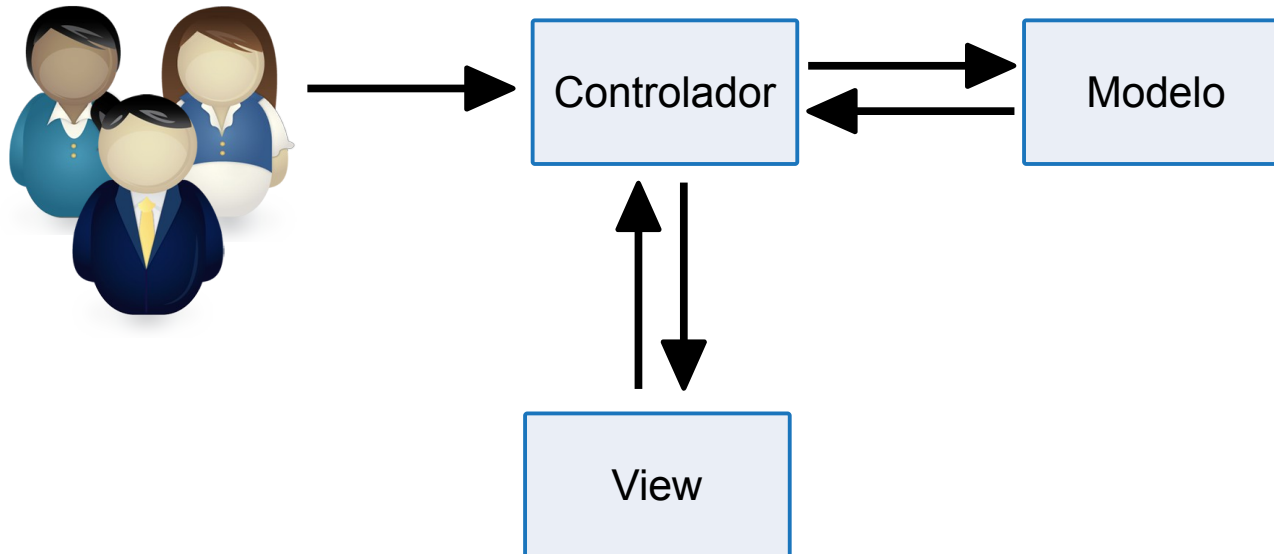




Prof. David Fernandes de Oliveira
Instituto de Computação
UFAM

O MVC

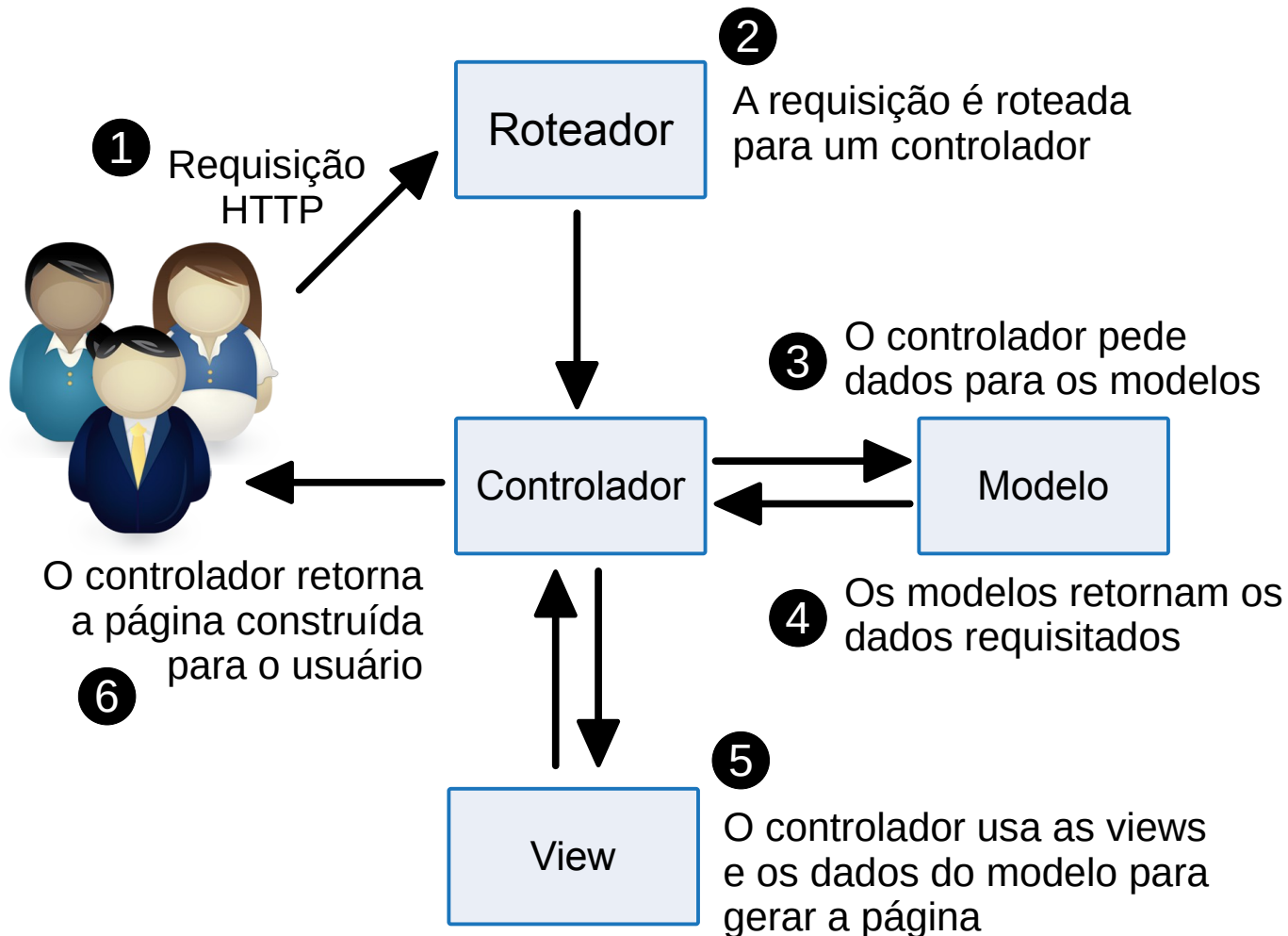
- O MVC é um padrão de arquitetura de software que separa as aplicações em 3 camadas: **Modelos**, **Views** e **Controladores**
 - O objetivo de separar a arquitetura nas três camadas é facilitar a organização, compreensão e a manutenção do código
- Frameworks Web MVC: Yii2, Laravel, Sails, Adonis,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



