Programação na Internet

Turma i52d

Lesson 35
Simplifying Code Flows with Promises

Promise

https://en.wikipedia.org/wiki/Futures_and_promises

Container of an asynchronous result:

⇒ May hold a successful or failure result.

Asynchronous alternative idioms:

1. _callback_ `(err, data) => {...}` -- `err` and `data` are 2 possible results
2. `EventEmitters` `.on('error’, callback)` e `.on('data’, callback)`.
3. **`Promise` ⇔ Java CompletableFuture, .Net Task.**
4. `async` / `await` -- in most environments e.g. .Net, Python, Js, etc except Java
5. suspend functions
6. etc

Promise

3 possible states:

- Pending
- Fulfilled (success)
- Rejected (error)

```
….then(…) – receives a continuation
….then(
  val => …, // executed when it is fulfilled
  err => …) // executed when it is rejected
….catch(
  err => …) // executed when it is rejected
// both return new Promises that allow chaining through then/catch
```

Promise

```
• new Promise((res, rej) => …)      // state Pending
• Promise.resolve()                 // state Fulfilled
• Promise.resolve().then(… => res) // state Fulfilled with res

• ….then(val => …, err => …)  // returns a new Promise
```

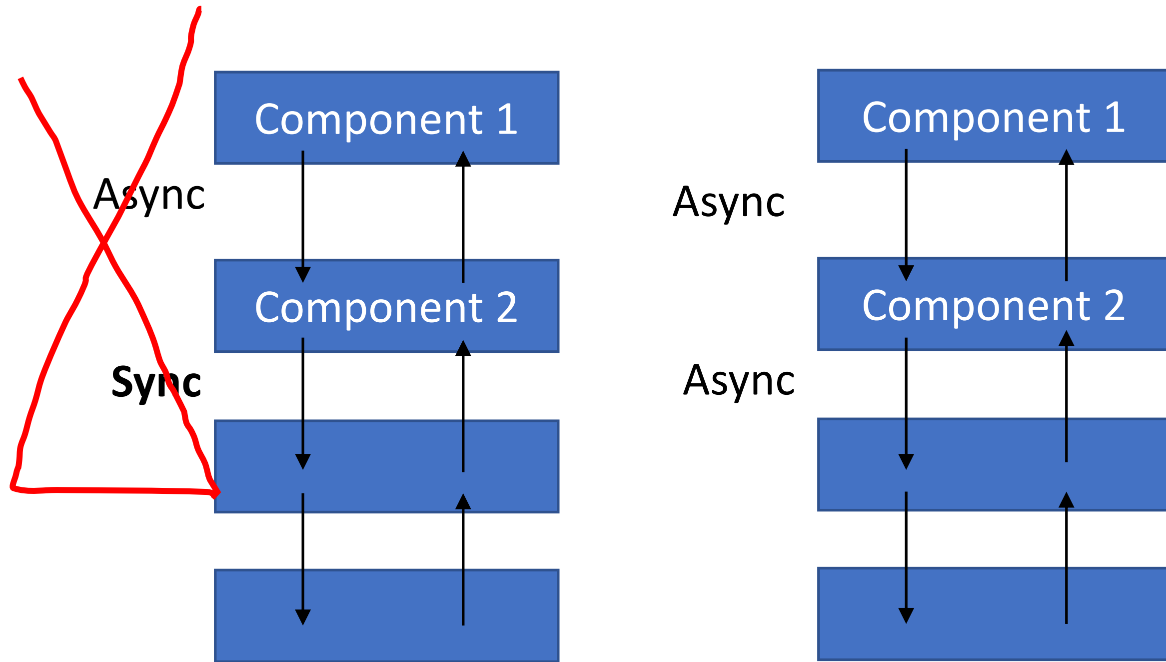
- The callback (or continuation) will be performed when the previous Promise is completed(*fulfilled* or *rejected*)

- The result of the new Promise will be the result of the continuation.

