NODE.JS

Part II

- a command-line application that takes in a web URL as input and downloads its contents locally into a file
- The npm dependencies are
- `request`: A library to streamline HTTP calls
- `mkdirp`: A small utility to create directories recursively

```
function spider(url, callback) {
var filename = utilities.urlToFilename(url);
fs.exists(filename, function(exists) { //[1]
if(!exists) {
    console.log("Downloading " + url);
    request(url, function(err, response, body) {
    //[2]
    if(err) {
        callback(err);
    } else {
        mkdirp(path.dirname(filename),
        function(err) { //[3] if(err) {
                                  callback(err);
                                  } else {
                                      fs.writeFile(filename, body, function(err){
                                              if(err) {//[4]
                                          callback(err);
                                      } else {
                                          callback(null, filename, true);
                                      }
                                  });
                                  }
                                  });
                                  }
                                  });
                                  } else {
                                      callback(null, filename, false);
                                  }
                                  });
                              }
```

WEB-SPIDER APPLICATION

1. Checks if the URL was already downloaded by verifying that the corresponding file was not already created:

```
fs.exists(filecodename, function(exists) …
```

2. If the file is not found, the URL is downloaded using the following line of code:

```
request(url, function(err, response, body) …
```

3. Then, we make sure whether the directory that will contain the file exists or not:

```
mkdirp(path.dirname(filename), function(err) …
```

4. Finally, we write the body of the HTTP response to the filesystem:

```
fs.writeFile(filename, body, function(err) …
```

CALLBACK-HELL

- Even though the algorithm we implemented is really straightforward, the resulting code has several levels of indentation and is very hard to read.
- Implementing a similar function with direct style blocking API would be straightforward, and there would be very few chances to make it look so wrong
- The situation where the abundance of closures and in-place callback definitions transform the code into an unreadable and unmanageable blob is known as **callback hell**.
- It's one of the most well recognized and severe anti-patterns in Node.js and JavaScript in general.
- The typical structure of a code affected by this problem looks like

```
asyncFoo(function(err) {
asyncBar(function(err) {
asyncFooBar(function(err) {
[...]
});
});
});
```

CALLBACK-HELL

```
asyncFoo(function(err) {
    asyncBar(function(err) {
      asyncFooBar(function(err) {
            [...]
            });
        });
    });
```

- We can see how code written in this way assumes the shape of a pyramid due to the deep nesting and that's why it is also colloquially known as the *pyramid of doom*.
- The most evident problem with code such as the preceding one is the poor readability.
- Due to the nesting being too deep, it's almost impossible to keep track of where a function ends and where another one begins.
- Another issue is caused by the overlapping of the variable names used in each scope.
- When writing asynchronous code, the first rule to keep in mind is to not abuse closures when defining callbacks
- Most of the times, fixing the callback hell problem does not require any library, fancy technique, or change of paradigm but just some common sense

HELL TO HEAVEN-HOW TO REACH

These are some basic principles that can help us keep the nesting level low and improve the organization of our code in general:

- You must exit as soon as possible. Use `return`, `continue`, or `break`, depending on the context, to immediately exit the current statement instead of writing (and nesting) complete `if/else` statements. This will help keep our code shallow.
- You need to create named functions for callbacks, keeping them out of closures and passing intermediate results as arguments.
- Naming our functions will also make them look better in stack traces.
- You need to modularize the code. Split the code into smaller, reusable functions whenever it's possible.

```
if(err) {
callback(err);
} else {
//code to execute when there are no
errors
}
```

```
if(err) {
return callback(err);
}
//code to execute when there are no errors
```

- With this simple trick, we immediately have a reduction of the nesting level of our functions; it is easy and doesn't require any complex refactoring.
- We should never forget that the execution of our function will continue even after we invoke the callback.
- It is then important to insert a `return` instruction to block the execution of the rest of the function.
- Also note that it doesn't really matter what output is returned by the function; the real result (or error) is produced asynchronously and passed to the callback.
- The return value of the asynchronous function is usually ignored.
- This property allows us to write shortcuts such as the following:
- `return callback(...)`
- Instead of the slightly more verbose ones such as the following:
- `callback(...)`
- `return;`

