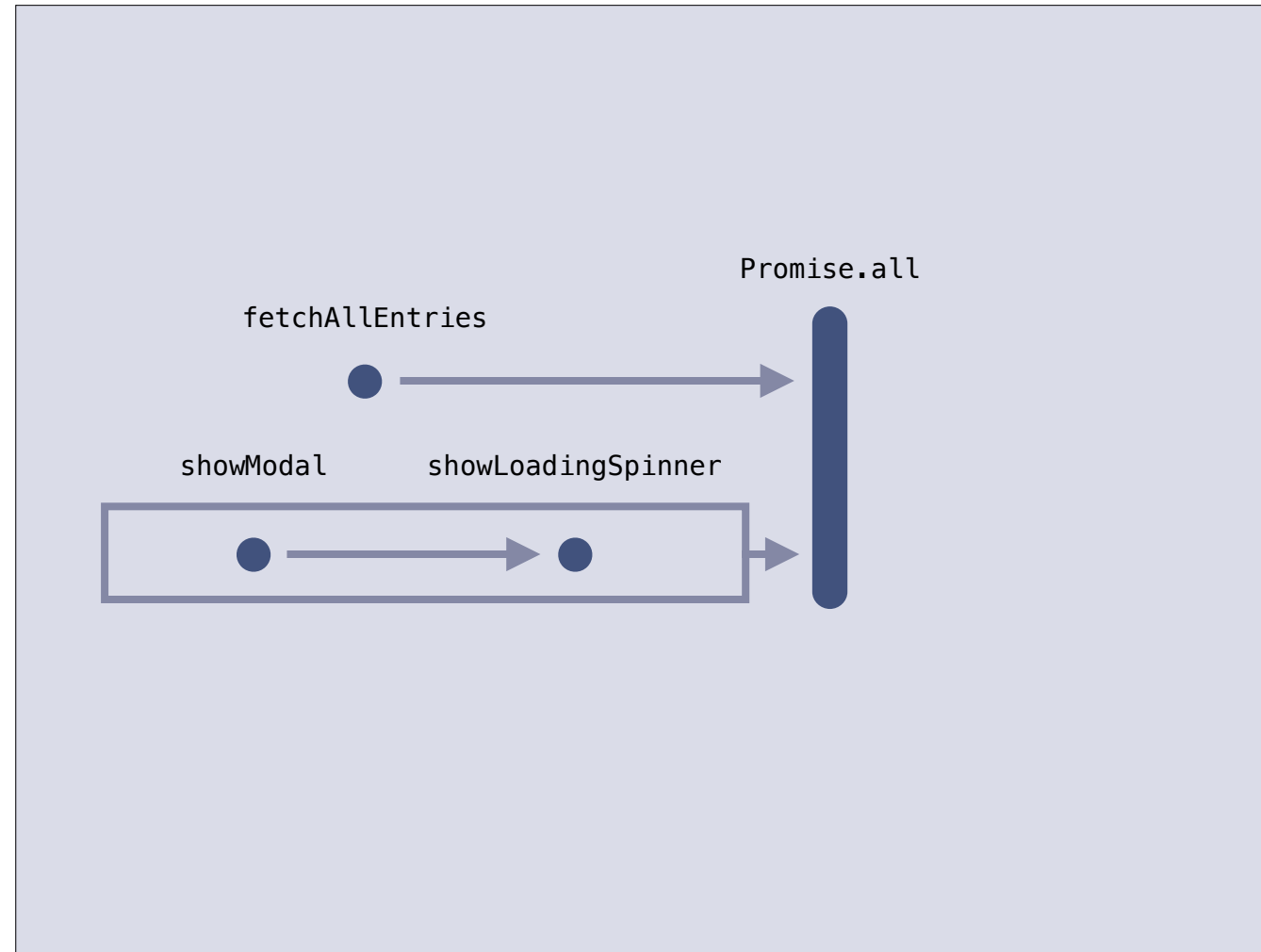
Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow



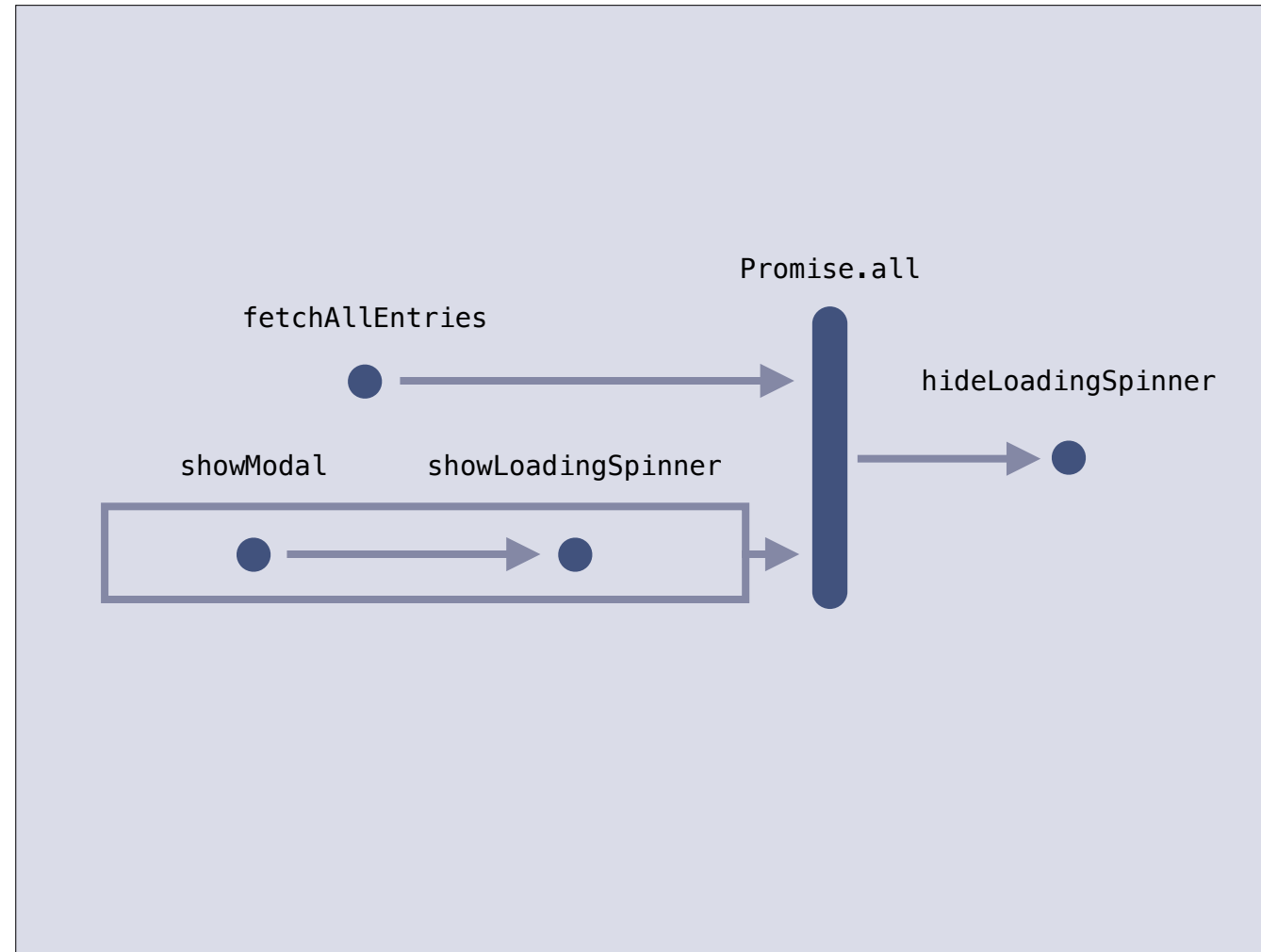
Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow



