WACAD016 – Fundamentos de Teste de Software - Aula 02

Júlia Luiza

jlslc@icomp.ufam.edu.br

Cronograma: Aula 02

Testes de integração no back-end com Jest e Supertest

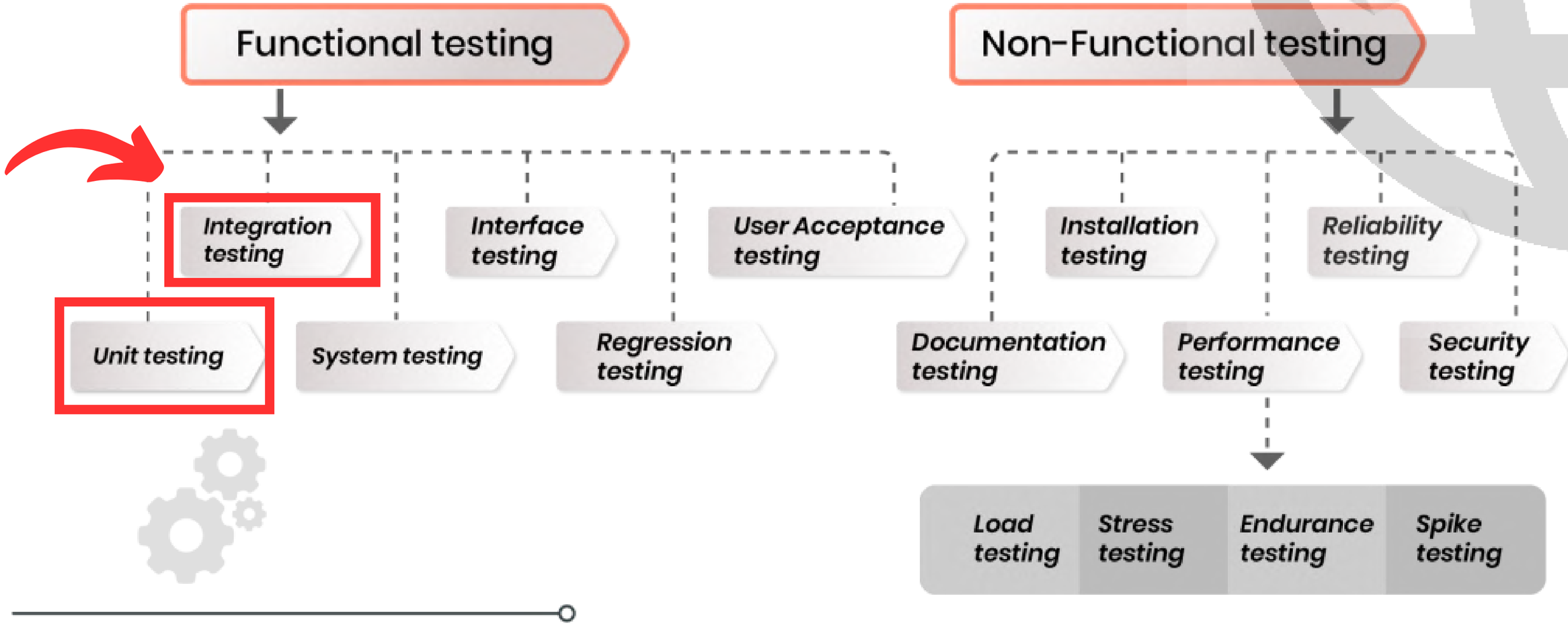
- Como escrever testes de integração;
- Exemplo de teste automatizado para requisições (camada serviço) de um projeto express + mysql, utilizando Supertest e banco de dados de teste;
- Boas práticas em testes de integração;

Testes com Jest no front-end com React

- Instalação do Jest em um projeto React;
- Como escrever testes unitários utilizando Jest e react-testing-library;
- Prática com testes unitários para o front-end;
- Boas práticas de testes unitários no front-end.

E os testes de integração na prática?

TYPES OF SOFTWARE TESTING



Testes de integração

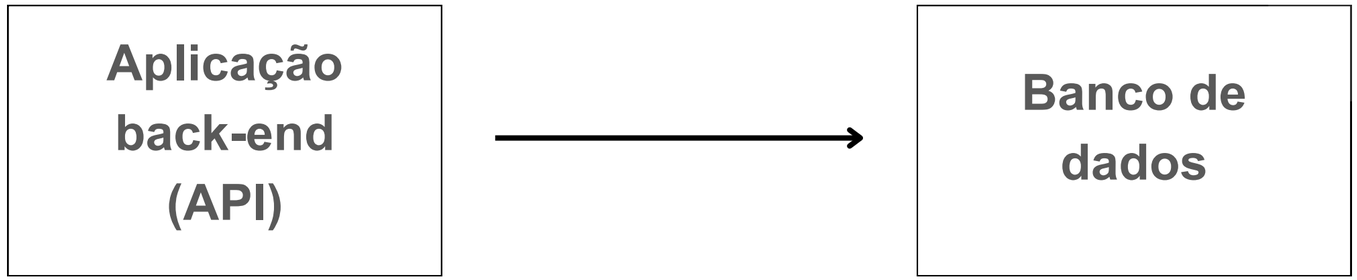
- Múltiplos componentes, com lógicas integradas e efeitos colaterais são testados;
- **Valida o funcionamento das unidades de software de forma integrada;**
- Complementares aos testes unitários;
- Exemplos: chamada de um módulo para outro, chamada externa, acesso ao banco de dados, acesso a API.