PROMISES

- ❑Promises are an abstraction that allow an asynchronous function to return an object called a **promise**, which represents the eventual result of the operation.
- ❑In the promises jargon, we say that a promise is **pending** when the asynchronous operation is not yet complete, it's **fulfilled** when the operation successfully completes, and **rejected** when the operation terminates with an error.
- ❑Once a promise is either fulfilled or rejected, it's considered **settled.**
- ❑ To receive the fulfillment value or the error (*reason*) associated with the rejection, we can use the `then()` method of the promise.
- ❑ The following is its signature:

```
promise.then([onFulfilled], [onRejected])
```

- ❑ Where `onFulfilled()` is a function that will eventually receive the fulfillment value of the promise, and `onRejected()` is another function that will receive the reason of the rejection (if any).
- ❑ Both functions are optional

PROMISE BASED API

```
asyncOperation(arg, function(err, result) {
    if(err) {
        //handle error
}
    //do stuff with result
    });
```

```
asyncOperation(arg)
.then(function(result) {
//do stuff with result
}, function(err) {
//handle error
});
```

- ❑One crucial property of the `then()` method is that it synchronously returns another promise.
- ❑If any of the `onFulfilled()` or `onRejected()` functions return a value `x`, the promise returned by the `then()` method will be as follows:
- ❑• Fulfill with `x` if `x` is a value
- ❑• Fulfill with the fulfillment value of `x` if `x` is a promise or a **thenable**
- ❑• Reject with the eventual rejection reason of `x` if `x` is a promise or a thenable

PROMISE

- A thenable is a promise-like object with a `then()` method.
- This term is used to indicate a promise that is *foreign* to the particular promise implementation in use.
- This feature allows us to build chains of promises, allowing easy aggregation and arrangement of asynchronous operations in several configurations.
- Also, if we don't specify an `onFulfilled()` or `onRejected()` handler, the fulfillment value or rejection reasons are automatically forwarded to the next promise in the chain

