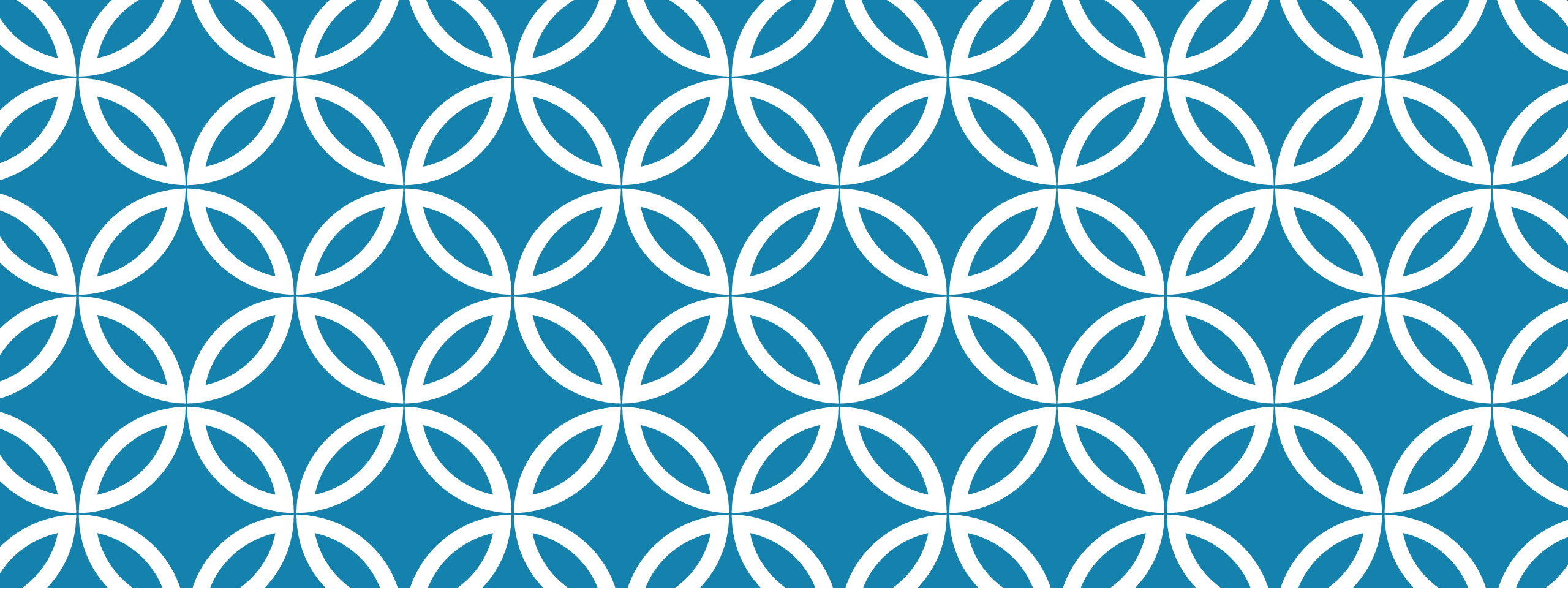


NODE JS

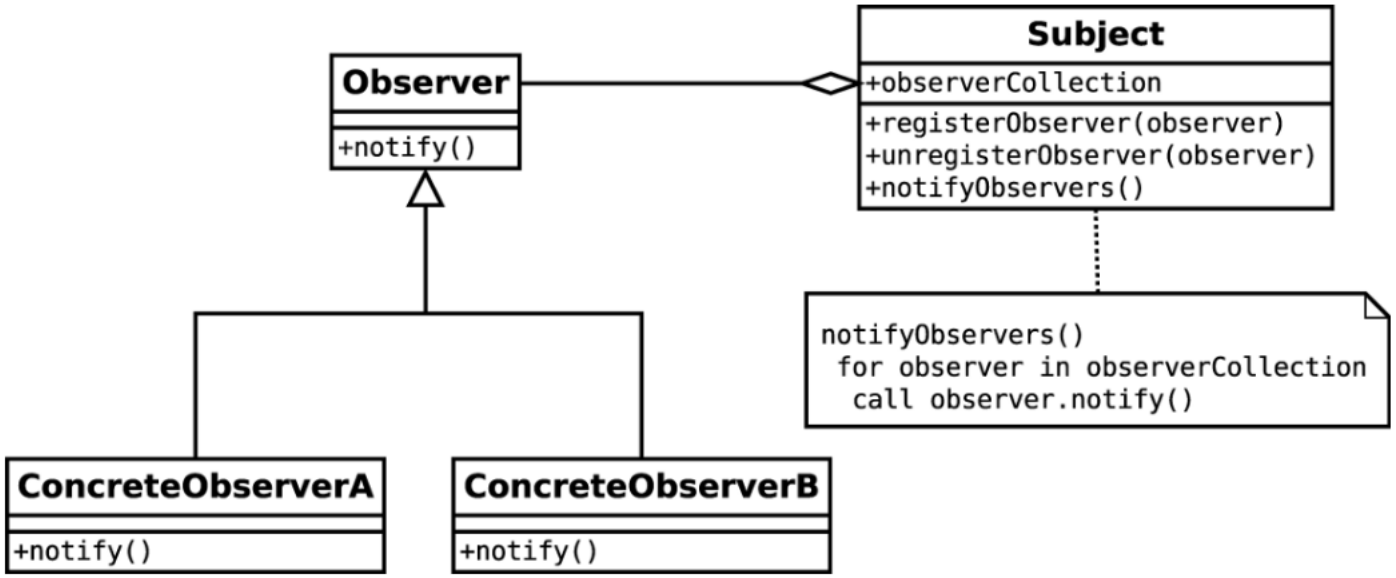
Part III (design patterns)

THE OBSERVER PATTERN

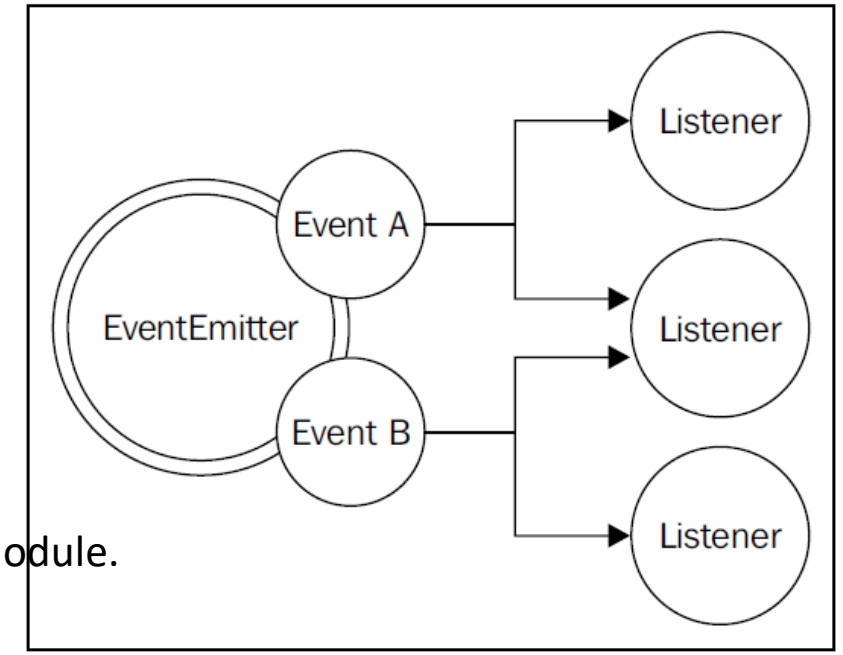
- Together with reactor, callbacks, and modules, this is one of the pillars of the platform and an absolute prerequisite for using many node-core and userland modules
- Pattern (observer): defines an object (called subject), which can notify a set of observers (or listeners), when a change in its state happens.
- The main difference from the callback pattern is that the subject can actually notify multiple observers, while a traditional continuation-passing style callback will usually propagate its result to only one listener, the callback
- The observer pattern is already built into the core and is available through the **EventEmitter** class.

The `EventEmitter` class allows us to register one or more functions as listeners, which will be invoked when a particular event type is fired

THE OBSERVER PATTERN



EVENT EMITTER



- The `EventEmitter` is a prototype, and it is exported from the `events` core module.
- Obtaining a reference of the EventEmitter
  - `var EventEmitter = require('events').EventEmitter;`
  - `var eeInstance = new EventEmitter();`
- The essential methods of the `EventEmitter` are given as follows.