Testes de integração na prática

- Com relação a **sintaxe** do Jest, podemos escrever os testes de integração no back-end da mesma forma, com a mesma organização e utilizando os *matchers* disponíveis;
- **Mas,** considerando que agora iremos testar unidades integradas, precisaremos também de alguns recursos a mais, dependendo do que queremos testar:
  - No caso de testes de integração envolvendo comunicação com APIs, por exemplo, **precisaremos decidir entre mockar a API ou comunicar-se diretamente com ela**, sabendo que poderá haver instabilidadades;
  - Outro exemplo é caso quisermos testar a comunicação com o banco de dados da aplicação. Nesse caso, **precisamos configurar um banco de dados dedicado para os testes acessarem e utilizarem**, pois não queremos criar registros "teste" no banco de dados original.

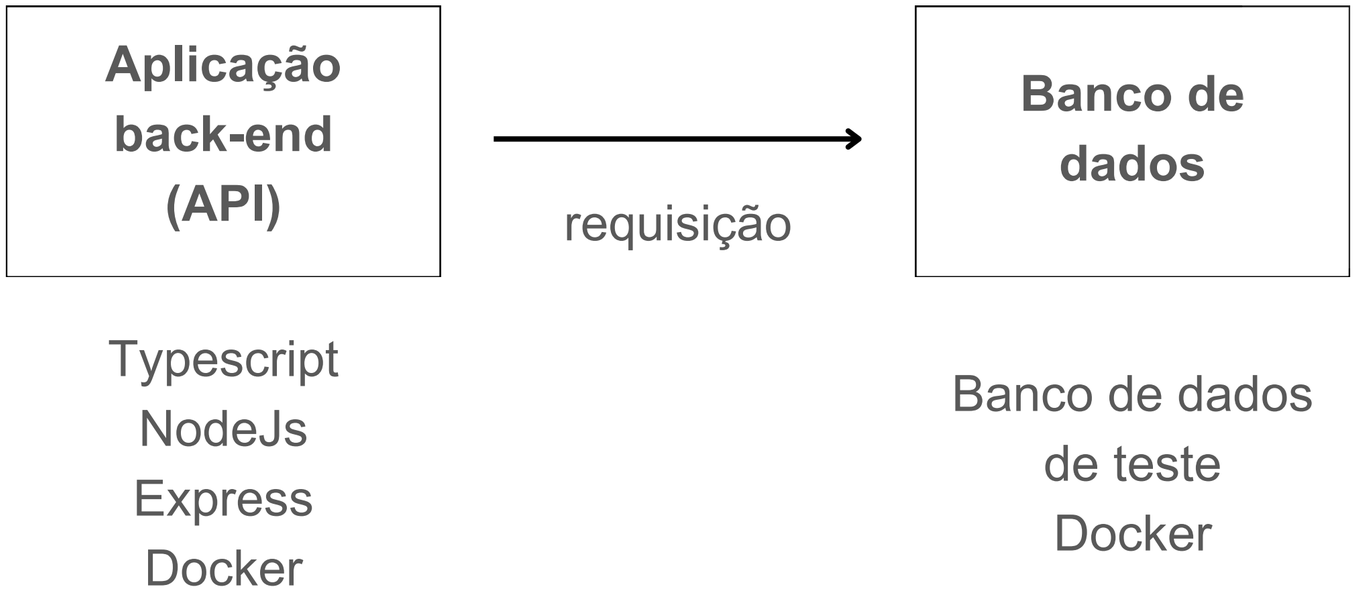
Testes de integração na prática

- Para o exemplo de cenário de loja virtual, optaremos por realizar testes de integração para testar a seguinte comunicação:



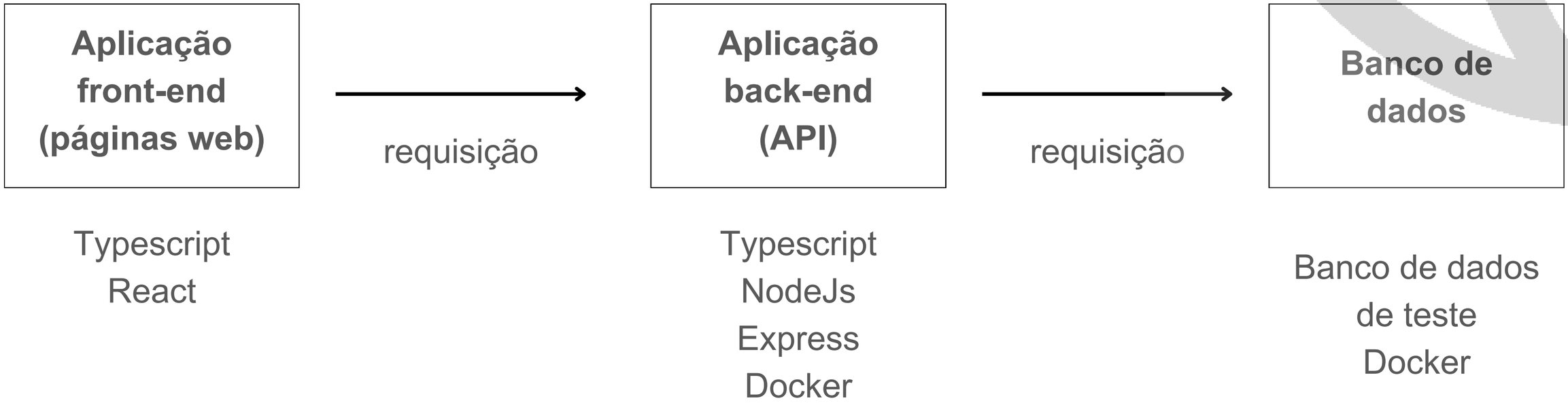
Testes de integração na prática

- Para o exemplo de cenário de loja virtual, optaremos por realizar testes de integração para testar a seguinte comunicação:



