Error handling (with Promises)

> This article was originally published at https://gist.github.com/joepie91/c8d8cc4e6c2b57889446. It only applies when using Promise chaining syntax; when you use `async` / `await`, you are instead expected to use `try` / `catch`, which unfortunately does not support error filtering.

There's roughly three types of errors:

1. **Expected errors** - eg. "URL is unreachable" for a link validity checker. You should handle these in your code at the top-most level where it is practical to do so.
2. **Unexpected errors** - eg. a bug in your code. These should crash your process (yes, really), they should be logged and ideally e-mailed to you, and you should fix them right away. You should never catch them for any purpose other than to log the error, and even then you should make the process crash.
3. **User-facing errors** - not really in the same category as the above two. While you can represent them with error objects (and it's often practical to do so), they're not really errors in the programming sense - rather, they're user feedback. When represented as error objects, these should only ever be handled at the top-most point of a request - in the case of Express, that would be the error-handling middleware that sends a HTTP status code and a response.

Would I still need to use try/catch if I use promises?

Sort of. Not the usual `try`/`catch`, but eg. Bluebird has a `.try` and `.catch` equivalent. It works like synchronous `try`/`catch`, though - errors are propagated upwards automatically so that you can handle them where appropriate.

Bluebird's `try` isn't identical to a standard JS `try` - it's more a 'start using Promises' thing, so that you can also wrap synchronous errors. That's the magic of Promises, really - they let you handle synchronous and asynchronous errors/values like they're one and the same thing.

Below is a relatively complex example, that uses a custom 'error filter' (predicate) function, because filesystem errors have a name but not a special error type. The error filtering is only available in Bluebird, by the way - 'native' Promises don't have the filtering.

```
/* UPDATED: This example has been changed to use the new object predicates, that were
 * introduced in Bluebird 3.0. If you are using Bluebird 2.x, you will need to use the
 * older example below, with the predicate function. */

var Promise = require("bluebird");
var fs = Promise.promisifyAll(require("fs"));

Promise.try(function(){
    return fs.readFileAsync("./config.json").then(JSON.parse);
}).catch({code: "ENOENT"}, function(err){
    /* Return an empty object. */
    return {};
}).then(function(config){
    /* `config` now either contains the JSON-parsed configuration file, or an empty object if no
    configuration file existed. */
});
```

If you are still using Bluebird 2.x, you should use predicate functions instead:

```
/* This example is ONLY for Bluebird 2.x. When using Bluebird 3.0 or newer, you should
 * use the updated example above instead. */

var Promise = require("bluebird");
var fs = Promise.promisifyAll(require("fs"));

var NonExistentFilePredicate = function(err) {
    return (err.code === "ENOENT");
};

Promise.try(function(){
    return fs.readFileAsync("./config.json").then(JSON.parse);
}).catch(NonExistentFilePredicate, function(err){
    /* Return an empty object. */
    return {};
}).then(function(config){
    /* `config` now either contains the JSON-parsed configuration file, or an empty object if no
```

```
configuration file existed. */
});
```

Revision #1
Created 2024-12-11 12:42:06 UTC by joepie91
Updated 2024-12-11 18:44:19 UTC by joepie91