Extra notes - Client-side Design and Development

- Dr Nick Hayward

JS - Generators and Promises

A brief introduction to generators and promises with plain JavaScript.

Contents

Intro Generators and Promises are new to plain JavaScript with the introduction of ES6.

Generators are a special type of function, which produce multiple values per request whilst suspending their execution between these requests.

In JS, *generators* are useful to help simplify convoluted loops, suspend and resume code execution, &c. Each benefit also helps write simple, more elegant *async* code.

Promises are a new, built-in object to simply help development of *async* code. As such, a promise becomes a placeholder for a value not currently available, but one that will be available later.

Async code and execution JS relies on a single-threaded execution model.

If we query a remote server using standard code execution, we block the UI until a response is received and various operations completed.

We may modify our code to use callbacks, which will be invoked as a task completes. This should work without blocking the UI but it can quickly create a spaghetti mess of code, error handling, and logic.

Generators and *Promises* are an elegant solution to this mess and proliferation of code.

Generators A generator function *generates* a sequence of values. However, this is commonly not all at once but on a request basis.

A generator is explicitly asked for a new value, and it will either return a value or a response of no more values.

After producing a requested value, a generator will then suspend instead of ending its execution. The generator will then resume when a new value is requested.

e.g.

```
//generator function
function* nameGenerator() {
  yield "emma";
  yield "daisy";
  yield "rosemary";
}
```

We define our generator function by appending an *asterisk* after the keyword. We may then use the `yield` keyword within the body of the generator to request and retrieve individual values.

We may then consume these generated values using a standard loop, or perhaps the new `for-of` loop.

iterator object If we make a call to the body of the generator, an iterator object will be created.

We may now communicate with and control the generator using the iterator object.

e.g.

```
// generator function
function* NameGenerator() {
  yield "emma";
}
// create an iterator object
const nameIterator = NameGenerator();
```

The iterator object, `nameIterator` , exposes various methods including the `next` method. We may use `next` to control the iterator, and request its next value. e.g.

```
// get a new value from the generator with the 'next' method
const name1 = nameIterator.next();
```

The `next` method executes the generator's code to the next yield expression. It then returns an object with the value of the yield expression, and a property `done` set to *false* if a value is still available.

This boolean will switch to *true* if there is no value for the next requested yield. This way we know the iterator for the generator has now finished.

iterate over iterator object We may simply iterate over the iterator object to return each value per available yield expression.

The `for-of` loop, for example, works well

```
// iterate over iterator object
for(let iteratorItem of NameGenerator()) {
  if (iteratorItem !== null) {
    console.log("iterator item = "+iteratorItem+index);
  }
}
```

This is noticeably clearer than an equivalent loop with `while` .

call generator within a generator We may also call a generator from within another generator,

e.g.

```
//generator function
function* NameGenerator() {
  yield "emma";
  yield "rose";
  yield "celine";
  yield* UsernameGenerator();
  yield "yvaine";
}

function* UsernameGenerator() {
  yield "frisby67";
  yield "trilby72";
}
```

We may then use the initial generator, `NameGenerator` , as normal.

practical generator - recursive traversal of the DOM The document object model, or DOM, is tree-like structure of HTML nodes. Every node, except the root, has exactly one parent, and the potential for zero or more child nodes.

As such, DOM traversal is a common and necessary requirement of client-side development.

However, we may now use generators to help iterate over the DOM tree.

e.g.

```
// generator function - traverse the DOM
function* DomTraverseGenerator(htmlElem) {
  yield htmlElem;
  htmlElem = htmlElem.firstElementChild;
  // transfer iteration control to another instance of the current generator
  // - enables sub iteration...
  while (htmlElem) {
    yield* DomTraverseGenerator(htmlElem);
    htmlElem = htmlElem.nextElementSibling;
  }
}
```

A benefit to this generator-based approach for DOM traversal is that callbacks are not required. Simpler code ensues, and we're able to consume the generated sequence of nodes with a simple `for-of` loop without using callbacks.