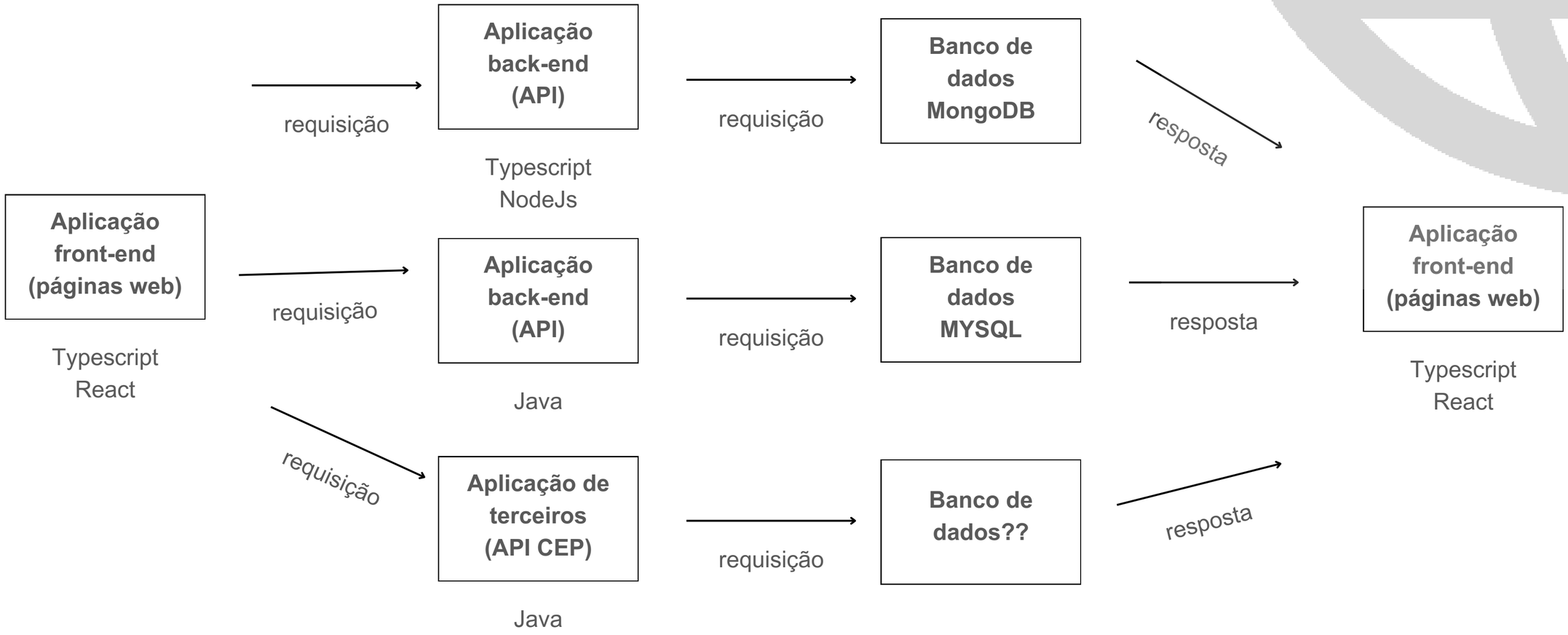
Testes de integração na prática

- Outro cenário mais completo seria:



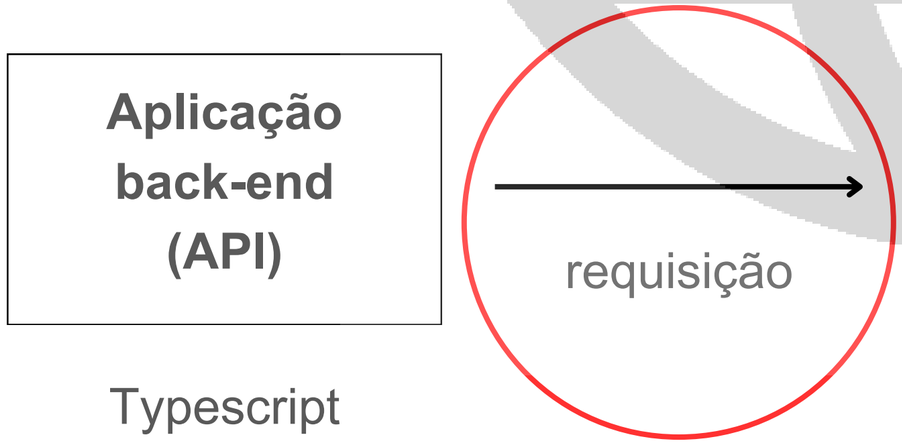
Testes de integração na prática

- Outro cenário mais completo ainda seria:



Testes de integração na prática: requisição

- **Considerando o nosso cenário de teste,** para resolver a parte de requisição utilizaremos a biblioteca auxiliar **Supertest;**
- **"A motivação deste módulo é fornecer uma abstração de alto nível para testar HTTP";**




```
npm install supertest --save-dev
```

Typescript
NodeJs
Express
Docker

https://www.npmjs.com/package/supertest

Testes de integração na prática: requisição

```
app.get('/user', function(req, res) {
  res.status(200).json({ name: 'john' });
});
```

```
const request = require('supertest');
```

```
describe('GET /user', function() {
  it('responds with json', function(done) {
    request(app)
      .get('/user')
      .set('Accept', 'application/json')
      .expect('Content-Type', /json/)
      .expect(200, done);
  });
});
```

https://www.npmjs.com/package/supertest

Configurando Jest e Supertest

Vamos praticar?

