Introdução a Express

Prof. Dr. Anderson Rodrigues

Introdução ao Express

- **O que é o Express ?**
- **Por que usar o Express ?**
- **Configuração inicial de um projeto Express**
- **Criando um servidor básico com Express**
- **Roteamento em Express**

O que é o Express ?

- Express é uma popular biblioteca/framework web para Node.js
- Criada para fornecer uma maneira mais simples e flexível de ligar com as complexidades do desenvolvimento de APIs
- Fornece uma camada de abstração sobre o protocolo HTTP
- Ele fornece um conjunto robusto de recursos, incluindo roteamento flexível, middleware para processamento de requisições e respostas, suporte para manipulação de sessões, cookies e autenticação, entre outros.

Por que usar Express ?

- Simplicidade e minimalismo
- Flexibilidade
- Grande ecossistema de middleware
- Desempenho e escalabilidade
- Suporte da comunidade
- Robustez e maturidade
- Facilidade de teste
- Compatibilidade com middleware e ferramentas

Configuração inicial

- Instalação do Express
- Criação do arquivo principal
- Configuração do servidor Express
- Inicialização do servidor
- Cuidados necessários

Configuração inicial

- Criar a pasta que servirá como raiz
- Inicializar o projeto dentro da pasta raiz
  - `npm init -y`
- Instalar o express
  - `npm install -D typescript`
  - `npm install express`
  - `npm install -D @types/express`
  - `npm install -D @nestjs/schematics`
  - `npm install -D ts-node`
- Inicializar o compilador
  - `npx tsc --init`

Configuração do ambiente

- Ajustar ou criar o arquivo tsconfig.json na pasta raiz do projeto:

```
{
   "compilerOptions": {
       "target": "ES6",
       "module": "commonjs",
       "rootDir": "./",
       "typeRoots": ["./node_modules/@types"],
       "types": ["node", "express"],
       "sourceMap": true,
       "outDir": "dist/",
       "esModuleInterop": true,
       "forceConsistentCasingInFileNames": true,
       "strict": true,
       "skipLibCheck": true
    }
}
```

Configuração do ambiente

- Criar o arquivo nest-cli.json na pasta raiz do projeto:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src/",
  "entryFile" : "app.ts”
}
```

Configuração do ambiente

- Ajustar ou criar arquivo package.json na pasta raiz do projeto:

```
{
    "scripts": {
    "build": "tsc",
    "start": "ts-node src/app.ts",
    "start:debug": "ts-node --
inspect --require ts-node/register
src/app.ts",
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2"
  },
  "devDependencies": {
    "@nestjs/schematics":
"^10.1.1",
    "@types/express": "^4.17.21",
    "ts-node": "^10.9.2",
    "typescript": "^5.4.3"
  }
}
```

Criação do arquivo principal

- Criar a pasta que conterá o código fonte. Por convenção normalmente **src**
- Criar o arquivo app.ts TypeScript para o servidor. Este arquivo será usado para iniciar o servidor e deve ser criado dentro da pasta **src.**
- Dentro do arquivo app.ts importe a biblioteca Express e crie uma instância do aplicativo
  - `import express, { Request, Response } from 'express'`
  - `const app = express();`

Configuração do servidor

- Edite o arquivo **app.ts** de acordo com as próximas instruções.

- Defina em qual porta o servidor ‘ouvirá’ as chamadas
  - `const port = 3000;`

- Definir a rota raiz para o servidor

```
app.get('/',(req: Request, res: Response)=> {
        res.send('Bem vindo à minha API');
}
```

- Iniciar o Middleware para processar solicitações JSON
  - `app.use(express.json);`

Iniciar o servidor

- Dizer ao servidor para começar a ‘ouvir’ pela porta definida