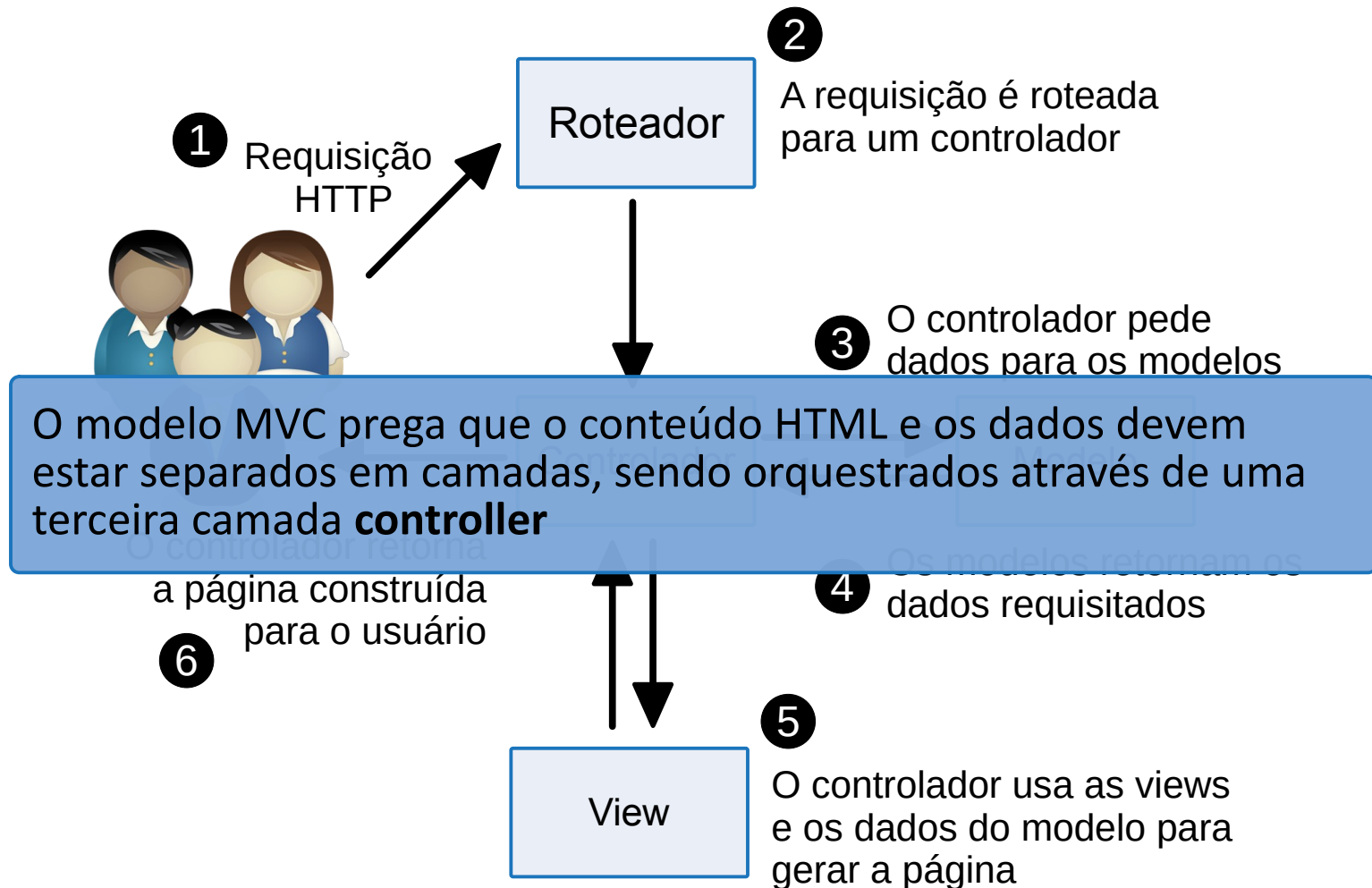
O MVC

- 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

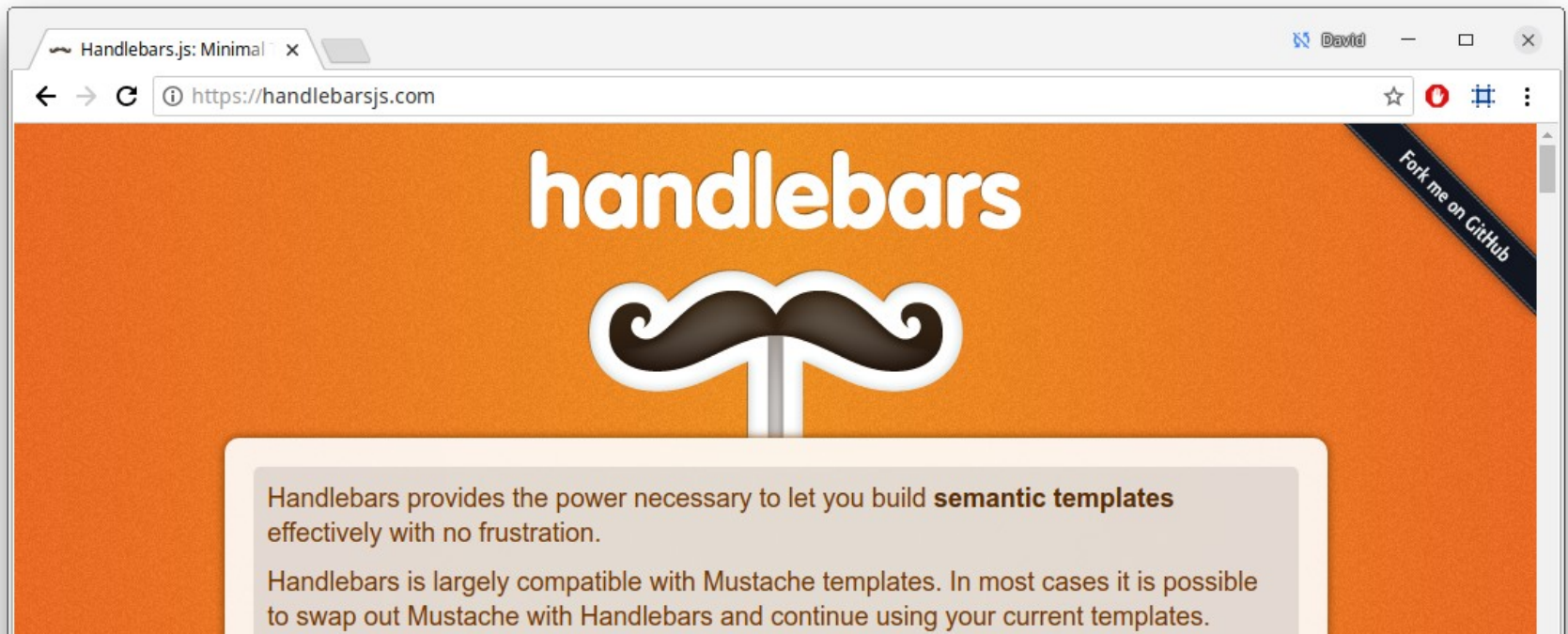


O MVC



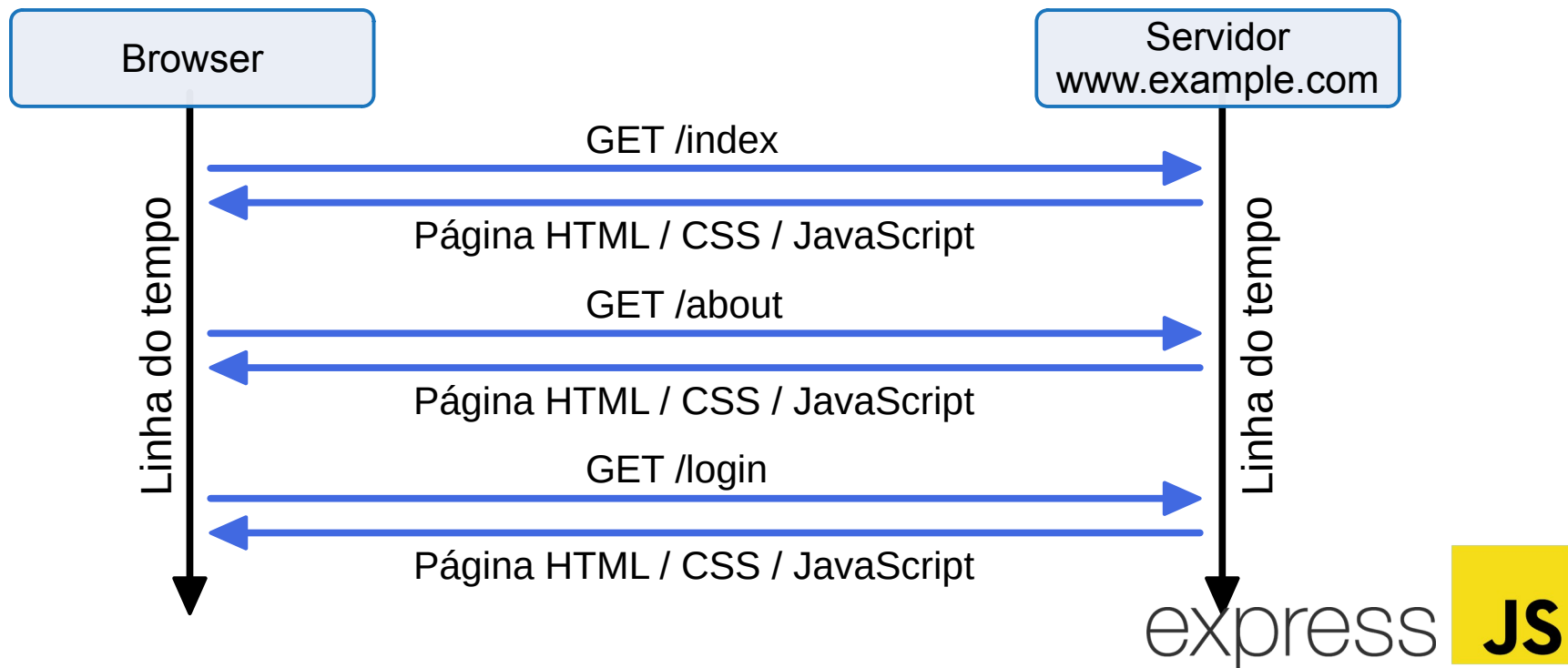
Views

- 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



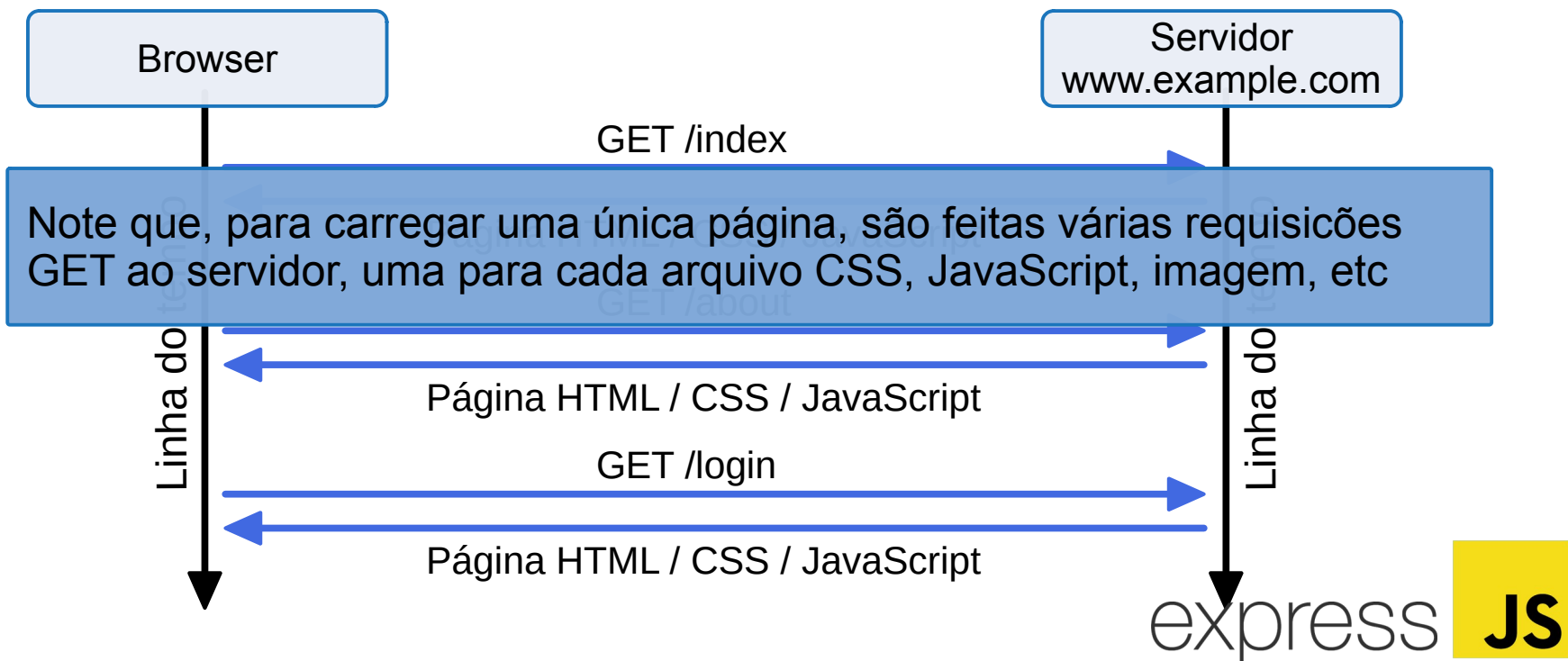
Sistemas MVC

- 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



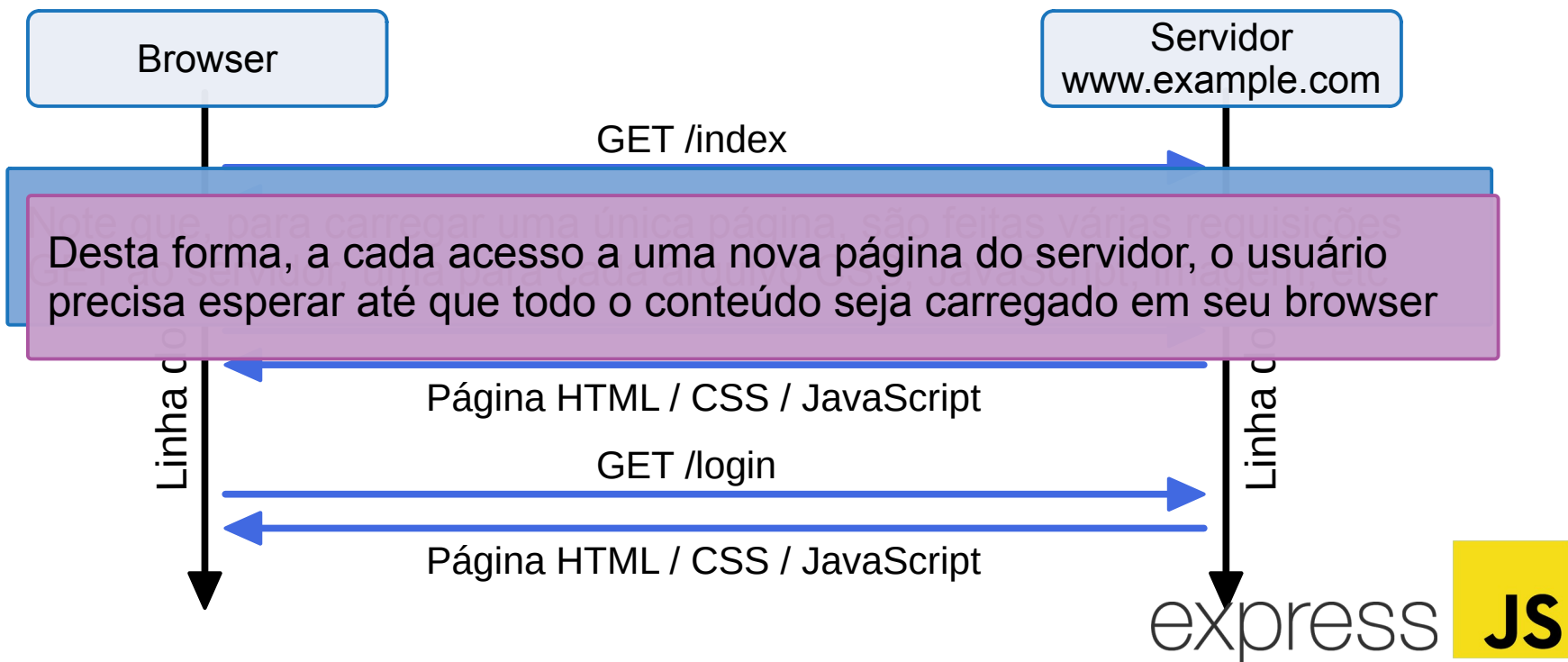
Sistemas MVC

- 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**



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

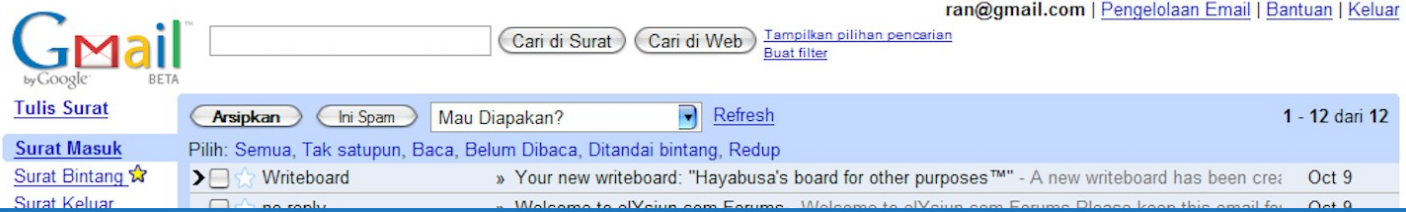
Anda sekarang memakai 0 MB (0%) dari kuota Anda sebesar 2654 MB.