1. Clonar repositório https://github.com/julialuiza/LojaVirtualWA
2. Realizar o setup de acordo com o README

Jest

1. Instalar Jest na parte back-end, utilizando npm
    a. *npm install --save-dev jest*
2. Instalar outras dependências necessárias para o projeto TS:
    a. *npm install --save-dev babel-jest @babel/core @babel/preset-env*
    b. *npm install --save-dev @babel/preset-typescript*

Supertest

1. Intalar Supertest na parte back-end, utilizando npm
    a. *npm install supertest --save-dev*

Configurando Jest e Supertest

Vamos praticar?

Jest + Babel

1. Como dito anteriormente, dependendo das tecnologias do projeto, serão necessárias algumas configurações a mais para que o Jest funcione corretamente. No caso do nosso projeto, necessário instalar também:
   a. *npm install --save-dev @babel/plugin-proposal-decorators*
   b. *npm install --save-dev @babel/plugin-transform-flow-strip-types*
   c. *npm install --save-dev @babel/plugin-proposal-class-properties*
2. Depois, criar arquivo babel.config.js na raiz do projeto back-end para explicitar os presets e plugins necessários (ver imagem ao lado)

babel.config.js U

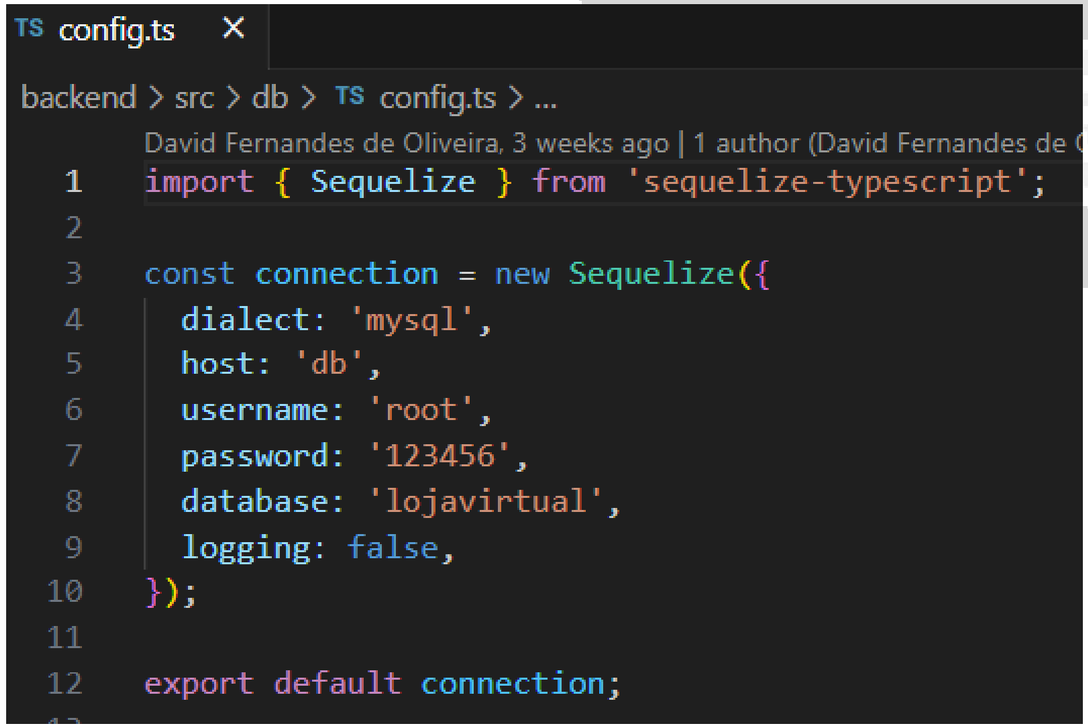
backend > babel.config.js > ...

```
1   module.exports = {
2     presets: [
3       ['@babel/preset-env', { targets: { node: 'current' } }],
4       '@babel/preset-typescript',
5     ],
6     plugins: [
7       ['@babel/plugin-proposal-decorators', { legacy: true }],
8       ['@babel/plugin-transform-flow-strip-types'],
9       ['@babel/plugin-proposal-class-properties', { loose: true }],
10    ],
11  };
12
```


```
module.exports = {
 presets: [
  ['@babel/preset-env', { targets: { node: 'current' } }],
  '@babel/preset-typescript',
 ],
 plugins: [
  ['@babel/plugin-proposal-decorators', { legacy: true }],
  ['@babel/plugin-transform-flow-strip-types'],
  ['@babel/plugin-proposal-class-properties', { loose: true }],
 ],
};
```

Testes de integração na prática: banco de dados de teste

- Como dito anteriormente também, quando executarmos nossos testes de integração, **não queremos que as alterações sejam feitas diretamente no banco de dados de produção;**
- Para resolver isso, geralmente criamos uma réplica do BD com propósito de servir apenas para testes;
- Em nosso projeto back-end, já possuímos esse banco de dados de teste, mas como padrão no arquivo *config.ts* estamos utilizando o BD de 'produção'.



Testes de integração na prática: banco de dados de teste

TS config.ts

backend > src > db > TS config.ts > ...