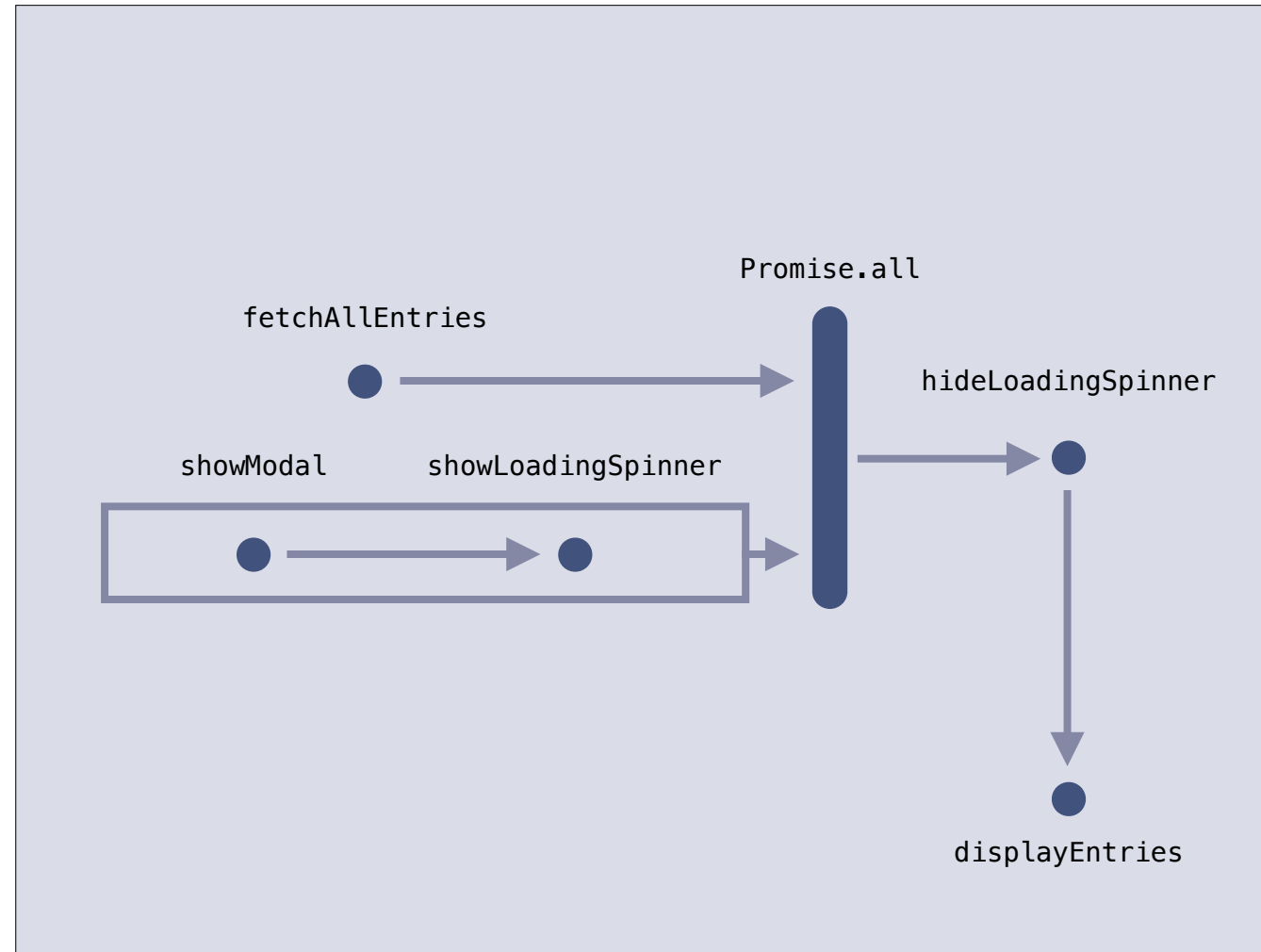
Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow



Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow



Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow



Note the function call, and follow the paths of the diagram to see how the different promises can each step through the entrance animation process flow

```
function fetchAllEntries() {  
  return new Promise(function(resolve, reject) {  
    request({  
      url: 'https://some.important.resource/entries',  
      method: 'GET',  
      success: resolve,  
      error: reject  
    });  
  });  
}
```

Similar to before, we can isolate the fetchAllEntries method
We wrap this method within a promise

```
function showModal() {  
  return $.Velocity.animate(this.$box, {  
    opacity: 1,  
    width: '100%',  
    height: '100%'  
  }, {  
    duration: 1000,  
    easing: 'easeInOut'  
  });  
}
```

Next, we need to show the modal. We have selected \$.Velocity to animate with, and have shimmed in our promise library into the library as a dependency


```
function showLoadingSpinner() {  
    var $spinner = this.$spinner;  
  
    return $.Velocity.animate($spinner, {  
        opacity: 1  
    }, {  
        duration: 500  
    })  
    .then(function() {  
        $spinner.addClass('isActive');  
    });  
}
```

Notice how we use velocity again, and actually add in another chain in the link, creating a small yet still evident subchain

```
function hideLoadingSpinner() {  
    var $spinner = this.$spinner;  
  
    return $.Velocity.animate($spinner, {  
        opacity: 0  
    }, {  
        duration: 500  
    })  
    .then(function() {  
        $spinner.removeClass('isActive');  
    });  
}
```

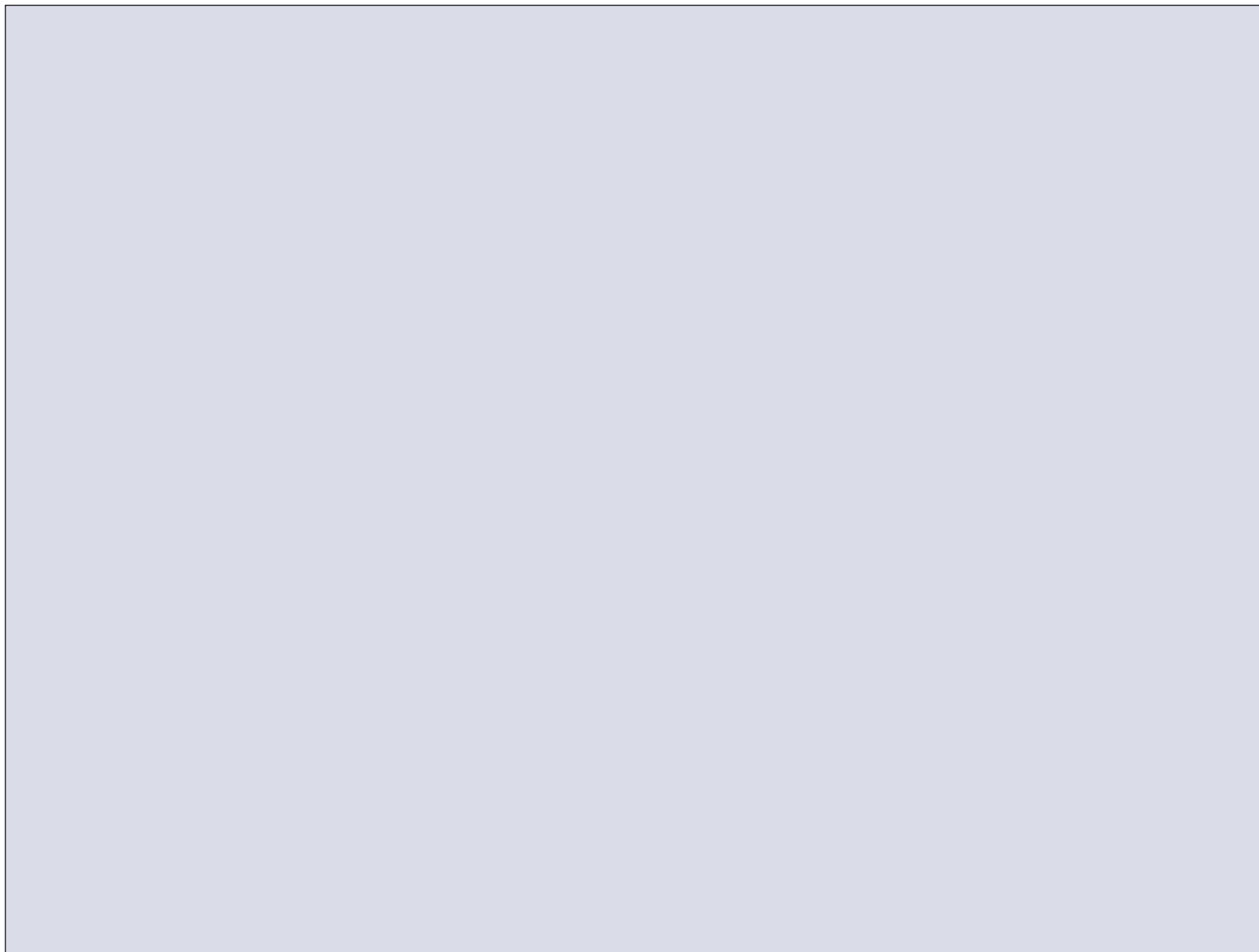
Notice how we use velocity again, and actually add in another chain in the link, creating a small yet still evident subchain

```
function displayEntries() {  
  // TODO: Display the entries  
  var $contents = this.$contents;  
  
  return $.Velocity.animate($contents, {  
    opacity: 1  
  }, {  
    duration: 500  
  });  
  .then(function() {  
    $contents.addClass('isActive');  
  });  
}
```

Notice how we use velocity again, and actually add in another chain in the link, creating a small yet still evident subchain

```
function transitionToPage() {  
  return Promise.all([  
    this.fetchAllEntries(),  
    this.showModal()  
      .then(this.showLoadingSpinner.bind(this))  
  ])  
  .then(this.hideLoadingSpinner.bind(this))  
  .then(this.displayEntries.bind(this));  
}
```