```
app.listen(port, () => {
    console.log(`Servidor rodando em http://localhost:${port}`);
});
```

- Executar o servidor
  - `npm run build`
  - `npm run start`
- Testar o servidor
  - abra esta url no seu browser: `http://localhost:3000`

Cuidados necessários

- **Segurança:** Certifique-se de implementar práticas de segurança recomendadas para proteger sua API contra ataques comuns, como injeção de SQL, XSS, CSRF, entre outros.
- **Validação de Dados:** Valide todas as entradas de dados do cliente para evitar problemas como dados inválidos ou maliciosos.
- **Tratamento de Erros:** Implemente um middleware de tratamento de erros para lidar com erros de forma adequada e consistente em toda a sua API.
- **Testes:** Escreva testes automatizados para verificar o funcionamento correto da sua API e garantir que quaisquer alterações futuras não causem regressões.
- **Documentação:** Documente sua API de forma clara e abrangente, incluindo descrições de cada endpoint, parâmetros de requisição, respostas esperadas, entre outros. Isso ajudará os desenvolvedores que consumirão sua API a entenderem como usá-la corretamente.

Manipulando requisições e respostas no Express

Prof. Dr. Anderson Rodrigues

Senac FACULDADE

Métodos HTTP em Express

- GET:
  - O método GET é usado para **solicitar dados** de um recurso específico no servidor.
  - Ele é seguro e idempotente, o que significa que fazer várias solicitações GET para o mesmo recurso deve sempre retornar os mesmos resultados e não deve alterar o estado do servidor.
  - Em uma aplicação Express, você pode definir um manipulador de rota GET.

```
app.get('/api/get', (req, res) => {
    // Lógica para recuperar e enviar dados de
    usuários
});
```

Idempotente: *Que tem a propriedade de poder ser aplicado mais do que uma vez sem que o resultado se altere*

Métodos HTTP em Express

- POST:
  - O método POST é usado para **enviar dados** para o servidor para criar ou atualizar um recurso.
  - Ele é não idempotente, o que significa que fazer várias solicitações POST para criar o mesmo recurso resultará em diferentes resultados ou em múltiplas instâncias do recurso.
  - Em uma aplicação Express, você pode definir um manipulador de rota POST

```
app.post('/api/post', (req, res) => {
    // Lógica para criar um novo usuário com os dados enviados no corpo da requisição
    res.status(200).send("Recebido");
});
```

Idempotente: *Que tem a propriedade de poder ser aplicado mais do que uma vez sem que o resultado se altere*

Métodos HTTP em Express

- PUT:
  - O método PUT é usado para **enviar dados** para o servidor para atualizar um **recurso existente**.
  - Ele é idempotente, o que significa que fazer várias solicitações PUT para atualizar o mesmo recurso deve sempre resultar no mesmo estado do recurso.
  - Em uma aplicação Express, você pode definir um manipulador de rota PUT para lidar com solicitações PUT.

