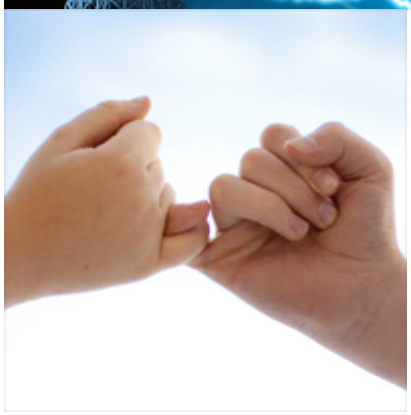
Advertise Here



Promise-Based Validation

Pavan Podila on Apr 25th 2013 with 21 Comments

Tutorial Details

-
- **Difficulty:** Advanced
- **Completion Time:** 1 Hour

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

The concept of "Promises" has changed the way we write asynchronous JavaScript. Over the past year, many frameworks have incorporated some form of the Promise pattern to make asynchronous code easier to write, read and maintain. For example, jQuery added $.Deferred(), and NodeJS has the Q and jspromise modules that work on both client and server. Client-side MVC frameworks, such as EmberJS and AngularJS, also implement their own versions of Promises.

But it doesn't have to stop there: we can rethink older solutions and apply Promises to them. In this article, we'll do just that: validate a form using the Promise pattern to expose a super simple API.

What is a Promise?

> Promises notify the result of an operation.

Simply put, Promises notify the result of an operation. The result can be a success or a failure, and the operation, itself, can be anything that abides by a simple contract. I chose to use the word *contract* because you can design this contract in several different ways. Thankfully, the development community reached a consensus and created a specification called Promises/A+.

Only the operation truly knows when it has completed; as such, it is responsibile for notifying its result using the Promises/A+ contract. In other words, it *promises* to tell you the final result on completion.

The operation returns a `promise` object, and you can attach your callbacks to it by using the `done()` or `fail()` methods. The operation can notify its outcome by calling `promise.resolve()` or `promise.reject()`, respectively. This is depicted in the following figure:

Using Promises for Form Validation

Let me paint a plausible scenario.

> We can rethink older solutions and apply Promises to them.

Client–side form validation always begins with the simplest of intentions. You may have a sign–up form with *Name* and *Email* fields, and you need to ensure that the user provides valid input for both fields. That seems fairly straightforward, and you start implementing your solution.

You are then told that email addresses must be unique, and you decide to validate the email address on the server. So, the user clicks the submit button, the server

checks the email's uniqueness and the page refreshes to display any errors. That seems like the right approach, right? Nope. Your client wants a slick user experience; visitors should see any error messages without refreshing the page.

Your form has the *Name* field that doesn't require any server-side support, but then you have the *Email* field that requires you to make a request to the server. Server requests means `$.ajax()` calls, so you will have to perform email validation in your callback function. If your form has multiple fields that require server-side support, your code will be a nested mess of `$.ajax()` calls in callbacks. Callbacks inside callbacks: "Welcome to callback hell! We hope you have a miserable stay!".

So, how do we handle callback hell?

The Solution I Promised

Take a step back and think about this problem. We have a set of operations that can either succeed or fail. Either of these results can be captured as a `Promise`, and the operations can be anything from simple client-side checks to complex server-side validations. Promises also give you the added benefit of consistency, as well as letting you avoid conditionally checking on the type of validation. Lets see how we can do this.

> As I noted earlier, there are several promise implementations in the wild, but I will focus on jQuery's $.Deferred() Promise implementation.

We will build a simple validation framework where every check immediately returns either a result or a Promise. As a user of this framework, you only have to remember one thing: *"it always returns a Promise"*. Lets get started.

