Sincronizando Funções no NodeJS

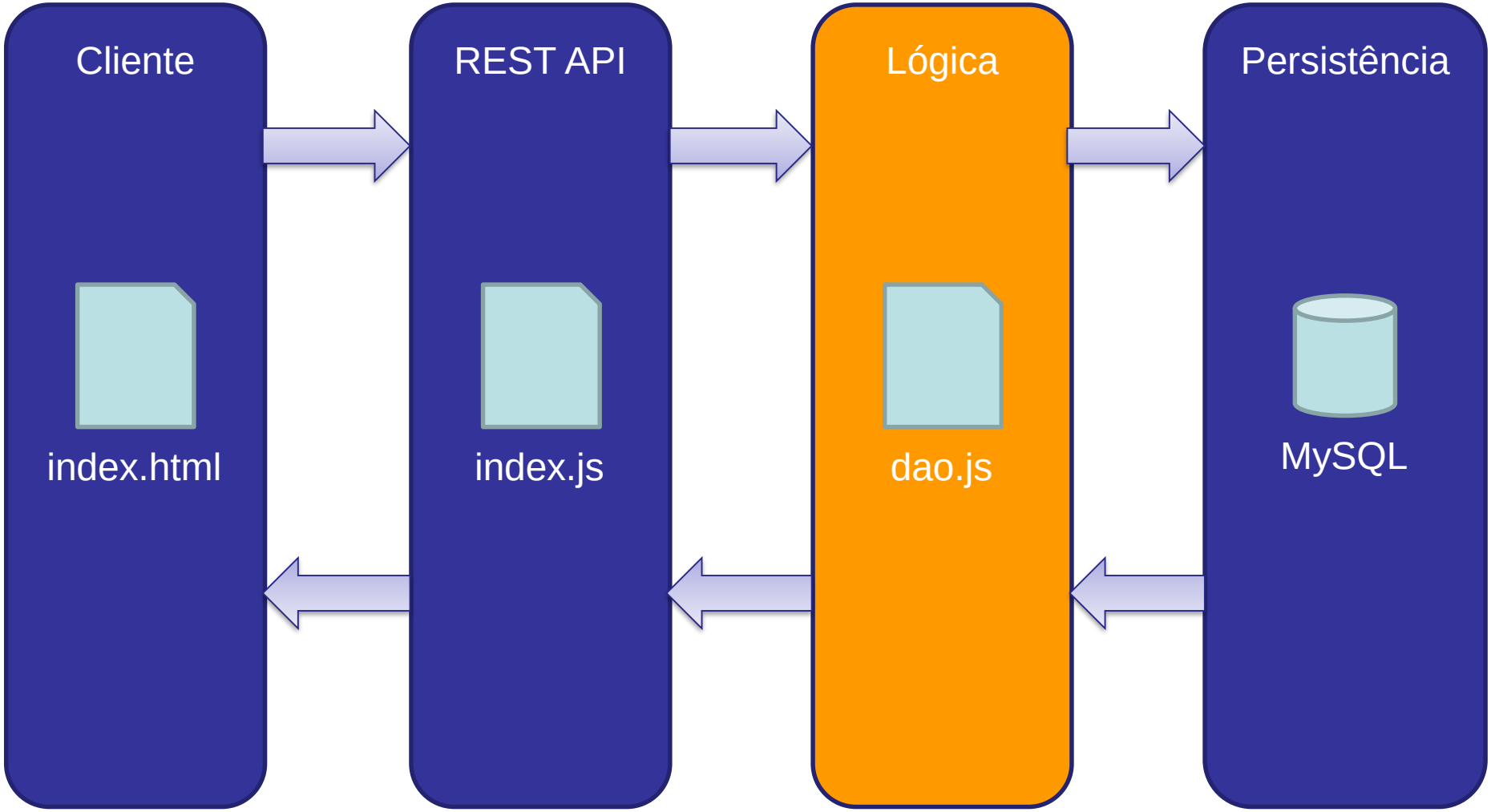
EC021 - Tópicos Avançados II

Sistemas Distribuídos

Prof. Adauto Mendes | E-mail: adauto.mendes@inatel.br