  • `on(event, listener)`: This method allows you to register a new listener (a function) for the given event type (a string)

  • `once(event, listener)`: This method registers a new listener, which is then removed after the event is emitted for the first time

  • `emit(event, [arg1], […])`: This method produces a new event and provides additional arguments to be passed to the listeners

  • `removeListener(event, listener)`: This method removes a listener for the specified event type

EVENT EMITTERS

- All the preceding methods will return the `EventEmitter` instance to allow chaining.
- There is a big difference between a listener and a traditional Node.js callback; in particular, the first argument is not an error, but it can be any data passed to emit() at the moment of its invocation.
- The `listener` function has the signature, `function([arg1], […])`, so it simply accepts the arguments provided the moment the event is emitted.
- Inside the listener, `this` refers to the instance of the `EventEmitter` that produces the event

EMITTING EVENTS

- The `EventEmitter` - as it happens for callbacks - cannot just throw exceptions when an error condition occurs, as they would be lost in the event loop if the event is emitted asynchronously.
- Instead, the convention is to emit a special event, called `error`, and to pass an `Error` object as an argument.
- A common dilemma when defining an asynchronous API is to check whether to use an `EventEmitter` or simply accept a callback.
- The general differentiating rule is semantic: callbacks should be used when a result must be returned in an asynchronous way; events should instead be used when there is a need to communicate that something has just happened
- Another case where the EventEmitter might be preferable is when the same event can occur multiple times, or not occur at all.
- A callback, in fact, is expected to be invoked exactly once, whether the operation is successful or not.
- On the server-side, the Socket instance extends the Node.js EventEmitter class.
- On the client-side, the Socket instance uses the event emitter provided by the component-emitter library, which exposes a subset of the EventEmitter methods.

