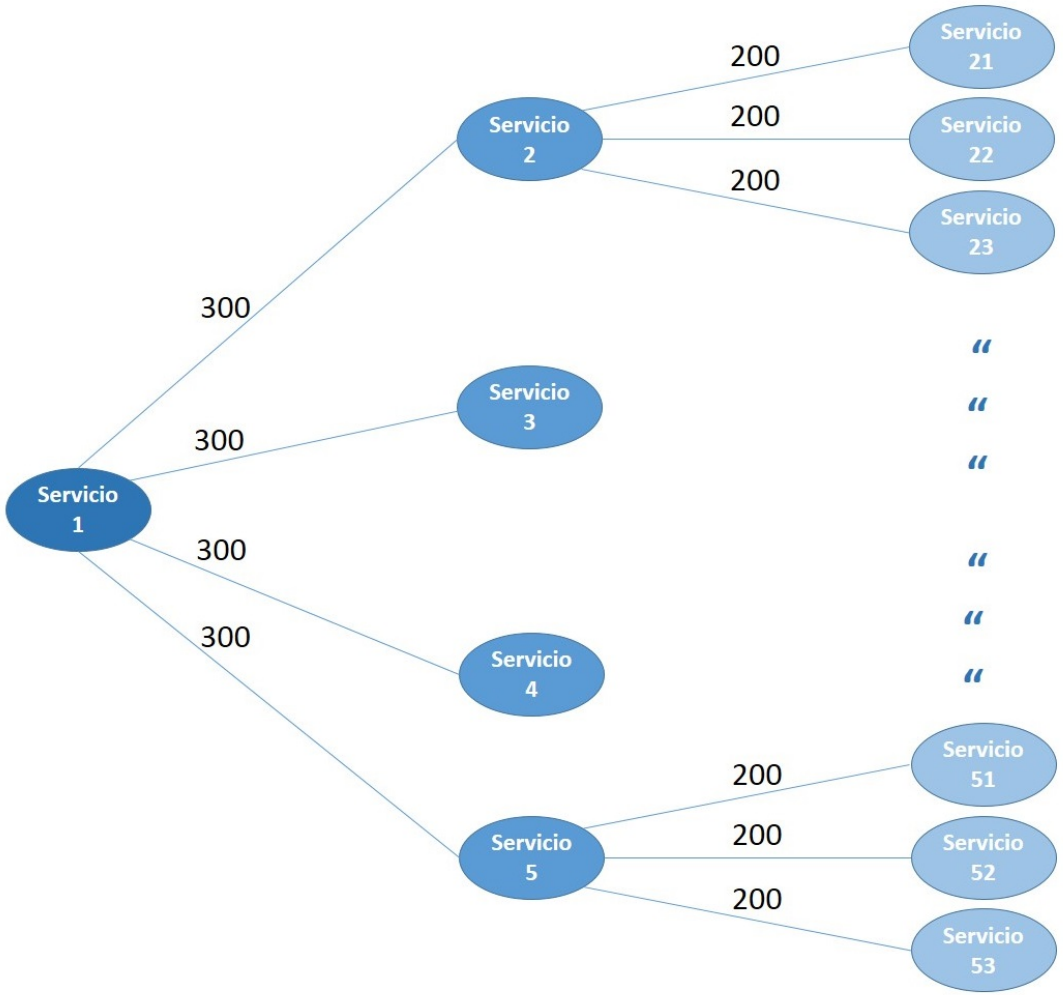
Programación asincrónica

Carlos Lombardi

Universidad Nacional de Quilmes – Argentina

Sesiones sobre microservicios

Asincronismo – por qué



Ejecución sincrónica:
$(300 + 200 * 3) * 4 =$
3600 ms.

Ejecución asincrónica:
$300 + 200 = 500$ ms.

Callbacks

En lugar de

```
let tempBsAs = getTemp('BsAs')
```

Accedemos a valores via callbacks.

```
let tempBsAs = 0
getTemp('BsAs', function(error, result) {
    tempBsAs = result
})
```

Callbacks – salen del flujo de ejecución

```
getTemp('BsAs', function(error, result) {
    console.log('Temperatura en Buenos Aires:  ' + result)
})
console.log('después de la invocación')
```

¿En qué orden aparecen los mensajes?

Callbacks – uso ingenuo

Un caso similar, con varias invocaciones.

