Advanced Software Engineering Internet Applications

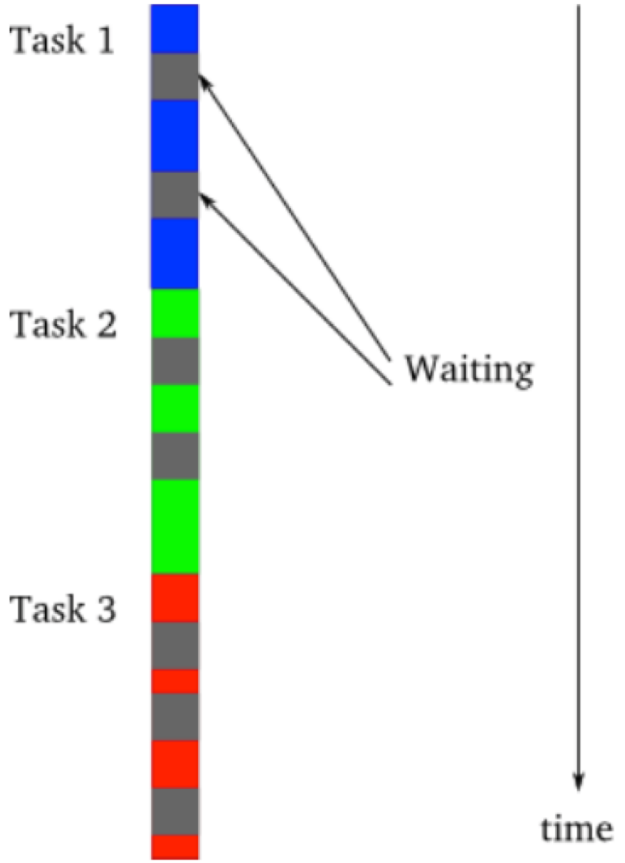
A Few Words about Asynchronous Programming

Overview

1. Motivation
2. Definition / Alternative Models
3. Illustrative Example (very simple in Python)
4. Evolution of Async Programming within Node.js
   - Callbacks
   - Promises
   - async/await
5. Wrapping up
   - Recap
   - Node.js Model Summary
6. Bibliography

1. Motivation – 1

In a single-threaded synchronous program, if n (independent) tasks have to be performed, the CPU wastes a lot of time waiting (e.g. on disk or network I/O).



From : http://cs.brown.edu/courses/cs168/f16/content/docs/async.pdf

1. Motivation – 2

The only reason behind asynchronous programming is **making code go fast** without consuming too much memory resources.

For a busy (REStful) server with a lot of connections and little memory resources (e.g. embarked on a smart thing), it offers **much better scalability** than other solutions. (e.g. a multi-thread server needs more memory resources and does not scale as well).

2. Definition / Alternatives – 1

Definition 1

Asynchronous programming is a style of concurrent programming, which means making many things at once.

The way to do it is to use the processor at its maximum by liberating it, while slower tasks (e.g. reading / writing on a disk, receiving / sending network packets, or executing database requests) are taking place.

Three models are possible to implement concurrent programming :

- Using multiple processes. The OS does the work but processes are quite heavy and resources demanding.
- Using a multi-threaded synchronous model. Threads are less resources consuming than processes, but they often share access to the same spaces and programming with events and many threads can become quite a headache.
- Using an asynchronous programming language and / or library.

2. Definition / Alternatives – 2

Definition 2

Asynchronous programming is a style of concurrent programming, in which tasks release the CPU during waiting periods so that other tasks can use it.

It is based on cooperative multi-tasking.