David Fernandes de Oliveira, 3 weeks ago | 1 author (David Fernandes de

```ts
import { Sequelize } from 'sequelize-typescript';

const connection = new Sequelize({
  dialect: 'mysql',
  host: 'db',
  username: 'root',
  password: '123456',
  database: 'lojavirtual',
  logging: false,
});

export default connection;
```

Configuração atual

.env

```
# Backend
PORT_BACK=3333

# Frontend
PORT_FRONT=3366

# Database Development
PORT_MYSQL=3320
MYSQL_DATABASE=lojavirtual
MYSQL_ROOT_PASSWORD=123456

# Database Test
PORT_MYSQL_TEST=3321
MYSQL_DATABASE_TEST=lojavirtual_test
MYSQL_ROOT_PASSWORD_TEST=123456

# PhpMyAdmin
PORT_PMA=8010
```

Configuração BD de teste

Testes de integração na prática: banco de dados de teste

Assim, precisamos resolver duas situações:

1. Alterar o banco de dados que será acessado durante a execução dos testes;
2. **Identificar que estamos executando os testes** para realizar a mudança do banco de dados
   a. Para essa parte, utilizaremos uma biblioteca auxiliar chamada **"cross-env"**
   b. E iremos adaptar o script de execução do jest para informar que estamos em ambiente de teste

```
npm install --save-dev cross-env
```

https://www.npmjs.com/package/cross-env

```
"scripts": {
  "start": "nodemon -e js,json,ts,yaml src/index.ts",
  "start:prod": "node build/index.js",
  "build": "npx tsc",
  "tsc:status": "tsc --diagnostics",
  "test": "cross-env NODE_ENV=test jest"        You, 1
},
```

```
npm test

> express@1.0.1 test
> cross-env NODE_ENV=test jest
```

Testes de integração na prática: banco de dados de teste

Agora que já conseguimos identificar que estamos em ambiente de teste, ainda precisamos:

1. Alterar o banco de dados que será acessado durante a execução dos testes
    a. Para isso, utilizaremos a variavel de ambiente **NODE_ENV** que acabamos de configurar;
    b. Conferindo a variável, alteramos as informações de acesso ao banco de dados no arquivo **config.ts** da seguinte forma:

```
/*      You, now • Uncommitted changes
const connection = new Sequelize({
  dialect: 'mysql',
  host: 'db',
  username: 'root',
  password: '123456',
  database: 'lojavirtual',
  logging: false,
});
 */

const connection = new Sequelize({
  dialect: 'mysql',
  host: process.env.NODE_ENV !== 'test' ? 'db' : 'localhost',
  port: process.env.NODE_ENV !== 'test' ? 3306 : 3321,
  username: 'root',
  password: '123456',
  database:
    process.env.NODE_ENV !== 'test' ? 'lojavirtual' : 'lojavirtual_test',
  logging: false,
});
```

Testes de integração na prática: banco de dados de teste

```
/*      You, now • Uncommitted changes
const connection = new Sequelize({
  dialect: 'mysql',
  host: 'db',
  username: 'root',
  password: '123456',
  database: 'lojavirtual',
  logging: false,
});
 */

const connection = new Sequelize({
  dialect: 'mysql',
  host: process.env.NODE_ENV !== 'test' ? 'db' : 'localhost',
  port: process.env.NODE_ENV !== 'test' ? 3306 : 3321,
  username: 'root',
  password: '123456',
  database:
    process.env.NODE_ENV !== 'test' ? 'lojavirtual' : 'lojavirtual_test',
  logging: false,
});
```

Configuração atual

.env

```
# Backend
PORT_BACK=3333

# Frontend
PORT_FRONT=3366

# Database Development
PORT_MYSQL=3320
MYSQL_DATABASE=lojavirtual
MYSQL_ROOT_PASSWORD=123456

# Database Test
PORT_MYSQL_TEST=3321
MYSQL_DATABASE_TEST=lojavirtual_test
MYSQL_ROOT_PASSWORD_TEST=123456

# PhpMyAdmin
PORT_PMA=8010
```

Configuração BD de teste

Testes de integração na prática

Beleza, e agora podemos começar a escrever o teste em si?

Só mais 2 coisas!

1. Para realizar a requisição na API de back-end por meio dos testes, **precisamos ter acesso a variável que guarda nossa instância do express();**
2. Por conta do Jest executar os testes de forma paralela, **precisamos indicar que caso seja ambiente de teste, a porta de execução da API não pode ser a mesma que a padrão,** para evitar conflito.

```
export const server = new Api();
```

backend\src\index.ts

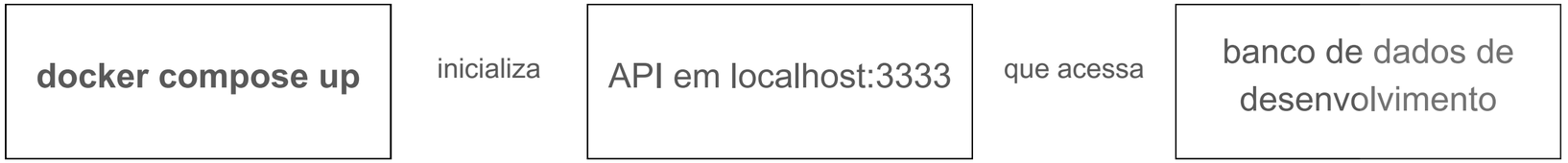
```
private async router() {
  this.server.use(router);

  try {
    if (process.env.NODE_ENV !== 'test') {
      this.server.listen(api.defaultPort);
    }
  } catch (err) {
    console.error(err);
    throw error;
  }
}
```