[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**



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?](#)

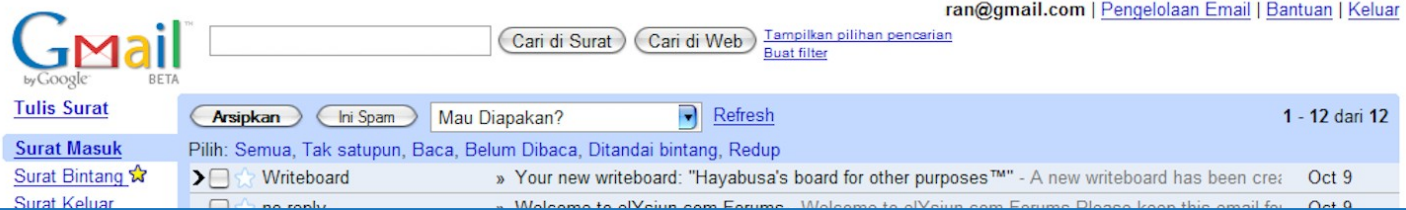
Anda sekarang memakai 0 MB (0%) dari kuota Anda sebesar 2654 MB.

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

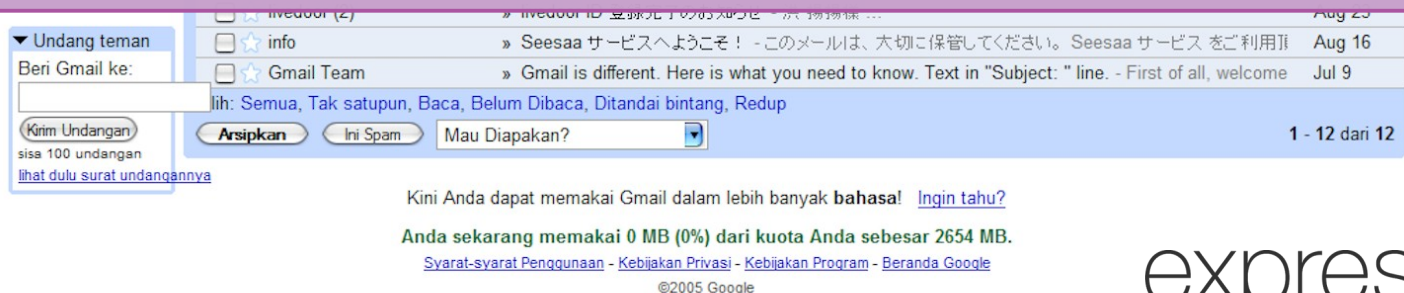
©2005 Google

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**

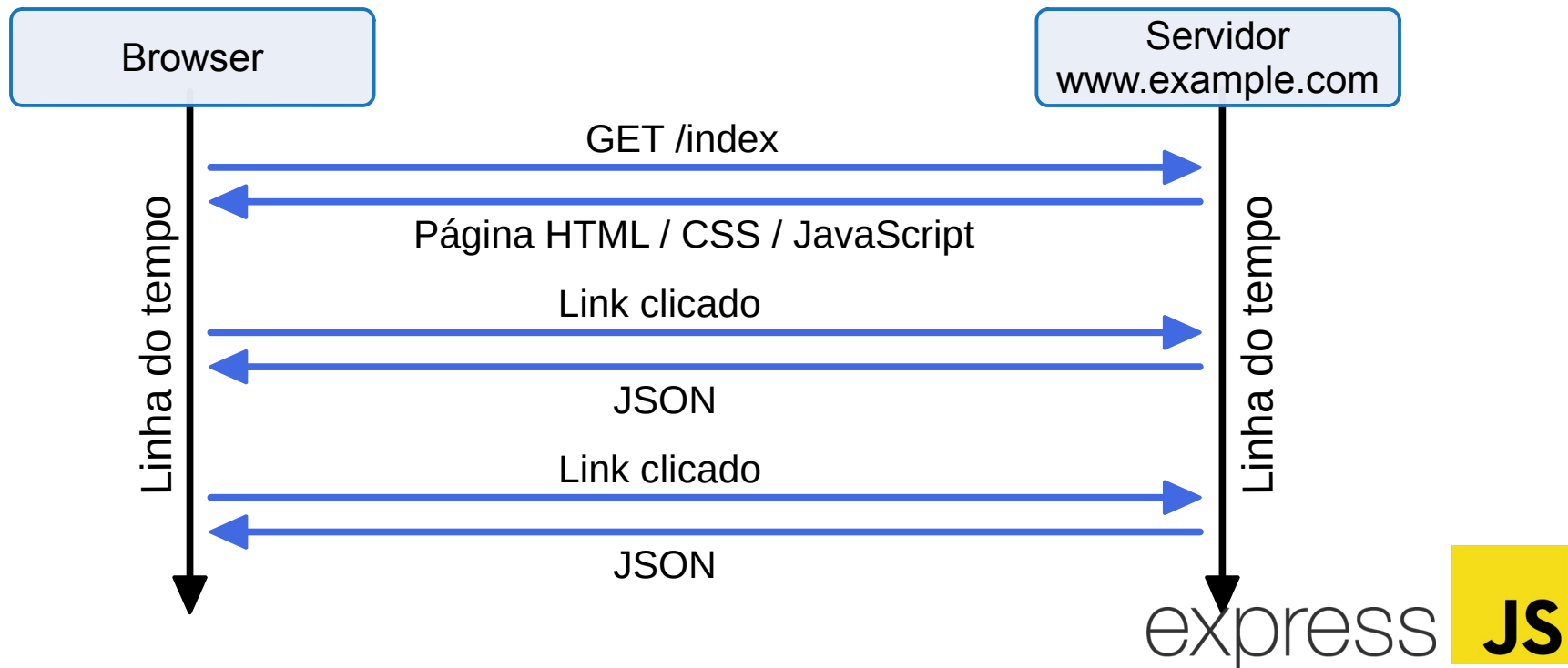


Foi uma das primeiras vezes que se conseguiu trazer a sensação de uso de uma aplicação desktop para dentro de um sistema Web - e esse é um dos pilares das single-page applications



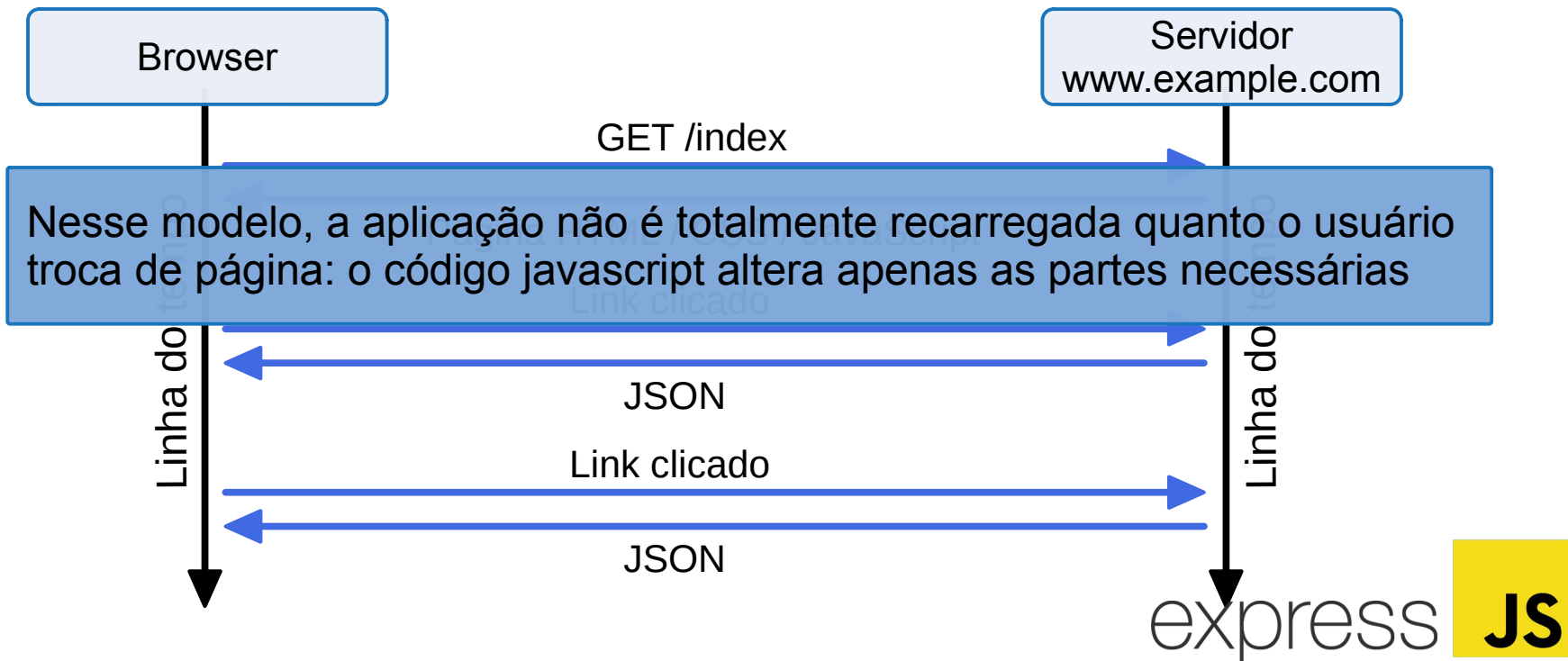
Single-Page Applications

- 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**

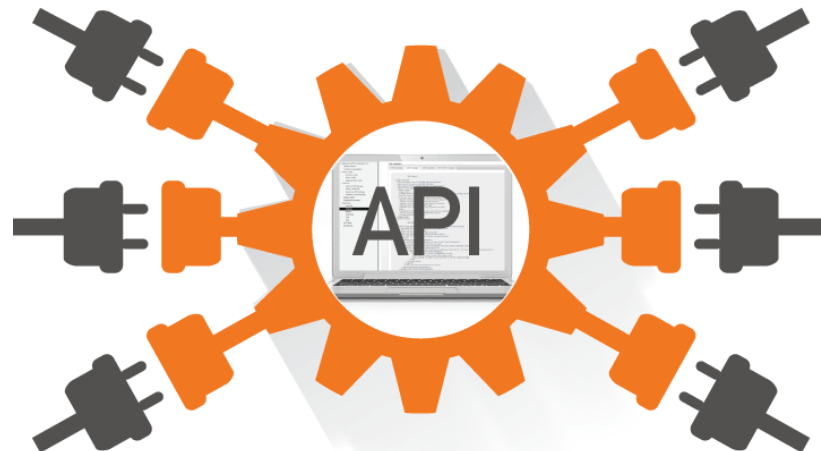
- Técnicas mais consolidadas
- **Search Engine Optimization** mais simples
- Mais fácil de implementar
- Menor acoplamento com código javascript no lado cliente