From : http://cs.brown.edu/courses/cs168/f16/content/docs/async.pdf

2. Definition / Alternatives – 3

Processes vs. Threads vs. Async

	Processes	Threads	Async
Optimize waiting periods	Yes (preemptive)	Yes (preemptive)	Yes (cooperative)
Use all CPU cores	Yes	No 	No
Scalability	Low (ones/tens)	Medium (hundreds)	High (thousands+)
Use blocking std library functions	Yes	Yes	No
GIL interference	No	Some	No

From : https://www.youtube.com/watch?v=iG6fr81xHKA&feature=youtu.be

3. Illustrative Example

```python
from time import sleep

def hello():
    print('Hello')
    sleep(3)
    print('World!')

if __name__ == '__main__':
    for i in range(10):
        hello()

```

```python
import asyncio
loop = asyncio.get_event_loop()

async def hello():
    print('Hello')
    await asyncio.sleep(3)
    print('World!')

if __name__ == '__main__':
    for i in range(10):
        loop.create_task(hello())
    loop.run_forever()
```

See all examples on GitHub: https://bit.ly/asyncpython

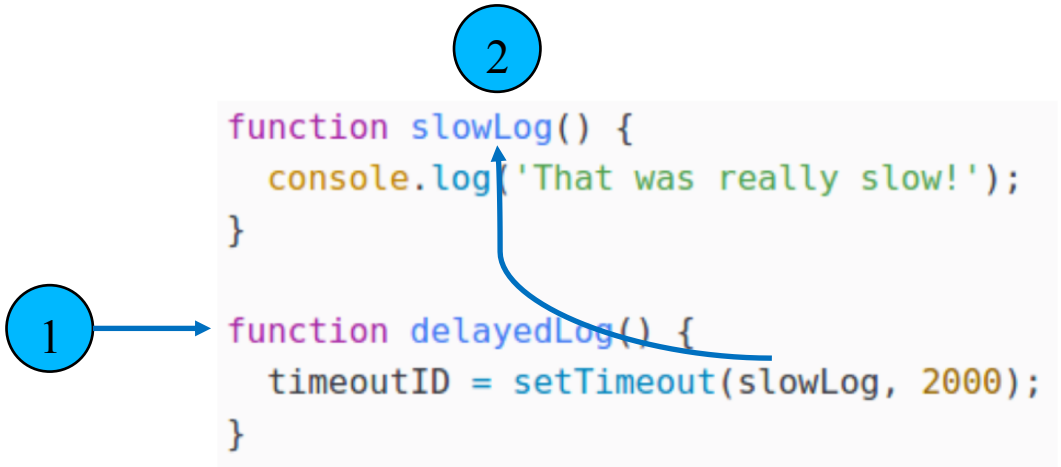
4. Node.js : Callbacks – 1

Callbacks are functions passed as parameter and called upon completion

Exemple: **setTimeOut()**

```
var timeoutID = scope.setTimeout(function[, delay, param1,
param2, ...]);
var timeoutID = scope.setTimeout(function[, delay]);
var timeoutID = scope.setTimeout(code[, delay]);
```

The **setTimeout()** method sets a timer which executes a function or specified piece of code once after the timer expires.



4. Node.js : Callbacks – 2

We will use a **resolveAfterXSeconds()** function all along the examples

```
function resolveAfter2Seconds (done, x) {
  setTimeout(function () {
    done(x);
  }, 2000)
}

var resolveAfter3Seconds = (done, x) => setTimeout(() => done(x), 3000);
```

param

Running asynchronous functions sequentially

```
resolveAfter2Seconds(function (x) {
    console.log('waited 2 seconds');
    resolveAfter3Seconds(function (y) {
        console.log('waited 5 seconds, got '+y);
    } x);
} 20);
```

4. Node.js : Callbacks – 3

Asynchronous code that should execute sequentially can lead to extreme situations

```
step1(value, function(data {
  step2(data, function(data2 {
    step3(data2, function(data3){
      step4(data3, function(data4){
        //INSTRUCTIONS
      });
    });
  };
};
```

This is referred as callback hell

4. Node.js : Promises – 1

Meet promises

```
step1(value, function(data){
  step2(data, function(data2){
    step3(data2, function(data3){
      step4(data3, function(data4){
         //INSTRUCTIONS
      });
    });
  });
});
```

```
step1(value)
  .then(step2)
  .then(step3)
  .then(step4)
```