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

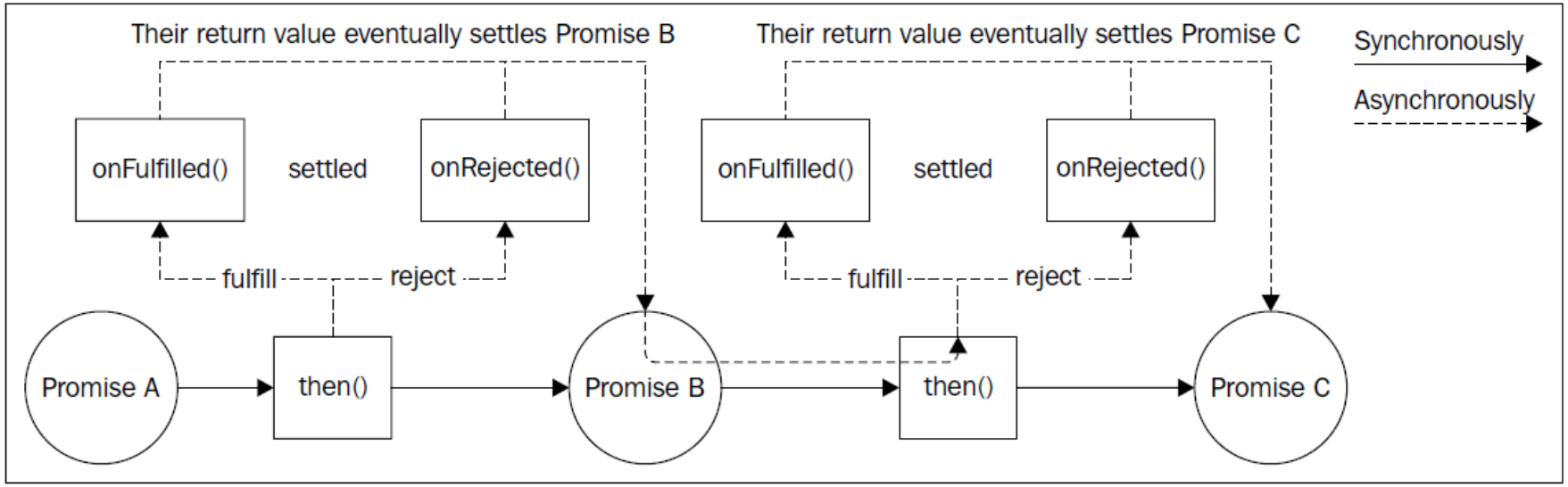
PROMISE CHAIN

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

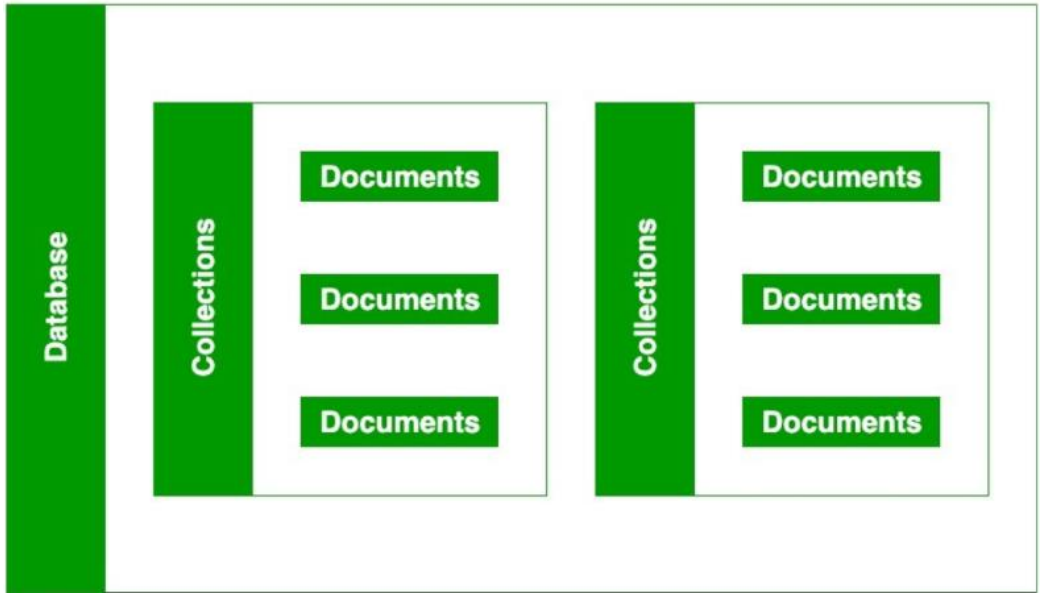
```
asyncOperation(arg)
.then(function(result1) {
    //returns another promise
    return asyncOperation(arg2);
    })
.then(function(result2) {
//returns a value
return 'done';
})
.then(undefined, function(err) {
//any error in the chain is caught
here
});
```

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"));
```

https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/

MONGOOSE

- To connect a Node.js application to MongoDB, we use a library called **Mongoose**
- **Mongoose-** elegant MongoDB object modeling for Node.js
- Mongoose is an ODM (Object Data Modeling) library for MongoDB
- Mongoose helps with data modeling, schema enforcement, model validation, and general data manipulation
- `const mongoose = require("mongoose");`
- Connect method is invoked with URL and user credentials

```
mongoose.connect("mongodb://localhost:27017/newCollection", {
        useNewUrlParser: true,
        useUnifiedTopology: true
});
```

https://www.mongodb.com/try/download/community

SCHEMA

- Mongoose forces a semi-rigid schema from the beginning.
- With Mongoose, developers must define a Schema and Model.
- A schema is a structure, that gives information about how the data is being stored in a collection.
- A Mongoose schema maps directly to a MongoDB collection.

```
const messageSchema = {
            userName: String,
            message: String,
      };
```

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

CONFIGURING THE DATABASE

Another schema

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

- Models take your schema and apply it to each document in its collection.
- Models are responsible for all document interactions like creating, reading, updating, and deleting (CRUD).
- 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);
```

- The first argument passed to the model should be the singular form of your collection name.
- Mongoose automatically changes this to the plural form, transforms it to lowercase, and uses that for the database collection name.