- 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

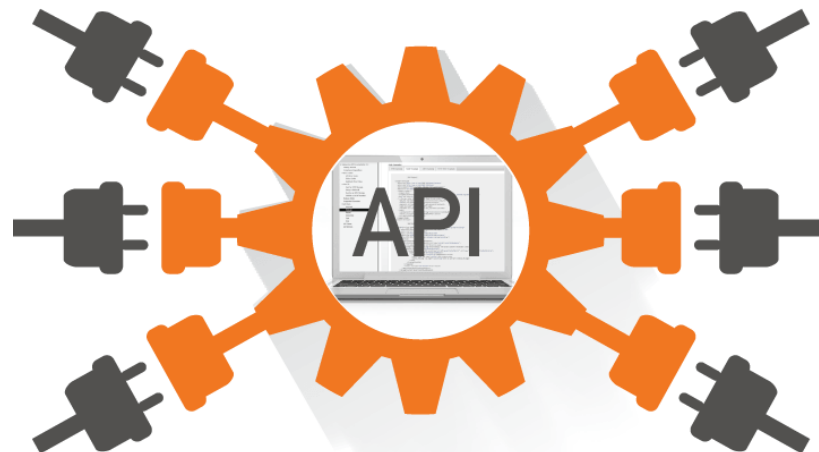
REST APIs

- 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 desenvolvido através de vários tipos de frameworks frontend, como o **React**, **Angular** e o **Vue.js**
- aplicação



Sistema de Loja Virtual

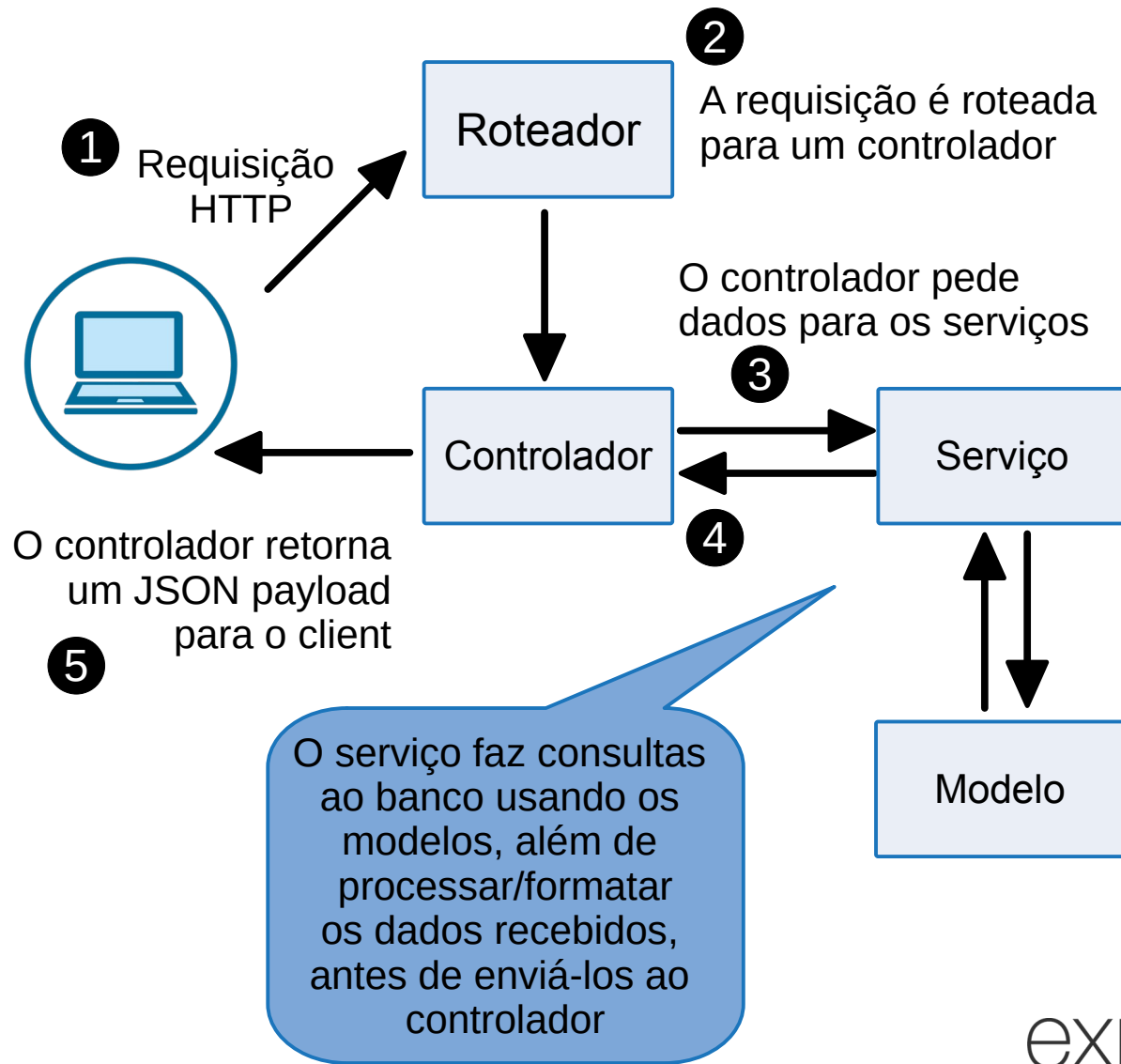
- Como prática de desenvolvimento nesta etapa do curso, cada aluno irá desenvolver um sistema SPA para uma loja virtual
- Além da API, que será desenvolvida no presente módulo, a loja virtual também terá um frontend, testes integrados, além de uma estratégia de CI/CD



Movendo para o padrão REST

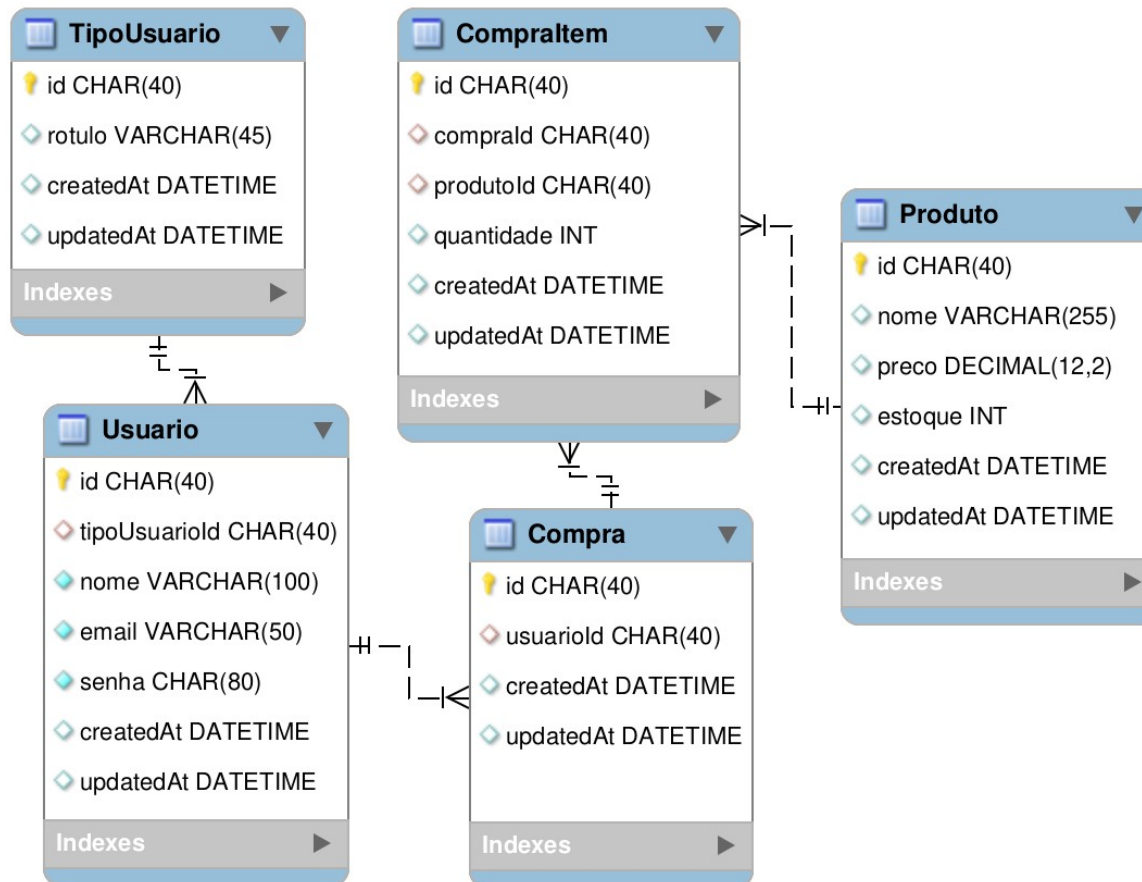
- Nas aplicações MVC, cada página é contruída através de uma action de um dado controlador
 - Por exemplo a **função** about do controlador **main** tem por objetivo construir e retornar o conteúdo HTML da página **/about**
- Nas aplicações REST, por outro lado, as páginas de uma aplicação são definidas 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



Esquema de banco de dados e ORM

- Nossa aplicação usará o ORM Prisma, e os modelos serão definidos no diretório **prisma/schema.prisma**

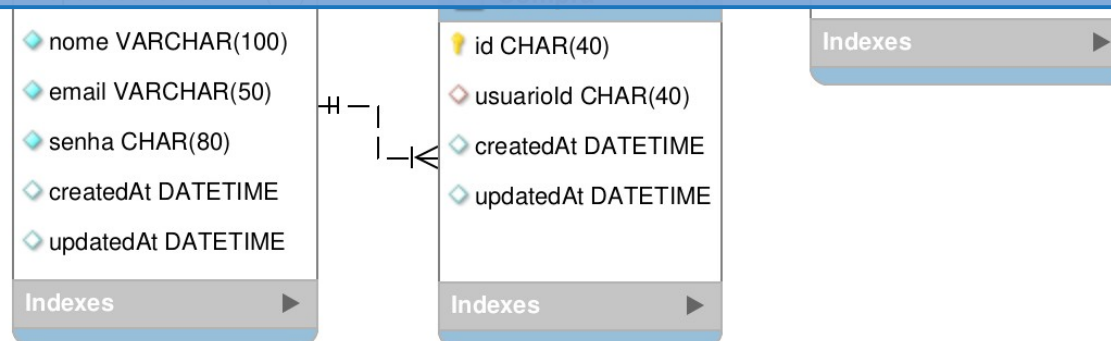


Esquema de banco de dados e ORM

- Nossa aplicação usará o ORM Prisma, e os modelos serão definidos no diretório **prisma/schema.prisma**