Validator Framework using Promises

I think it's easier to appreciate the simplicity of Promises from the consumer's point of view. Lets say I have a form with three fields: Name, Email and Address:

```
<form>
  <div class="row">
    <div class="large-4 columns">
      <label>Name</label>
      <input type="text" class="name"/>
    </div>
  </div>

  <div class="row">
    <div class="large-4 columns">
      <label>Email</label>
      <input type="text" class="email"/>
    </div>
  </div>

  <div class="row">
    <div class="large-4 columns">
      <label>Address</label>
      <input type="text" class="address"/>
    </div>
  </div>

</form>
```

I will first configure the validation criteria with the following object. This also serves as our framework's API:

```
var validationConfig = {
  '.name': {
    checks: 'required',
    field: 'Name'
  },
  '.email': {
    checks: ['required'],
    field: 'Email'
  },
  '.address': {
    checks: ['random', 'required'],
    field: 'Address'
  }
};
```

The keys of this config object are jQuery selectors; their values are objects with the following two properties:

- `checks`: a string or array of validations.
- `field`: the human-readable field name, which will be used for reporting errors for that field

We can call our validator, exposed as the global variable `V`, like this:

```
V.validate(validationConfig)
  .done(function () {
      // Success
  })
  .fail(function (errors) {
      // Validations failed. errors has the details
  });
```

Note the use of the `done()` and `fail()` callbacks; these are the default callbacks for handing a Promise's result. If we happen to add more form fields, you could simply augment the `validationConfig` object without disturbing the rest of the setup (the Open–Closed Principle in action). In fact, we can add other validations, like the uniqueness constraint for email addresses, by extending the validator framework (which we will see later).

So that's the consumer–facing API for the validator framework. Now, let's dive in and see how it works under the hood.

Validator, Under the Hood

The validator is exposed as an object with two properties:

- `type`: contains the different kinds of validations, and it is also serves as the extension point for adding more.
- `validate`: the core method that performs the validations based upon the provided config object.

The overall structure can be summarized as:

```
var V = (function ($) {

var validator = {

  /*
  * Extension point - just add to this hash
  *
  * V.type['my-validator'] = {
  *   ok: function(value){ return true; },
  *   message: 'Failure message for my-validator'
  *   }
```

```
    */
    type: {
      'required': {
         ok: function (value) {
             // is valid ?
         },
         message: 'This field is required'
      },

      ...
    },

    /**
     *
     * @param config
     * {
     *   '<jquery-selector>': string | object | [ string ]
     * }
     */
    validate: function (config) {

      // 1. Normalize the configuration object

      // 2. Convert each validation to a promise

      // 3. Wrap into a master promise

      // 4. Return the master promise
    }
  };

  })(jQuery);
```

The `validate` method provides the underpinnings of this framework. As seen in the comments above, there are four steps that happen here:

1. **Normalize the configuration object.**

This is where we go through our config object and convert it into an internal representation. This is mostly to capture all the information we need to carry out the validation and report errors if necessary:

```
function normalizeConfig(config) {
  config = config || {};

  var validations = [];

  $.each(config, function (selector, obj) {

    // make an array for simplified checking
    var checks = $.isArray(obj.checks) ? obj.checks : [obj.checks]

    $.each(checks, function (idx, check) {
      validations.push({
        control: $(selector),
        check: getValidator(check),
        checkName: check,
        field: obj.field
      });
    });

  });
  return validations;
}

function getValidator(type) {
  if ($.type(type) === 'string' && validator.type[type]) return va

  return validator.noCheck;
}
```

This code loops over the keys in the config object and creates an internal representation of the validation. We will use this representation in the `validate` method.

The `getValidator()` helper fetches the validator object from the `type` hash. If we don't find one, we return the `noCheck` validator which always returns true.

2. **Convert each validation to a Promise.**

Here, we ensure every validation is a Promise by checking the return value of

`validation.ok()`. If it contains the `then()` method, we know it's a Promise (this is as per the Promises/A+ spec). If not, we create an ad-hoc Promise that resolves or rejects depending on the return value.

```
validate: function (config) {
  // 1. Normalize the configuration object
  config = normalizeConfig(config);
  var promises = [],
    checks = [];

  // 2. Convert each validation to a promise
  $.each(config, function (idx, v) {
    var value = v.control.val();
    var retVal = v.check.ok(value);

    // Make a promise, check is based on Promises/A+ spec
    if (retVal.then) {
      promises.push(retVal);
    }
    else {
      var p = $.Deferred();

      if (retVal) p.resolve();
      else p.reject();

      promises.push(p.promise());
    }
    checks.push(v);
  });
  // 3. Wrap into a master promise

  // 4. Return the master promise
}
```

3. Wrap into a master Promise.

We created an array of Promises in the previous step. When they all succeed, we want to either resolve once or fail with detailed error information. We can do this by wrapping all of the Promises into a single Promise and propagate the result. If everything goes well, we just resolve on the master promise.

