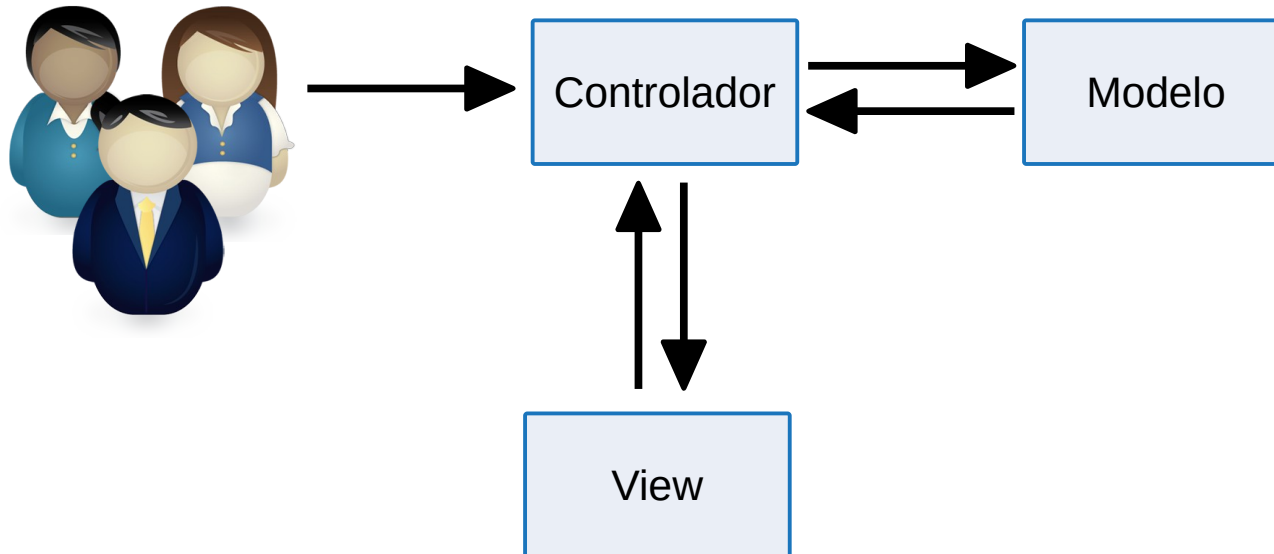




Prof. David Fernandes de Oliveira
Instituto de Computação
UFAM

O MVC

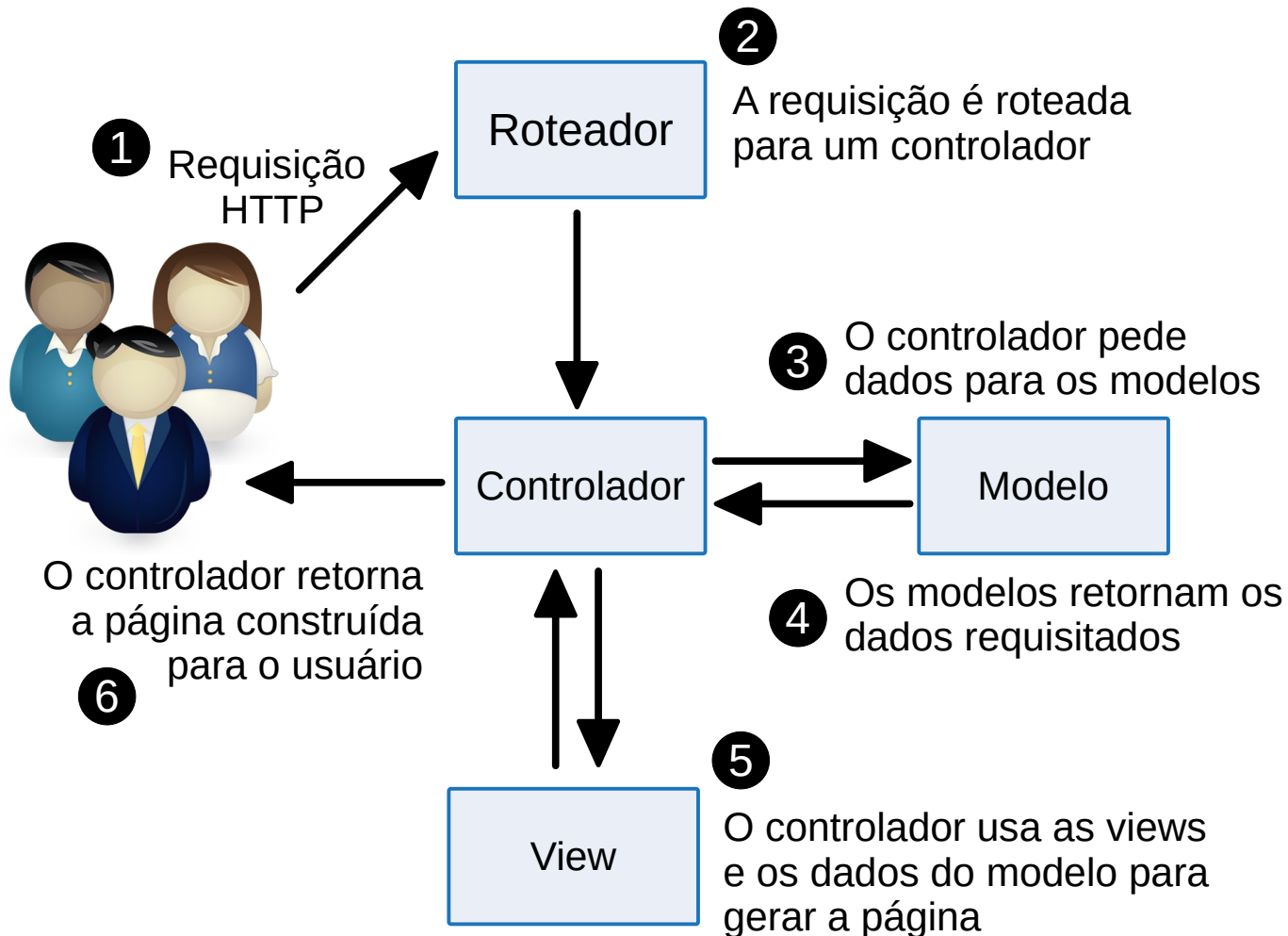
- O MVC é um padrão de arquitetura de software que separa as aplicações em 3 camadas: **Modelos**, **Views** e **Controladores**
 - O objetivo de separar a arquitetura nas três camadas é facilitar a organização, compreensão e a manutenção do código
- Frameworks Web MVC: Yii2, Laravel, Sails, Adonis, Django, etc



