Generators

Similar to how Promises allow us to avoid callback hell, Generators allow us to flatten our code - giving our asynchronous code a synchronous feel. Generators are essentially functions which we can pause their execution and subsequently return the value of an expression.

A simple example of using generators is shown below:

```
function* sillyGenerator() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
}

var generator = sillyGenerator();
> console.log(generator.next()); // { value: 1, done: false }
> console.log(generator.next()); // { value: 2, done: false }
> console.log(generator.next()); // { value: 3, done: false }
> console.log(generator.next()); // { value: 4, done: false }
```

Where next will allow us to push our generator forward and evaluate a new expression. While the above example is extremely contrived, we can utilize Generators to write asynchronous code in a synchronous manner:

```
// Hiding asynchronousity with Generators

function request(url) {
    getJSON(url, function(response) {
        generator.next(response);
    });
}
```

And here we write a generator function that will return our data:

```
function* getData() {
    var entry1 = yield request('http://some_api/item1');
    var data1  = JSON.parse(entry1);
    var entry2 = yield request('http://some_api/item2');
    var data2  = JSON.parse(entry2);
}
```

By the power of `yield`, we are guaranteed that `entry1` will have the data needed to be parsed and stored in `data1`.

While generators allow us to write asynchronous code in a synchronous manner, there is no clear and easy path for error propagation. As such, as we can augment our generator with Promises:

```
function request(url) {
    return new Promise((resolve, reject) => {
        getJSON(url, resolve);
    });
}
```

And we write a function which will step through our generator using `next` which in turn will utilize our `request` method above to yield a Promise:

```
function iterateGenerator(gen) {
    var generator = gen();
    (function iterate(val) {
        var ret = generator.next();
        if(!ret.done) {
            ret.value.then(iterate);
        }
    })();
}
```

By augmenting our Generator with Promises, we have a clear way of propagating errors through the use of our Promise `.catch` and `reject`. To use our newly augmented Generator, it is as simple as before:

```
iterateGenerator(function* getData() {
    var entry1 = yield request('http://some_api/item1');
    var data1  = JSON.parse(entry1);
    var entry2 = yield request('http://some_api/item2');
    var data2  = JSON.parse(entry2);
});
```

We were able to reuse our implementation to use our Generator as before, which shows their power. While Generators and Promises allow us to write asynchronous code in a synchronous manner while retaining the ability to propagate errors in a nice way, we can actually begin to utilize a simpler construction that provides the same benefits: async-await.