Botlhale Village
Working together for ICT innovation and growth in Africa

BELGIUM CAMPUS iTversity
It's the way we're wired

WPR252

BELGIUM CAMPUS
iTversity
It's the way we're wired

Node's Programming Model

Synchronous vs. Asynchronous

- Node.js uses asynchronous application design
- Most languages performs synchronous I/O (also known as blocking I/O) which means that when they begin some I/O operations(suck as disk read, network call), they sits idle until the operation completes.
- Typically, languages that use blocking I/O calls are also multi-threaded: therefore one thread is idle, another thread can perform some meaningful work.
- Javascript is single-threaded.
- Node.js uses callback functions, event emitters and promises to achieve asynchronous operations

Callbacks

- A callback function is function that is invoked at the completion of an asynchronous operation with the results of the operation passed as a function arguments.
- A callback function is passed as an argument and it should be the last argument, then the callback is invoked inside function.
- The code below make use of callback and setTimeout time.
- addStudent adds a student to the array and getStudents prints array objects.
- The idea is first add a student to the array and then print the new array contents.
- SetTimeout is used resemble the database operations: it take longer to add data to the database than extract it

```
let student = ['peter', 'john', 'matha', 'thato'];
function addStudent(stud,callback) {
    setTimeout(function () {
        student.push(stud)
        callback();
    },2000)
}
function getstudents() {
    setTimeout(function () {
        for (var i = 0; i < student.length; i++) {
            console.log(student[i]);
        }
    },1000)
}
addStudent('mike', getstudents);
```

Promise

- A Promise object represents an operation which *has produced or will eventually produce* a value.
- Promises provide a robust way to wrap the (possibly pending) result of asynchronous work, mitigating the problem of deeply nested callbacks (known as "callback hell").
- A promise is an executor function which has a resolve and a reject callback
- each of these callbacks returns a value with resolve returning a promised value while the reject callback returning an error object.

A promise can be in one of three states:

1. *pending* — The underlying operation has not yet completed, and the promise is *pending* fulfillment.
2. *fulfilled* — The operation has finished, and the promise is *fulfilled* with a *value*. This is analogous to returning a
   value from a synchronous function.
3. *rejected* — An error has occurred during the operation, and the promise is *rejected* with a *reason*. This is analogous to throwing an error in a synchronous function.

Structure of a promise


```
const promise = new Promise((resolve, reject) => {
// Perform some work (possibly asynchronous)
if (/* Work has successfully finished and produced "value" */) {
resolve(value);
} else {
// Something went wrong because of "reason"
// The reason is traditionally an Error object, although // this
is not required or enforced.
let reason = new Error(message);
reject(reason);
// Throwing an error also rejects the promise.
throw reason;
}
});
```

Structure of a promise

The then and **catch** methods can be used to attach fulfillment and rejection callbacks:

```
promise.then(value => {
// Work has completed successfully,
// promise has been fulfilled with "value"
}).catch(reason => {
// Something went wrong,
// promise has been rejected with "reason"
});
Find an example of date promise on
```

```
let studentDB = [
    { name: 'jack', WPR252: 67, STAT252: 76, MAT252: 56 },
    { name: 'zane', WPR252: 71, STAT252: 76, MAT252: 51 },
    { name: 'thato', WPR252: 57, STAT252: 76, MAT252: 62 }
]
function findStudent(studname) {
    let position = new Promise(function (resolve, reject) {
        let index = studentDB.map(obj => obj.name).indexOf(studname);
        if (index !== -1) {
            resolve(index);
        } else {
            reject('student does not exists in the database')
        }
    })
    return position;
}
findStudent('th').then(function (fromResolve) { console.log(studentDB[fromResolve])
})
.catch(function (fromReject) { console.log(fromReject); });
```

```
function checkPassedModules(student) {
    return new Promise(function (resolve) {
        if (studentDB[student].MAT252 >= 50 && studentDB[student].STAT252>=50 &&
studentDB[student].WPR252>=50) {
            console.log('the student passed all the modules');
        } else {
            console.log('the student has to redo some modules');
        }
        resolve(student);
    })
}
//findStudent('thato').then(checkPassedModules)
//    .catch(function (fromReject) { console.log(fromReject); });
function average(student) {
    return new Promise(function (resolve) {
        console.log((studentDB[student].MAT252 + studentDB[student].STAT252 +
studentDB[student].WPR252)/3);
    })
}
findStudent('thato').then(checkPassedModules).then(average)
    .catch(function (fromReject) { console.log(fromReject); });
```

Events

- In Node.js applications, Events and Callbacks concepts are used to provide concurrency.
- Event-Driven Programming makes use of the following concepts:

❖ An Event Handler is a callback function that will be called when an event is triggered.

❖ A Main Loop listens for event triggers and calls the associated event handler for that event.

- Node.js has a useful module called EventEmitter that enable Event-Driven Programming.
- Imagine a chat room where an alert is send to everyone when a new user joins the chat room.
- an event listener for a userJoined event is needed.
- First, we'll write a function that will act as our event listener, then we can use EventEmitters on method to set the listener.

```
const EventEmitter = require('events').EventEmitter;
const chatRoomEvents = new EventEmitter;
function userJoined(username){
// Assuming we already have a function to alert all users.
alertAllUsers('User ' + username + ' has joined the chat.'); }
// Run the userJoined function when a 'userJoined' event is triggered.
chatRoomEvents.on('userJoined', userJoined);
```

The next step would be to make sure that our chat room triggers a userJoined event whenever someone logs in so that our event handler is called.

EventEmitter has an emit method that we use to trigger the event.

We would want to trigger this event from within a login function inside of our chatroom module.

```
function login(username){ chatRoomEvents.emit('userJoined', username); }
```

To remove event listeners in EventEmitter we can use the removeListener or removeAllListeners method.