Blockchain Development
Week: 5
Title: Node.js

Dr Ian Mitchell

Middlesex University,
Dept. of Computer Science,
London

September 26, 2019

Aims & Objectives

- Overview of Javascript
- Overview of Nodejs [2]
- A/Synchronous Programming
- Examples

Node.js

- Client-side script
    - GUI
    - Web
    - Mobile
    - JS, CSS, HTML
- Server-side script
    - Web
    - REST
    - HTTP
    - Ajax
    - Messaging
    - lang: ASP.Net, Java, PHP
    - tools: MySQL

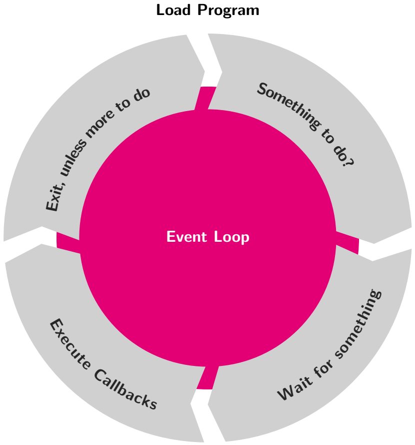
Middleware

- I/O-bound
- Server-side have to wait
- input query
- output result
- all processes halted
- Distributed
- Node.js

Felix Geisendø" rfer

"Everything runs in parallel except your code"

- Events
- Callbacks
- Listening
- Create callback functions that get executed in response to listening to events
- Non-blocking



Node.js

Single-Threaded and Highly Parallel

- Run code

Backwardism

-

Why?

- Composer
- Asynchronous
- Non-blocking
- Single-Threaded
- Event-based

Synchronous

- Sequence

Listing

```
console.log('Start');
console.log('End');
```

Synchronous

- Sequence

Output

```
Start
End
```

Listing

```
console.log('Start');
console.log('End');
```

Asynchronous

- `SetTimeout`(*fn*, *ms*)
- Exec. fn after ms
- Order $\neq$ Code
- Non-blocking, continue to execute program

Listing

```
console.log('Start');
setTimeout( () => {console.log('Callback');},2000);
console.log('End');
```

Asynchronous

- `SetTimeout`(*fn*, *ms*)
- Exec. fn after ms
- Order $\neq$ Code
- Non-blocking, continue to execute program

Listing

```
console.log('Start');
setTimeout( () => {console.log('Callback');},2000);
console.log('End');
```

Output

```
Start
End
Callback
```

Asynchronous

Listing

```
console.log('Start');
setTimeout( () => {console.log('1st Callback')
      ;},2000);
setTimeout( () => {console.log('2nd Callback');},
      0);
console.log('End');
```

- Again
- Order $\neq$ Code
- Non-blocking, continue to execute program

Asynchronous

Output

```
Start
End
2nd Callback
1st Callback
```

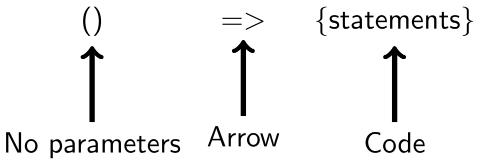
- Again
- Order $\neq$ Code
- Non-blocking, continue to execute program

Listing

```
console.log('Start');
setTimeout( () => {console.log('1st Callback')
      ;},2000);
setTimeout( () => {console.log('2nd Callback');},
      0);
console.log('End');
```

Arrow Function: No Parameter

Syntax

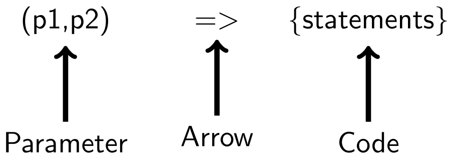


Arrow Function: Single Parameter

Syntax

$$\begin{tikzcd}
\text{(p1)} & => & \{\text{statements}\} \\
\text{Parameter} \arrow[u] & \text{Arrow} \arrow[u] & \text{Code} \arrow[u]
\end{tikzcd}$$

Arrow Function: Multiple Parameter
Syntax



```
(p1,p2)    =>    {statements}
```

Arrow Functions
Comparison

Listing without

```
var add = function(x,y){return x+y;}
console.log(add(3,7));
```

Listing with

```
var add = (x,y) => x+y;
console.log(add(3,7));
```

Arrow Functions
Benefits

- Shorter
- Bind `this` lexically
-

Anonymous Callback

Definition

Passing a function as an argument

Example

```
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

mathOperate(10,5,function(u,v){return u*v
     ;});
mathOperate(10,5,function(u,v){return u+v
     ;});
```