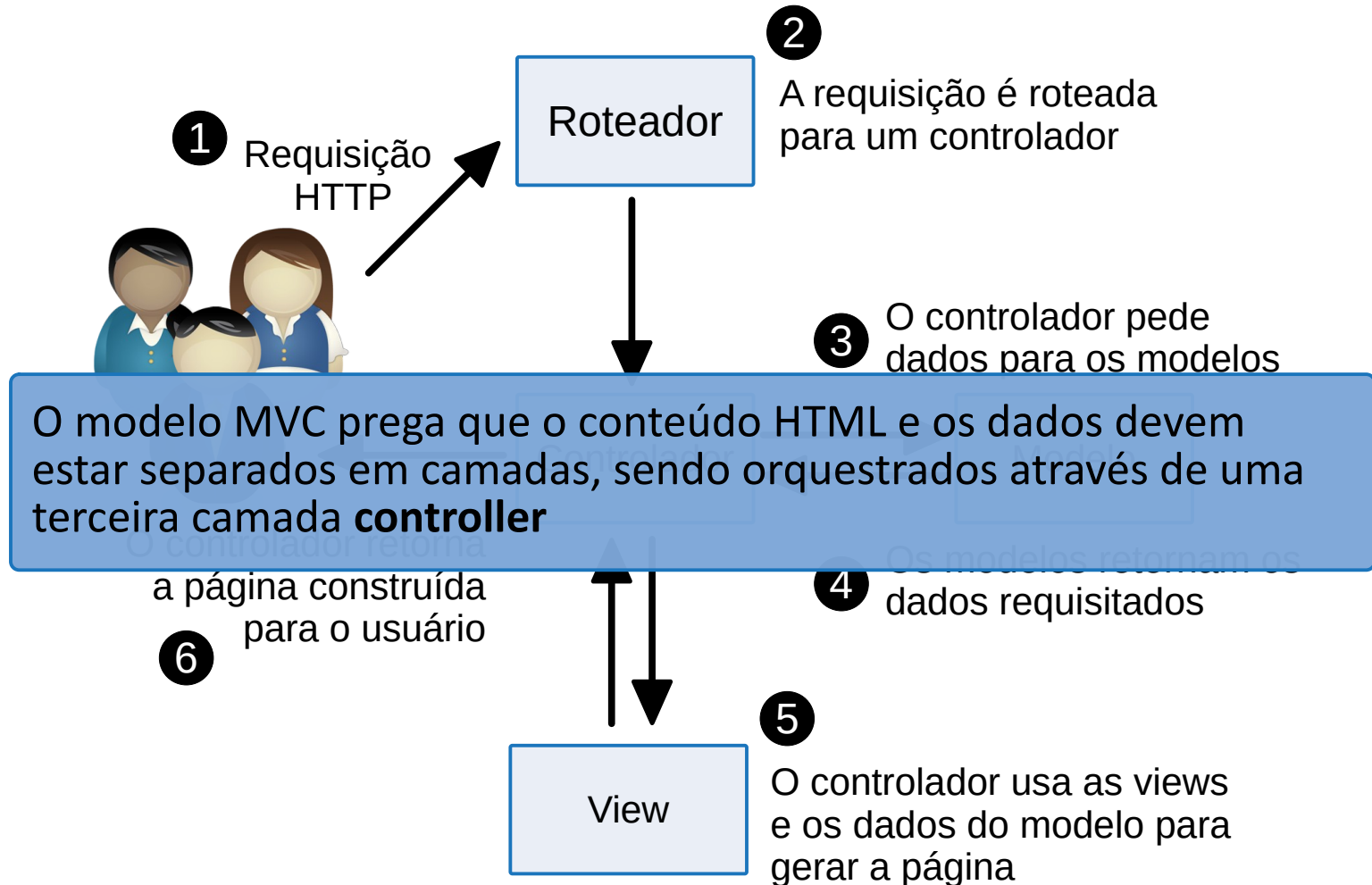
O MVC

- No modelo MVC, as 3 camadas possuem funções específicas e estão conectadas entre si:
 - **Modelo**, responsável pela leitura e escrita dos dados provenientes do SGBD utilizado pela aplicação
 - **Visão**, responsável por gerar o conteúdo HTML que será enviado para o usuário para que esse possa interagir com a aplicação
 - **Controlador**, responsável por responder as requisições dos usuários, fazendo uso dos modelos e apresentando os resultados através das views