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
getTemp('BsAs', function(error, result) {
    tempBsAs = result
})
getTemp('Rawson', function(error, result) {
    tempRawson = result
})
getTemp('Salta', function(error, result) {
    tempSalta = result
})
console.log('Temperatura en Buenos Aires: ' + tempBsAs)
console.log('Temperatura en Rawson: ' + tempRawson)
console.log('Temperatura en Salta: ' + tempSalta)
```

Callbacks – corregimos

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
getTemp('BsAs', function(error, result) {
  tempBsAs = result
  getTemp('Rawson', function(error, result) {
    tempRawson = result
    getTemp('Salta', function(error, result) {
      tempSalta = result
      console.log('Temperatura en Buenos Aires: ' + tempBsAs)
      console.log('Temperatura en Rawson: ' + tempRawson)
      console.log('Temperatura en Salta: ' + tempSalta)
    })
  })
})
```

Callbacks – corregimos

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
getTemp('BsAs', function(error, result) {
  tempBsAs = result
  getTemp('Rawson', function(error, result) {
    tempRawson = result
    getTemp('Salta', function(error, result) {
      tempSalta = result
      console.log('Temperatura en Buenos Aires: ' + tempBsAs)
      console.log('Temperatura en Rawson: ' + tempRawson)
      console.log('Temperatura en Salta: ' + tempSalta)
    })
  })
})
```

Bienvenidos al callback hell . . .

Callbacks – corregimos

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
getTemp('BsAs', function(error, result) {
  tempBsAs = result
  getTemp('Rawson', function(error, result) {
    tempRawson = result
    getTemp('Salta', function(error, result) {
      tempSalta = result
      console.log('Temperatura en Buenos Aires: ' + tempBsAs)
      console.log('Temperatura en Rawson: ' + tempRawson)
      console.log('Temperatura en Salta: ' + tempSalta)
    })
  })
})
```

Bienvenidos al callback hell . . .

. . . y las llamadas no dejan de ser sincrónicas.

Promises (1)

Alternativa a los callbacks.

En lugar de

```
let tempBsAs = 0
getTemp('BsAs', function(error, result) {
    tempBsAs = result
})
```

Hacemos

```
let tempBsAs = 0
getTempPromise('BsAs')
  .then(function(result) { tempBsAs = result })
```

Promises (2)

Una Promise es un objeto que encapsula una operación asincrónica.

- Entiende los mensajes `then` y `catch`. Los parámetros de estos mensajes son funciones.
- Cuando la operación termina, se invoca a la función del `then`, con el resultado como parámetro.
- Si la operación da error, se invoca a la función del `catch`, con el resultado como parámetro.

```
let tempBsAs = 0
getTempPromise('BsAs')
  .then(function(result) { tempBsAs = result })
  .catch(function(theError) { console.log(theError) })
```

Promises – salimos del callback hell

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
getTempPromise('BsAs')
.then(function(result) {
    tempBsAs = result
    return getTempPromise('Rawson')
}).then(function(result) {
    tempRawson = result
    return getTempPromise('Salta')
}).then(function(result) {
    tempSalta = result
    console.log('Temperatura en Buenos Aires: ' + tempBsAs)
    console.log('Temperatura en Rawson: ' + tempRawson)
    console.log('Temperatura en Salta: ' + tempSalta)
})
```

De callback a promise

- Se usa constructor de la clase `Promise`.
- El parámetro es una función que llama a la función original.
- El secreto está en cómo armar el callback.

```
function getTempPromise(city) {
  return new Promise(function(fulfill,reject) {
    getTemp(city, function(theError,response,body) {
      if (theError) { reject(theError) }
      else { fulfill(body) }
    })
  })
}
```

Las funciones `fullfill` y `reject` van a corresponder a los argumentos de la invocación a `.then` y `.catch`.

Promises – paralelizamos operaciones

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
Promise.all([
    getTempPromise('BsAs'),
    getTempPromise('Rawson'),
    getTempPromise('Salta')])
.spread(function(tempBsAs, tempRawson, tempSalta) {
    console.log('Temperatura en Buenos Aires: ' + tempBsAs)
    console.log('Temperatura en Rawson: ' + tempRawson)
    console.log('Temperatura en Salta: ' + tempSalta)
})
```

- Uso de `Promise.all` para paralelizar requests.
- Uso de `.spread` en lugar de `.then`.

Promise.all – el manejo de error sigue siendo simple

```
let tempBsAs = 0 ;  let tempRawson = 0 ;  let tempSalta = 0
Promise.all([
    getTempPromise('BsAs'),
    getTempPromise('Rawson'),
    getTempPromise('Salta')])
.spread(function(tempBsAs, tempRawson, tempSalta) {
    console.log('Temperatura en Buenos Aires: ' + tempBsAs)
    console.log('Temperatura en Rawson: ' + tempRawson)
    console.log('Temperatura en Salta: ' + tempSalta)
})
.catch(function(theError) {
    ... manejo del error ...
})
```