Chaining Promises

```
/**
 * Retrieves the top tracks (limit) of the favourite artists
 * for the given username.
 * Notice it returns a single Array flatten with those tracks.
 *
 * @param {String} username
 * @returns {Promise<Array<Track>>}
 */
function getTopTracks(username, limit) {
  return users
    .getUser(username)                                                    // Promise<User>
    .then(user => user.artists)                                           // Promise<Array<Artist>
    .then(artists => artists.map(artist => lastfm.getTopTracks(artist))) // Promise<Array<Promise<Array<String>>>>
    .then(arr => Promise.all(arr))                                        // Promise<Array<Array<String>>>
    .then(tracks => tracks.map(arr => arr.slice(0, limit)))               // Promise<Array<Array<String>>>
    .then(tracks => tracks.flat())                                        // Promise<Array<String>>>
}
```

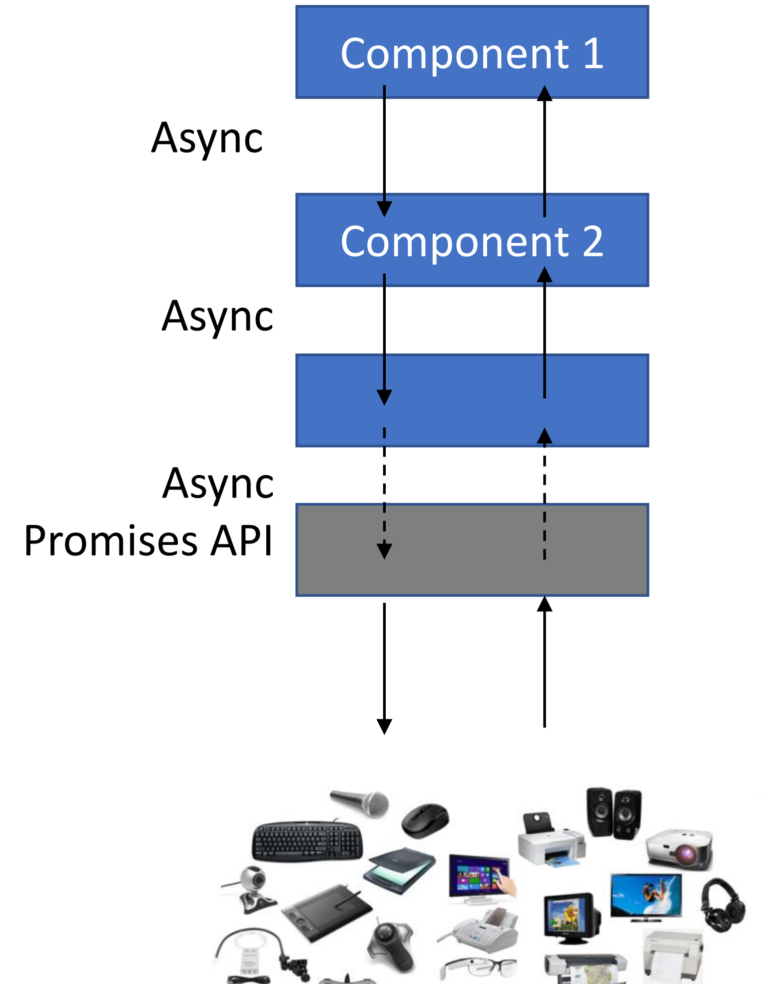
Remember



You do NOT need !!!!

1. Pending Promises

```
return new Promise((resolve, reject) => {
    ...
}) // Pending
```



You do NOT need !!!!

1. Pending Promises
2. `express` auxiliary modules for Promises:
   - `express-promise-router`
   - `express-async-handler`
   - **`etc`**

You do NOT need !!!!

1. Pending Promises
2. `express` auxiliary modules for Promises
3. Auxiliary promises library e.g. `bluebird`
4. Auxiliary fetch mock libraries e.g. `fetch-mock`, `fetch-mock-jest`, etc

You do NOT need !!!!

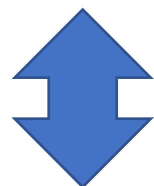
1. Pending Promises
2. `express` auxiliary modules for Promises
3. Auxiliary promises library e.g. `bluebird`
4. Auxiliary fetch mock libraries e.g. `fetch-mock`, `fetch-mock-jest`, etc

DON'T STRESS with ASYNC / AWAIT

- You will forget try/catch and you will get troubles!!!!
- You will run independent tasks sequentially and loose scalability !!!!!

Async/await only syntactic sugar

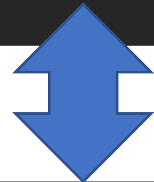
```
const promiseOfstatus = getStatusCode(url)
promiseOfstatus.then(status => ... continuation...) // TELL
```



```
const status = await getStatusCode(url) // ASK
```

Async/await only syntactic sugar

```
const promiseOfstatus = getStatusCode(url)
promiseOfstatus
     .then(status => ... continuation...) // TELL
     .catch(err => next(err))
```



```
try {
     const status = await getStatusCode(url) // ASK
} catch(err) {
     next(err)
}
```