O MVC

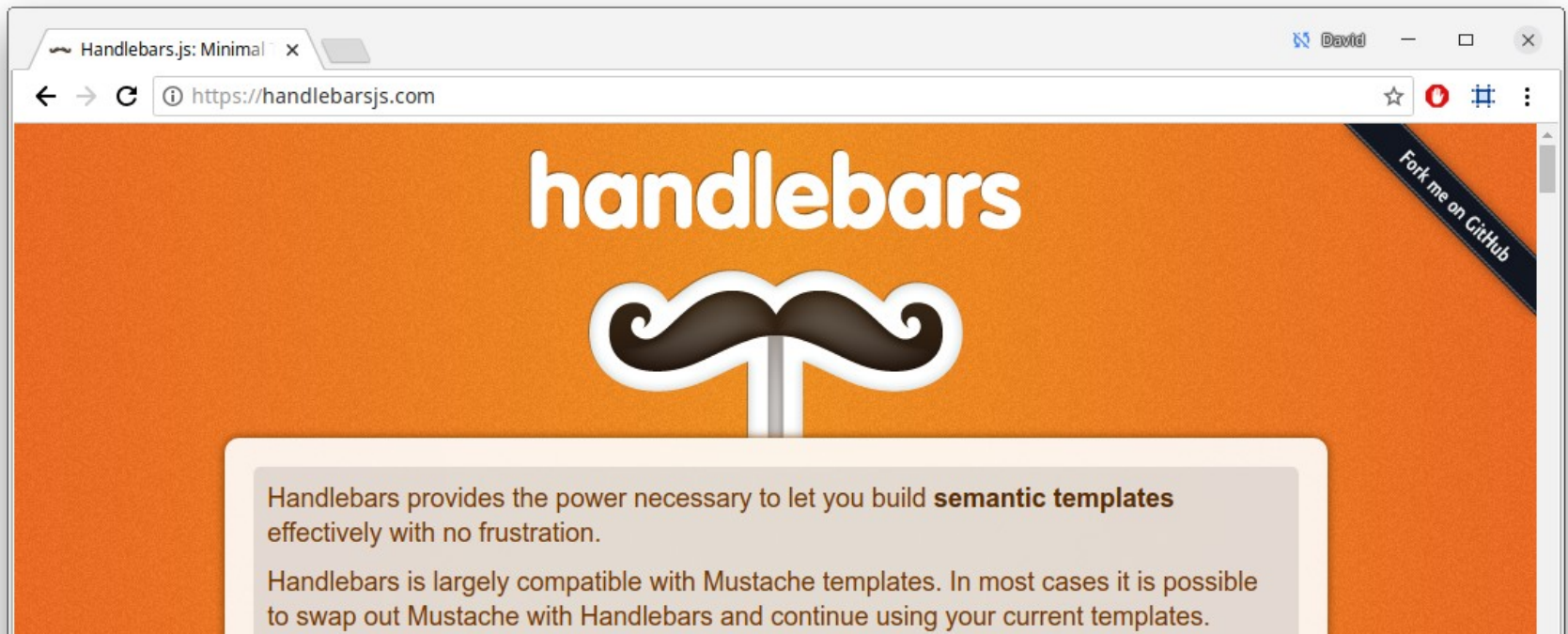


O MVC



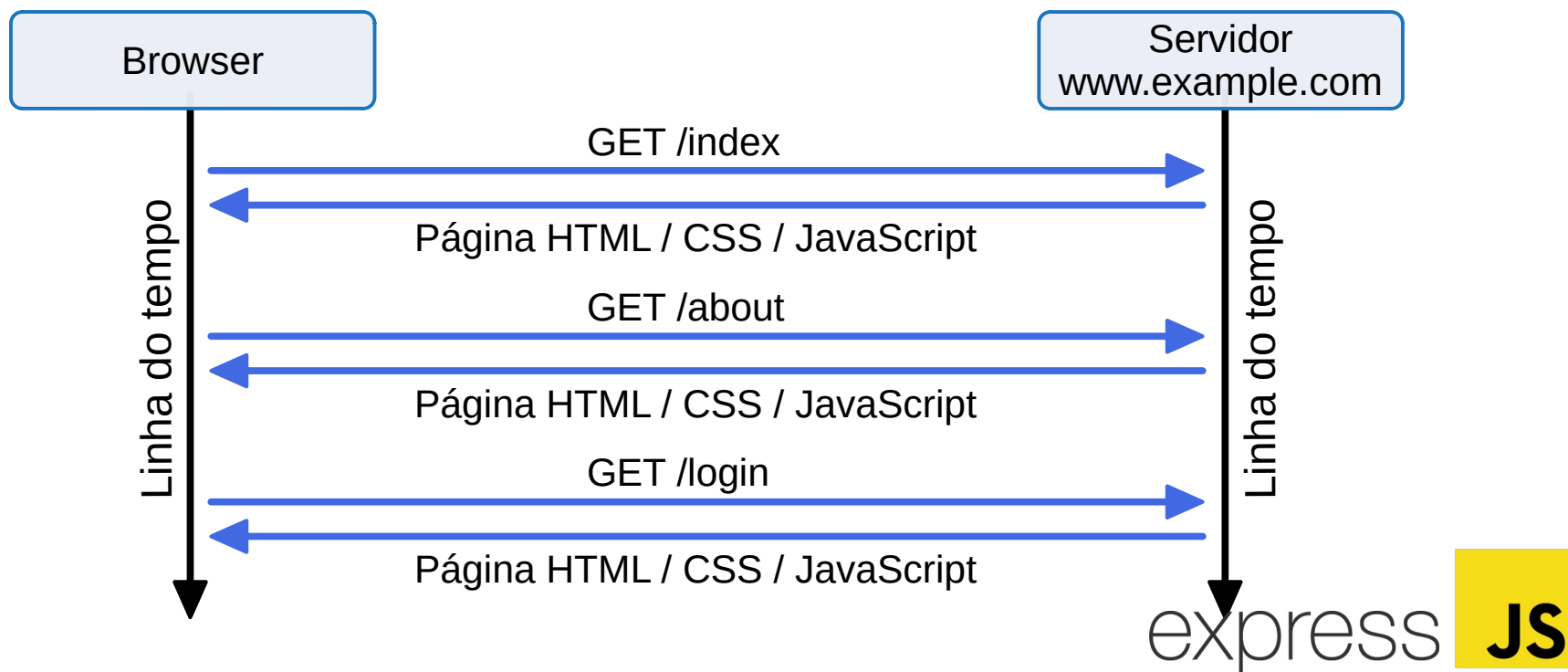
Views

- As **views** são responsáveis por gerar automaticamente o código HTML que é enviado pelo usuário a cada requisição
 - Fazem parte do modelo **MVC** – **Model, View, Controller**
- Existem muitas engines de views disponíveis para o Express, dentre as quais destaca-se: EJS, Handlebars, Pug e Mustache



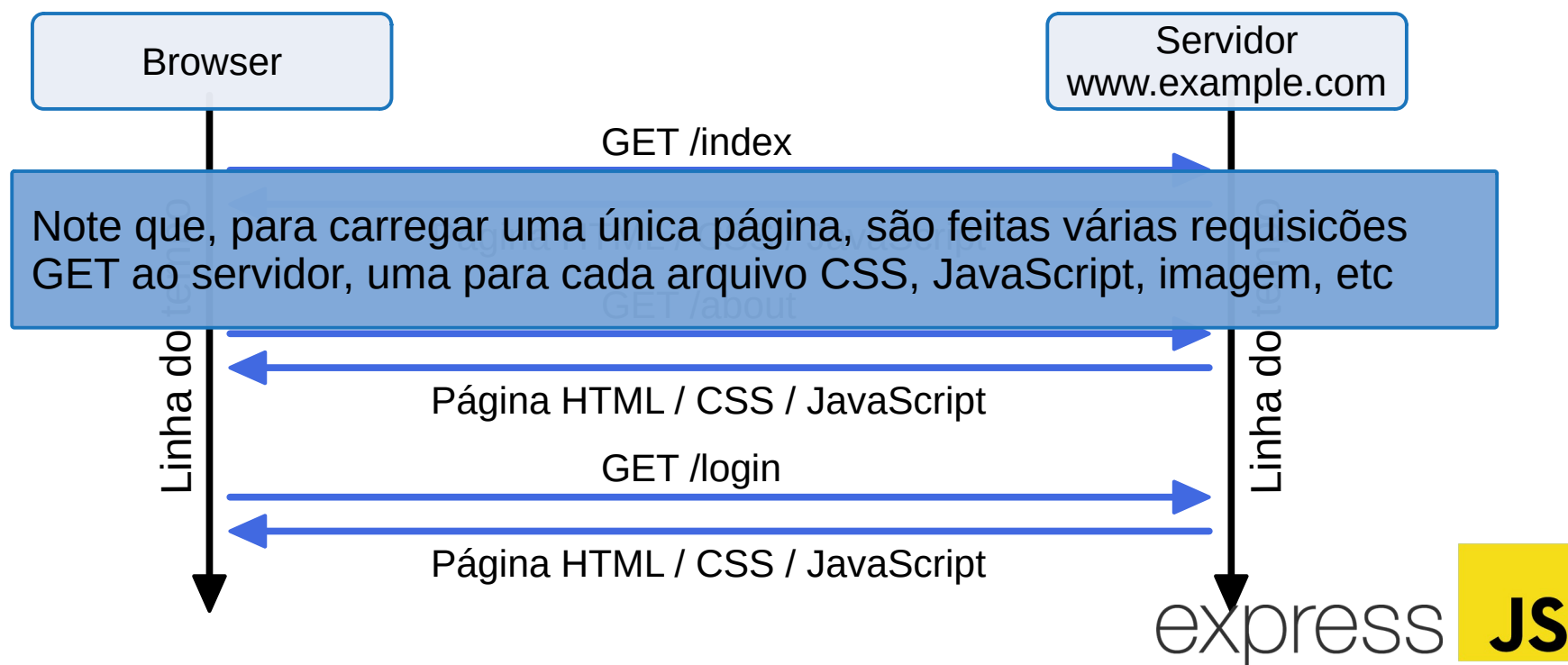
Sistemas MVC

- Nos **sistemas MVC**, o servidor é responsável por executar a maior parte da lógica da aplicação
- A cada requisição ao sistema, o servidor precisa retornar o todo o conteúdo HTML, CSS e JavaScript do recurso desejado



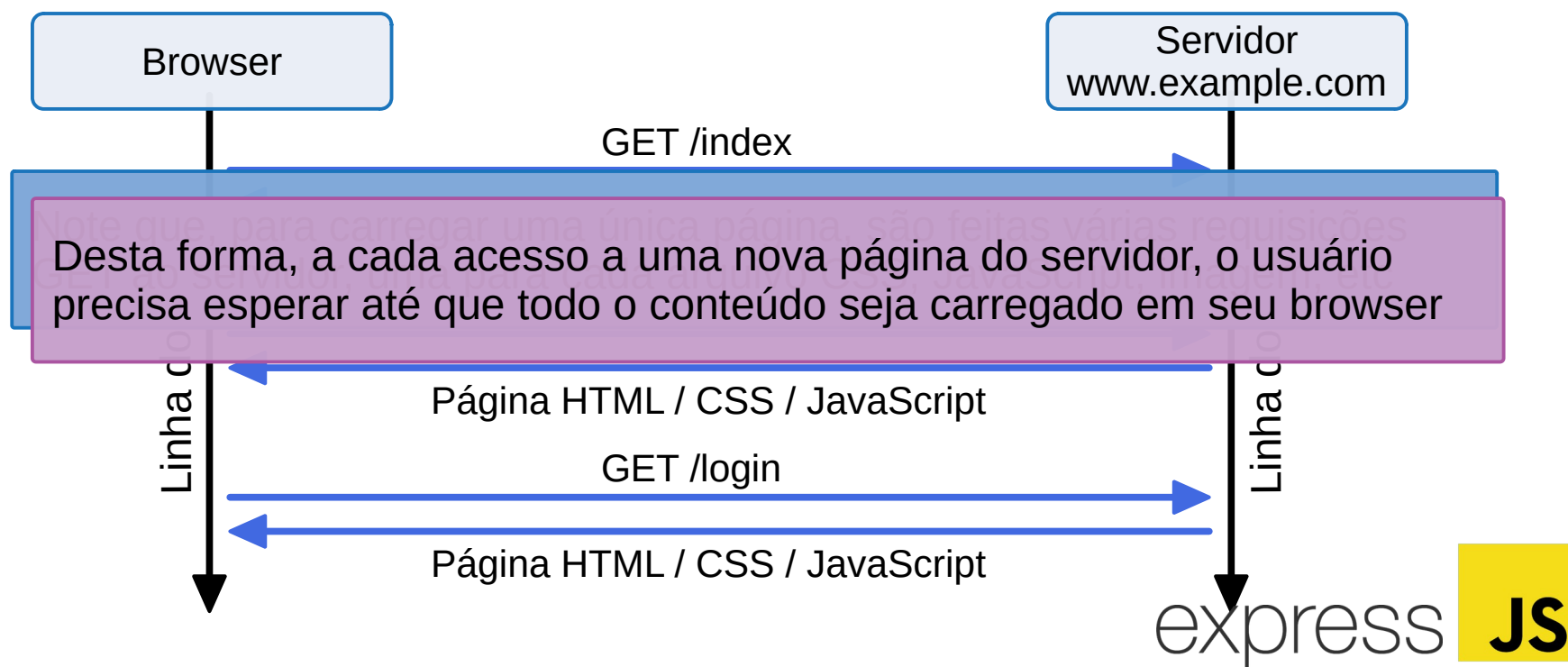
Sistemas MVC

- Nos **sistemas MVC**, o servidor é responsável por executar a maior parte da lógica da aplicação
- A cada requisição ao sistema, o servidor precisa retornar o todo o conteúdo HTML, CSS e JavaScript do recurso desejado