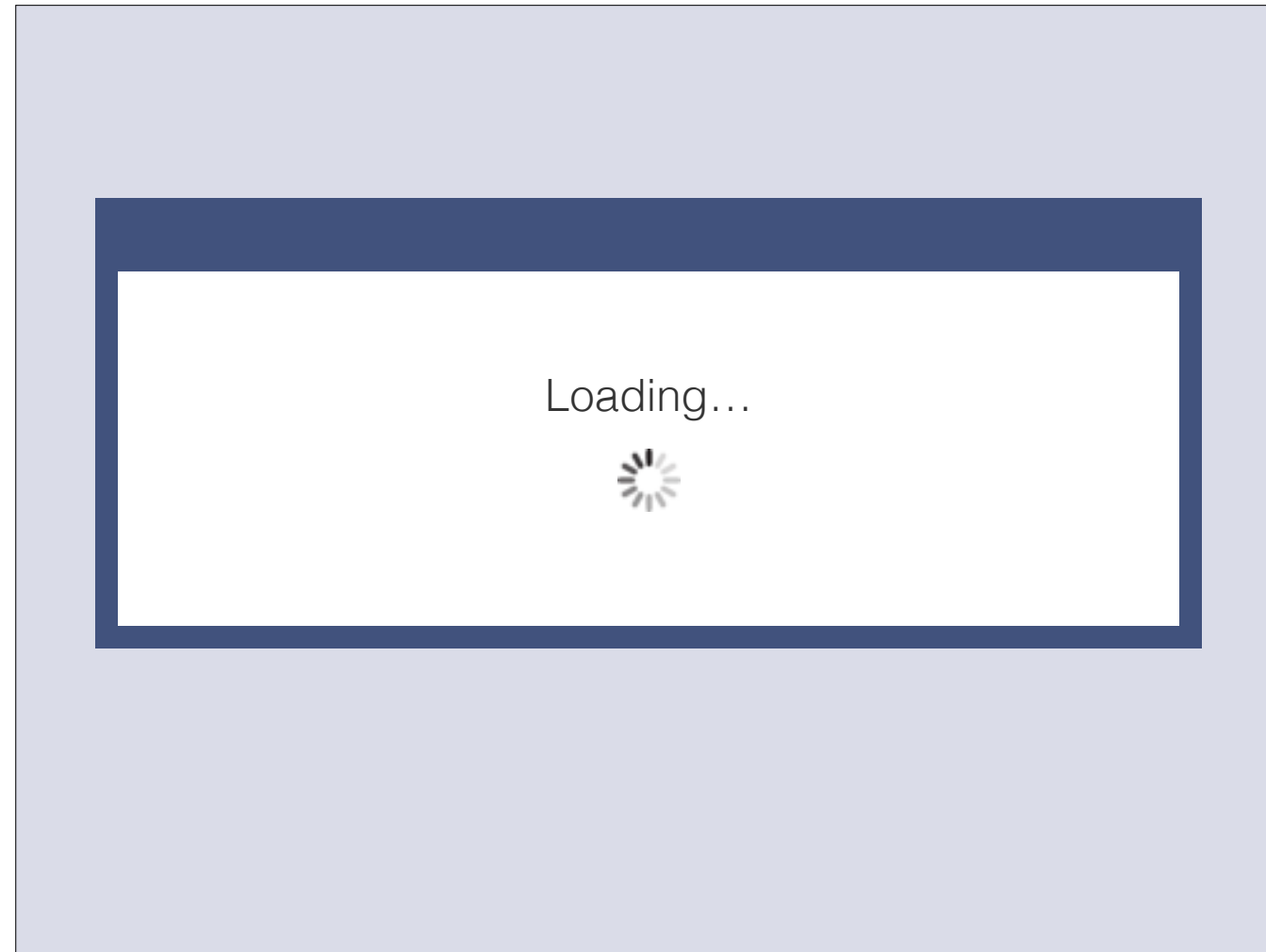
The final synopsis of the promise chain reveals the degree of nesting. Note that each of the more complex next components could themselves



Now we can see our entrance animation again, but with all of the methods called out individually



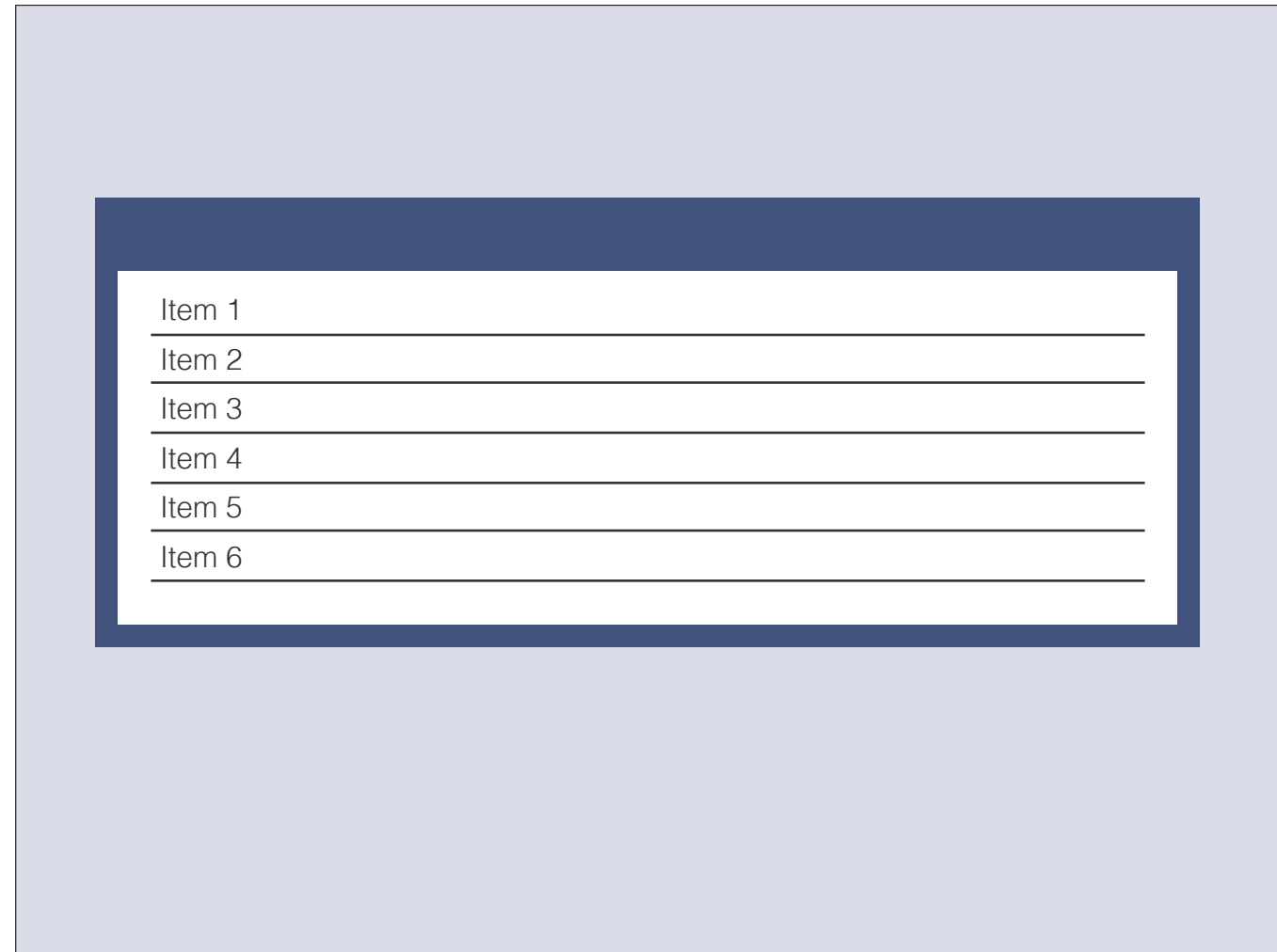
Now we can see our entrance animation again, but with all of the methods called out individually



Now we can see our entrance animation again, but with all of the methods called out individually



Now we can see our entrance animation again, but with all of the methods called out individually



Now we can see our entrance animation again, but with all of the methods called out individually

- Interface
- Error Bubbling
- Synopsis
- Rejection
- Nesting
- Looping

Now that we've learned nesting, we can apply this a second time into looping



Looping

Dynamic Chaining

Looping is effectively a dynamic or repeated form of nesting

Problem

- Many repeated steps
- Steps have many small steps

Looping is something that is the most complex pattern, and one that takes me a long time to wrap my brain around and also shows up very infrequently. When it does show up though, having a pattern to follow has been integral

Solution

- Queue it up
- Gracefully handle nested promises

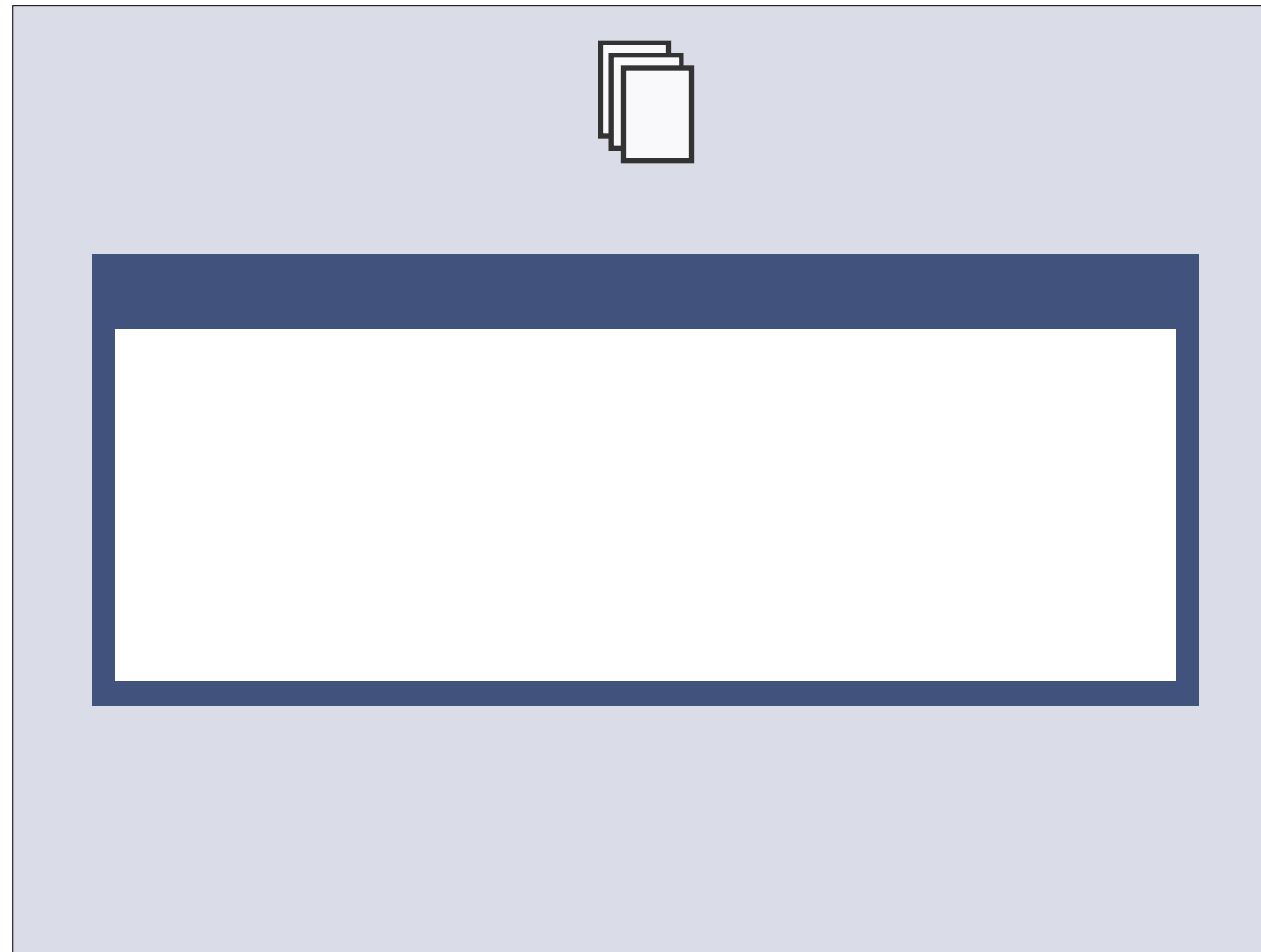
These processes are automatic. We must solve it in a way that doesn't require us to interact.