In the above example, we're also able to use generators to separate our code. For example, we have code that is producing values, a HTML node in this example, from code consuming the sequence of generated values. In this example, we're using the `for-of` loop to log visited nodes.

exchanging data with a generator We can also send data to a generator, thereby enabling bi-directional communication.

So, we may request data, then process it, and return an updated value when necessary to a generator.

The easiest option is to simply consider a generator like a normal function, sending data using function call arguments.

```
// generator function - send data to generator - receive standard argument
function* MessageGenerator(data) {
  // yield a value - generator returns an intermediary calculation
  const message = yield(data);
  yield("Greetings, "+ message);
}

const messageIterator = MessageGenerator("Hello World");
const message1 = messageIterator.next();
console.log("message = "+message1.value);

const message2 = messageIterator.next("Hello again");
console.log("message = "+message2.value);
```

The first call with the `next()` method requests a new value from the generator, which will return the initial passed argument. The generator is then suspended.

The second call using `next()` will resume the generator, again requesting a new value. However, it also sends a new argument into the generator using the `next()` method. This newly passed argument value becomes the complete value for this yield, thereby replacing the previous value `Hello World` .

So, we can achieve the required bi-directional communication with a generator. We may use `yield` to return data from a generator, and then use the iterator's `next()` method to pass data back to the generator.

detailed structure of generators Generators work in a detailed manner as follows,

- **suspended start** - none of the generator code is executed when it first starts
- **executing** - execution either starts at the beginning or resumes where it was last suspended. This state is created when the iterator's `next()` method is called…code must exist in generator for execution
- **suspended yield** - whilst executing, a generator may reach `yield` . It will then create a new object carrying the return value, yields this object, and then suspends execution at the point of the yield…
- **completed** - a `return` statement or lack of code to execute will cause the generator to move to a *complete* state

Promises A *promise* is similar to a placeholder for a value we currently do not have, but we would like later. In effect, it's a guarantee we'll eventually receive the result to an asynchronous request, computation &c.

A result will be returned, either a value or an error.

So, we commonly use *promises* to fetch data from a server.

e.g.

```
// use built-in Promise constructor
// - pass callback function with two parameters (resolve & reject)
const testPromise = new Promise((resolve, reject) => {
  resolve("test return");
  // reject("an error has occurred trying to resolve this promise...");
});

// use `then` method on promise - pass two callbacks for success and failure
testPromise.then(data => {
  // output value for promise success
  console.log("promise value = "+data);
}, err => {
  // output message for promise failure
  console.log("an error has been encountered...");
});
```

We may use the built-in *Promise* constructor to create a new promise object. We can then pass a function, which is a standard arrow function in the above example.

This function for a Promise is commonly known as an *executor* function, and includes two parameters, `resolve` and `reject` .

The *executor* function is called immediately as the *Promise* object is being constructed. The `resolve` argument is called manually when we need the promise to resolve successfully. The second argument, `reject` , will be called if an error occurs.

This example uses the *promise* by calling the built-in `then` method available on the *promise* object. This `then` method accepts two callback functions, `success` and `failure` .

`success` is called if the *promise* resolves successfully, whilst the `failure` callback is available if there is an error.

callbacks and async Async code is useful to prevent execution blocking, and delays in the browser, in particular as we execute long-running tasks.

This issue is often solved using *callbacks*. In effect, we provide a callback that's invoked when the task is completed.

However, such long running tasks may result in errors. Therefore, the issue with callbacks is that we can't use built-in constructs such as `try-catch` statements.

e.g.

```
try {
  getJSON("data.json", function() {
    // handle return results...
  });
} catch (e) {
  // handle errors...
}
```

This won't work as expected because the code executing the callback is not usually executed in the same step of the event loop as the code running the long task.

So, errors will usually get lost as part of this long running task.