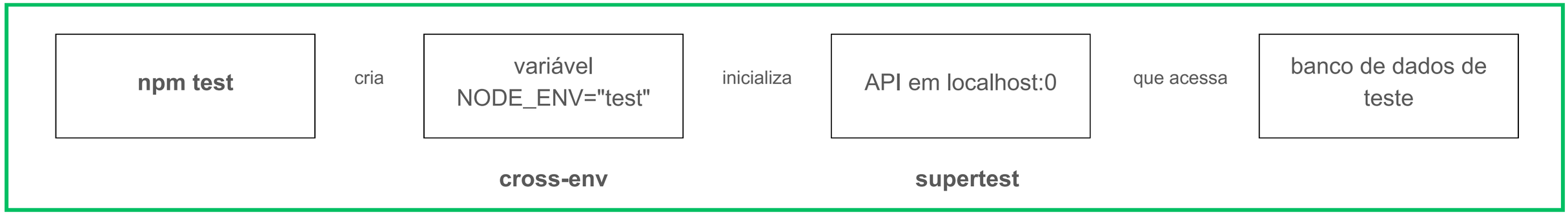
backend\src\server.ts

Testes de integração na prática

Beleza, agora vai

Antes, só para resumir, a nossa configuração de teste ficou da seguinte forma:





Testes de integração na prática

Beleza, agora vai mesmo

Finalmente um exemplo de teste de integração no código:

```
import request from 'supertest';
import { server } from '../../../index';
import connection from '../../../db/config';
import { TiposUsuarios } from '../../tipoUsuario/tipoUsuario.constants';

describe('Usuario Service', () => {
  beforeAll(async () => {
    await server.bootstrap();
  });

  it('should create new user', async () => {
    const randomEmailNumber = Math.random().toFixed(10);

    const res = await request(server.server)
      .post('/v1/usuario')
      .send({
        nome: 'Web teste',
        email: `web.teste${randomEmailNumber}@gmail.com`,
        tipoUsuarioId: '7edd25c6-c89e-4c06-ae50-c3c32d71b8ad',
        senha: '12345678',
      });

    expect(res.statusCode).toEqual(201);
    expect(res.body.nome).toEqual('Web teste');
    expect(res.body.email).toEqual(`web.teste${randomEmailNumber}@gmail.com`);
    expect(res.body.tipoUsuarioId).toEqual(TiposUsuarios.ADMIN);
  });

  afterAll(async () => {
    await connection.close();
  });
});
```

Testes de integração na prática

Quebrando em partes para entendermos melhor:

- No bloco **beforeAll(),** estamos de forma assíncrona inicializando o nosso servidor e conectando no banco de dados através da função bootstrap();
- No bloco **afterAll(),** estamos acessando a conexão aberta de banco de dados e a fechando, visto que não é mais necessário, pois os testes já terminaram de executar.

```
import request from 'supertest';
import { server } from '../../../index';
import connection from '../../../db/config';
import { TiposUsuarios } from '../../tipoUsuario/tipoUsuario.constants';

describe('Usuario Service', () => {
  beforeAll(async () => {
    await server.bootstrap();
  });
```

```
  afterAll(async () => {
    await connection.close();
  });
});
```

Testes de integração na prática

Quebrando em partes para entendermos melhor:

- No bloco **it(),** que contem o teste em si, estamos utilizando o **request do supertest** para realizar uma operação de **post** na API com caminho v1/usuario;
- Para que o usuário cadastrado seja sempre diferente, utilizamos uma ~~gambi~~ função auxiliar Math.random() para gerar emails aleatórios;
- Depois de esperar pela resposta (async/await), utilizamos o **expect** e **matchers** do Jest para verificar o resultado da request.

```
it('should create new user', async () => {
  const randomEmailNumber = Math.random().toFixed(10);

  const res = await request(server.server)
    .post('/v1/usuario')
    .send({
      nome: 'Web teste',
      email: `web.teste${randomEmailNumber}@gmail.com`,
      tipoUsuarioId: '7edd25c6-c89e-4c06-ae50-c3c32d71b8ad',
      senha: '12345678',
    });

  expect(res.statusCode).toEqual(201);
  expect(res.body.nome).toEqual('Web teste');
  expect(res.body.email).toEqual(`web.teste${randomEmailNumber}@gmail.com`);
  expect(res.body.tipoUsuarioId).toEqual(TiposUsuarios.ADMIN);
});
```

Testes de integração na prática

```
::ffff:127.0.0.1 - - [27/Jul/2023:17:36:43 +0000] "POST /v1/usuario HTTP/1.1" 2
  console.log
    conteudo de res.body: {
      id: '26a880e0-2ca4-11ee-a0cd-c1fb00d43e3f',
      nome: 'Web teste',
      email: 'web.teste0.6044248762@gmail.com',
      tipoUsuarioId: '7edd25c6-c89e-4c06-ae50-c3c32d71b8ad',
      updatedAt: '2023-07-27T17:36:42.992Z',
      createdAt: '2023-07-27T17:36:42.992Z'
    }

      at Object.log (src/resources/usuario/tests/usuario.service.test.js:23:13)

 PASS  src/resources/usuario/tests/usuario.service.test.js
  Usuario Service
    ✓ should create new user (216 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.129 s, estimated 6 s
```

```
describe('Usuario Service', () => {
  beforeAll(async () => {
    await server.bootstrap();
  });

  it('should create new user', async () => {
    const randomEmailNumber = Math.random().toFixed(10);
       You, 2 hours ago • chore: add integration test setup; add i
    const res = await request(server.server)
      .post('/v1/usuario')
      .send({
        nome: 'Web teste',
        email: `web.teste${randomEmailNumber}@gmail.com`,
        tipoUsuarioId: '7edd25c6-c89e-4c06-ae50-c3c32d71b8ad',
        senha: '12345678',
      });

    console.log('conteudo de res:', res);

    expect(res.statusCode).toEqual(201);
    expect(res.body.nome).toEqual('Web teste');
    expect(res.body.email).toEqual(`web.teste${randomEmailNumber}@gmail.com`);
    expect(res.body.tipoUsuarioId).toEqual(TiposUsuarios.ADMIN);
  });

  afterAll(async () => {
    await connection.close();
  });
});
```