Automatic means that the solution must also gracefully handle steps in between. Queue up a set of these, or reduce all of your events down to one promise



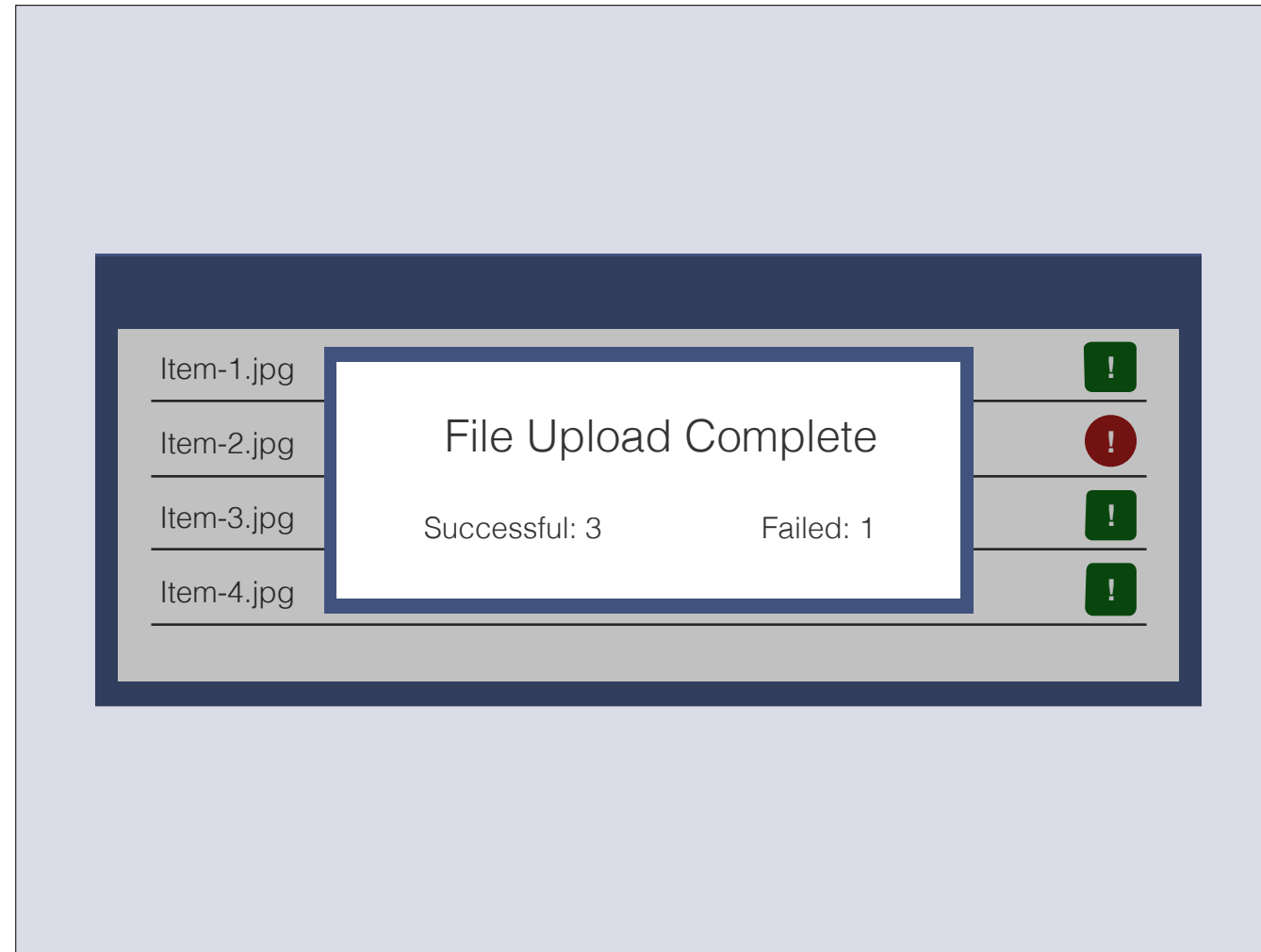
Consider a strict RESTful system. I'm sure everyone's done this and had both great success and great failure. This example is derived from the need to upload a file and also perform strict RESTful calls. In this example, we have a requirement to load each file individually (in serial)

Note the different states our interface goes when a drag & drop action occurs.



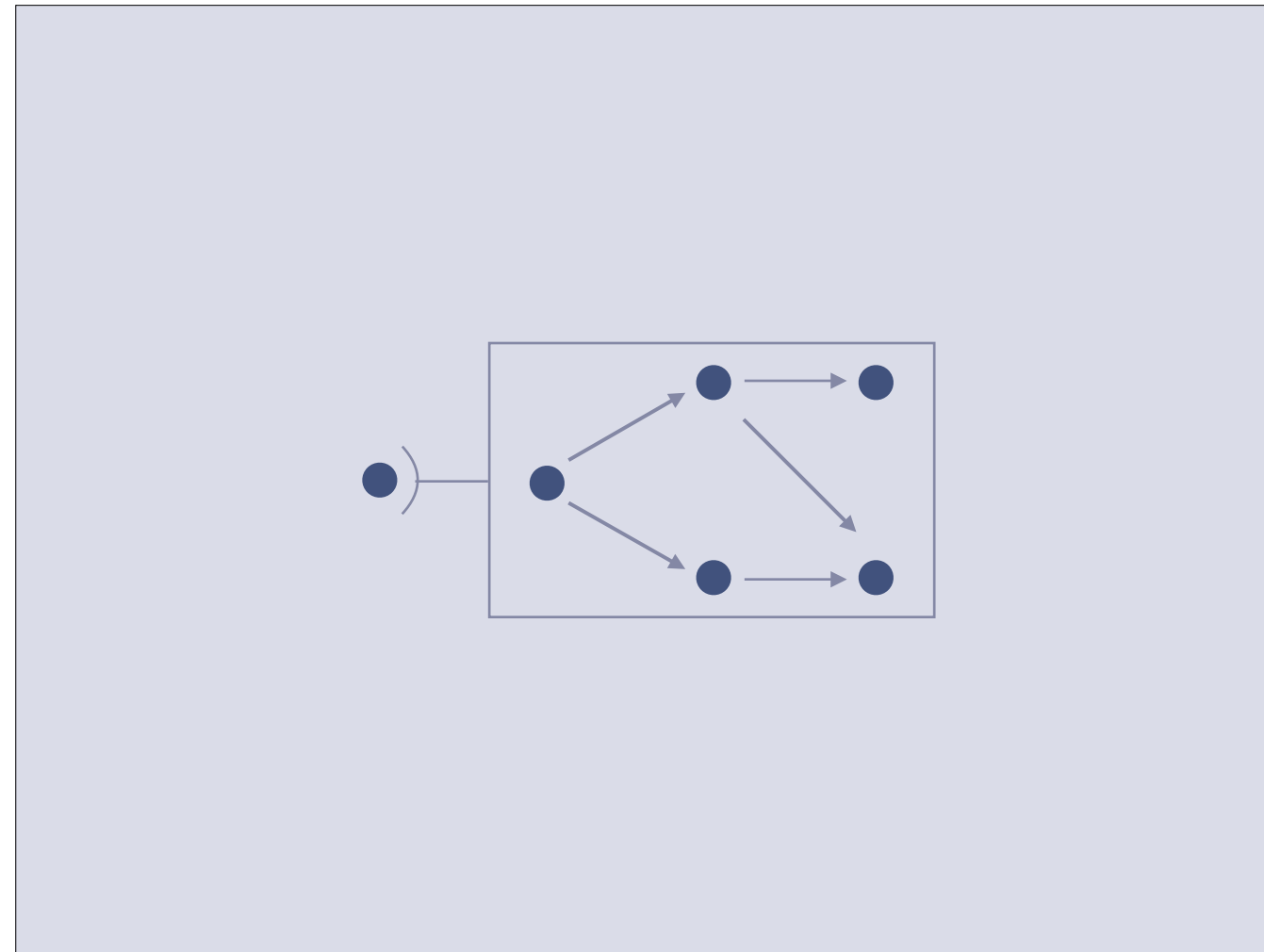
Consider a strict RESTful system. I'm sure everyone's done this and had both great success and great failure. This example is derived from the need to upload a file and also perform strict RESTful calls. In this example, we have a requirement to load each file individually (in serial)

Note the different states our interface goes when a drag & drop action occurs.