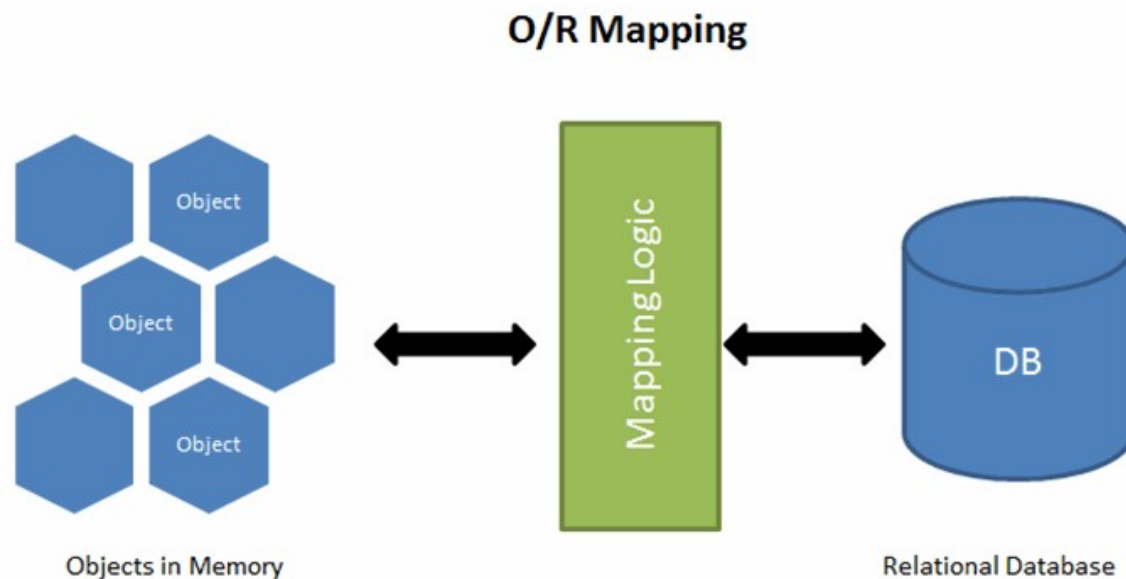
Descrição das Tabelas:

- **Usuario:** dados dos usuários da aplicação
- **TipoUsuario:** determina se o usuário é cliente ou admin
- **Compra:** compras feitas pelos clientes
- **Compraltem:** itens das compras feitas
- **Produto:** produtos vendidos na loja

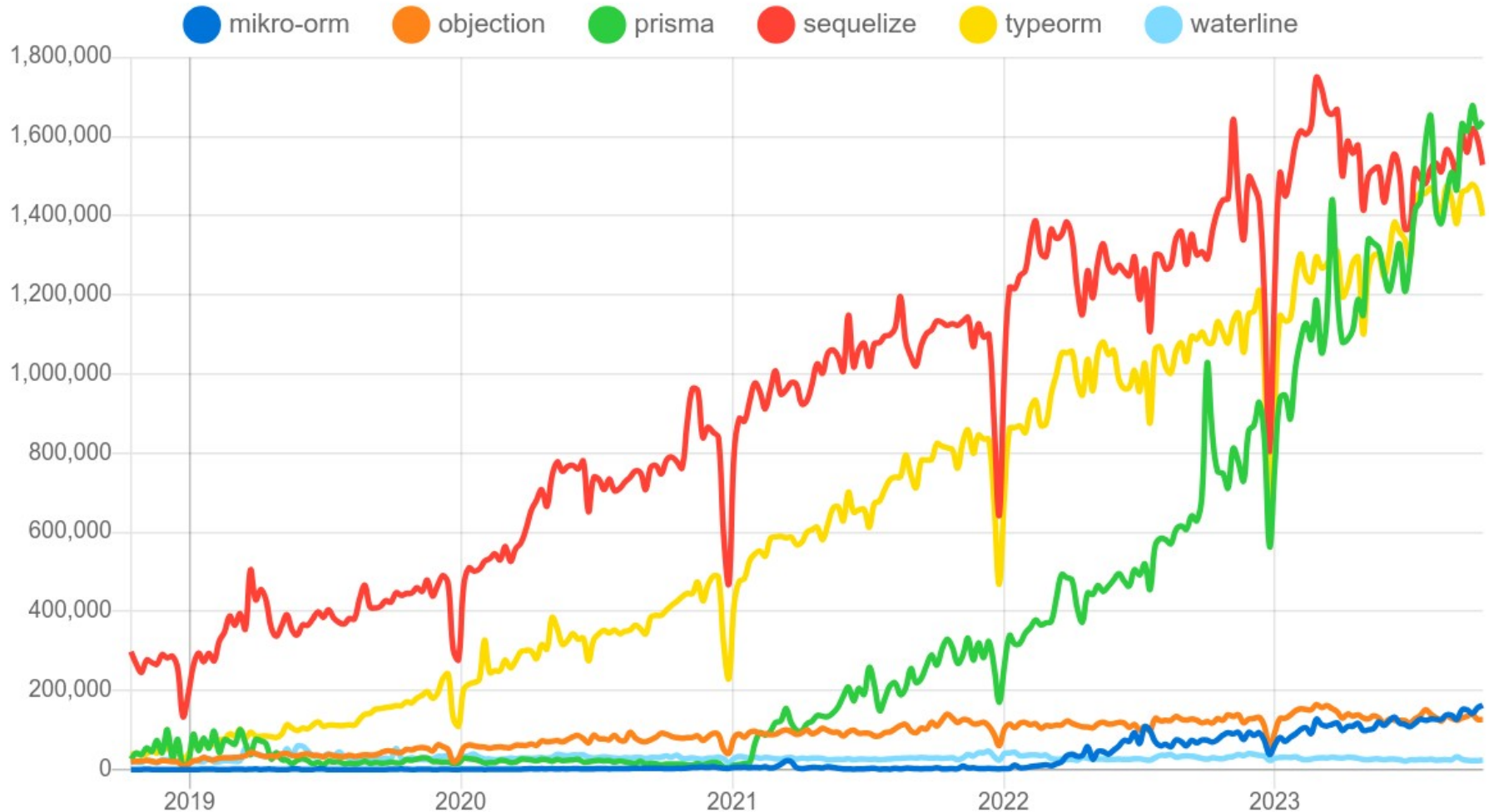


Object Relational Mapper - ORM

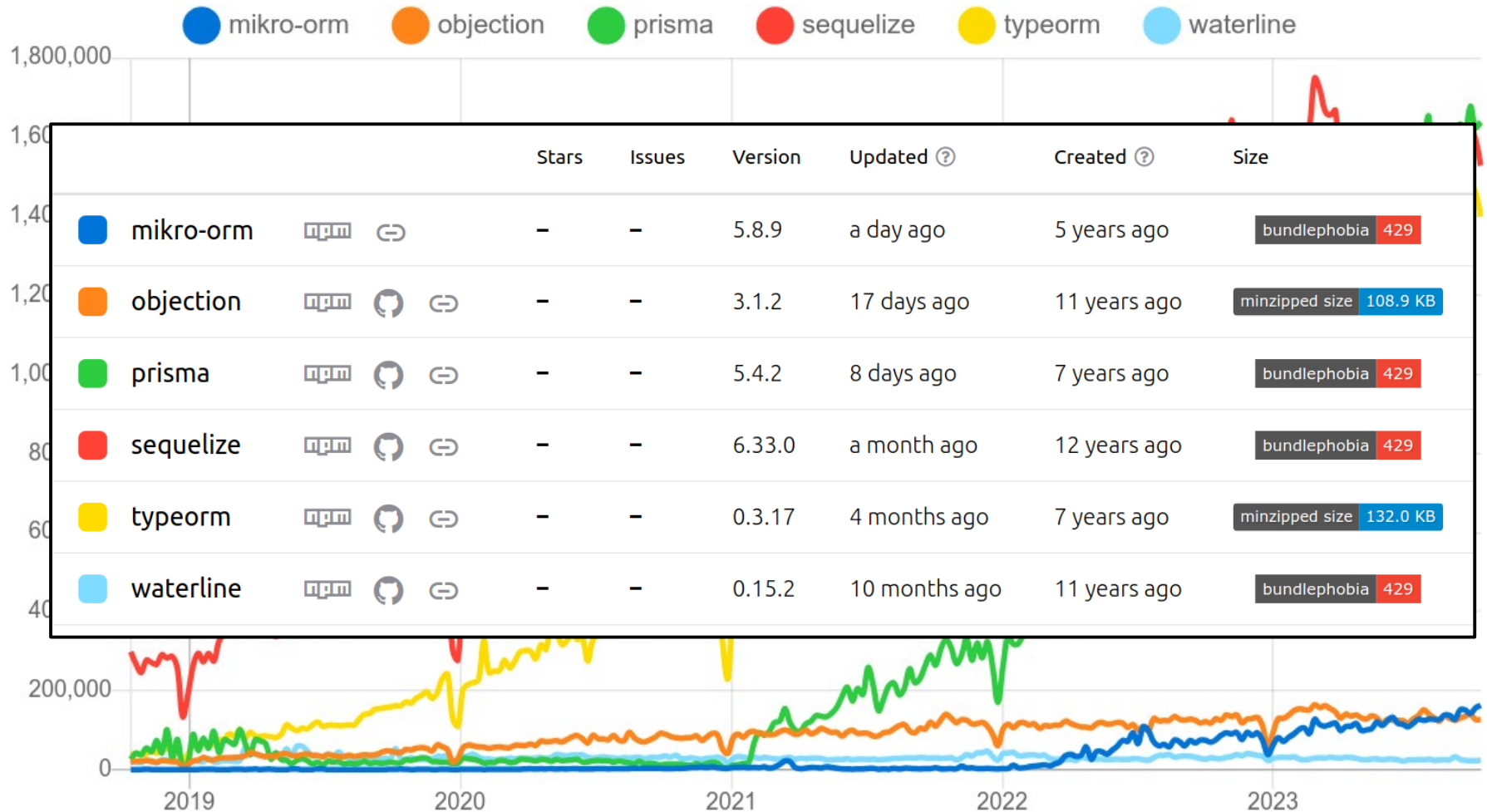
- ORM é uma técnica que permite **consultar e manipular dados** de um database usando o paradigma de orientação a objetos
- Desta forma, o acesso aos dados não é feito através da linguagem SQL, e sim através de objetos