Sistemas MVC

- Nos **sistemas MVC**, o servidor é responsável por executar a maior parte da lógica da aplicação
- A cada requisição ao sistema, o servidor precisa retornar o todo o conteúdo HTML, CSS e JavaScript do recurso desejado



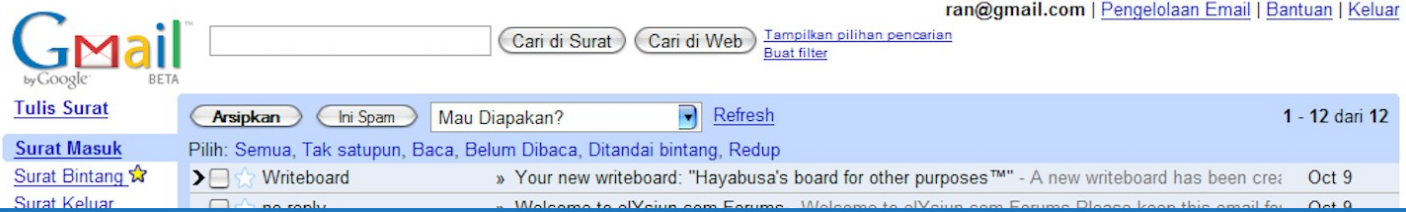
Uma revolução chamada AJAX

- Em 2005, **Jesse Garrett** introduziu o conceito de AJAX, tornando possível a criação de páginas muito mais dinâmicas
- Um dos primeiros grandes sistemas a usar essa nova tecnologia de forma realmente produtiva foi o **Gmail**



Uma revolução chamada AJAX

- Em 2005, **Jesse Garrett** introduziu o conceito de AJAX, tornando possível a criação de páginas muito mais dinâmicas
- Um dos primeiros grandes sistemas a usar essa nova tecnologia de forma realmente produtiva foi o **Gmail**



Utilizando requisições assíncronas e em background, o Gmail conseguiu trazer uma excelente experiência realmente nova para os usuários



Kini Anda dapat memakai Gmail dalam lebih banyak bahasa! [Ingin tahu?](#)

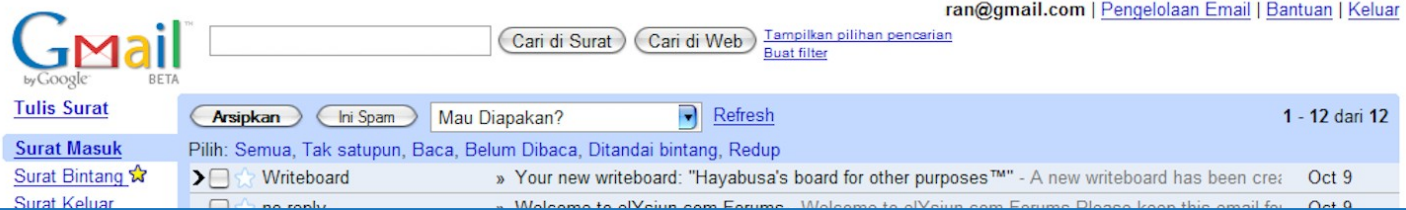
Anda sekarang memakai 0 MB (0%) dari kuota Anda sebesar 2654 MB.

[Syarat-syarat Penggunaan](#) - [Kebijakan Privasi](#) - [Kebijakan Program](#) - [Beranda Google](#)

©2005 Google

Uma revolução chamada AJAX

- Em 2005, **Jesse Garrett** introduziu o conceito de AJAX, tornando possível a criação de páginas muito mais dinâmicas
- Um dos primeiros grandes sistemas a usar essa nova tecnologia de forma realmente produtiva foi o **Gmail**



Foi uma das primeiras vezes que se conseguiu trazer a sensação de uso de uma aplicação desktop para dentro de um sistema Web - e esse é um dos pilares das single-page applications



Kini Anda dapat memakai Gmail dalam lebih banyak bahasa! [Ingin tahu?](#)

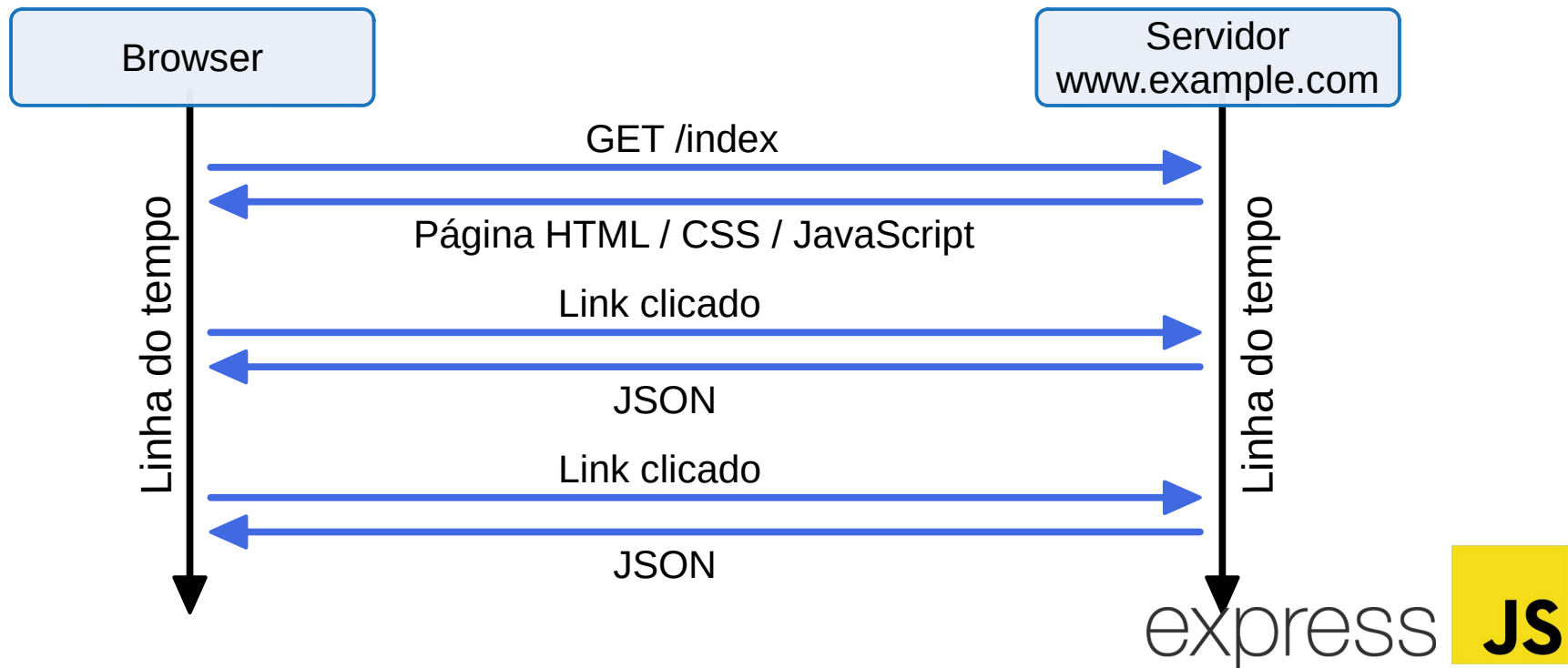
Anda sekarang memakai 0 MB (0%) dari kuota Anda sebesar 2654 MB.