Another issue with callbacks is nesting, often to the point where the code may become difficult to read, understand, and debug.

A third issue is trying to run parallel callbacks. Performing a number of parallel steps becomes inherently tricky and error prone.

further detail on promises A *promise* starts in a pending state where we know nothing about the return value. In this state, the promise is often known as an *unresolved* promise.

During execution, if the promise's *resolve* function is then called, it will now move into its *fulfilled* state. The return value is now available.

If there is an error or issue, or the *reject* method is explicitly called, the promise will simply move into a *rejected* state. The return value is no longer available, but an error is available.

Either of these states means that the promise can now no longer switch state. i.e from rejected to fulfilled and vice-versa…

So, an example of working with a promise may be as follows

- code starts (execution is ready)
- promise is now executed and starts to run
- promise object is created
- promise continues until it resolves
  - successful return, artificial timeout &c.
- code for the current promise is now at an end
- promise is now resolved
  - value is available in the promise
- then work with resolved promise and value
  - call `then` method on promise and returned value…
  - this callback is scheduled for successful *resolve* of the promise
  - this callback will always be asynchronous regardless of state of promise…

explicitly reject promises Two standard ways to reject a promise.

- explicit rejection of promise

```
const promise = new Promise((resolve, reject) => {
  reject("explicit rejection of promise");
});
```

Once the promise has been rejected, an error callback will always be invoked, e.g. through the calling of the `then` method

```
promise.then(
  () => fail("won't be called..."),
  error => pass("promise was explicitly rejected...");
);
```

We may also chain a `catch` method to the `then` method as an alternative to the error callback. e.g.

```
promise.then(
  () => fail("won't be called..."))
  .catch(error => pass("promise was explicitly rejected..."));
```

a real-world promise We may use a promise to perform an asynchronous action on the client-side by fetching data from a server.

For example, we may use the built-in `XMLHttpRequest` object to perform this fetching of data.

e.g.

```
// create a custom get json function
function getJSON(url) {
  // create and return a new promise
  return new Promise((resolve, reject) => {
    // create the required XMLHttpRequest object
    const request = new XMLHttpRequest();
    // initialise this new request - open
    request.open("GET", url);
    // register onload handler - called if server responds
    request.onload = function() {
      try {
        // make sure response is OK - server needs to return status 200 code...
        if (this.status === 200) {
          // try to parse json string
          // - if success, resolve promise successfully with value
          resolve(JSON.parse(this.response));
        } else {
          // different status code, exception parsing JSON &c. - reject the promise...
          reject(this.status + " " + this.statusText);
        }
      } catch(e) {
        reject(e.message);
      }
    };

    // if error with server communication - reject the promise...
    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };

    // send the constructed request to get the JSON
    request.send();
  });
}

// call getJSON with required URL, then method for resolve object, and catch for error
getJSON("test.json").then(response => {
  // check return value from promise...
  response !== null ? "response obtained" : "no response";
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
  // - not much to show due to return of jsonp from flickr...
    console.log('error found = ', err);
});
```

This example has created a `getJSON` function that returns a *promise*. This enables us to register callbacks for *success* and *failure* for asynchronously getting the JSON from a server.

With the XMLHttpRequest object, we have two available events

- `onload` - triggered as the browser receives a response from the server
- `onerror` - triggered as an error is received

Each of these event handlers are triggered asynchronously as required by the browser.

For the `onload` event, we're checking the return code of the response. In effect, was the requested loaded successfully. This will give us the required 200 status code, otherwise we can simply reject the promise.

We surround the `JSON.parse` method in a `try-catch` statement so we can check the return JSON code for syntax errors. If an exception occurs, again we may simply reject the promise.

Once we've successfully resolved the promise, we may use our `getJSON` function to work with the response.

chain promises By calling `then` on the returned promise itself creates a new *promise*. So, we may then register an additional callback if this promise is now resolved successfully.