Introdução

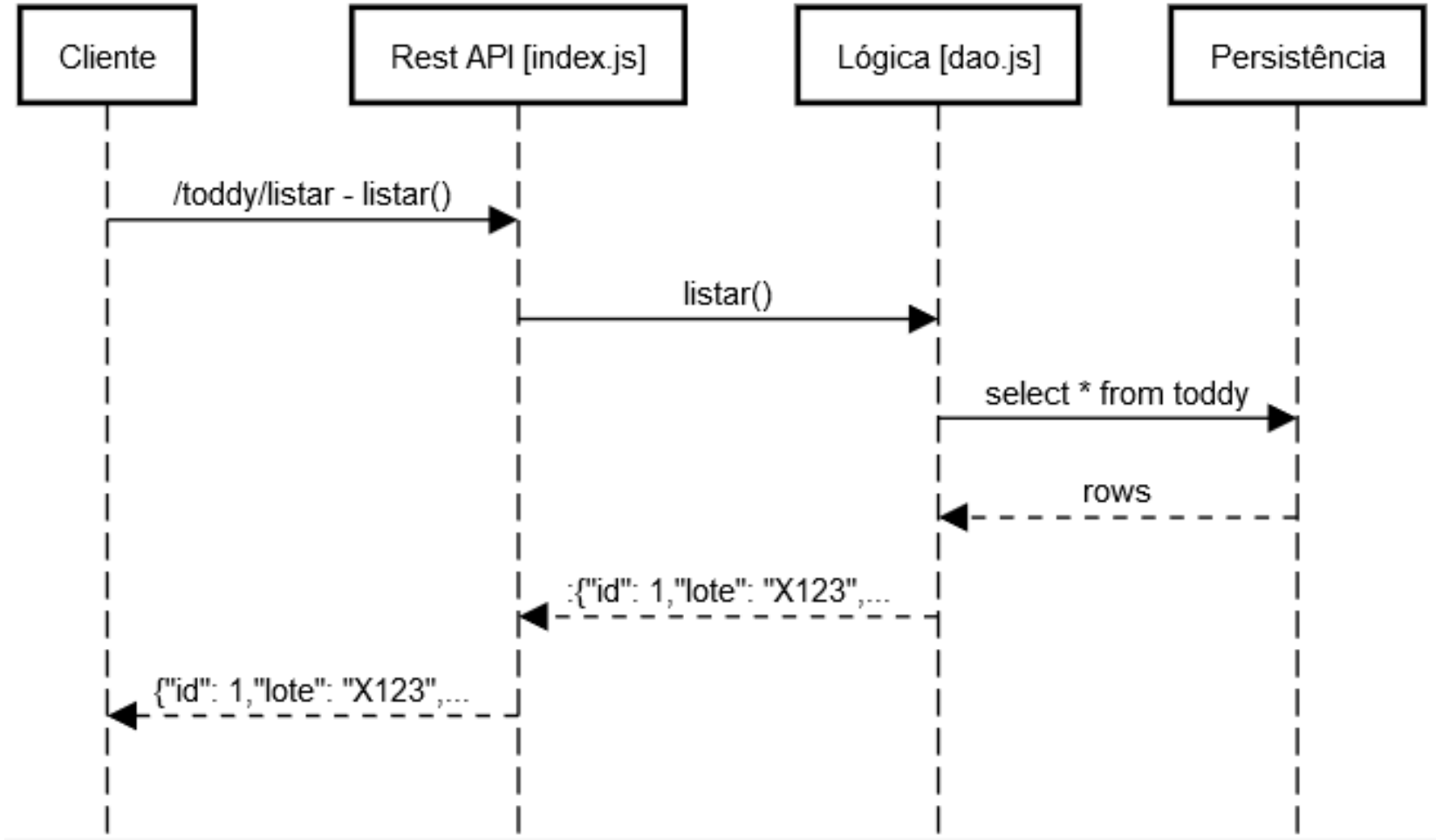
- Na prática anterior separamos a camada de lógica do sistema da camada de acesso ao servidor (Rest API).