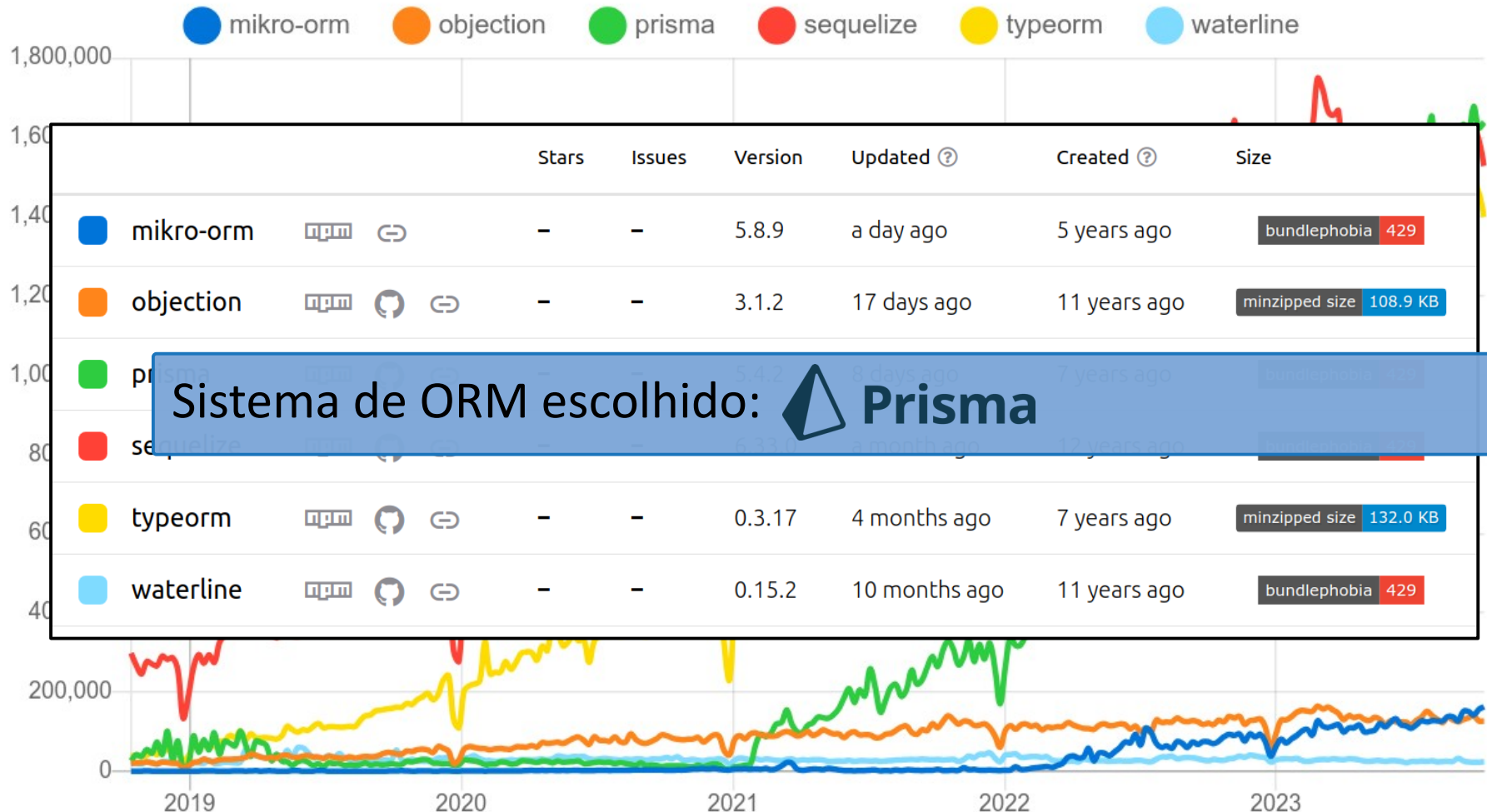
Escolhendo o ORM



Escolhendo o ORM



Escolhendo o ORM





- Use os comandos abaixo para instalar o **Prisma** como dependência de desenvolvimento

```
$ npm install -D prisma
```

- Após isso, execute o comando abaixo para que o Prisma crie os arquivos iniciais de configuração

```
$ npx prisma init
```

~/d/e/backend

```
david@coyote ~/dev/expApi/backend [main]× $ npx prisma init
```

✓ Your Prisma schema was created at `prisma/schema.prisma`
You can now open it in your favorite editor.

warn You already have a `.gitignore` file. Don't forget to add ``.env`` in it to not commit any private information.

```
$ npm install -D prisma
```

Next steps:

1. Set the `DATABASE_URL` in the `.env` file to point to your existing database. If your database has no tables yet, read <https://pris.ly/d/getting-started>
2. Set the `provider` of the `datasource` block in `schema.prisma` to match your database: `postgresql`, `mysql`, `sqlite`, `sqlserver`, `mongodb` or `cockroachdb`.
3. Run `prisma db pull` to turn your database schema into a Prisma schema.
4. Run `prisma generate` to generate the Prisma Client. You can then start querying your database.

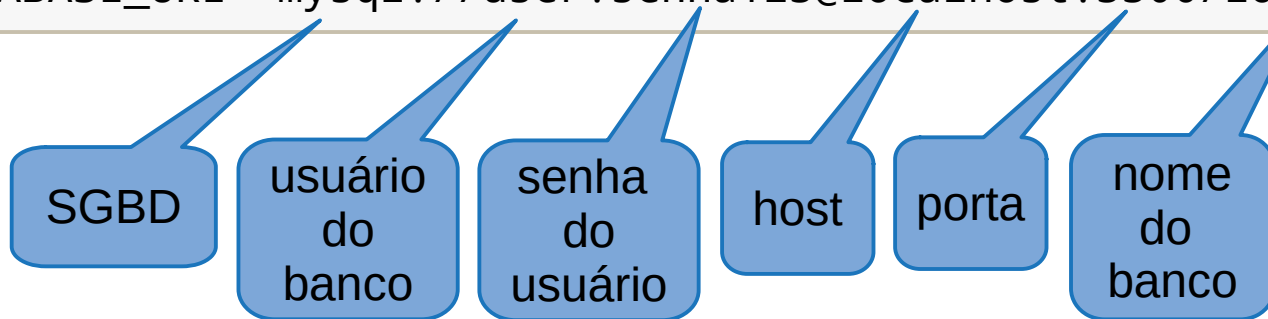
More information in our documentation:
<https://pris.ly/d/getting-started>

```
david@coyote ~/dev/expApi/backend [main]× $
```



- O comando **prisma init** adiciona uma variável `DATABASE_URL` no arquivo **.env**, contendo uma string de conexão com o banco
- Será preciso editar o valor dessa variável conforme a realidade do banco de dados utilizado

```
DATABASE_URL="mysql://user:senha123@localhost:3306/loja"
```



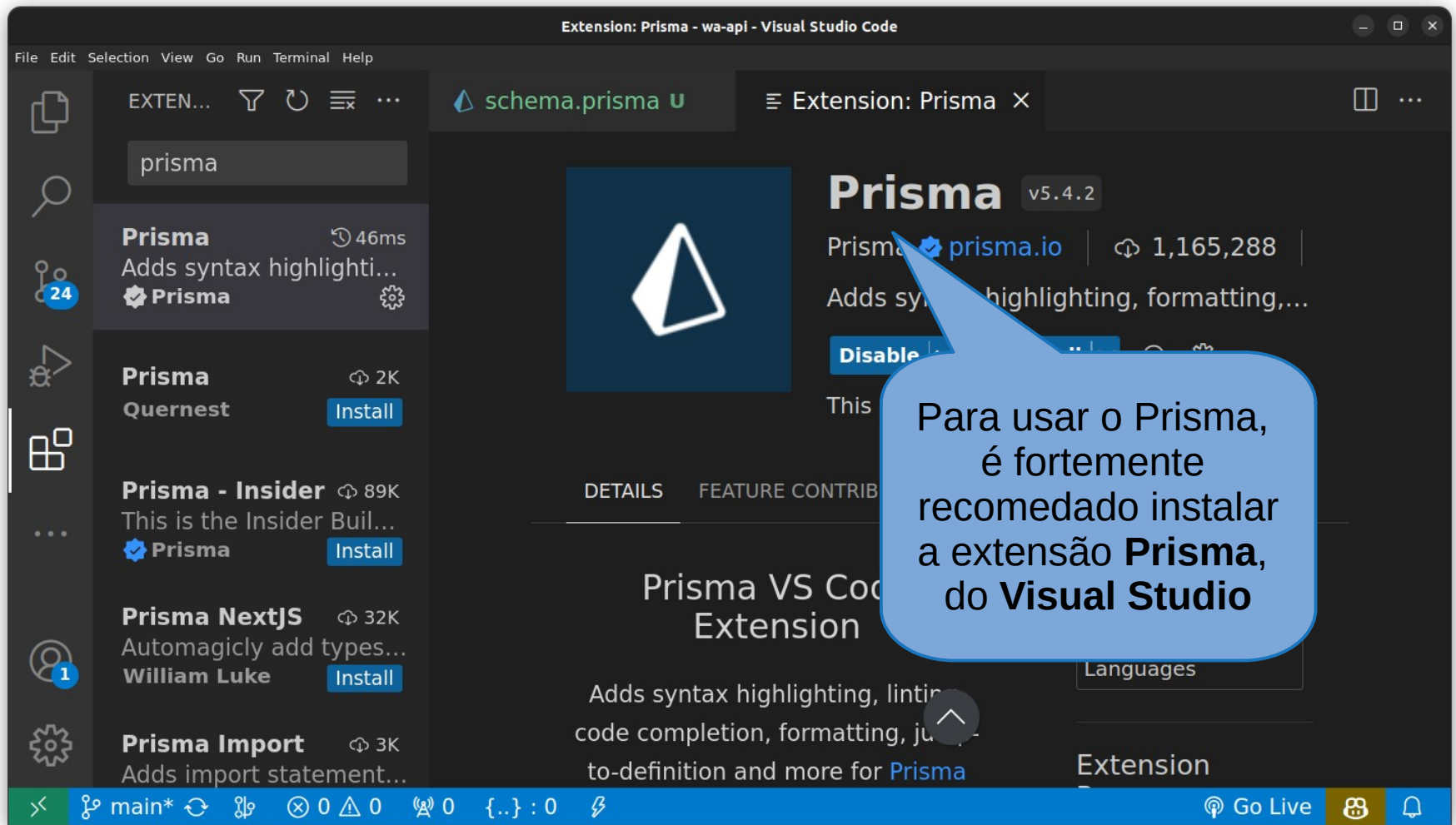


- O comando **prisma init** também cria um arquivo **prisma/schema.prisma**, onde serão criados os modelos da aplicação
- Nesse arquivo, é importante alterar o **provider** para **mysql**, conforme mostrado abaixo

```
generator client {  
  provider = "prisma-client-js"  
}  
  
datasource db {  
  provider = "mysql"  
  url      = env("DATABASE_URL")  
}
```

Mudar o
provider
para mysql

A blue speech bubble with a tail pointing to the "mysql" value in the code block above.



Adicionando um modelo

- Ainda no arquivo **prisma/schema.prisma**, vamos adicionar o primeiro modelo de nossa aplicação

```
model Produto {  
  id          String   @id @default(uuid()) @db.Char(40)  
  nome        String   @unique @db.VarChar(100)  
  preco       Decimal  @db.Decimal  
  estoque     Int       @db.Int  
  createdAt   DateTime @default(now()) @map("created_at")  
  updatedAt   DateTime @updatedAt @map("updated_at")  
  
  @@map("produtos")  
}
```

Adicionando um modelo

- Após isso, usamos o comando **npx prisma migrate dev --name create-produto-table** para gerar a migração e criar a tabela produtos

```
~ /d/e/backend
david@coyote ~/dev/expApi/backend [main]× $ npx prisma migrate dev --name create-produto-table
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": MySQL database "loja" at "localhost:3306"

Applying migration `20231019144451_create_produto_table`

The following migration(s) have been created and applied from new schema changes:

migrations/
├─ 20231019144451_create_produto_table/
└─ migration.sql

Your database is now in sync with your schema.

✓ Generated Prisma Client (v5.4.2) to ./node_modules/@prisma/client in 63ms
```

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/produto/?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/>