- Why?
- Dynamic

Anonymous Callback

Output

```
result: 50
result: 15
```

- Why?
- Dynamic

Example

```
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

mathOperate(10,5,function(u,v){return u*v
     ;});
mathOperate(10,5,function(u,v){return u+v
     ;});
```

Named Callback

- Why?
- Dynamic
- trigger automatic updates
- `setInterval(fn,ms)`

Example

```
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

function times(u,v){return u*v;}
function add(u,v){return u+v;}
function mod(u,v){return u%v;}

mathOperate(10,7,times);
mathOperate(10,7,add);
mathOperate(10,7,mod);
```

Named Callback

Output

```
result: 70
result: 17
result: 3
```

- Why?
- Dynamic
- trigger automatic updates
- `setInterval(fn,ms)`

Example

```javascript
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

function times(u,v){return u*v;}
function add(u,v){return u+v;}
function mod(u,v){return u%v;}

mathOperate(10,7,times);
mathOperate(10,7,add);
mathOperate(10,7,mod);
```

Promises [1]

Definition [3]

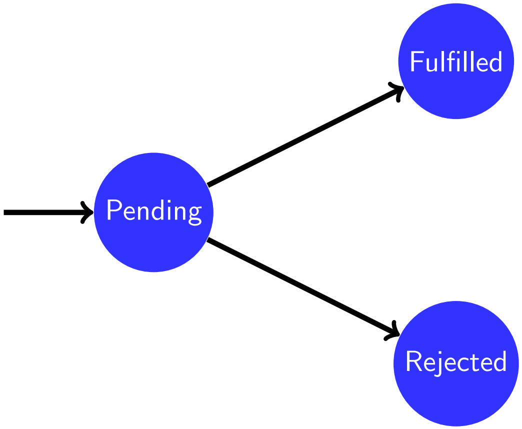
A promise is an object that serves as a placeholder for a value. That value is usually the result of an async[hronous] operation.... When an async function is called it can immediately return a promise object. Using that object, you can register callbacks that will run when the operation succeeds or an error occurs.

Promise States [3]

Definitions

Pending:	The operation has not begun or is in progress.
Fulfilled:	The operation has completed.
Rejected:	The operation could not be completed.

Promise States Relationships[3]



Promise Analogy

Child: Please can I have some sweets?

Parent: I will give you some when you complete your homework

Promise Pending

Promise Analogy

Child: Please can I have some sweets?

Parent: I will give you some when you complete your homework

Promise Pending

⋮

Child: Can I have some sweets now!

Parent: Have you completed your homework?

Child: No.

Parent: Then you cannot have any sweets.

Promise Rejected

Promise Analogy

Child:	Please can I have some sweets?
Parent:	I will give you some when you complete your homework

Promise Pending

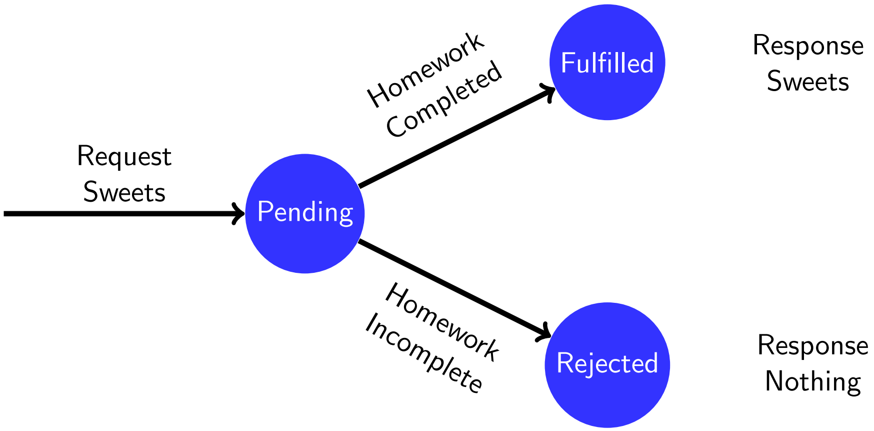
Child:	Can I have some sweets now?
Parent:	Have you completed your homework?
Child:	Yes.
Parent:	Well done, I will go and get you some.

Promise Pending

Parent:	They are on the table.