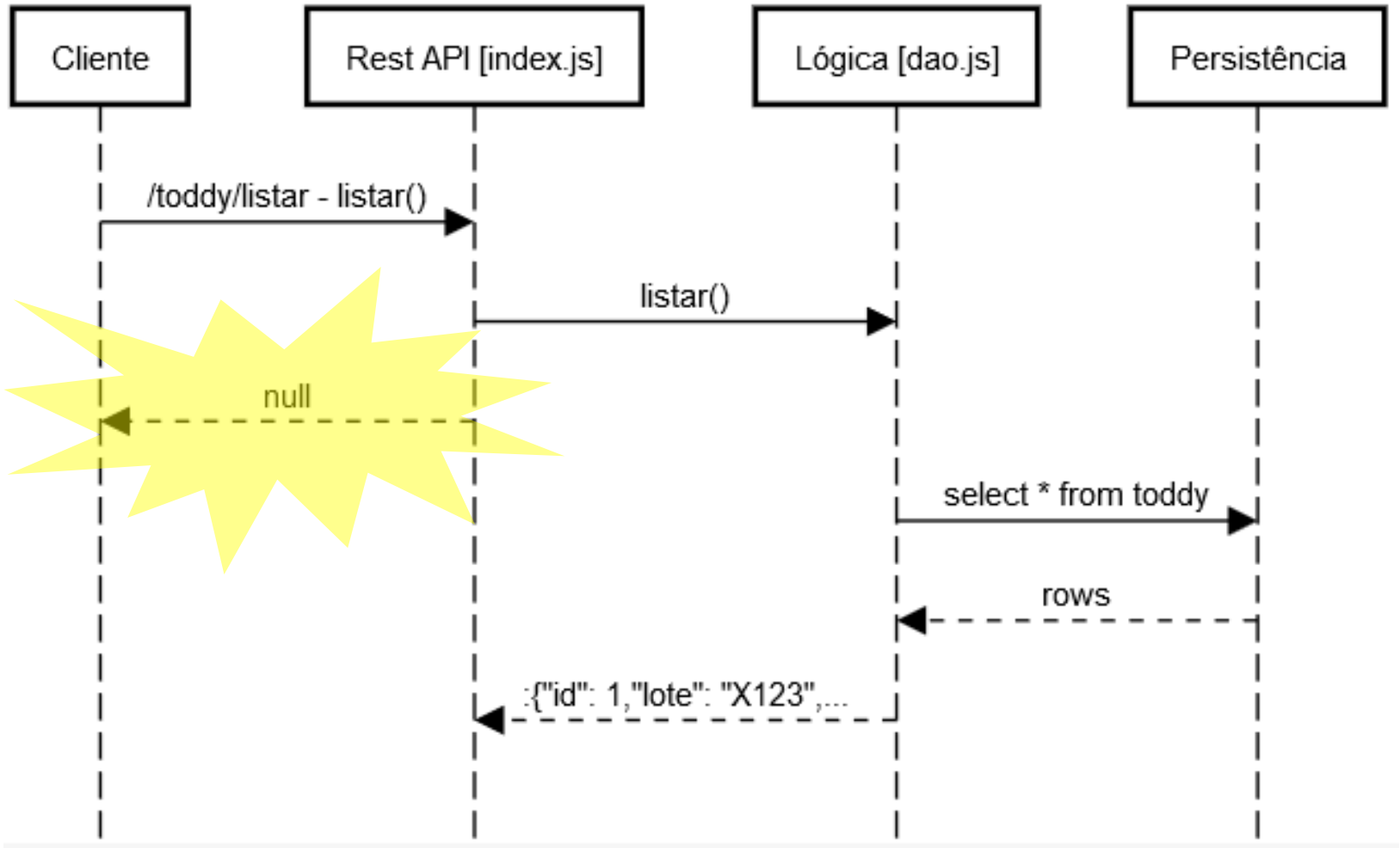
Introdução

- O problema é que ao fazer isso e dado que o NodeJS funciona no modelo de I/O não bloqueante (ou seja, assíncrono) todas as chamadas que fazemos ao middleware está retornando NULL.
- Isto ocorre pois as funções não interrompem a sua execução para aguardar o retorno de outra função que foi chamada.

Resultado esperado



Resultado obtido



Sincronizando chamadas

- Para resolver os problemas de sincronismo entre as chamadas de funções no NodeJS podemos utilizar de algumas funcionalidades já contidas no seu ambiente do NodeJS:
  - Callbacks
  - Promises
  - Async/Await => para que este mecanismo funcione com o Restify é necessário instalar a dependência ‘restify-async-wrap’
  - Módulos externos: sync, sync-request...

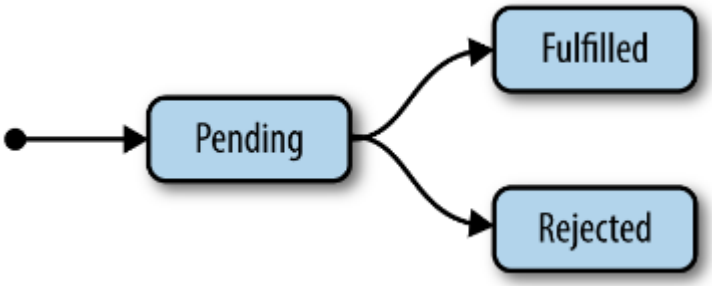
Utilizando Promises

- O maior desafio em trabalhar dentro do modelo assíncrono do JavaScript é gerenciar a ordem de execução através de uma série de etapas e lidar com quaisquer erros que possam surgir. As **Promises** abordam esse problema fornecendo uma maneira de organizar os retornos de chamada em etapas discretas que são mais fáceis de ler e manter. E quando ocorrem erros, eles podem ser manipulados fora da lógica de aplicativo principal sem a necessidade de verificações padrão em cada etapa (funcionamento análogo ao bloco try/catch).