- A promise represents the result of an asynchronous operation.
- They can also be called futures (mostly in other languages)
- They don’t require language-specific support

4. Node.js : Promises – 2

Creating a promise

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}
```

Using promises

```
resolveAfter2Seconds(20)
  .then(resolveAfter3Seconds)
  .then(console.log);
```

4. Node.js : Promises – 3

Querying a MySQL DB using promises

```
var mysql = require('promise-mysql');
var connection;

mysql.createConnection({
  host: 'localhost',
  user: 'sauron',
  password: 'theonetruering',
  database: 'mordor'
}).then(function(conn){
  connection = conn;
  return connection.query('select `id` from hobbits where `name`="frodo"');
}).then(function(rows){
  // Query the items for a ring that Frodo owns.
  var result = connection.query('select * from items where `owner`="'
                                + rows[0].id + '" and `name`="ring"');
  connection.end();
  return result;
}).then(function(rows){
  // Logs out a ring that Frodo owns
  console.log(rows);
});
```

4. Node.js : Promises – 4

Catching errors

```
var mysql = require('promise-mysql');
var connection;

mysql.createConnection({
  host: 'localhost',
  user: 'sauron',
  password: 'theonetruering',
  database: 'mordor'
}).then(function(conn){
  connection = conn;
  return connection.query('select * from tablethatdoesnotexist');
}).then(function(){
  var result = connection.query('select * from hobbits');
  connection.end();
  return result;
}).catch(function(error){
  if (connection && connection.end) connection.end();
  //logs out the error
  console.log(error);
});
```

4. Node.js : async/await – 1

> The purpose of `async` / `await` functions is to simplify the behavior of using promises synchronously and to perform some behavior on a group of `Promises`. Just as `Promises` are similar to structured callbacks, `async` / `await` is similar to combining generators and promises. [3]

```
async function add1(x) {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter3Seconds(30);
  return x + a + b;
}