Boas práticas em testes de integração

- **Nomear bem os testes:** é essencial nomear de forma semântica os blocos de testes e os testes em si; dessa forma, quando algum teste falhar será fácil identificar qual parte do código está com problemas.
- **Planejar** antes de iniciar a implementação;
- Diferente dos testes de unidade, os testes de integração tendem a demorar mais tempo na execução, portanto **deve-se focar no teste de fluxos críticos;**
- Não esquecer de **resetar os dados entre os testes,** para garantir um ambiente mais estável e evitar conflito entre os testes, causando falhas inesperadas.
- **Não ignore os testes.**

Cronograma: Aula 02

Testes de integração no back-end com Jest e Supertest

- Como escrever testes de integração;
- Exemplo de teste automatizado para requisições (camada serviço) de um projeto express + mysql, utilizando Supertest e banco de dados de teste;
- Boas práticas em testes de integração;

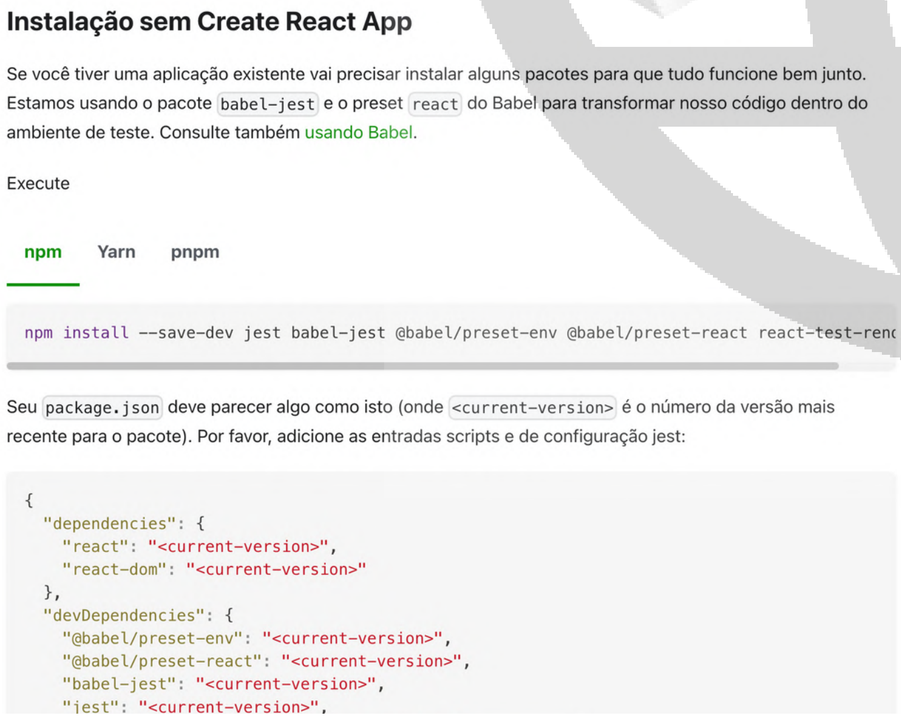
Testes com Jest no front-end com React

- Instalação do Jest em um projeto React;
- Como escrever testes unitários utilizando Jest e react-testing-library;
- Prática com testes unitários para o front-end;
- Boas práticas de testes unitários no front-end.

E o front-end nessa história?

Instalando Jest no front-end com React

- No caso do front-end com React, criá-lo com ***npx create-react-app my-app*** já faz com que o Jest seja incluído por padrão no projeto;
- Caso contrário, será necessário instalar e configurar manualmente o Jest:
  - *npm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-renderer*
  - Configurar arquivo babel.config.js.



Instalação sem Create React App

Se você tiver uma aplicação existente vai precisar instalar alguns pacotes para que tudo funcione bem junto. Estamos usando o pacote `babel-jest` e o preset `react` do Babel para transformar nosso código dentro do ambiente de teste. Consulte também usando Babel.

Execute

npm Yarn pnpm

```
npm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-rend
```