We may now chain as many `then` methods as necessary. In effect, we create a sequence of promises, which hopefully will each be resolved one after another.

Instead of creating deeply nested callbacks, we may now simply chain such methods to our initial resolved promise.

To catch an error we may chain a final `catch` call. So, if we're simply interested in a failure for the overall chain, we may use the `catch` method for the overall chain, e.g.

```
getJSON().then()
.then()
.then()
.catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
  // - not much to show due to return of jsonp from flickr...
    console.log('error found = ', err);
});
```

If a failure occurs in any of the previous promises, the `catch` method will be called.

waiting for multiple promises Promises also make it easy to wait for multiple, independent asynchronous tasks.

With `Promise.all` , we may wait for a number of promises.

e.g.

```
// wait for a number of promises - all
Promise.all([
// call getJSON with required URL
// - `then` method for resolve object, and `catch` for error
getJSON("notes.json"),
getJSON("metadata.json")]).then(response => {
  // check return value from promise...
  // - response[0] = notes.json, response[1] = metadata.json &c.
  if (response[0] !== null) {
        console.log("response obtained");
    console.log("notes = ", JSON.stringify(response[0]));
    console.log("metadata = ", JSON.stringify(response[1]));
    }
}).catch((err) => {
  // Handle any error that occurred in any of the previous promises in the chain.
  // - not much to show due to return of jsonp from flickr...
    console.log('error found = ', err);
});
```

The order of execution for tasks doesn't matter for `Promise.all`, and indeed whether some have finished or not. By using the `Promise.all` method, we are simply stating that we want to wait.

This method, `Promise.all`, accepts an array of promises, and then creates a *new* promise that will resolve successfully when all passed promises resolve. However, it will reject if a single one of the passed promises fails.

The return promise is an array of succeed values as responses. In effect, one succeed value for each passed in promise.

racing promises We may also setup competing promises, with an effective prize to the first promise to resolve or reject. This might be useful for querying multiple APIs, databases, &c.

To race our promises, we may use the `Promise.race` method, e.g.

```
Promise.race(
  [
  // call getJSON with required URL
  // - `then` method for resolve object, and `catch` for error
  getJSON("notes.json"),
  getJSON("metadata.json")]).then(response => {
    if (response !== null) {
        console.log(`response obtained - ${response} won...`);
    }
  }).catch((err) => {
    // Handle any error that occurred in any of the previous promises in the chain.
    // - not much to show due to return of jsonp from flickr...
    console.log('error found = ', err);
  });
);
```

This method accepts an array of promises, and returns a completely new resolved or rejected *promise* for the first of resolved or rejected promises.

Combining generators and promises By combining generators and promises, the idea is simply to put code that uses asynchronous tasks in the *generator*. Then, we can execute and use the *generator* with that async code.

As we reach some async code in the generator, we can create a *promise* that represents the value of the async task. At the same point, we yield from the *generator* to avoid blocking the logic and the UI.

When the *promise* has been resolved, we may then continue the execution of our generator by calling its `next` method. We may, of course, call this method as many times as necessary during the lifecycle of the generator.

So, an example might be as follows

- `async` function takes a *generator*, calls it, and creates the required *iterator*
  - use *iterator* to resume generator execution as needed
  - declare a *handle* function - handles one return value from *generator*
    - one iteration of iterator
  - if generator result is a *promise* & resolves successfully - use iterator's `next` method
    - promise value sent back to generator
    - generator resumes execution
  - if error, *promise* gets rejected
    - error thrown to generator using iterator's `throw` method
  - continue generator execution until it returns `done`

- `generator` - executes up to each `yield getJSON()`
  - *promise* created for each `getJSON()` call
  - value is fetched async - generator is paused whilst fetching value…
  - control flow is returned to current invocation point in `handle` function whilst paused
- `handle` function
  - yielded value to `handle` function is a promise
  - able to use `then` and `catch` methods with promise object
    - registers success and error callback
    - execution is able to continue