add1(10).then(v => {
  console.log(v);  // prints 60 after 5 seconds.
});
```

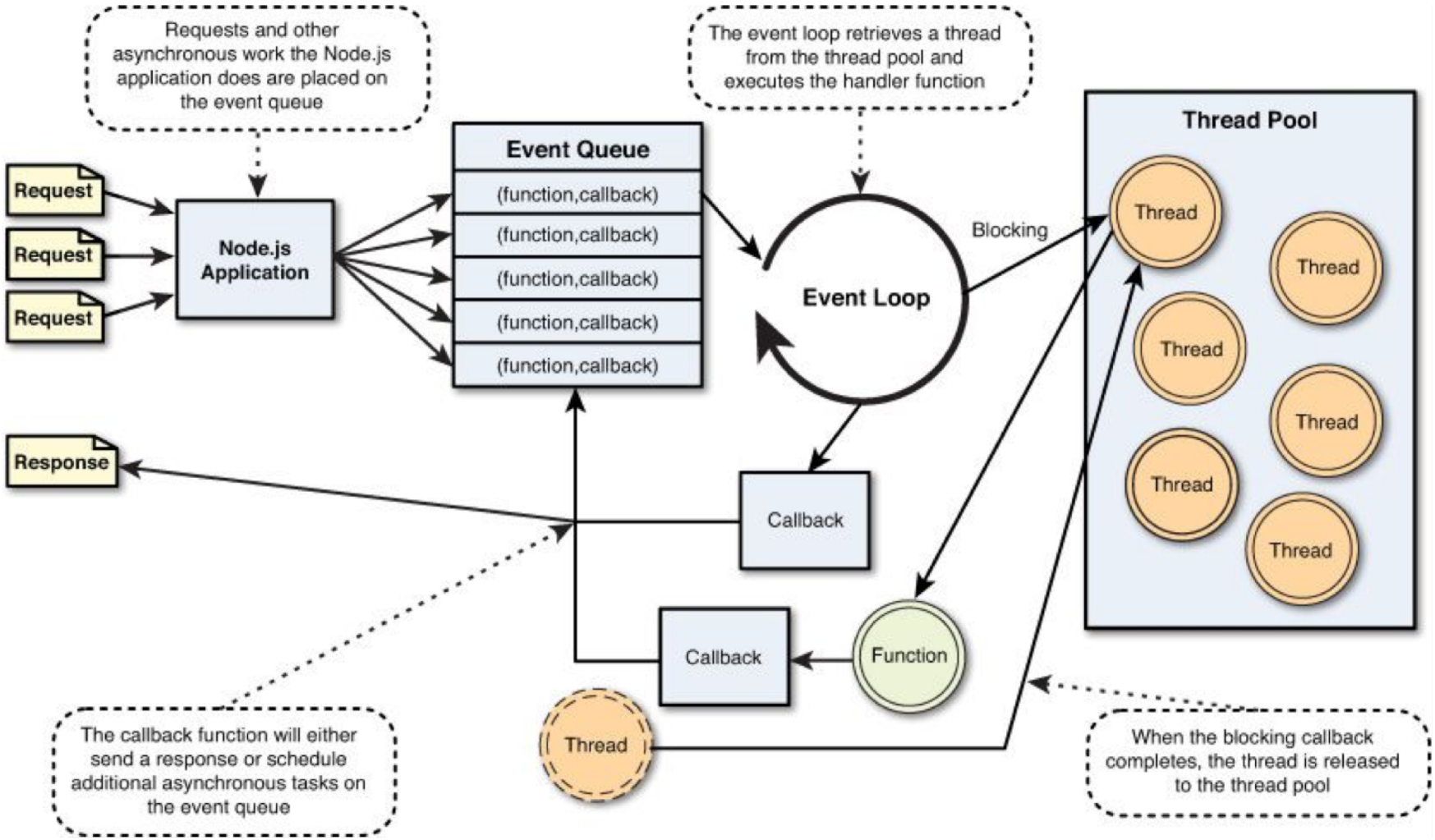
4. Node.js : async/await – 2

Let's rewrite the previous DB query example using async/await

```
var mysql = require('promise-mysql');
var connection;

async function query() {
  try {
    connection = await mysql.createConnection({
      host: 'localhost',
      user: 'sauron',
      password: 'theonetruering',
      database: 'mordor'
    })
    var rows = await connection.query('select `id` from hobbits where `name`="frodo"');
    var result = await connection.query('select * from items where `owner`="' + rows[0].id
                                        + '" and `name`="ring"');
    console.log(result);
  }
  catch(e) {
    console.log(error);
  }
  finally {
    if (connection && connection.end) connection.end();
  }
}
```

5. Wrapping up : Node Model Summary



5. Wrapping up : Recap

- A **callback** is a primitive of asynchronous programming
  - often leads to hard to read, unmaintainable code
- A **promise** is a pattern to flatten the code
  - software pattern = doesn't require language-specific support
  - some caveats
    - no branching
    - long chain of promises are hard to read
    - a lot of libs doesn't have built-in promises (yet?)
- **async/await** keywords simplify the behaviour of using promises
  - language feature
    - ECMAScript 2017 (ES8) feature (use promises)
    - Python 3.5 feature (built upon coroutines)
- **Generators** (coroutines) are another approach to asynchronous programming
  - these are functions that return intermediates values
    - not widely used in JavaScript
    - extensively used in Python

6. Bibliography

- Miguel Grinberg Asynchronous Python for the Complete Beginner PyCon 2017
  - https://youtu.be/iG6fr81xHKA
- Implementing Promises by Forbes Lindesay
  - https://www.promisejs.org/implementing/
- Async function – JavaScript
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Dayley, B. (2014). Node.js, MongoDB and AngularJS web development.