[Syarat-syarat Penggunaan](#) - [Kebijakan Privasi](#) - [Kebijakan Program](#) - [Beranda Google](#)

©2005 Google

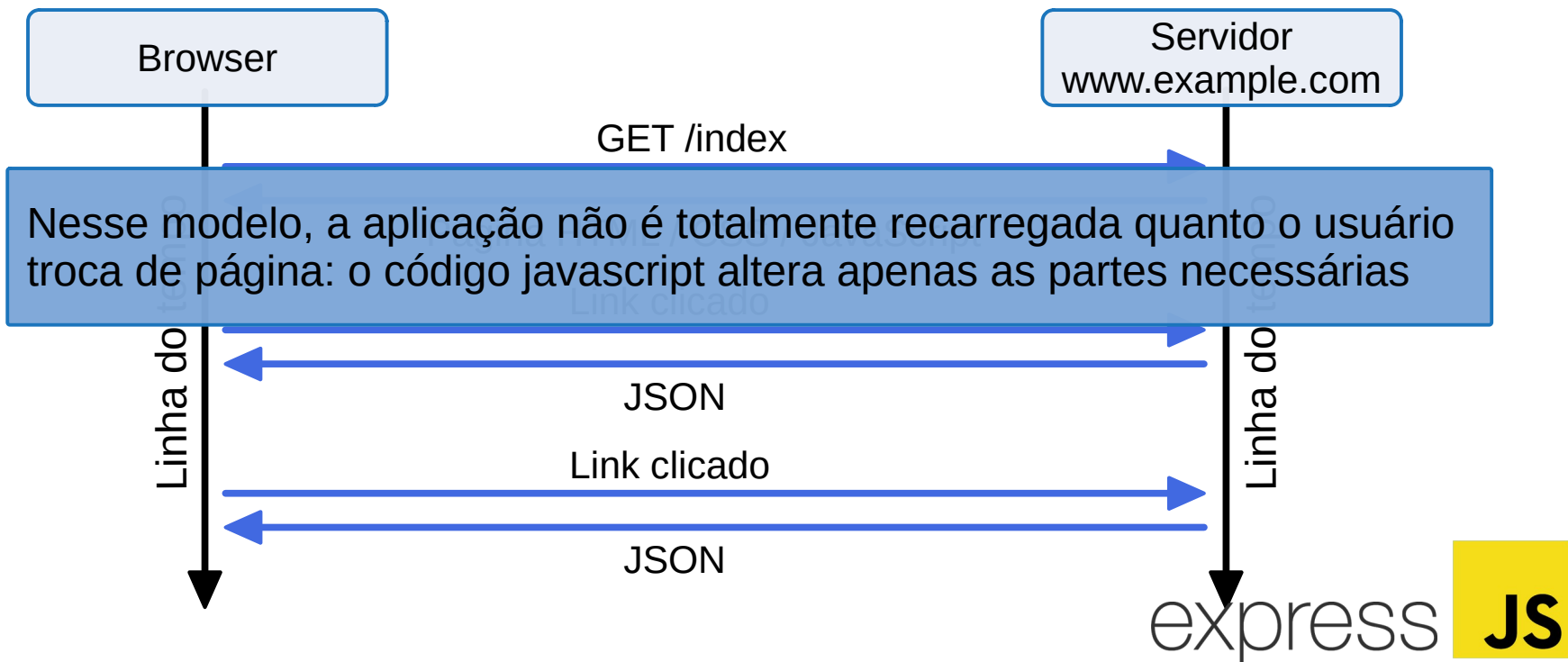
Single-Page Applications

- Um SPA é uma aplicação web que roda em uma única página, de uma forma similar à uma aplicação desktop ou mobile
- Executam a maior parte da lógica da aplicação no browser, comunicando-se com o servidor através de APIs



Single-Page Applications

- Um SPA é uma aplicação web que roda em uma única página, de uma forma similar à uma aplicação desktop ou mobile
- Executam a maior parte da lógica da aplicação no browser, comunicando-se com o servidor através de APIs



Vantagens e Desvantagens

- Vantagens das **Single-Page Applications**

- Páginas mais reativas, interação com o usuário mais fluida
- Alto desacoplamento entre backend e frontend
- Vários frameworks para o frontend

- Desvantagens

- Requer uma política de **Search Engine Optimization** diferenciada
- Carregamento inicial com muito mais código
- Requer conhecimentos sólidos de programação Javascript
- Perigo de descontinuidade das bibliotecas usadas, ou geração de novas versões incompatíveis com as anteriores

Vantagens e Desvantagens

- Vantagens dos **Sistemas Web Tradicionais**

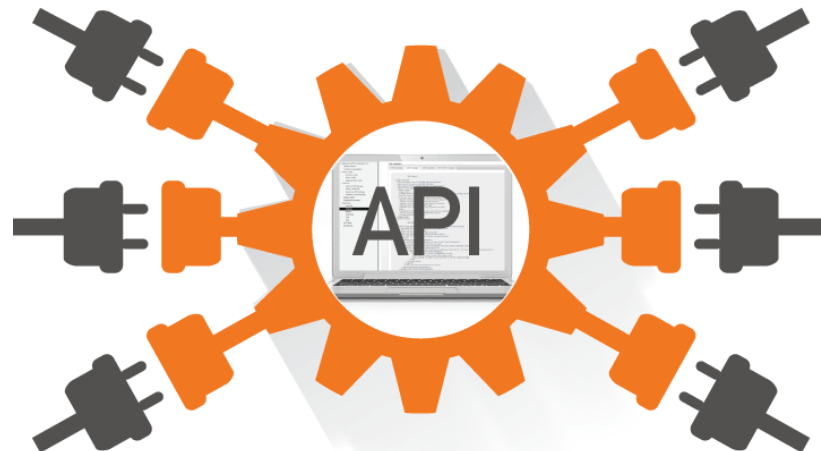
- Técnicas mais consolidadas
- **Search Engine Optimization** mais simples
- Mais fácil de implementar
- Menor acoplamento com código javascript no lado cliente

- Desvantagens:

- Experiência de usuário inferior, pois todo o conteúdo da página é recarregada a cada nova requisição
- Forte acoplamento dentre frontend e backend
- Arquitetura defasada

REST APIs

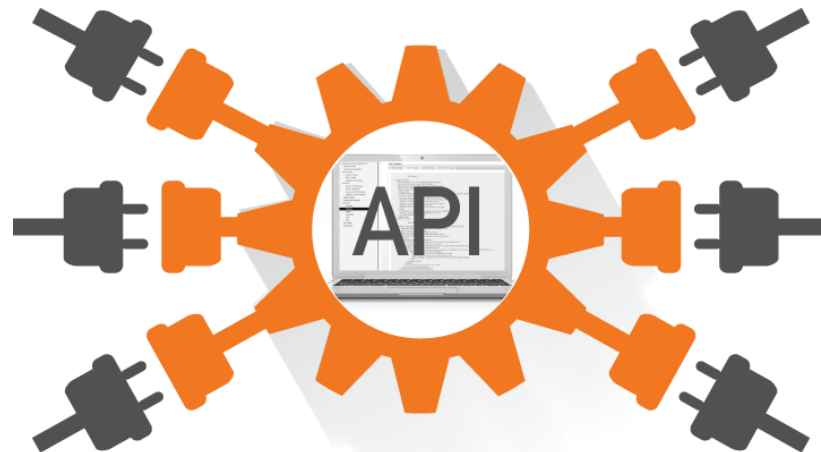
- O REST – Representational State Transfer – é caracterizado como um paradigma de desenvolvimento de software semelhante aos webservice
 - Nesse paradigma, um serviço (normalmente chamado de API) é fornecido para acesso e manipulação dos dados de uma aplicação
- API – Application Programming Interface – é um conjunto de rotinas usadas na comunicação entre duas partes de uma aplicação