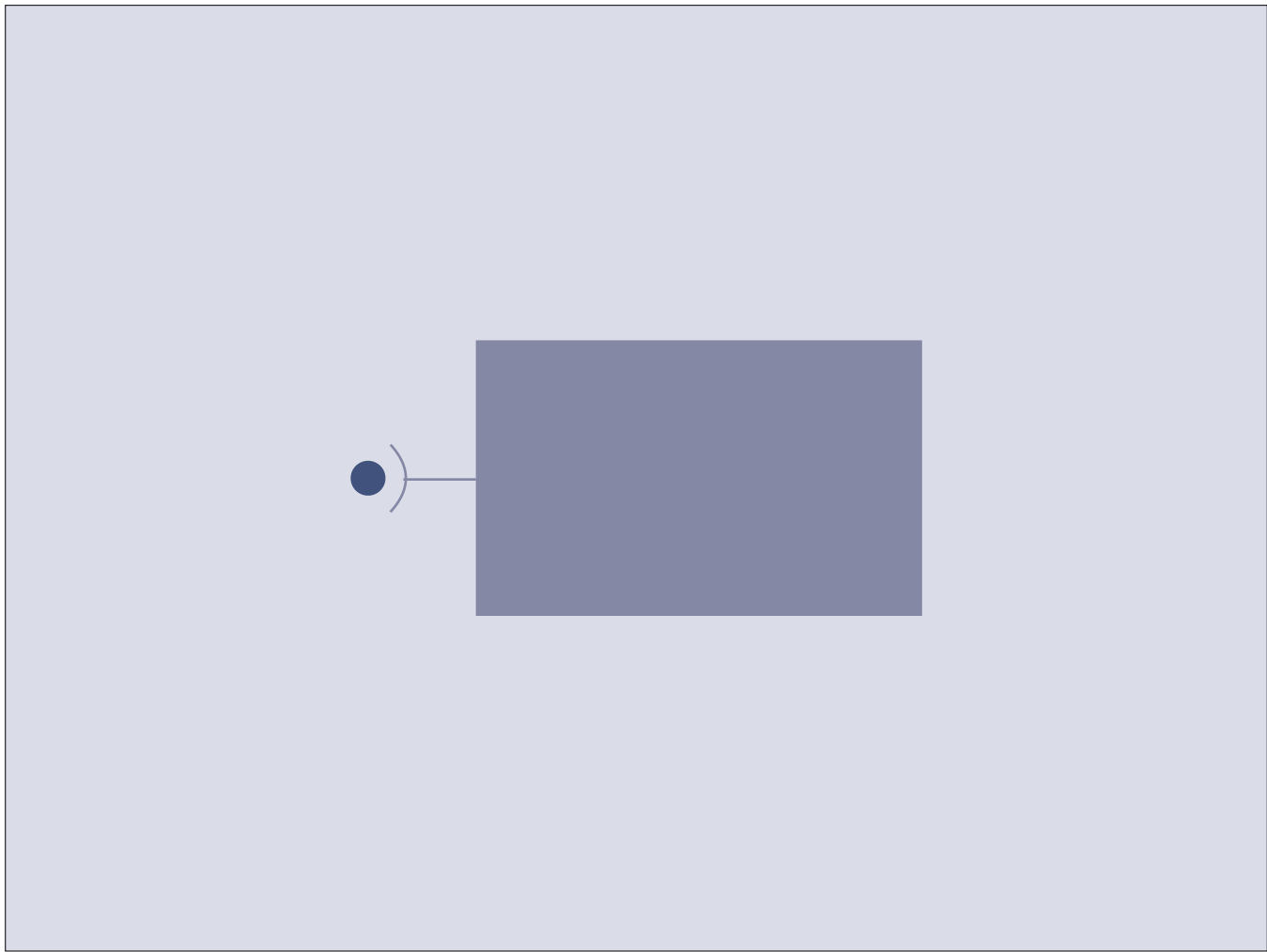


Consider a strict RESTful system. I'm sure everyone's done this and had both great success and great failure. This example is derived from the need to upload a file and also perform strict RESTful calls. In this example, we have a requirement to load each file individually (in serial)

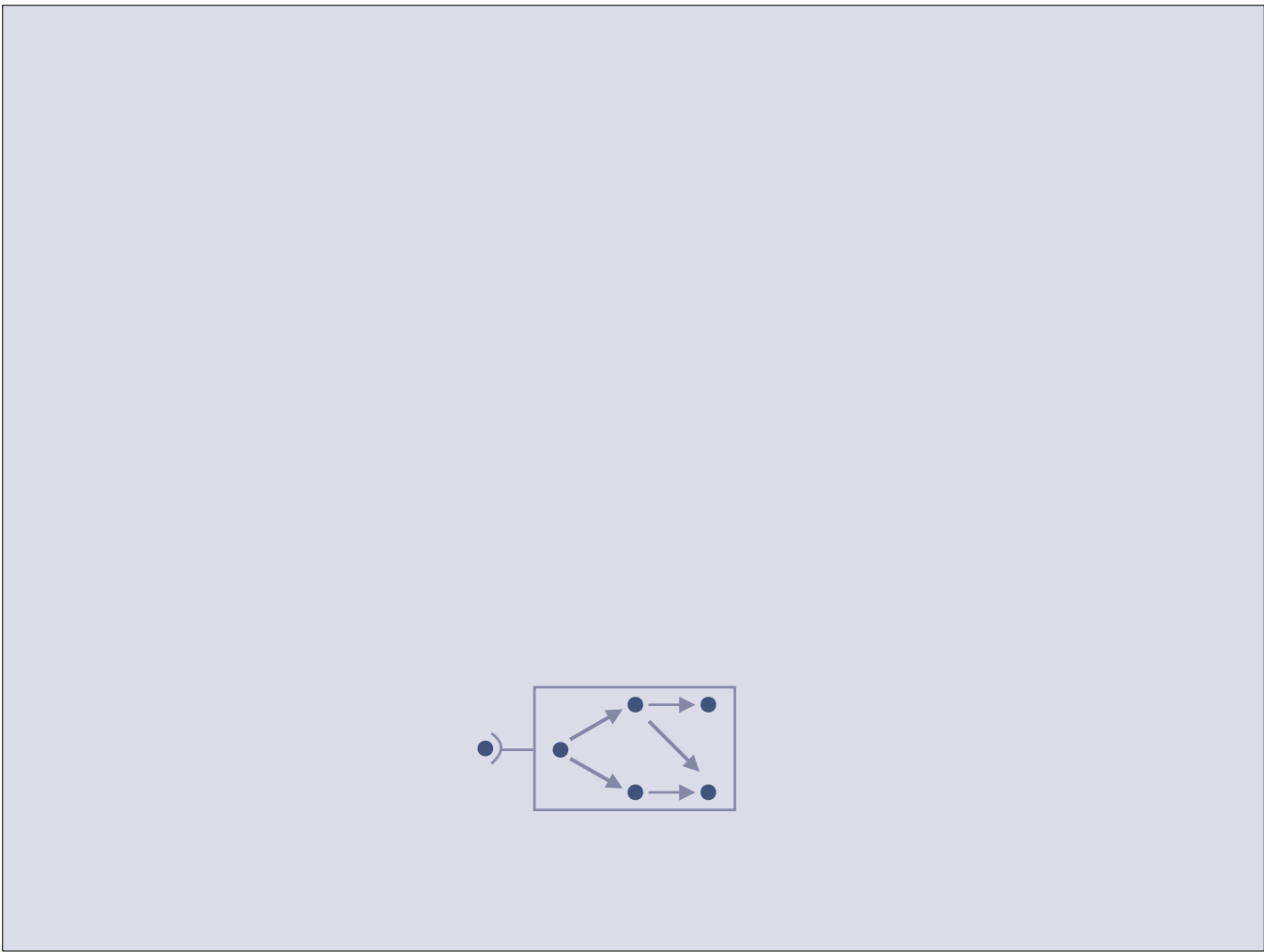
Note the different states our interface goes when a drag & drop action occurs.