Promise Fulfilled

Promise States Relationships[3]



Promise Syntax[2]

- Creation: object

Listing

```
var somePromise = new promise((resolve,
    reject)=>{
//do asynchronous stuff here
  });

```

Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn

Listing

```
var somePromise = new promise((resolve,
    reject)=>{
//do asynchronous stuff here
  });

```

Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn
- Asynchronous

Listing

```
var somePromise = new promise((resolve,
    reject)=>{
//do asynchronous stuff here
  });

```

Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn
- Asynchronous
- Resolve, Reject

Listing

```
var somePromise = new promise((resolve,
    reject)=>{
//do asynchronous stuff here
  });

```

Promise Example
Resolve

Output

```
success:  It worked
```

Listing

```
var somePromise = new Promise((resolve,reject)=>{
  setTimeout(()=>{
      resolve('It worked');
      resolve('It worked again');//won't run - promises can only either be resolved or rejected once
      },500);

});

somePromise.then((message) => {
  console.log('success:',message);},
  (errorMessage) => {
  console.log('Failure:',errorMessage);
});
```

Promise Example

Reject

Output

```
Failure:  It Failed
```

Listing

```
var somePromise = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    reject('It Failed');
      },500);

});

somePromise.then((message) => {
  console.log('success:',message);},
  (errorMessage) => {
  console.log('Failure:',errorMessage);
});
```

Call a Promise
.then

- .then is a callback function
  - success
  - failure
- two callback functions
- Reject or Resolve
- Only reject once
- Only resolve once
- Pending for 500ms

Listing

```
somePromise.then((message) => {
  console.log('success:',message);},
  (errorMessage) => {
  console.log('Failure:',errorMessage);
});
```

Return a Promise

Resolve

Output

sum:109

Listing

```
var asyncAdd = (a,b) => {
  return new Promise((resolve,reject) =>{
    setTimeout(()=>{
      if( (typeof a === 'number') && (typeof b === 'number')) {
        resolve(a+b);
      } else {
        reject('enter two numbers');
      }
    },500);
  });

};

asyncAdd(34,75).then(
  //first callback is the success - resolve case
  (message)=>{console.log('Sum:',message);},
  //second callback is the failure - reject case
  (errorMessage)=>{console.log('Error:',errorMessage);}
  );
```

Return a Promise

Resolve

Output

Error: enter two numbers

Listing

```
var asyncAdd = (a,b) => {
  return new Promise((resolve,reject) =>{
    setTimeout(()=>{
      if( (typeof a === 'number') && (typeof b === 'number')) {
        resolve(a+b);
      } else {
        reject('enter two numbers');
      }
    },500);
  });

};

asyncAdd(5,'a').then(
  //first callback is the success - resolve case
  (message)=>{console.log('Sum:',message);},
  //second callback is the failure - reject case
  (errorMessage)=>{console.log('Error:',errorMessage);}
  );
```

Promise Chaining [3]

Listing

```
function printStr(prev,curr,t){
    return new Promise((resolve,reject)=>{
    setTimeout(
      ()=>{ if((typeof(prev)=='string') && (typeof(curr)=='string')
     ){
            resolve(prev+curr)
         } else {
            reject('Enter Strings only')
            }
         },
      t);
    })
}
function printAll(){
  printStr('','A',2500)
  .then( (result)=>printStr(result,'B',250))
  .then( (result)=>printStr(result,'C',25))
  .then( (result)=>console.log(result))
  .catch( result=>console.log(result))
}
async function printAll2(){
  let str=''
  str=await printStr(str,'X',2500)
  str=await printStr(str,'Y',250)
  str=await printStr(str,'Z',25)
  console.log(str);
}
printAll();
setTimeout(()=>{printAll2()},5000)
```

Promise Chaining [3]

Output

```
ABC
XYZ
```

Listing

```
function printStr(prev,curr,t){
    return new Promise((resolve,reject)=>{
    setTimeout(
      ()=>{ if((typeof(prev)=='string') && (typeof(curr)=='string')
     ){
          resolve(prev+curr)
        } else {
          reject('Enter Strings only')
          }
        },
      t);
    })
}
function printAll(){
  printStr('','A',2500)
  .then( (result)=>printStr(result,'B',250))
  .then( (result)=>printStr(result,'C',25))
  .then( (result)=>console.log(result))
  .catch( result=>console.log(result))
}
async function printAll2(){
  let str=''
  str=await printStr(str,'X',2500)
  str=await printStr(str,'Y',250)
  str=await printStr(str,'Z',25)
  console.log(str);
}
printAll();
setTimeout(()=>{printAll2()},5000)
```

Summary

- Promises
- passing functions as parameters
- Asynchronous code
- Synchronous code
- Promise chaining
- callback hell
- async, await, let

References I

[1] B. Liskov and L. Shrira. "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems". In: *Int. Conf. on Programming Language Design and Implementation (SIGPLAN'88)*. 1988, pp. 260–267.

[2] A. Mead. *Learning Node.js Development*. Packt, 2018.

[3] D. Parker. *Javascript with Promises*. 1st ed. O'Reilly, 2015.

Web Resources

- `http://hyperledger.org`
- `https://nodejs.org`