CONFIGURING THE DATABASE

Another schema

```
const messageSchema = {
        userName: String,
        message: String,
};
```

- The first argument passed to the model should be the singular form of your collection name.
- Mongoose automatically changes this to the plural form, transforms it to lowercase, and uses that for the database collection name.
- One would create a schema/model file for each schema that is needed

```
import mongoose from 'mongoose';
const { Schema, model } = mongoose;

const messageSchema = {
        userName: String,
        message: String,
    };

const Message = model('Message', messageSchema);
export default Message;
```

STORING DATA

```
const mess = new Message({
        message: "Hello class",
        userName: "Chandreyee",
        });
await mess.save();
```

- 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,
});


await contact.save();
```

STORING DATA

- The create() method instantiates and saves the object in one action

```
Message.create({ message: 'small chat msg', userName:'chandreyee'})
      .then(result => {console.log(result);})
                  .catch(err => {});
```

- // or, for inserting large batches of documents

```
Contact.insertMany([{ query: 'small' }])
```

- To find a matching data from the collection

```
 const firstMessage=Message.findOne({userName:'chac'}).then((docs)=>{
       console.log("Result :",docs);
   })
   .catch((err)=>{
       console.log(err);
});
```

HANDLING ERRORS

If you only need to handle Promise transitions to the Rejected state, rather than passing a null first parameter to then(), you can instead use the catch() method which accepts a single callback, executed when the Promise transitions to the Rejected state

```
.catch((err)=>{
        console.log(err);
});
```

FIND AND UPDATE

- Mongo DB way of querying the database

```
const messageWhere = await Blog.where(“userName").equals(“CR1");

console.log(messageWhere)

const messageWhere = await
Blog.where(“userName").equals(“CR1").select(“message”);
```

Select is used for projection here

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 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().then(contact=>{
      res.send(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');
        });
```

REQUIRE SYNCHRONY

- The essential concept to remember is that everything inside a module is private unless it's assigned to the `module.exports` variable.
- The contents of this variable are then cached and returned when the module is loaded using `require()`.

- `require` function is synchronous.
- In fact, it returns the module contents using a simple direct style, and no callback is required
- In its early days, Node.js used to have an asynchronous version of `require()`, but it was soon removed because it was overcomplicating a functionality that was actually meant to be used only at initialization time, and where asynchronous I/O brings more complexities than advantages.
- The term *dependency hell*, describes a situation whereby the dependencies of a software, in turn depend on a shared dependency, but require different incompatible versions.
- Node.js solves this problem elegantly by loading a different version of a module depending on where the module is loaded from.
- All the merits of this feature go to `npm` and also to the resolving algorithm used in the `require` function

RESOLVING DEPENDENCIES

- The `resolve()` function takes a module name (which we will call here, `moduleName`) as input and it returns the full path of the module.
- This path is then used to load its code and also to identify the module uniquely.
- The resolving algorithm can be divided into the following three major branches:
- • **File modules**: If `moduleName` starts with "`/`" it's considered already an absolute path to the module and it's returned as it is. If it starts with "`./`", then `moduleName` is considered a relative path, which is calculated starting from the requiring module.
- • **Core modules**: If `moduleName` is not prefixed with "`/`" or "`./`", the algorithm will first try to search within the core Node.js modules.
- • **Package modules**: If no core module is found matching `moduleName`, then the search continues by looking for a matching module into the first `node_modules` directory that is found navigating up in the directory structure starting from the requiring module.

RESOLVING DEPENDENCIES

- The algorithm continues to search for a match by looking into the next `node_modules` directory up in the directory tree, until it reaches the root of the filesystem.
- The resolving algorithm is applied transparently for us when we invoke require(); however, if needed, it can still be used directly by any module by simply invoking require.resolve().

- `myApp`, `depB`, and `depC` all depend on `depA`; however,they all have their own private version of the dependency
- Calling `require('depA')` from `/myApp/foo.js` will load `/myApp/node_modules/depA/index.js`
- Calling `require('depA')` from `/myApp/node_modules/depB/bar.js` will load `/myApp/node_modules/depB/node_modules/depA/index.js`

```
myApp
├── foo.js
└── node_modules
    ├── depA
    │   └── index.js
    ├── depB
    │   ├── bar.js
    │   └── node_modules
    │       └── depA
    │           └── index.js
    └── depC
        ├── foobar.js
        └── node_modules
            └── depA
                └── index.js
```

- 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;`