REST APIs

- O REST – Representational State Transfer – é caracterizado como um paradigma de desenvolvimento de software semelhante aos webservice
 - Nesse paradigma, um serviço (normalmente chamado de API) é fornecido para acesso e manipulação dos dados de uma aplicação
- O Front, que consome os dados da API, pode ser desenvolvida através de vários tipos de frameworks frontend, como o React, Angular e o Vue.js.

aplicação



Elementos de uma Requisição

- O **endpoint** é o caminho usado para fazer uma requisição, possuindo um **resource** e opcionalmente uma **query string**

<http://api.minhaloja.com/produtos/?tipo=livros>
resource ou path query string

- O **método HTTP** define o tipo de ação desejada pela requisição, sendo que os métodos mais usados são:
 - **Get**, usado para buscar dados do servidor
 - **Post**, usado para enviar dados para o servidor
 - **Put** e **Patch**, usado para atualizar dados
 - **Delete**, usado para apagar registros no servidor

Elementos de uma Requisição

- O **body** é o corpo da mensagem enviada na requisição, e é usado apenas com os métodos POST, PUT e PATCH
- Os **HTTP status codes** servem para indicar se uma requisição HTTP foi corretamente concluída
- Os principais códigos utilizados para as respostas de um endpoint são o 200 (OK), o 201 (CREATED), o 204 (NO CONTENT), o 404 (NOT FOUND) e o 400 (BAD REQUEST).



HTTP Cats

http.cat







HTTP Cats

Usage:

Descrição dos vários HTTP Status Code: <https://http.cat/>

`https://http.cat/[status_code]`

Note: If you need an extension at the end of the URL just add `.jpg`.

 100 Continue	 101 Switching Protocols	 102 Processing
		

Elementos de uma Requisição

- Os **parâmetros** permitem passar dados adicionais sobre a consulta desejada
- Para ler o valor de **id** dentro de uma action, podemos usar o atributo **param** de **req** (objeto da requisição do usuário):

```
const read = async (req: Request, res: Response) => {  
  const { id } = req.params;  
  try {  
    const produto = await getProduto(id);  
    if (produto === null) {  
      res.status(404).json({ message: 'Produto não encontrado' });  
    } else {  
      res.status(200).json(produto);  
    }  
  } catch (err) {  
    res.status(500).json(err);  
  }  
};
```

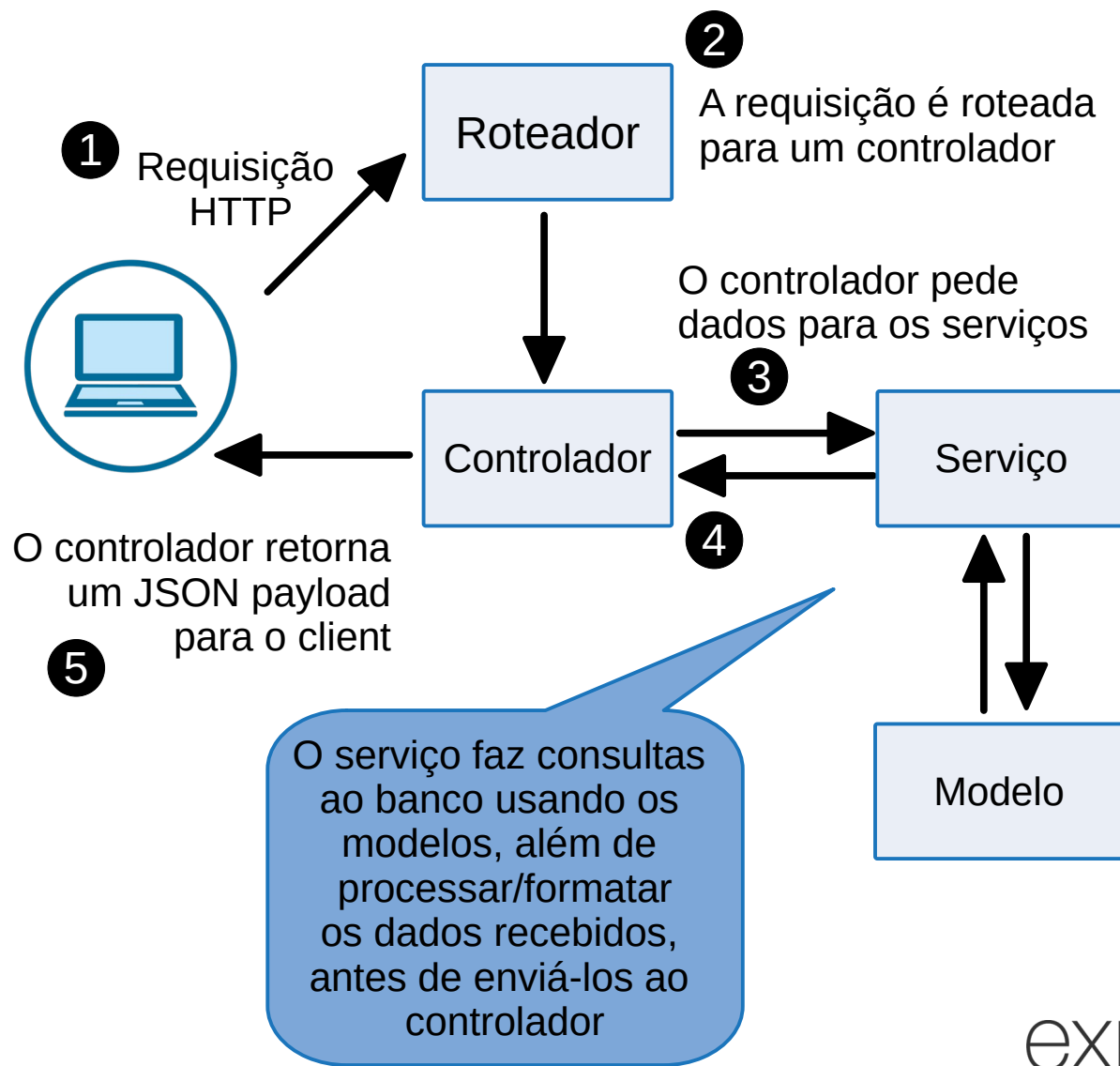
Lendo o
Parâmetro
id

'Found' });