For errors, we can read from our internal validation representation and use it for reporting. Since there can be multiple validation failures, we loop over the `promises` array and read the `state()` result. We collect all of the rejected promises into the `failed` array and call `reject()` on the master promise:

```
// 3. Wrap into a master promise
var masterPromise = $.Deferred();
$.when.apply(null, promises)
  .done(function () {
    masterPromise.resolve();
  })
  .fail(function () {
    var failed = [];
    $.each(promises, function (idx, x) {
      if (x.state() === 'rejected') {
        var failedCheck = checks[idx];
        var error = {
          check: failedCheck.checkName,
          error: failedCheck.check.message,
          field: failedCheck.field,
          control: failedCheck.control
        };
        failed.push(error);
      }
    });
    masterPromise.reject(failed);
  });

// 4. Return the master promise
return masterPromise.promise();
```

4. **Return the master promise.**

Finally we return the master promise from the `validate()` method. This is the Promise on which the client code sets up the `done()` and `fail()` callbacks.

Steps two and three are the crux of this framework. By normalizing the validations into a Promise, we can handle them consistently. We have more control with a master Promise object, and we can attach additional contextual information that may be useful to the end user.

Using the Validator

See the demo file for a full use of the validator framework. We use the `done()` callback to report success and `fail()` to show a list of errors against each of the fields. The screenshots below show the success and failure states:

The demo uses the same HTML and validation configuration mentioned earlier in this article. The only addition is the code that displays the alerts. Note the use of the `done()` and `fail()` callbacks to handle the validation results.

```
function showAlerts(errors) {
  var alertContainer = $('.alert');
  $('.error').remove();

  if (!errors) {
    alertContainer.html('<small class="label success">All Passed</
  } else {
    $.each(errors, function (idx, err) {
      var msg = $('<small></small>')
          .addClass('error')
          .text(err.error);

      err.control.parent().append(msg);
    });
  }
}

$('.validate').click(function () {

  $('.indicator').show();
  $('.alert').empty();

  V.validate(validationConfig)
      .done(function () {
        $('.indicator').hide();
        showAlerts();
      })
      .fail(function (errors) {
        $('.indicator').hide();
        showAlerts(errors);
      });

});
```

Extending the Validator

I mentioned earlier that we can add more validation operations to the framework by extending the validator's `type` hash. Consider the `random` validator as an example. This validator randomly succeeds or fails. I know its not a useful validator, but it's worth

noting some of its concepts:

- Use `setTimeout()` to make the validation async. You can also think of this as simulating network latency.
- Return a Promise from the `ok()` method.

```
// Extend with a random validator
V.type['random'] = {
  ok: function (value) {
    var deferred = $.Deferred();

    setTimeout(function () {
      var result = Math.random() < 0.5;
      if (result) deferred.resolve();
      else deferred.reject();

    }, 1000);

    return deferred.promise();
  },
  message: 'Failed randomly. No hard feelings.'
};
```

In the demo, I used this validation on the *Address* field like so:

```
var validationConfig = {
  /* cilpped for brevity */

  '.address': {
    checks: ['random', 'required'],
    field: 'Address'
  }
};
```

Summary

I hope that this article has given you a good idea of how you can apply Promises to old problems and build your own framework around them. The Promise–based approach is a fantastic solution to abstract operations that may or may not run synchronously. You can also chain callbacks and even compose higher–order Promises from a set of other Promises.

The Promise pattern is applicable in a variety of scenarios, and you'll hopefully encounter some of them and see an immediate match!

References

- Promises/A+ spec
- jQuery.Deferred()
- Q
- jspromise

Like 122 people like this. Be the first of your friends.

Tags: promises

By Pavan Podila

I am a Financial Technologist building apps for Web and Mobile Platforms using Ruby, NodeJS and iOS. You can follow me on Twitter or read my Blog

Note: Want to add some source code? Type <pre><code> before it and </code>

</pre> after it. Find out more