```
app.put('/usuarios/:id', (req, res) => {
    const userId = req.params.id;
    // Lógica para atualizar o usuário com o ID
    fornecido com os dados enviados no corpo da
    requisição
});
```

Idempotente: *Que tem a propriedade de poder ser aplicado mais do que uma vez sem que o resultado se altere*

Métodos HTTP em Express

- DELETE:
  - O método DELETE é usado para solicitar a exclusão de um recurso específico no servidor.
  - Ele é idempotente, o que significa que fazer várias solicitações DELETE para excluir o mesmo recurso deve sempre resultar no mesmo estado do servidor.
  - Em uma aplicação Express, você pode definir um manipulador de rota DELETE para lidar com solicitações DELETE.

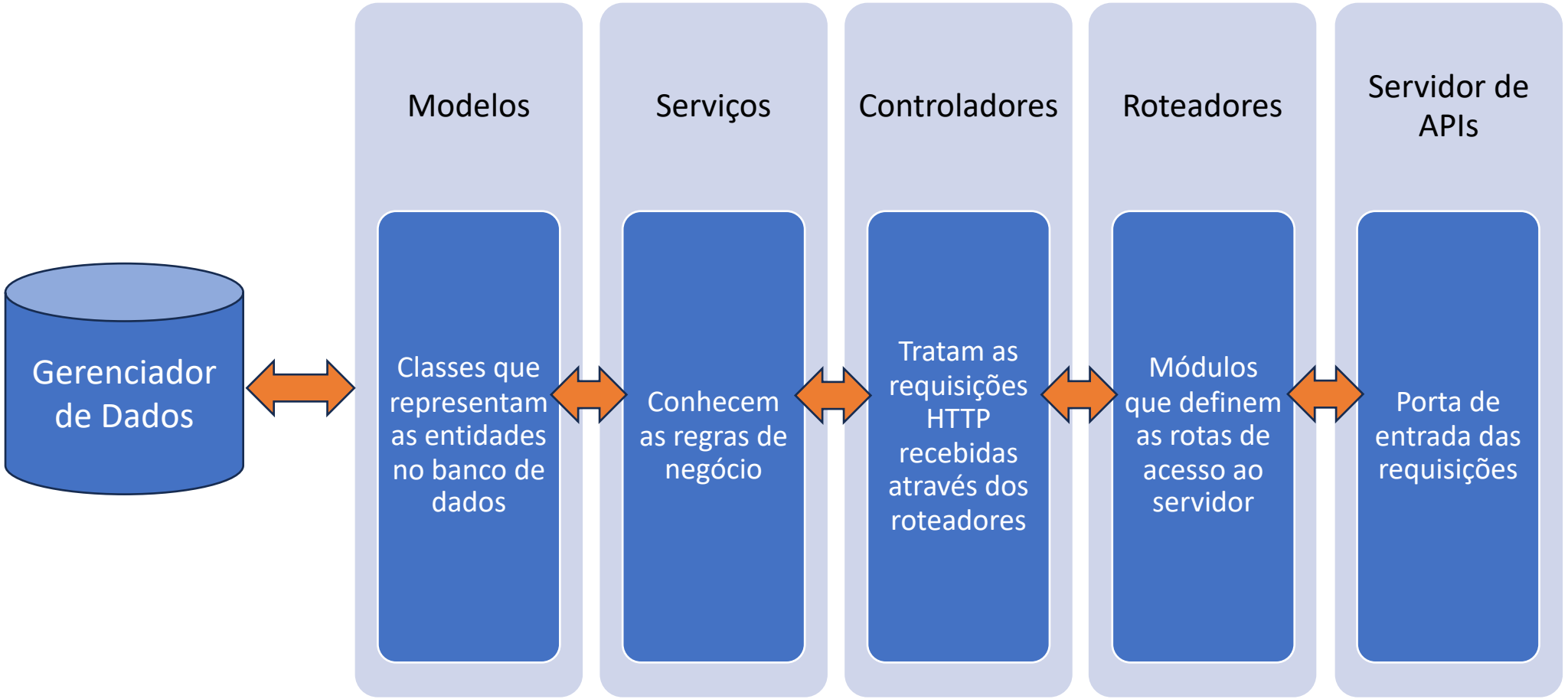
```
app.delete('/usuarios/:id', (req, res) => {
    const userId = req.params.id;
    // Lógica para excluir o usuário com o ID
    fornecido
});
```

Idempotente: *Que tem a propriedade de poder ser aplicado mais do que uma vez sem que o resultado se altere*

Definindo a Arquitetura do servidor de API

Prof. Dr. Anderson Rodrigues

Camadas do servidor de API



Camada de Modelo de Dados

- Importa os módulos necessários ao acesso ao gerenciador de banco de dados usado no projeto. Ex: Postgres, MySql, SQL Server, etc.
- Define as classes que representarão as entidades físicas no banco de dados.
- Garante a integridade e a corretude dos dados que serão **persistidos**.
- Trata os erros referentes às operações realizadas no banco de dados.

Camada de Serviços

- Implementa as funcionalidades referentes ao negócio da aplicação.
- Isola as funcionalidades do negócio das requisições HTTP
- Valida os dados de acordo com as regras do negócio. Ex: Validar CPF.
- Trata os erros referentes às regras do negócio.
- Implementa as funcionalidades referentes a segurança da aplicação como login, tipos de permissões, etc.

Camada de Controladores

- Recebe e trata as requisições recebidas através do módulo de roteadores
- Trata os erros referentes ao negócio da aplicação e às requisições HTTP recebidas.