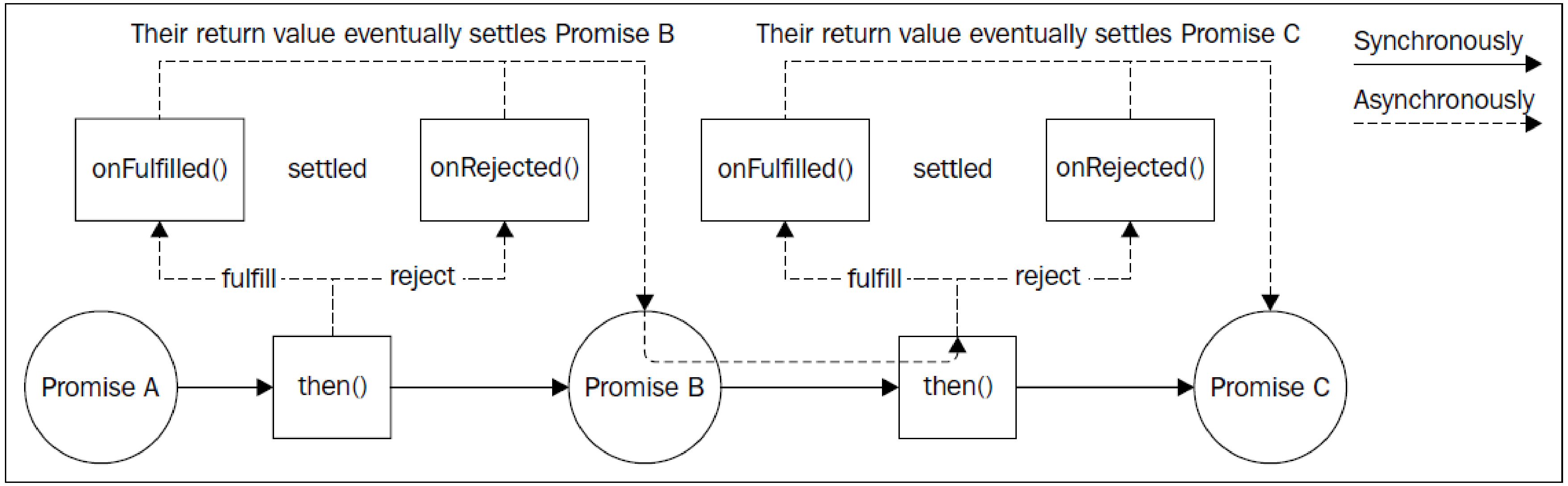
PROMISE CHAIN

With a promise chain, sequential execution of tasks suddenly becomes a trivial operation

```
asyncOperation(arg)
.then(function(result1) {
    //returns another promise
    return asyncOperation(arg2);
    })
.then(function(result2) {
//returns a value
return 'done';
})
.then(undefined, function(err) {
//any error in the chain is caught
here
});
```

PROMISE CHAIN

- If an exception is thrown (using the `throw` statement) from the `onFulfilled()` or `onRejected()` handler, the promise returned by the `then()` method will automatically reject with the exception as the rejection reason.
- This is a tremendous advantage over CPS, as it means that with promises, exceptions will propagate automatically across the chain



CONNECTING A DATABASE

- MongoDB is a NoSQL database used to store large amounts of data without any traditional relational database table.
- Instead of rows & columns, MongoDB used collections & documents to store data.
- A collection consists of a set of documents & a document consists of key-value pairs which are the basic unit of data in MongoDB

CONNECTING TO A DATABASE

Step 1: Initialize npm on the directory and install the necessary modules. Also, create the index file

```
$ npm i express mongoose
```

Step 2: Initialise the express app and make it listen to a port on localhost.

```
const express = require("express");

const app = express();

app.listen(3000, () => console.log("Server is
running"));
```

❑To connect a Node.js application to MongoDB, we have to use a library called **Mongoose**

❑`const mongoose = require("mongoose");`

❑Connect method is invoked with URL and user credentials

```
mongoose.connect("mongodb://localhost:27017/newCollection
", {

      useNewUrlParser: true,

      useUnifiedTopology: true

});
```

CONFIGURING THE DATABASE

A schema is defined

A schema is a structure, that gives information about how the data is being stored in a collection.

```
const contactSchema = {
email: String,
query: String,
};
```

Then we have to create a model using that schema which is then used to store data in a document as objects

```
const Contact = mongoose.model("Contact", contactSchema);
```

When you call mongoose.model() on a schema, Mongoose compiles a model for you.

The first argument is the singular name of the collection your model is for.

Mongoose automatically looks for the plural, lowercased version of your model name.

```
Contact.create({ query: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
});

// or, for inserting large batches of documents
Contact.insertMany([{ query: 'small' }], function(err) {

});
```

CRUD

Contact.find()

Contact.delete;

Contact.update();

```
async function run() {
// Create a new mongoose model
const personSchema = new mongoose.Schema({
name: String });
const Person = mongoose.model('Person',
personSchema);
// Create a change stream. The 'change'
event gets emitted when there's a // change
in the database
Person.watch(). on('change', data =>
console.log(new Date(), data));
```

STORING DATA

we are able to store data in our document

```
app.post("/contact", function (req, res) {
    console.log(req.body.email);
const contact = new Contact({
    email: req.body.email,
    query: req.body.query,
});
```

```
contact.save(function (err) {
    if (err) {
        throw err;
    } else {
        res.render("contact");
    }
});
```

MONGO DB CONNECTION

By using Mongoose we’re able to access the MongoDB database in an object-oriented way.

This means that we need to add a Mongoose schema

MongoDB's collections, by default, do not require their documents to have the same schema. That is:

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the document has substantial variation from other documents in the collection.

STORING FORM DATA WITH MULTIPLE FIELDS

```
app.post("/contact", function (req, res) {
    console.log(req.body.email);
const contact = new Contact({
    email: req.body.email,
    query: req.body.query,
});
```

contact = new Contact(req.body);

```
contact.save(function (err) {
    if (err) {
        throw err;
    } else {
        res.render("contact");
    }
});
```

STORING FORM DATA WITH MULTIPLE FIELDS

```
 let contact = new Contact(req.body);
 contact.save()
 .then(contact => {
            res.status(200).json({'contact': 'contact added
successfully'});
        })
        .catch(err => {
            res.status(400).send('adding new contact failed');
        });
```