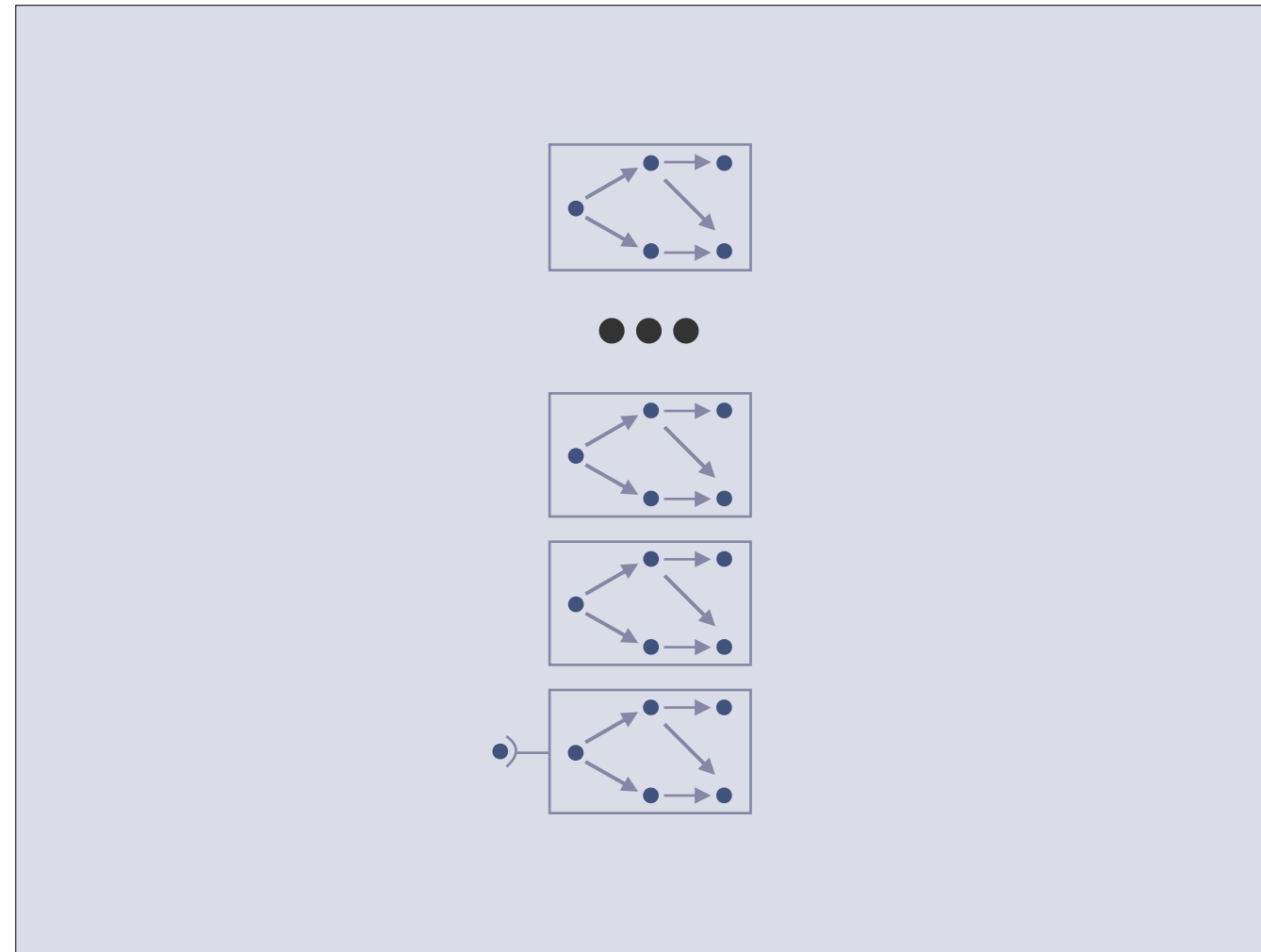
Here's a refresher again, we're going to need to remember the black boxing of our processes.



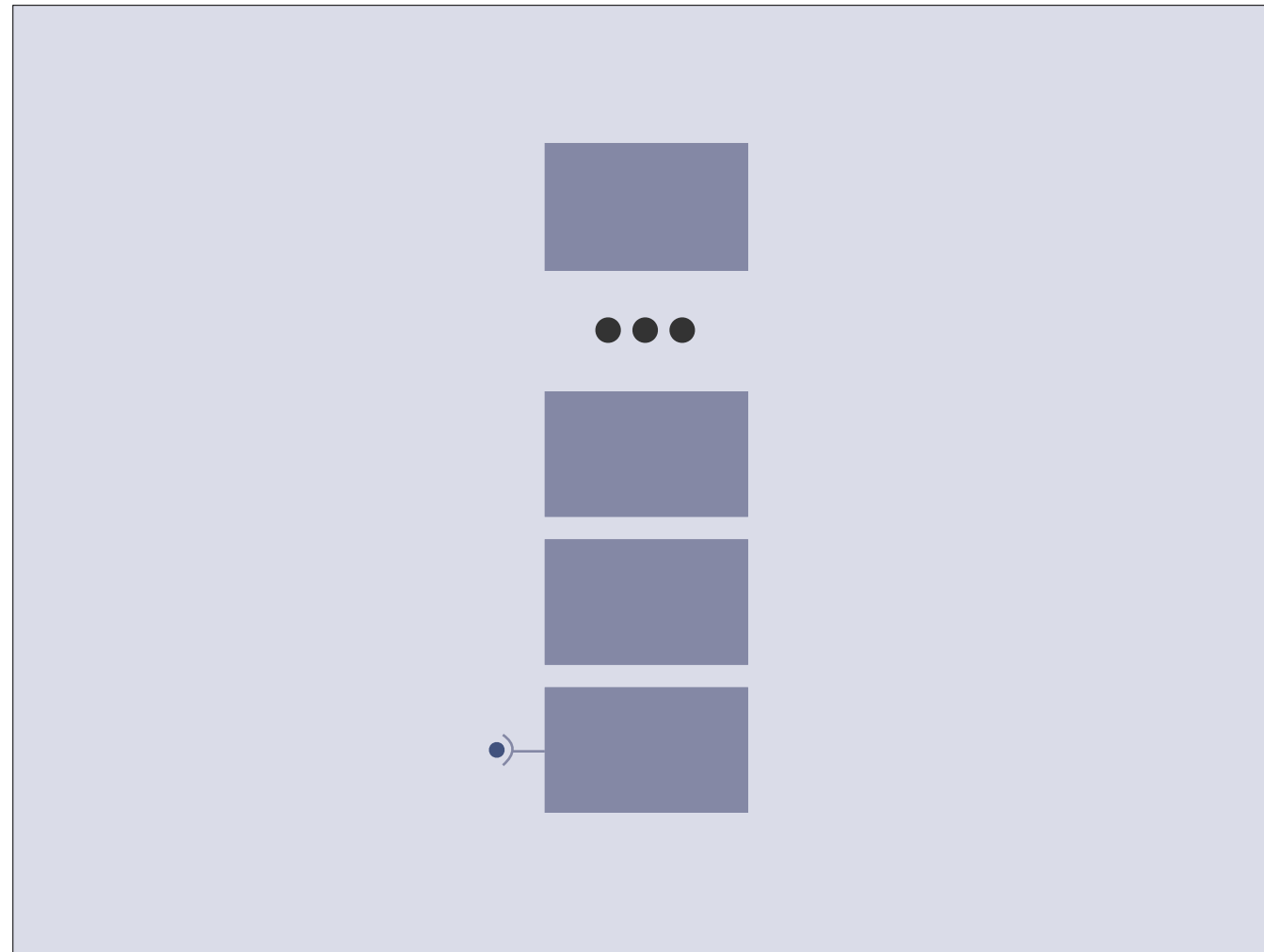
Here's a refresher again, we're going to need to remember the black boxing of our processes.



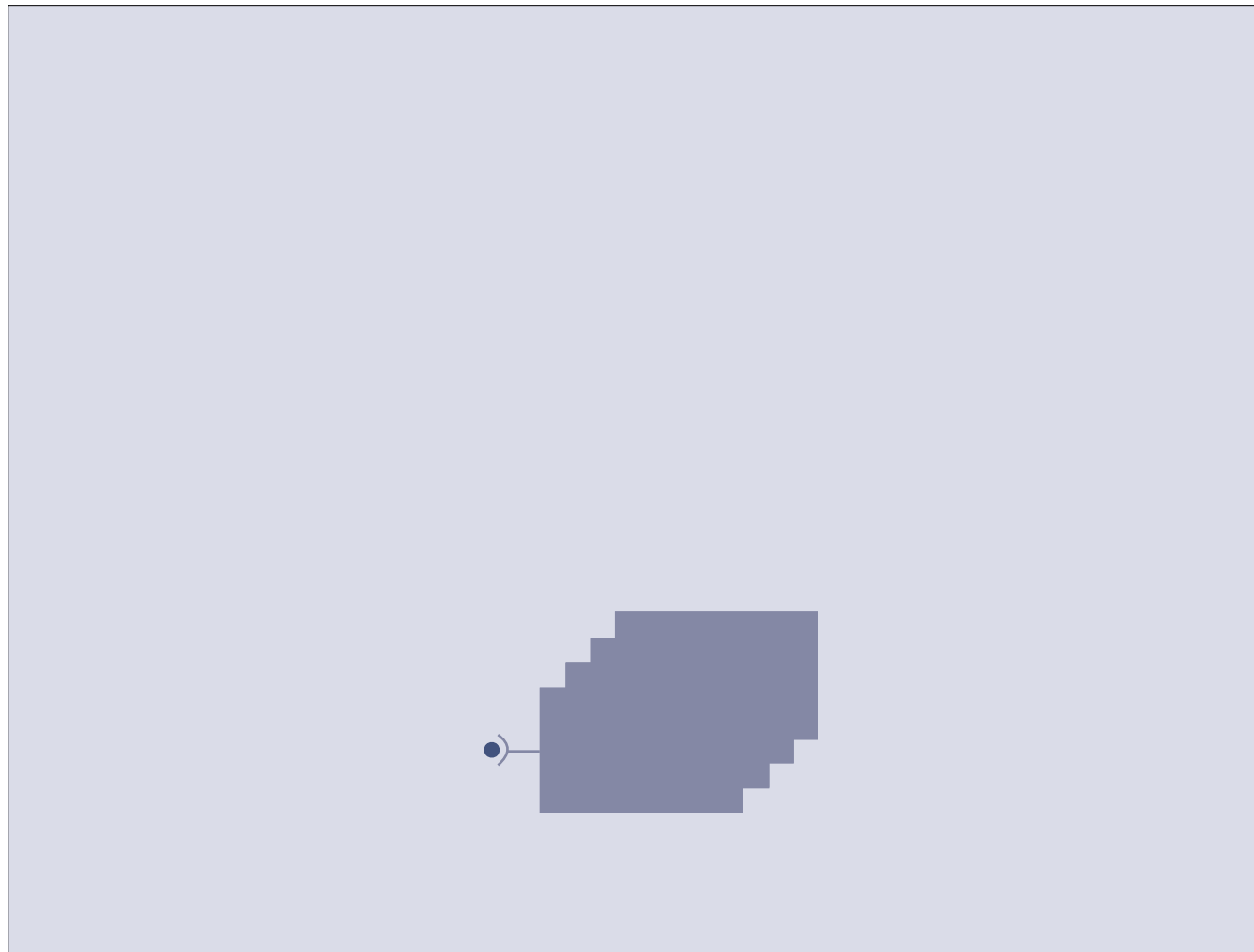
The trick is to set up the collection of subchains, and from those subchains, treat each one as just one link in the chain leading us to the completion of our large loop promise.