Movendo para o padrão REST

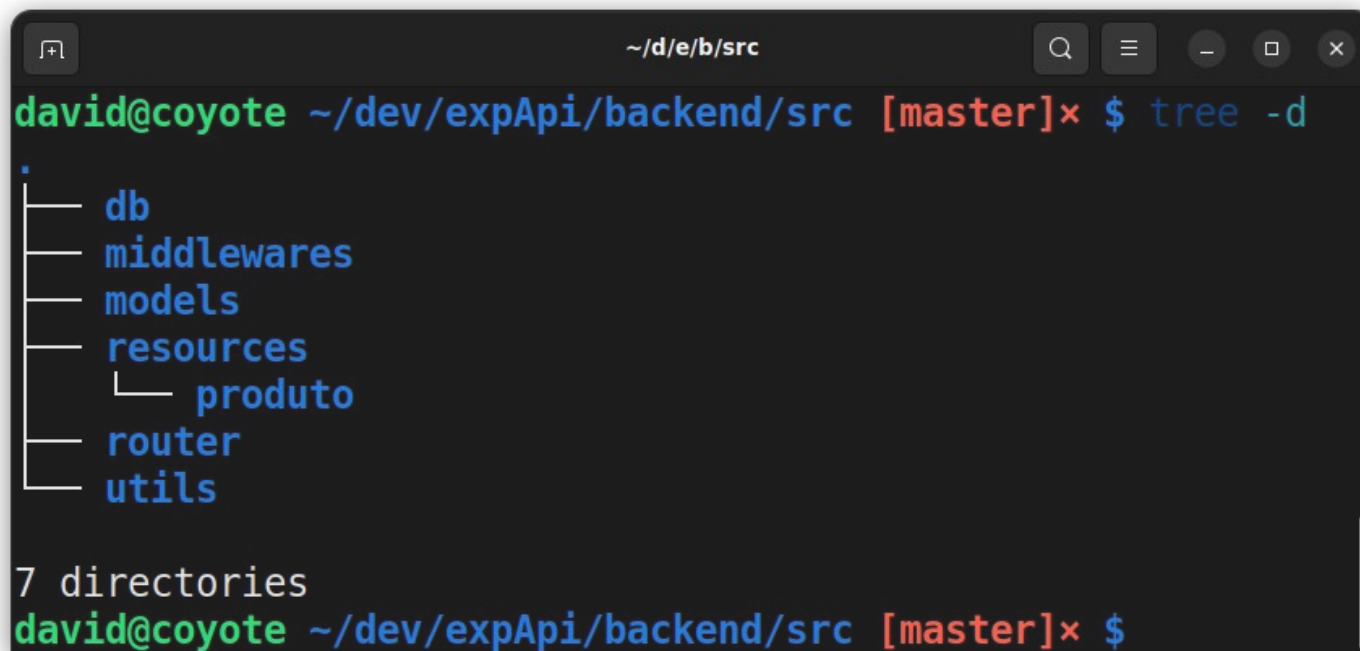
- Nas aplicações MVC, cada página é contruída através de uma action de um dado controlador
 - Por exemplo a **action** about do controlador **main** tem por objetivo construir e retornar o conteúdo da página **/about**
- Nas aplicações REST, por outro lado, as páginas de uma aplicação são definidas no lado Front e não no lado Back
 - O Back nesse caso é responsável por responder a chamadas HTTP do Front, realizando os processos de negócio e de persistência que sejam pertinentes a cada situação

Movendo para o padrão REST



Movendo para o padrão REST

- Para o padrão REST, optamos por organizar os arquivos de nossa aplicação usando o esquema abaixo
- O diretório **src** terá todos os arquivos fontes da aplicação, incluindo os diretórios **resources**, **middlewares** e **models**

A terminal window with a dark background and light blue text. The window title is ~/d/e/b/src. The prompt is david@coyote ~/dev/expApi/backend/src [master]x \$. The command tree -d has been executed, showing a directory tree with root '.', subdirectories db, middlewares, models, resources, router, and utils. The 'resources' directory has a subdirectory 'produto'. The output shows 7 directories.

```
~/d/e/b/src
david@coyote ~/dev/expApi/backend/src [master]x $ tree -d
.
├── db
├── middlewares
├── models
├── resources
│   └── produto
├── router
└── utils

7 directories
david@coyote ~/dev/expApi/backend/src [master]x $
```

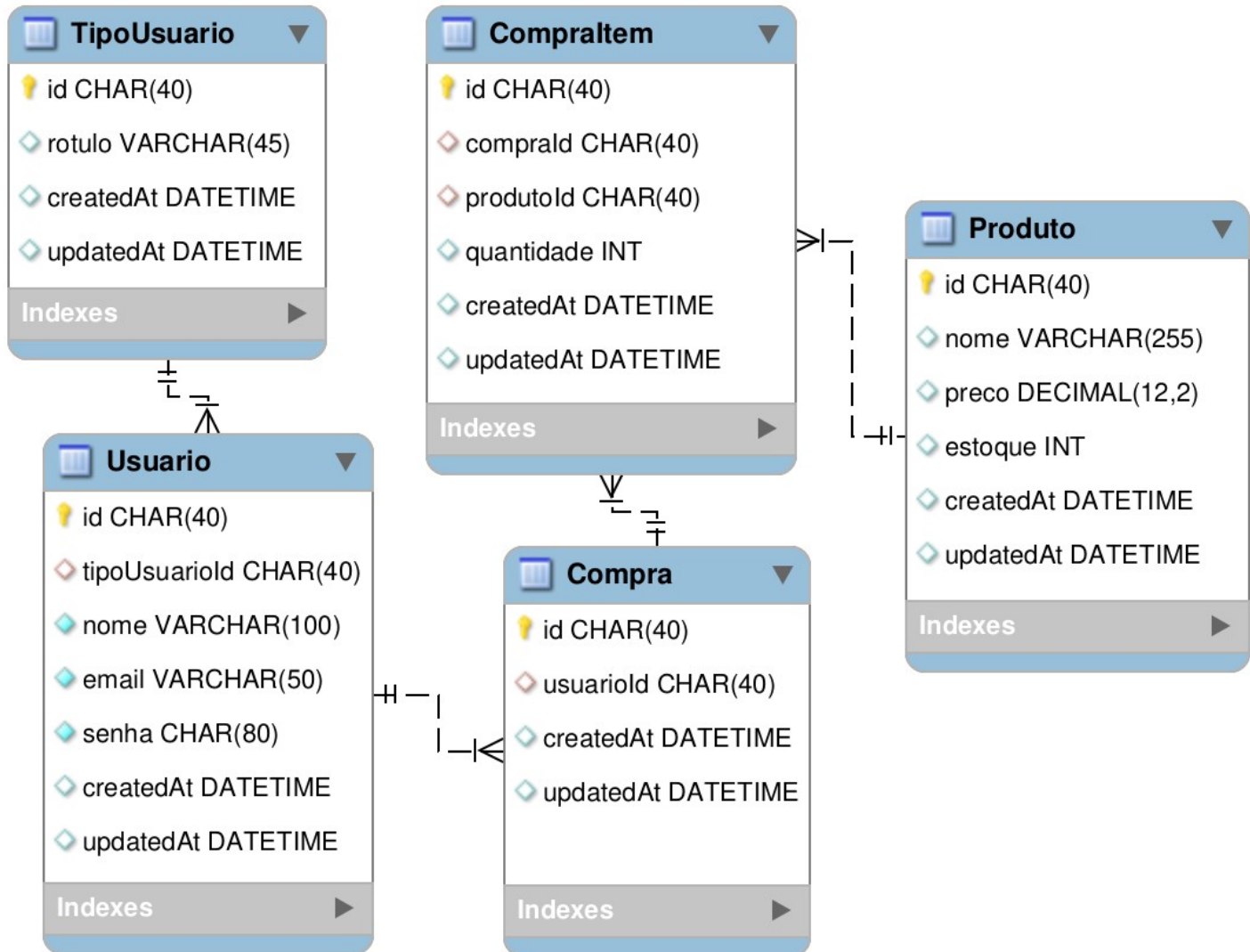
Movendo para o padrão REST

- Para o padrão REST, optamos por organizar os arquivos de nossa aplicação usando o esquema de pastas RESTful.
- O diretório **resources** terá um subdiretório para cada entidade da aplicação. Dentro dos subdiretórios, haverá um roteador, um controlador, um serviço e um arquivo de tipos de dados.

```
~|d/e/b/s|  
david@coyote ~/dev/expApi/backend/src [master]x $ tree -d  
├── db
```

```
~|d/e/b/s| produto  
david@coyote ~/dev/expApi/backend/src/resources/produto [master]x $ ls  
produto.controller.ts produto.service.ts  
produto.router.ts      produto.types.ts  
david@coyote ~/dev/expApi/backend/src/resources/produto [master]x $  
7 directories  
david@coyote ~/dev/expApi/backend/src [master]x $
```

Esquema do Banco de Dados



O Roteador

- O **Roteador** de cada resource contém as rotas associadas ao resource, referenciando as actions do controlador

```
// Arquivo src/resources/produto/produto.router.ts

import { Router } from 'express';
import produtoController from './produto.controller';
const router = Router();

// Produto controller
router.get('/', produtoController.index);
router.post('/', produtoController.create);
router.get('/:id', produtoController.read);
router.put('/:id', produtoController.update);
router.delete('/:id', produtoController.remove);

export default router;
```