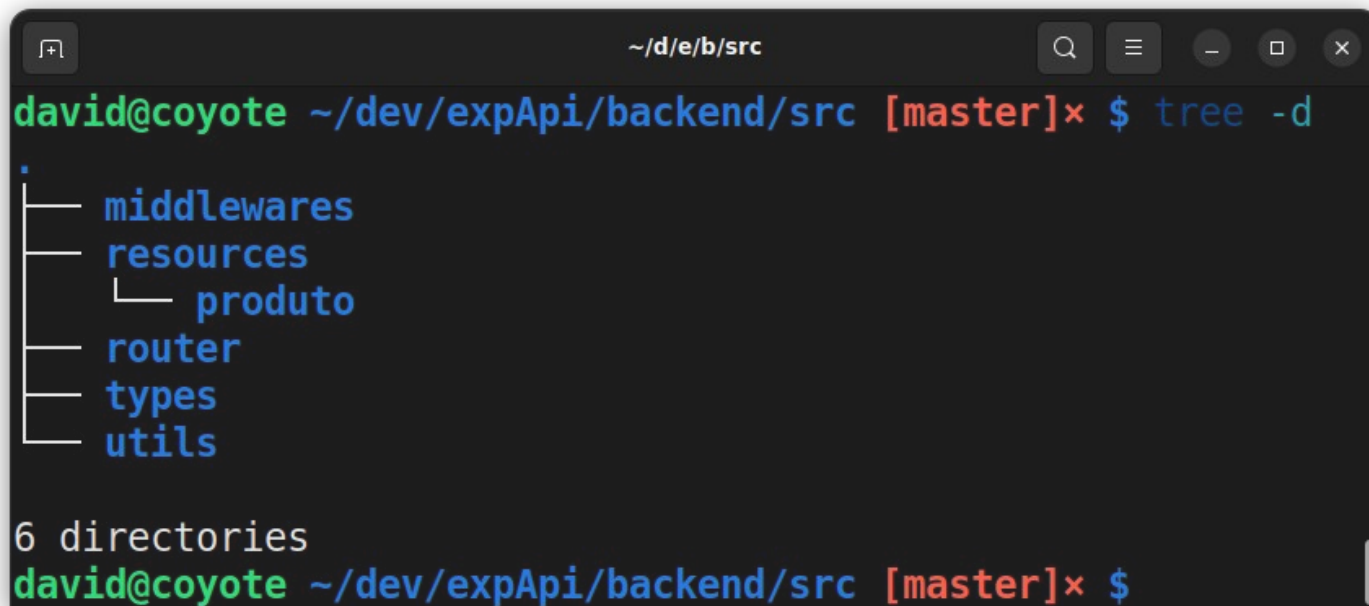
`https://http.cat/[status_code]`

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

 100 Continue	 101 Switching Protocols	 102 Processing
		

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, exceto os **modelos**

A terminal window with a dark background. The title bar shows the path ~/d/e/b/src. The prompt is david@coyote. The command tree -d has been executed, showing a directory tree with middlewares, resources (containing produto), router, types, and utils. Below the tree, it says '6 directories'. The prompt is now david@coyote.

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

6 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 recursos.
- O diretório **resources** terá um subdiretório para cada entidade da aplicação, exceto para o **middlewares**.

Dentro dos subdiretórios, haverá um roteador, um controlador, um serviço e um arquivo de tipos

O diretório **resources** terá um subdiretório para cada entidade da aplicação

```
~/dev/expApi/backend/src [master]x $ tree -d
├── middlewares
```

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


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 controlador

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

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

Embora não faça parte do CRUD, o objetivo da rota **/produto** é listar os produtos existentes

```
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 controlador

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

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

```
// Embora não faça parte do CRUD, o objetivo da rota /produto
```

Note que as rotas para **read**, **update** e **remove** terminam com a string **:id**, que representa um parâmetro utilizado para informar que produto se deseja **ler**, **atualizar** ou **apagar**

```
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/v1Router.ts
import express from 'express';
import produtoRouter from '../resources/produto/produto.router';

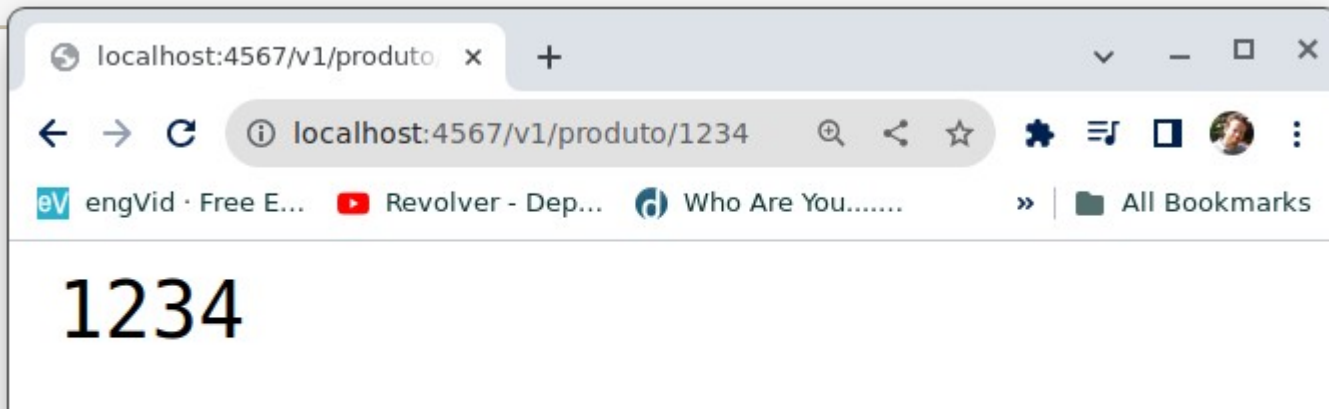
const router = express.Router();
router.use('/produto', produtoRouter);
export default router;
```

```
export default router;
```

O Roteador

- Os **parâmetros** permitem passar dados informações adicionais para o endpoint desejado
 - Por exemplo, na url **http://localhost:3000/produto/1234**, o valor do parâmetro **id** é 1234
- Para ler o valor de **id** dentro de uma função, podemos usar o atributo **param** de **req** (objeto da requisição do usuário):

```
async function read (req, res) {  
  const produtoId = req.params.id;  
  res.end(produtoId);  
},
```



O Roteador

- O Express possui um middleware chamado **json()**, que é usado para extrair os dados do corpo da requisição (**req.body**)
- Para usá-lo, basta inserir a linha abaixo no arquivo **src/index.ts** antes da chamada ao middleware router:

```
// Arquivo src/index.ts  
...  
app.use(express.json());  
app.use(router);
```

- Após isso, o **express.json()** irá extrair os dados do **request body** das requisições e copiá-los no objeto **req.body**

Camada de Serviços

- Os **serviços** têm como função orquestrar as regras de negócio e servir de intermediários entre controladores e modelos

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

import { PrismaClient, Produto } from '@prisma/client';
import { CreateProdutoDto } from '../produto.types';
const prisma = new PrismaClient();

export async function getAllProdutos(): Promise<Produto[]> {
  return await prisma.produto.findMany();
}

export async function createProduto(
  produto: CreateProdutoDto
): Promise<Produto> {
  return await prisma.produto.create({ data: produto });
}
```

Camada de Serviços

- Os **serviços** têm como função orquestrar as regras de negócio e servir de intermediários entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts  
  
import { PrismaClient, Produto } from '@prisma/client';
```

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>
```

```
produto: CreateProdutoDto  
) : Promise<Produto> {  
  return await prisma.produto.create({ data: produto });  
}
```

Camada de Serviços

- Os **serviços** têm como função orquestrar as regras de negócio e servir de intermediários entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts  
  
import { PrismaClient, Produto } from '@prisma/client';
```

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>
```

```
    produto: CreateProdutoDto  
): Promise<Produto> {  
    return await prisma.produto.create({ data: produto });  
}
```


Camada de Serviços

- Os **serviços** têm como função orquestrar as regras de negócio e servir de intermediários entre controladores e modelos

```
// Arquivo src/resources/produto/produto.service.ts  
  
import { PrismaClient, Produto } from '@prisma/client';
```

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>  
  produto.createProdutoDTO  
) : Promise<Produto> {  
  return await prisma.produto.create({ data: produto });  
}
```

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 { PrismaClient, Produto } from '@prisma/client';
import { CreateProdutoDto } from '../produto.types';
const prisma = new PrismaClient();

export async function createProduto(
  produto: CreateProdutoDto
): Promise<Produto> {
  return await prisma.produto.create({ data: 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';

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

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';

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

Camada do Controlador

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

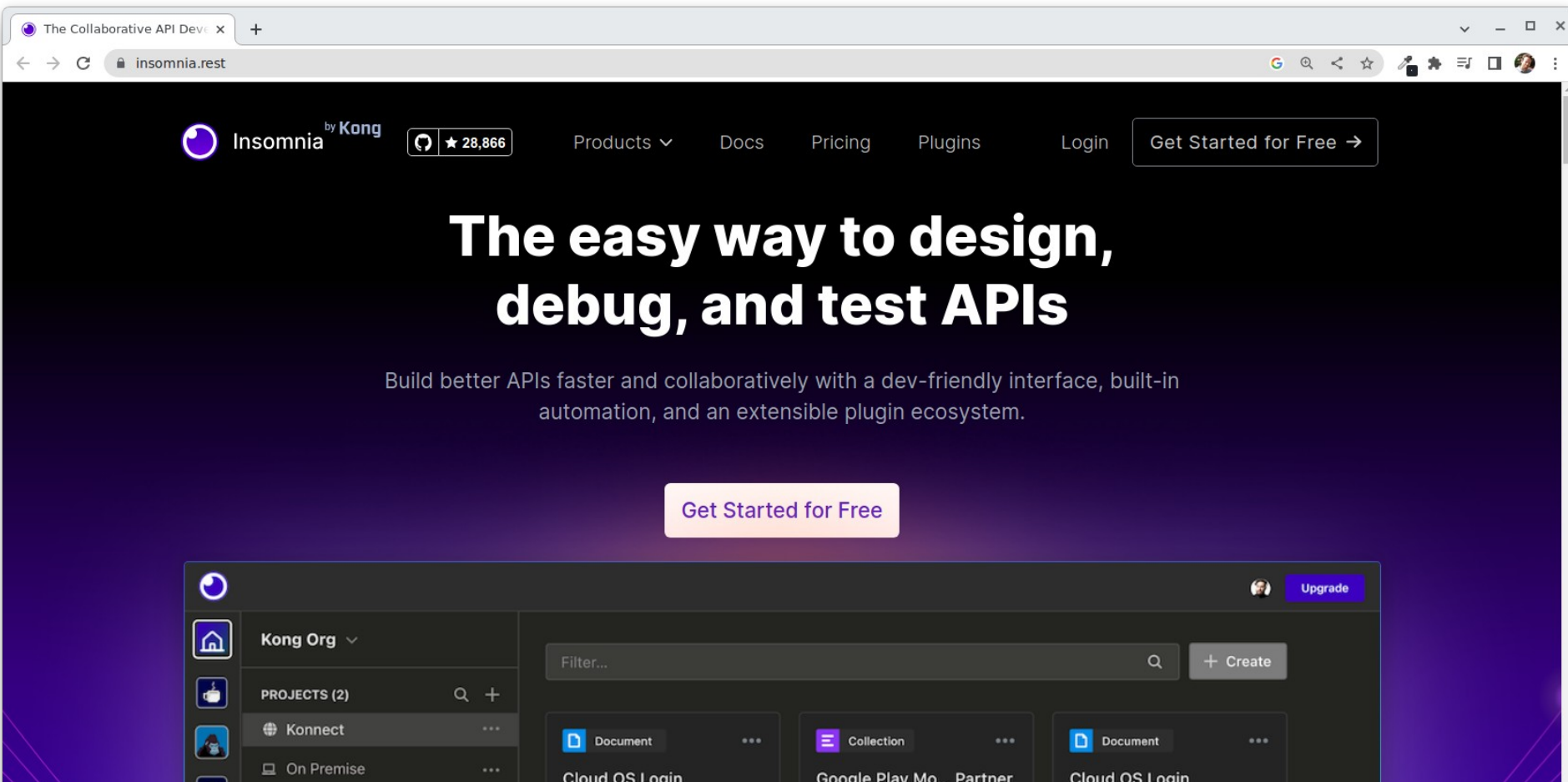
```
// Arquivo src/resources/produto/produto.controller.ts
```

É importante notar que erros na inserção de um registro disparam uma exceção no bloco try do Controlador, ocasionando um status 500 para a requisição.