The trick is to set up the collection of subchains, and from those subchains, treat each one as just one link in the chain leading us to the completion of our large loop promise.



The trick is to set up the collection of subchains, and from those subchains, treat each one as just one link in the chain leading us to the completion of our large loop promise.



The trick is to set up the collection of subchains, and from those subchains, treat each one as just one link in the chain leading us to the completion of our large loop promise.

```

function processFile(file, failedFiles) {
  var model;

  // TODO: Show the loading indicator, update the file name

  return volumeService.get()
    .then(function(volumeModel) {
      model = volumeModel;
      model.title = file.name;

      return volumeService.save(model);
    })
    .then(function() {
      return volumeService.uploadFile(model.uri, file);
    })
    .then(function() {
      // TODO: Hide the loading indicator
      // TODO: Show the success indicator
    }, function() {
      return volumeService.delete(model)
        .then(function() {
          failedFiles.push(file);

          // TODO: Show the failed indicator
        });
    });
}

```

In this example application, assume what we're uploading for is a medical device such as an MRI or CT scan that uses 3D images that need to be broken down into 2D image slices for processing on the browsers. In this example, we upload a file and perform the upload.

For this we're using angular.

Group 1: We get the new volume entry, and update the model to then save it.

Group 2: After this, we upload the volumeService, and upload the file.

Group 3: Do all our success stuff, or if failed, don't ACTUALLY stop, we need to prep the system to mark the files as failed so we can inform the user

```

function processFile(file, failedFiles) {
  var model;

  // TODO: Show the loading indicator, update the file name

  return volumeService.get()
    .then(function(volumeModel) {
      model = volumeModel;
      model.title = file.name;

      return volumeService.save(model);
    })
    .then(function() {
      return volumeService.uploadFile(model.uri, file);
    })
    .then(function() {
      // TODO: Hide the loading indicator
      // TODO: Show the success indicator
    }, function() {
      return volumeService.delete(model)
        .then(function() {
          failedFiles.push(file);

          // TODO: Show the failed indicator
        });
    });
}

```

In this example application, assume what we're uploading for is a medical device such as an MRI or CT scan that uses 3D images that need to be broken down into 2D image slices for processing on the browsers. In this example, we upload a file and perform the upload.

For this we're using angular.

Group 1: We get the new volume entry, and update the model to then save it.

Group 2: After this, we upload the volumeService, and upload the file.

Group 3: Do all our success stuff, or if failed, don't ACTUALLY stop, we need to prep the system to mark the files as failed so we can inform the user


```

function processFile(file, failedFiles) {
  var model;

  // TODO: Show the loading indicator, update the file name

  return volumeService.get()
    .then(function(volumeModel) {
      model = volumeModel;
      model.title = file.name;

      return volumeService.save(model);
    })
    .then(function() {
      return volumeService.uploadFile(model.uri, file);
    })
    .then(function() {
      // TODO: Hide the loading indicator
      // TODO: Show the success indicator
    }, function() {
      return volumeService.delete(model)
        .then(function() {
          failedFiles.push(file);

          // TODO: Show the failed indicator
        });
    });
}

```

In this example application, assume what we're uploading for is a medical device such as an MRI or CT scan that uses 3D images that need to be broken down into 2D image slices for processing on the browsers. In this example, we upload a file and perform the upload.

For this we're using angular.

Group 1: We get the new volume entry, and update the model to then save it.

Group 2: After this, we upload the volumeService, and upload the file.

Group 3: Do all our success stuff, or if failed, don't ACTUALLY stop, we need to prep the system to mark the files as failed so we can inform the user

```

function processFile(file, failedFiles) {
  var model;

  // TODO: Show the loading indicator, update the file name

  return volumeService.get()
    .then(function(volumeModel) {
      model = volumeModel;
      model.title = file.name;

      return volumeService.save(model);
    })
    .then(function() {
      return volumeService.uploadFile(model.uri, file);
    })
    .then(function() {
      // TODO: Hide the loading indicator
      // TODO: Show the success indicator
    }, function() {
      return volumeService.delete(model)
        .then(function() {
          failedFiles.push(file);

          // TODO: Show the failed indicator
        });
    });
}

```

In this example application, assume what we're uploading for is a medical device such as an MRI or CT scan that uses 3D images that need to be broken down into 2D image slices for processing on the browsers. In this example, we upload a file and perform the upload.

For this we're using angular.

Group 1: We get the new volume entry, and update the model to then save it.

Group 2: After this, we upload the volumeService, and upload the file.

Group 3: Do all our success stuff, or if failed, don't ACTUALLY stop, we need to prep the system to mark the files as failed so we can inform the user

```

function processFile(file, failedFiles) {
  var model;

  // TODO: Show the loading indicator, update the file name

  return volumeService.get()
    .then(function(volumeModel) {
      model = volumeModel;
      model.title = file.name;

      return volumeService.save(model);
    })
    .then(function() {
      return volumeService.uploadFile(model.uri, file);
    })
    .then(function() {
      // TODO: Hide the loading indicator
      // TODO: Show the success indicator
    }, function() {
      return volumeService.delete(model)
        .then(function() {
          failedFiles.push(file);

          // TODO: Show the failed indicator
        });
    });
}

```

In this example application, assume what we're uploading for is a medical device such as an MRI or CT scan that uses 3D images that need to be broken down into 2D image slices for processing on the browsers. In this example, we upload a file and perform the upload.

For this we're using angular.

Group 1: We get the new volume entry, and update the model to then save it.

Group 2: After this, we upload the volumeService, and upload the file.

Group 3: Do all our success stuff, or if failed, don't ACTUALLY stop, we need to prep the system to mark the files as failed so we can inform the user

```

function uploadAndLink(files) {
  if ($scope.isProcessing) {
    return $q.reject('Cannot upload while the upload controller is uploading');
  }

  $scope.isProcessing = true;

  var failedFiles = [];

  var process = _.reduce(files, function(process, file) {
    return process.then(
      processFile.bind(undefined, file, failedFiles)
    );
  }, $q.when({}));

  // Because the process file fails gracefully
  // there will be no error state to handle intentionally
  // TODO: Implement handleErroneousFiles
  return process.then(
    handleErroneousFiles.bind(undefined, failedFiles)
  ).then(function() {
    $scope.isProcessing = false;
  });
}

```

Our primary promise starts from uploadAndLink

Note the angular usage again

In this, we stay away from allowing another upload in the event that its processing, and allow whatever's controlling us to push out the upload errors

Group 1: This is where the magic is. Notice how we loop through each of our file uploads and process them individually.

Group 2: Once we've collected all of the failed files, we can handle them and finally display our errors collectively

```

function uploadAndLink(files) {
  if ($scope.isProcessing) {
    return $q.reject('Cannot upload while the upload controller is uploading');
  }

  $scope.isProcessing = true;

  var failedFiles = [];

  var process = _.reduce(files, function(process, file) {
    return process.then(
      processFile.bind(undefined, file, failedFiles)
    );
  }, $q.when({}));

  // Because the process file fails gracefully
  // there will be no error state to handle intentionally
  // TODO: Implement handleErroneousFiles
  return process.then(
    handleErroneousFiles.bind(undefined, failedFiles)
  ).then(function() {
    $scope.isProcessing = false;
  });
}

```

Our primary promise starts from uploadAndLink

Note the angular usage again

In this, we stay away from allowing another upload in the event that its processing, and allow whatever's controlling us to push out the upload errors

Group 1: This is where the magic is. Notice how we loop through each of our file uploads and process them individually.

Group 2: Once we've collected all of the failed files, we can handle them and finally display our errors collectively

```

function uploadAndLink(files) {
  if ($scope.isProcessing) {
    return $q.reject('Cannot upload while the upload controller is uploading');
  }

  $scope.isProcessing = true;

  var failedFiles = [];

  var process = _.reduce(files, function(process, file) {
    return process.then(
      processFile.bind(undefined, file, failedFiles)
    );
  }, $q.when({}));

  // Because the process file fails gracefully
  // there will be no error state to handle intentionally
  // TODO: Implement handleErroneousFiles
  return process.then(
    handleErroneousFiles.bind(undefined, failedFiles)
  ).then(function() {
    $scope.isProcessing = false;
  });
}

```

Our primary promise starts from uploadAndLink

Note the angular usage again

In this, we stay away from allowing another upload in the event that its processing, and allow whatever's controlling us to push out the upload errors

Group 1: This is where the magic is. Notice how we loop through each of our file uploads and process them individually.

Group 2: Once we've collected all of the failed files, we can handle them and finally display our errors collectively

```

function uploadAndLink(files) {
  if ($scope.isProcessing) {
    return $q.reject('Cannot upload while the upload controller is uploading');
  }

  $scope.isProcessing = true;

  var failedFiles = [];

  var process = _.reduce(files, function(process, file) {
    return process.then(
      processFile.bind(undefined, file, failedFiles)
    );
  }, $q.when({}));

  // Because the process file fails gracefully
  // there will be no error state to handle intentionally
  // TODO: Implement handleErroneousFiles
  return process.then(
    handleErroneousFiles.bind(undefined, failedFiles)
  ).then(function() {
    $scope.isProcessing = false;
  });
}

```

Our primary promise starts from uploadAndLink

Note the angular usage again

In this, we stay away from allowing another upload in the event that its processing, and allow whatever's controlling us to push out the upload errors

Group 1: This is where the magic is. Notice how we loop through each of our file uploads and process them individually.