- Garante que a corretude das respostas enviadas pela API

Camada de Roteadores

- Módulo que define as rotas de chamadas para as funcionalidades previstas na aplicação.
- Fornece os ***middlewares*** necessários para encaminhar as requisições HTTP.
- Trata os erros de comunicação com o servidor Express.

Camada do servidor

- Carrega as variáveis de ambiente.
- Realiza a inicialização do servidor Express.
- Realiza a configuração do servidor Express.
- Instancia as rotas de API.
- Realiza a conexão com o servidor de banco de dados.
- Recebe e envia dados solicitados através da API.
- Garante que todas as dependências da aplicação sejam atendidas.

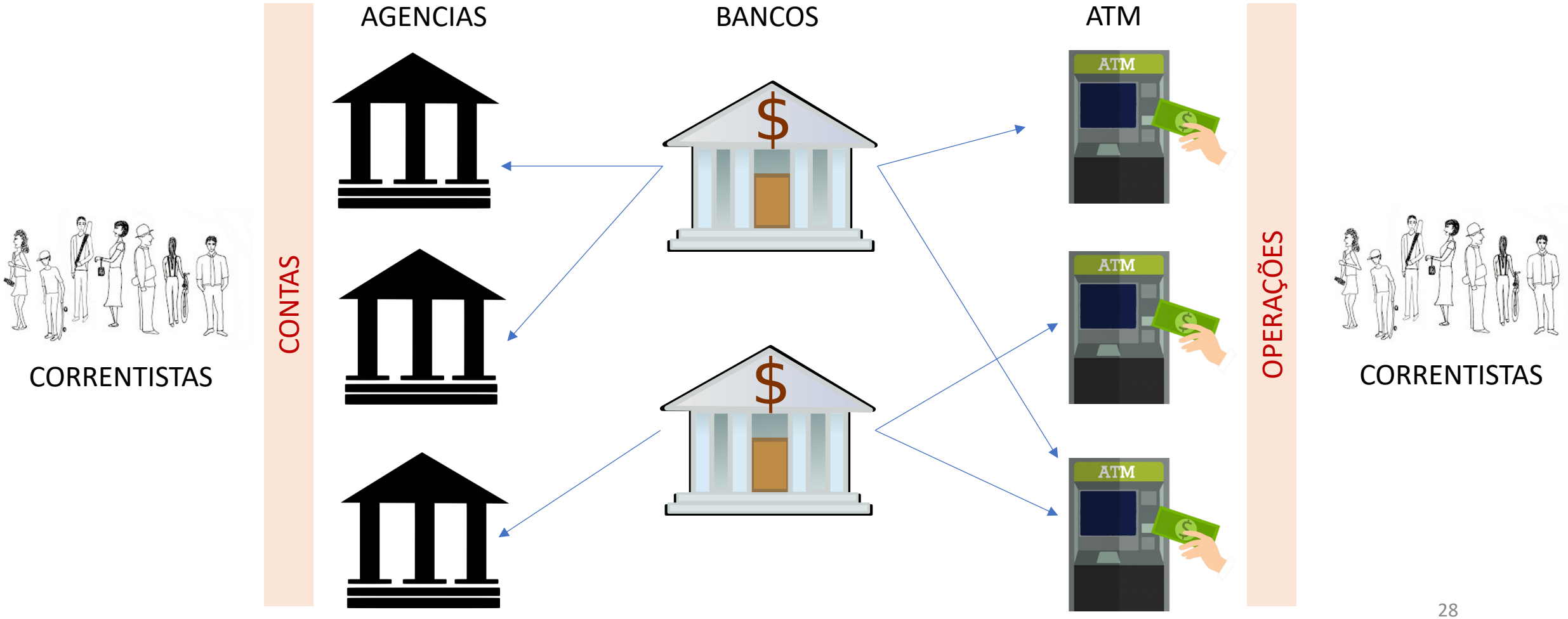
Vantagem do uso de camadas

- Facilita a manutenção da aplicação
- Facilita a realização de testes
- Facilita a escalabilidade da aplicação
- Facilita o trabalha em equipes
- Facilita a realização de mudanças em relação a tecnologia de persistência de dados.

Projeto Prático
Caixa Eletrônico

Prof. Dr. Anderson Rodrigues

Arquitetura do negócio



Objetivo geral da aplicação

- Permitir que correntistas realizem operações bancárias através de ATMs (caixas eletrônicos) descentralizados (fora das agências do banco).

Requisitos gerais

1. A aplicação deve permitir o cadastro de vários bancos
2. A aplicação deve permitir o cadastro de agências onde cada agência é vinculada a apenas um banco.
3. A aplicação deve permitir a abertura de contas bancárias para os correntistas onde apenas uma conta bancária poderá ser aberta por banco.
4. A aplicação deve permitir a realização de operações bancárias apenas através dos ATMs (caixas eletrônicos)
5. Um ATM (caixa eletrônico) pode ser credenciado por vários bancos;
6. Um correntista deve ser capaz de realizar operações de saque e depósito apenas em caixas eletrônicos credenciados pelo banco onde possui conta corrente.
7. As operações serão realizadas perante a informação do banco, da agência, da conta corrente e da senha do correntista.

Arquitetura da aplicação