```
const produto = req.body;
try {
  if (await jaExiste(produto.nome)) {
    return res.status(400).json({ msg: 'Produto já existe' });
  }
  const novoProduto = await createProduto(produto);
  res.status(201).json(novoProduto);
} catch (err) {
  res.status(500).json(err);
}
```

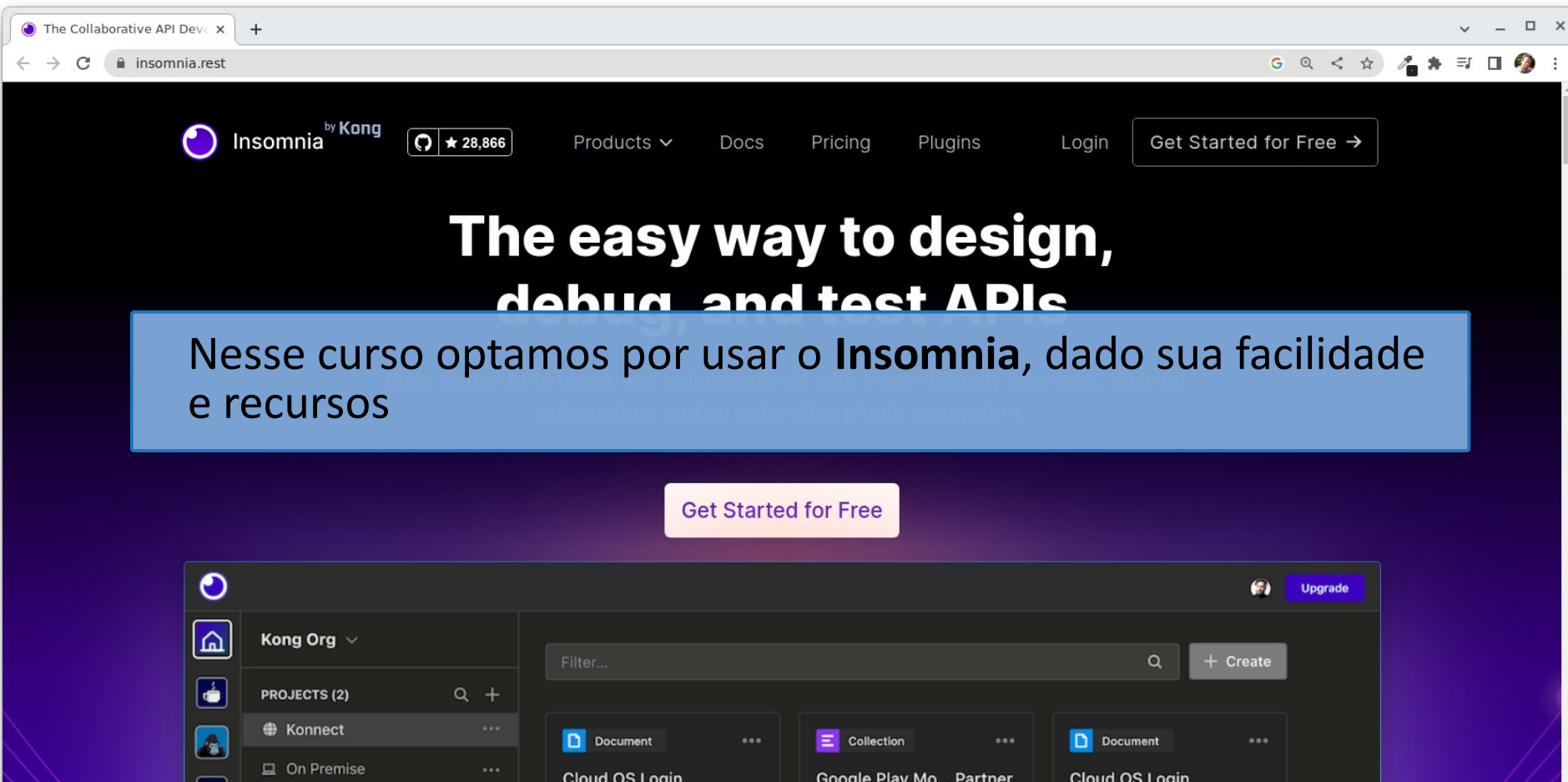
Insomnia

- Existem várias ferramentas para testar os endpoints de uma Api, como o **Insomnia**, o **Postman** e o **Thunder Client**



Insomnia

- Existem várias ferramentas para testar os endpoints de uma Api, como o **Insomnia**, o **Postman** e o **Thunder Client**



Podemos usar o plugin **insomnia-plugin-faker** para gerar requisições com dados fakes

Insomnia - Loja - Create Produto

Application Edit View Window Tools Help

No Envir

Filter

POST `_.base_url/v1/produto` Send

Auth Query Header 1 Docs

```
1
2  "nome": "Faker => Commerce ",
3  "preco": Faker => Commerce ,
4  "estoque": Faker => Random
5 }
```

Beautify JSON

201 Created 22.3 ms 179 B 2 Minutes Ago

Preview Header 7 Cookie Timeline

```
1 {
2   "id": "a09edab0-1a5d-11ee-ac7e-a531058f3526",
3   "nome": "Gorgeous Granite Bike",
4   "preco": 490,
5   "estoque": 33,
6   "updatedAt": "2023-07-04T11:26:32.412Z",

```

\$.store.books[*].author

GET Create Usuario

PUT Update Usuario

DEL Delete Usuario

Produto

GET Get All Produtos

GET Read Produto

POST Create Produto

PUT Update Produto

DEL Delete Produto

Insomnia - Loja - Create Produto

Application Edit View Window Tools Help

Insomnia / Loja ▾

No Environment ▾ Cookies

POST ▾ `_.base_url/v1/produto` Send

Filter

JSON ▾ Auth ▾ Query Header 1 Docs

1 {
2 "nome": "ABC",
3 "preco": Faker ⇒ Commerce,
4 "estoque": Faker ⇒ Random
}

500 Internal Server Error 7.87 ms 482 B 1 Minute Ago ▾

Preview ▾ Header 7 Cookie Timeline

1 {
2 "name": "SequelizeValidationError",
3 "errors": [
4 {
5 "message": "O nome do produto deve ter entre 4 e 50 caracteres",
6 "type": "Validation error",
}]
}

\$.store.books[*].author

GET Read
POST Create
PUT Update
DEL Delete Usuario

Produto

GET Get All Produtos
GET Read Produto
POST Create Produto
PUT Update Produto
DEL Delete Produto

Status 500 causados por erros de validação no Sequelize

Insomnia Preferences – v2022.3.0

General

Data

Themes

Keyboard

Account

Plugins

Import format will be automatically detected. Supported formats include: Insomnia v1, Insomnia v2, Insomnia v3, Insomnia v4, Postman, Postman Environment, HAR 1.2, cURL, Swagger 2.0, OpenAPI 3.0, WSDL

Your format isn't supported? [Add Your Own](#).

Export Data ▼

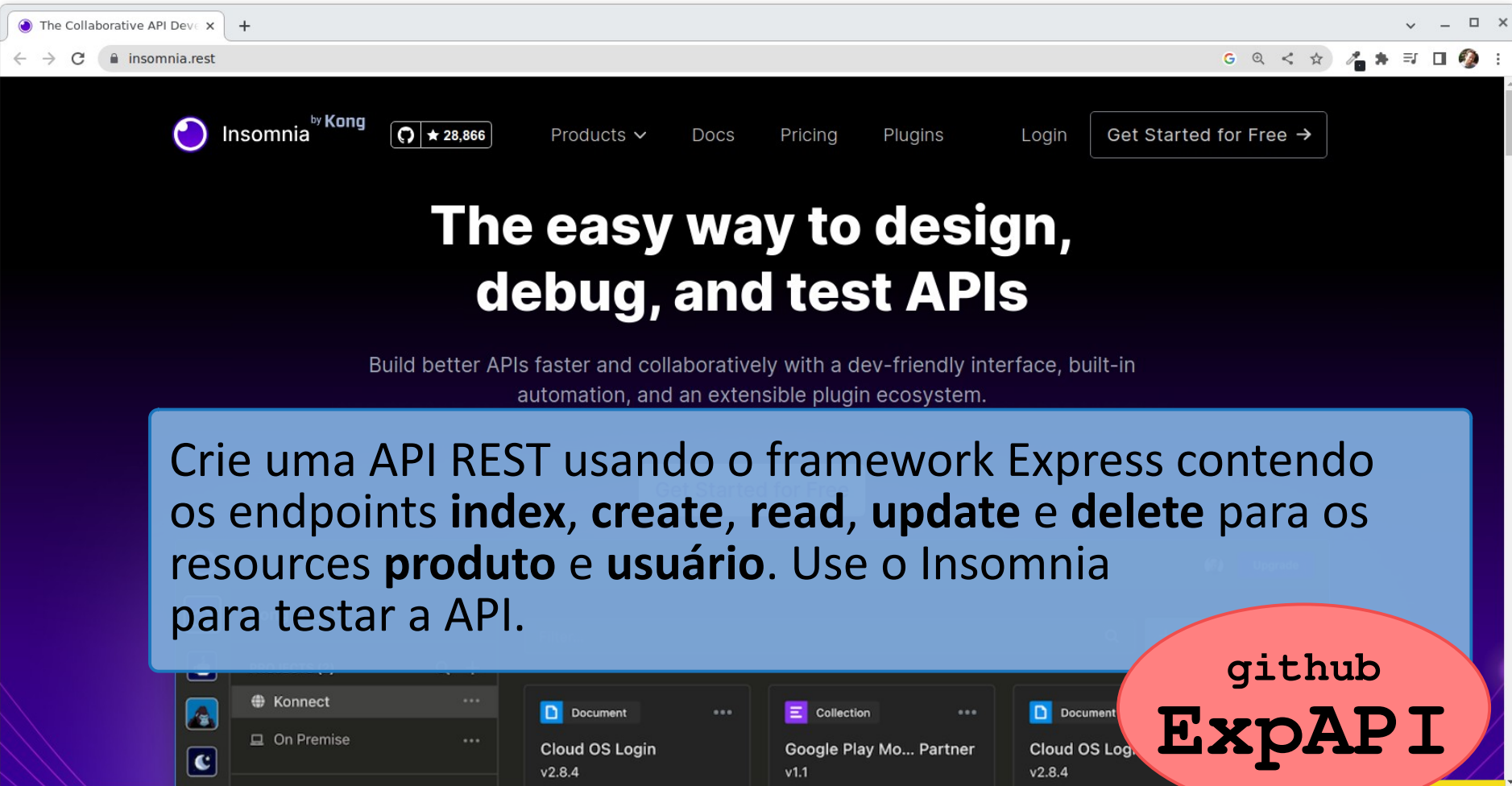
Import Data ▼

Create Run Button

** Tip: You can also paste Curl commands into the URL bar*

É possível exportar os endpoints gerados no Insomnia e salvá-los no repositório da aplicação (normalmente em **.insomina/insomia.json**). Use esse artifício para compartilhar os endpoints com os demais devs.

Exercício



Crie uma API REST usando o framework Express contendo os endpoints **index**, **create**, **read**, **update** e **delete** para os recursos **produto** e **usuário**. Use o Insomnia para testar a API.

github

ExpAPI

express **JS**