Group 2: Once we've collected all of the failed files, we can handle them and finally display our errors collectively



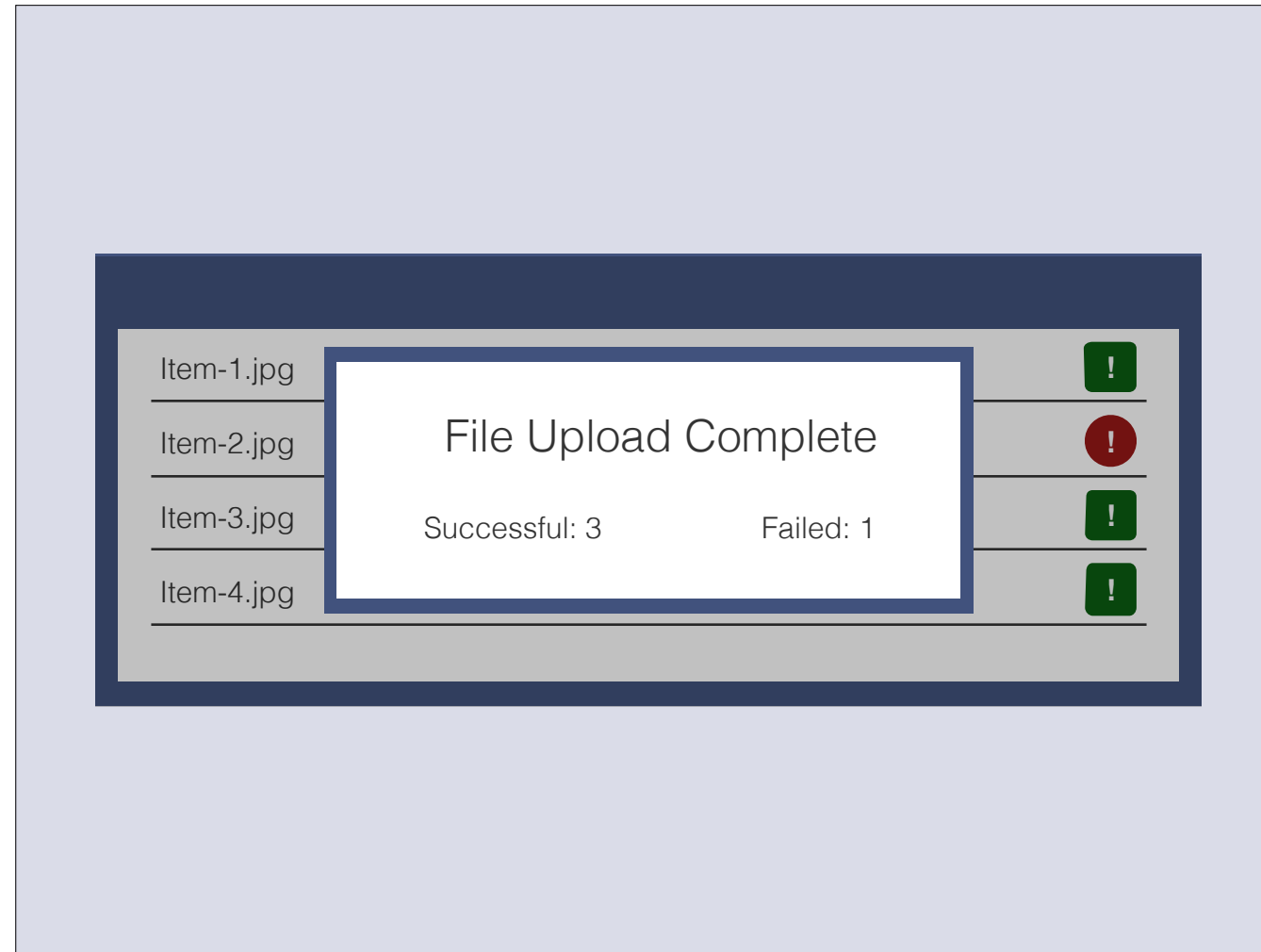
Lets review the previous animation and see the methods as they are invoked.



Lets review the previous animation and see the methods as they are invoked.

Item-1.jpg	!
Item-2.jpg	!
Item-3.jpg	!
Item-4.jpg	!

Lets review the previous animation and see the methods as they are invoked.



Lets review the previous animation and see the methods as they are invoked.

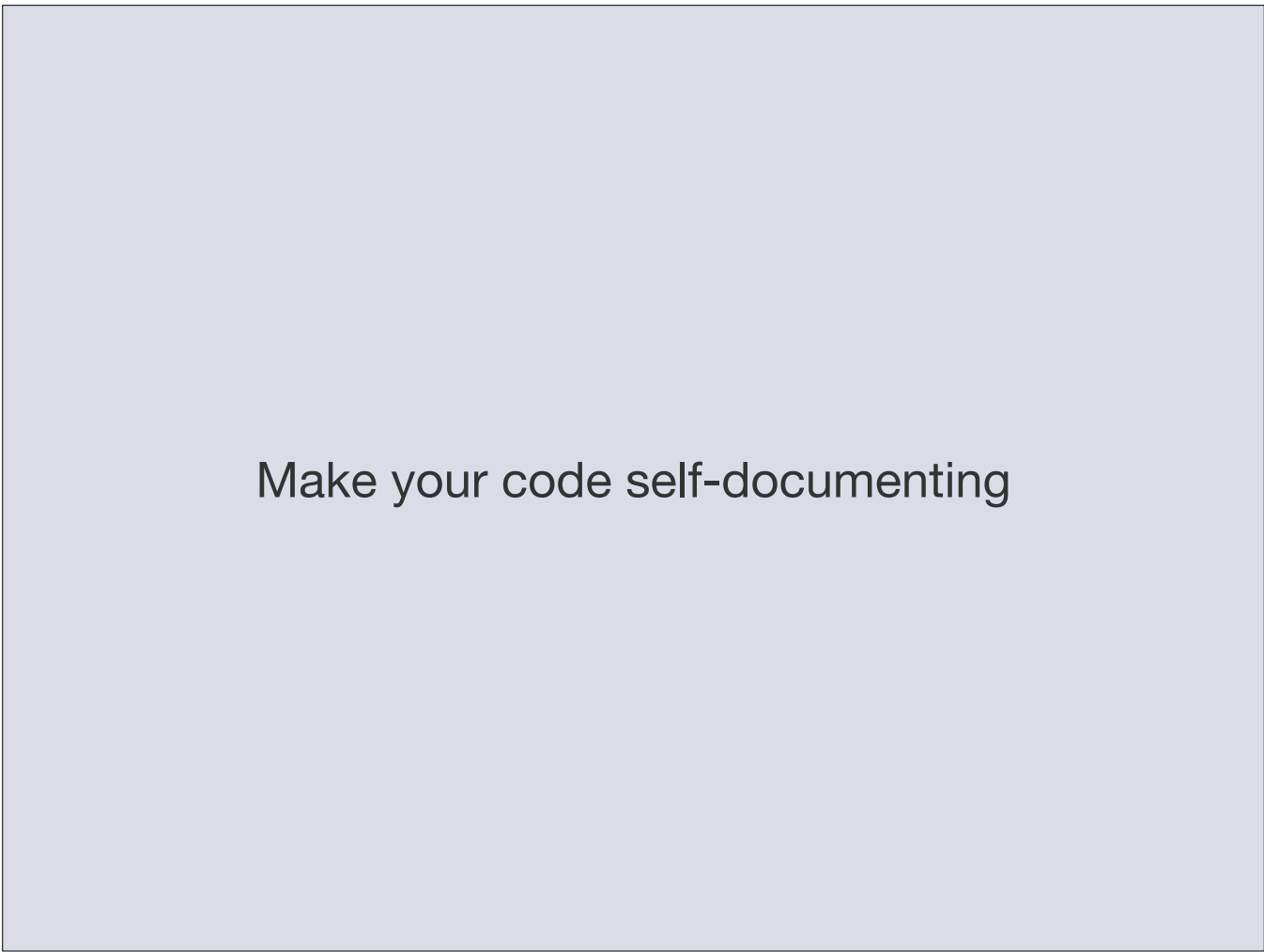
Takeaways

So I want to outline some takeaways. And remind you that while you may only take away some of this, it is HARD. Hard to Grok.



Utilize promises

Use promises when they're appropriate. Introduce the promises into the interface, don't go halfway, go all the way



Make your code self-documenting

Promises can aid readability. Their main purpose is to introduce readability in a synchronous pattern for an asynchronous process. Make your code self-documenting, be it by providing a promise synopsis, or creating a looped queue and nesting.



Be a control freak

Promise processes are hard. The patterns get complex and can be hard to follow. Utilize your rejection, control it, and make distinct decisions around rejection and handling rejection to work with your processes

Simplify the process

When you have a process that is hard to follow, or is very complex with many steps and many more steps, utilize your software engineering knowledge and break down the problem: simplify the process. You can easily provide a synopsis and also nest your process as needed. Even moving that into a loop



Beware rejection

Rejections, and errors in general, are often overlooked or not tested. Be sure to make sure your work has a way to see your rejection states and also has a way to maintain rejection. Not doing so can create oddities in your workflows

Experiment

Promises ARE hard. After experimenting, I've seen people find new ways to use promises, extending the promise pattern to areas that we don't even think of, but once you use, you don't want to go back.

Animations

Modals

Multi-step processes

Alerts/notifications

Workers

Questions?

Thank you for attending!