Utilizando Promises

- O estado de uma operação representada por uma Promise é armazenado dentro da própria Promise.
- A qualquer momento, uma operação pode não ter sido iniciada, estar em andamento, concluída ou interrompida e não pode ser concluída.
- Essas condições são representadas por três estados mutuamente exclusivos:
  - Pending: A operação não começou ou está em andamento.
  - Fulfilled: A operação foi concluída.
  - Rejected: A operação não pode ser completada.

Utilizando Promises



- Habitualmente nos referimos aos estados Fulfilled e Rejected como sucesso e erro, respectivamente. Existe uma diferença entre esses termos. Uma operação pode concluir com um erro (embora isso possa ser uma forma incorreta) e uma operação pode não ser concluída, porque foi cancelada mesmo que nenhum erro tenha ocorrido. Portanto, os termos Fulfilled e Rejected são melhores descrições para esses estados do que sucesso e erro.

Na Prática

- Na prática um objeto Promise deve ser encapsulado dentro de uma função que deve ser retornado. O formato de declaração de uma Promise segue a sintaxe abaixo:

```
funcaoTeste: function () {
    return new Promise(
        function (resolve, reject) {
            if (!erro) {
                resolve("retorne os resultados");
            } else {
                reject("retorne os erros");
            }
        }
    )
}
```

Na Prática

- A função que irá chamar a função que contém a Promise deve tratar seu retorno usando os métodos ‘then’ (para casos onde a Promise é fulfilled) e ‘catch’ (para casos em que a Promise é rejected):

```
function funcaoTeste(req, res, next) {
    dao.funcaoTeste()
        .then(function (result) {
            //Código que será executado
            //quando a Promise for Fulfilled
            console.log('result');
        })
        .catch(function (err) {
            //Código que será executado
            //quando a Promise for Rejected
            console.log('err');
        });
}
```

Na Prática

- Nota: a partir da versão 10 do NodeJS a instrução ‘finally’ passou a ser suportada:

```
function funcaoTeste(req, res, next) {
    dao.funcaoTeste()
        .then(function (result) {
            //Código que será executado
            //quando a Promise for Fulfilled
            console.log('result');
        })
        .catch(function (err) {
            //Código que será executado
            //quando a Promise for Rejected
            console.log('err');
        })
        .finally(() => {
            /** Encerrando método da REST API */
            next();
        });
}
```

Na Prática

- Também é possível escrever estas funções usando a sintaxe de “arrow functions”:

```
funcaoTeste: () => {
      return new Promise(
          (resolve, reject) => {
              if (!erro) {
                  resolve("retorne os resultados");
              } else {
                  reject("retorne os erros");
              }
          }
      )
}
```

Na Prática

```
var funcaoTeste = (req, res, next) => {
    dao.funcaoTeste()
        .then((result) => {
            //Código que será executado
            //quando a Promise for Fulfilled
        })
        .catch((err) => {
            //Código que será executado
            //quando a Promise for Rejected
        });
}
```

Pratica - Listar

- O método Listar da nossa API será refatorado como a seguir:

```
function listar(req, res, next) {
   dao.listar()
        .then((result) => {
            res.json(result);
        })
        .catch((err) => {
            res.json(err);
        });

    /** Encerrando método da REST API */
    next();
}
```

Pratica - Listar

- Dentro do arquivo dao.js altere o método 'listar' para que retorne uma Promise:

```
listar: function () {
        return new Promise(
            function (resolve, reject) {
                ...
            }
        )
    }
```

Prática - Teste

- Teste a chama aos endpoint /toddy/listar e veja se a consulta retorna resultados:

GET http://localhost:5000/toddy/listar

Pretty Raw Preview JSON


```
[
    {
        "id": 1,
        "lote": "X123",
        "conteudo": 200,
        "validade": "25/08/2018"
    },
    {
        "id": 2,
        "lote": "X1C",
        "conteudo": 250,
        "validade": "04/08/2019"
    },
    {
        "id": 3,
        "lote": "X123",
        "conteudo": 200,
        "validade": "25/08/2018"
    },
```


Prática

- Refatore os demais métodos para isolar toda lógica no arquivo dao.js.
- Teste-os via Postman.