O Roteador

- O **Roteador** de cada resource contém as rotas associadas ao resource, referenciando as actions do resource.

```
// Arquivo src/resources/produto/produto.router.ts
import { Router } from 'express';
```

O arquivo de rotas principal irá importar as rotas de cada resource

```
// Arquivo src/router/router.ts
import express from 'express';
import produtoRouter from '../resources/produto/produto.router';

const router = express.Router();
router.use('/produto', produtoRouter);
export default router;
```

```
export default router;
```

Camada de Serviços

- Os **serviços** são responsáveis por orquestrar as regras de negócio e servir de intermédio entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts

import { Produto } from '../../../models/Produto';
import { ProdCreateDto } from './produto.types';

const getAllProdutos = async (): Promise<Produto[]> => {
  const produtos = await Produto.findAll();
  return produtos.map((p) => p.toJSON());
};

const createProduto = async (produto: ProdCreateDto)
: Promise<Produto> => {
  return await Produto.create(produto);
};

export { getAllProdutos, createProduto };
```

Camada de Serviços

- Os **serviços** são responsáveis por orquestrar as regras de negócio e servir de intermédio entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts
```

```
import { Produto } from '../../../models/Produto';
```

Além das funções mostradas, o serviço de produtos precisa ter funções como:

```
const produtoJaExiste = async (nome: string): Promise<boolean> {
```

```
const getProduto = async (id: string): Promise<Produto>
```

```
const updateProduto = async (id: string, produto: ProdutoCreateDto):  
    Promise<[affectedCount: number]>
```

```
const removeProduto = async (id: string): Promise<number>
```

```
    return await Produto.create(produto);  
};
```

```
export { getAllProdutos, createProduto };
```

Camada de Serviços

- Os **serviços** são responsáveis por orquestrar as regras de negócio e servir de intermédio entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts
```

```
import { Produto } from '../../../models/Produto';
```

Além das funções mostradas, o serviço de produtos precisa ter funções como:

- Uma vantagem do uso de serviços é que suas funções podem ser utilizadas em outras partes da aplicação, diminuindo a réplica de códigos em vários arquivos.

```
Promise<[affectedCount: number]>
```

```
const removeProduto = async (id: string): Promise<number>
```

```
    return await Produto.create(produto);  
};
```

```
export { getAllProdutos, createProduto };
```


Camada de Serviços

- Os **serviços** são responsáveis por orquestrar as regras de negócio e servir de intermédio entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts  
  
import { Produto } from '../../../models/Produto';
```

Além das funções mostradas, o serviço de produtos precisa ter funções como:

- Uma vantagem do uso de serviços é que suas funções podem ser
- Outra vantagem dos serviços é que, caso se queira mudar o ORM da aplicação, o esforço será muito menor. Isso porque eles serão os únicos arquivos que usam os recursos do ORM para recuperar, atualizar e criar dados.

```
const removeProduto = async (id: string): Promise<number>  
  
  return await Produto.create(produto);  
};  
  
export { getAllProdutos, createProduto };
```

Data Transfer Objects (DTO)

- Os arquivos **resources/**/*types.ts** possuem as **interfaces** e **types**, em especial os **DTOs**, usados dentro do resource
- DTO é uma interface ou type usado para representar os objetos de dados que são trocados entre a API e as aplicações client
 - Por exemplo, para criar um novo produto, a aplicação cliente precisa enviar para a API os dados desse novo produto, sendo o formato desses dados é definido através de um DTO

Data Transfer Objects (DTO)

- Os DTOs geralmente contêm um subconjunto dos atributos de um dado modelo, e para gerá-los podemos usar o comando Pick

```
// Arquivo src/resources/produto/produto.types.ts  
  
import { Produto } from '../../../models/Produto';  
  
type ProdCreateDto = Pick<Produto, 'nome' | 'preco' | 'estoque'>;  
type ProdUpdateDto = Pick<Produto, 'nome' | 'preco' | 'estoque'>;  
  
export default { ProdCreateDto, ProdUpdateDto }
```

Cria um novo type contendo apenas as propriedades nome, preço e estoque do modelo Produto

Data Transfer Objects (DTO)

- Os DTOs são usados principalmente na camada de serviço, mas também podem ser utilizados nos controladores

```
// Arquivo src/resources/produto/produto.service.ts  
  
import { Produto } from '../../../models/Produto';  
import { ProdCreateDto } from './produto.types';  
  
const createProduto = async (produto: ProdCreateDto)  
: Promise<Produto> => {  
  return await Produto.create(produto);  
};
```

Camada do Controlador

- Os controladores são responsáveis por **receptionar** e **responder** as respostas dos usuários

```
// Arquivo src/resources/produto/produto.controller.ts

import { Request, Response } from 'express';
import { createProduto, jaExiste } from '../produto.service';
import { Produto } from '../../models/Produto';

const create = async (req: Request, res: Response) => {
  try {
    const jaExiste = await produtoJaExiste(req.body.nome);
    if (jaExiste) return res.status(400).json({
      message: 'Produto já existe'
    });
    const produto = await createProduto(req.body);
    res.status(201).json(produto);
  } catch (e) {
    res.status(500).json({ error: e });
  }
}
```

API Documentation & Design x +

swagger.io

Swagger
Supported by SMARTBEAR

Why Swagger? Tools Resources

Sign In Try Free

API Development for Everyone

Simplify API development for users, teams, and enterprises with the Swagger open source and professional toolset.

Find out how Swagger can help you design and document

Exercício: Crie uma API REST usando o framework Express contendo os endpoints **index**, **create**, **read**, **update** e **delete** para os resources **Produto** e **Usuário** (com senha criptografada).

anything?



The screenshot shows a REST client interface with a POST request to `base_url/produto:` that resulted in a **500 Internal Server Error**. The request body is a JSON object with `nome: "ABC"`, `preco: 771`, and `estoque: 427`. The response body is a JSON object indicating a Sequelize validation error for the `nome` field, which is too short (5 characters) as the validator requires between 5 and 40 characters.

Request:

```
POST base_url/produto:
{
  "nome": "ABC",
  "preco": 771,
  "estoque": 427
}
```

Response:

```
{
  "name": "SequelizeValidationError",
  "errors": [
    {
      "message": "O nome precisa ter entre 5 e 40 caracteres.",
      "type": "Validation error",
      "path": "nome",
      "value": "ABC",
      "origin": "FUNCTION",
      "instance": {
        "id": null,
        "nome": "ABC",
        "preco": 771,
        "estoque": 427,
        "updatedAt": "2021-09-13T11:23:35.019Z",
        "createdAt": "2021-09-13T11:23:35.019Z"
      },
      "validatorKey": "len",
      "validatorName": "len",
      "validatorArgs": [
        5,
        40
      ],
      "original": {
        "validatorName": "len",
        "validatorArgs": [

```

Note: Note que o array **errors** do Sequelize deve ser retornado, caso os dados enviados não satisfaçam algum validador definido.

Nodejs - Criando documentação com Swagger - Code/drops #85



API Development
for Everyone

Exercício: Após isso, instalar o Swagger (<https://swagger.io/>) para documentar a sua API. Dica: siga as orientações do vídeo <https://youtu.be/WhFx2heoFrA>



OpenAPI Specification

The power of Swagger tools starts with the OpenAPI Specification — the industry standard for RESTful API design



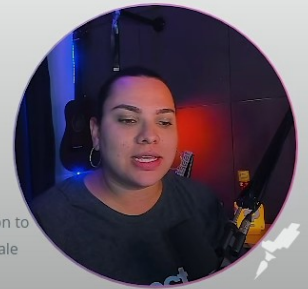
Open Source Tools

Individual tools to create, update and share OpenAPI definitions with consumers



SwaggerHub

SwaggerHub is the platform solution to support OpenAPI workflows at scale



Role para ver detalhes

express JS