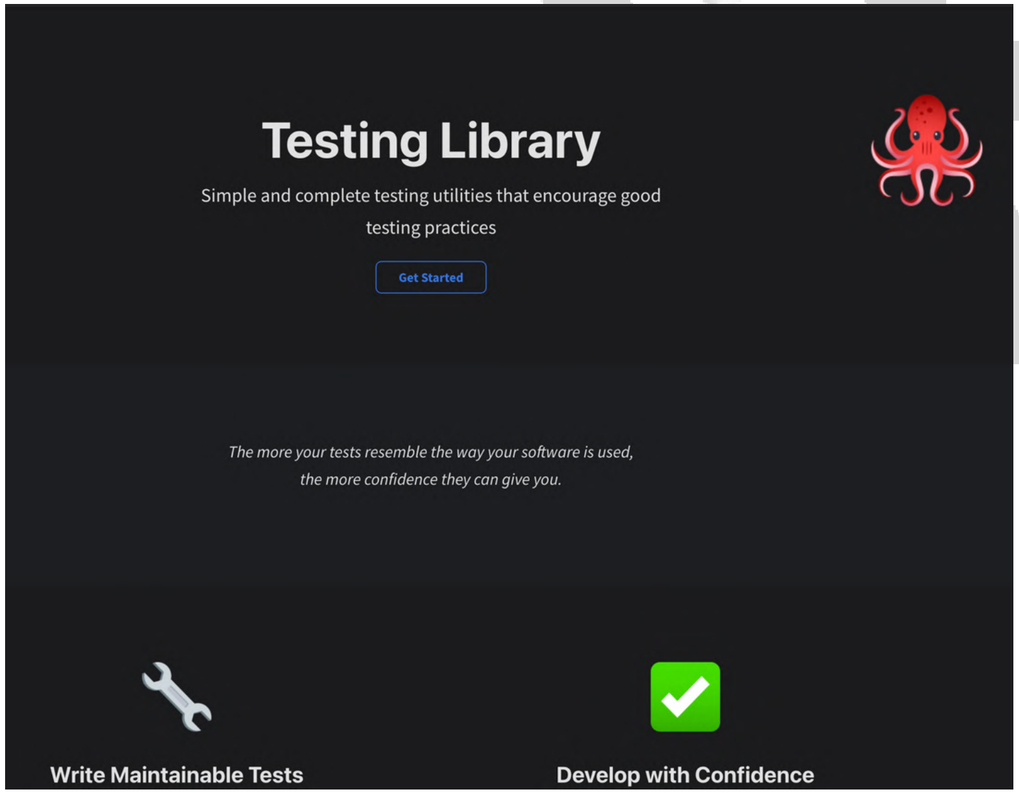
Seu `package.json` deve parecer algo como isto (onde `<current-version>` é o número da versão mais recente para o pacote). Por favor, adicione as entradas scripts e de configuração jest:

```
{
  "dependencies": {
    "react": "<current-version>",
    "react-dom": "<current-version>"
  },
  "devDependencies": {
    "@babel/preset-env": "<current-version>",
    "@babel/preset-react": "<current-version>",
    "babel-jest": "<current-version>",
    "jest": "<current-version>",
```

https://jestjs.io/pt-BR/docs/tutorial-react

Jest e React testing library para testes unitários

- Diferentemente do back-end, **para o front-end o Jest não faz tanto sentido sozinho**, pois os *matchers*, *expect* e demais recursos na maioria das vezes não são o suficiente para testar de fato os componentes front-end ;
- E é aí que entra a biblioteca **React Testing Library,** no caso de um projeto front com React**:**
  - **"Simple and complete testing utilities that encourage good testing practices"**
  - Como padrão, também vem embutido em aplicações criadas com *create-react-app;*
  - Se necessário, para instalar:
    - *npm install --save-dev @testing-library/react*



https://testing-library.com/

Jest e React testing library para testes unitários

Para diferenciar:

- **Jest:** um executor de teste que encontra testes, executa os testes e determina se os testes passaram ou falharam. Além disso, oferece **funções para suítes de teste, casos de teste e asserções.**
- **React Testing Library:** fornece **DOMs virtuais para testar componentes React.** Sempre que executamos testes sem um navegador da Web, devemos ter um DOM virtual para renderizar o aplicativo, interagir com os elementos e observar se o DOM virtual se comporta como deveria (como por exemplo, alterar a largura de um div em um clique de botão).

https://blog.logrocket.com/testing-react-apps-jest-react-testing-library/

render a component

```
import { render } from '@testing-library/react'

const result = render(<MyComponent />)
```

search the DOM

```
import { screen, render } from '@testing-library/react'

render(
  <label>
    Remember Me <input type="checkbox" />
  </label>,
)

const checkboxInput = screen.getByRole('checkbox', {
  name: /remember me/i,
})
```

interact with element

```
import userEvent from '@testing-library/user-event'

// userEvent simulates advanced browser interactions like
// clicks, type, uploads, tabbing etc
// Click on a button

userEvent.click(screen.getByRole('button'))

// Types HelloWorld in a text field
userEvent.type(screen.getByRole('textbox'), 'Hello World')
```

screen

debug(element)	Pretty print the DOM
...queries	Functions to query the DOM

search variants (result)

getBy	Element or Error
getAllBy	Element[] or Error
queryBy	Element or null
queryAllBy	Element[] or []
findBy	Promise<Element> or Promise<rejection>
findAllBy	Promise<Element[]> or Promise<rejection>

search types (result)

Role	...
LabelText	<label for="element" />
PlaceholderText	<input placeholder="username" />
Text	About
DisplayValue	<input value="display value" />
AltText	
Title	or <title />
TestId	<input data-testid='username-input' />

text matches

```
render(<label>Remember Me <input type="checkbox" /></label>)

screen.getByRole('checkbox', {name: /remember me/i})     // ✅
screen.getByRole('checkbox', {name: 'remember me'})      // ❌
screen.getByRole('checkbox', {name: 'Remember Me'})      // ✅

// other queries accept text matches as well
// the text match argument can also be a function
screen.getByText((text, element) => {/* return true/false
*/})
```

wait for appearance

```
test('movie title appears', async () => {
  render(<Movie />)

  // the element isn't available yet, so wait for it:
  const movieTitle = await screen.findByText(
    /the lion king/i,
  )

  // the element is there but we want to wait for it
  // to be removed
  await waitForElementToBeRemoved(() =>
    screen.getByLabelText(/loading/i),
  )

  // we want to wait until an assertion passes
  await waitFor(() =>
    expect(mockFn).toHaveBeenCalledWith('some arg'),
  )
})
```

render() options

hydrate	If true, will render with ReactDOM.hydrate
wrapper	React component which wraps the passed ui

React testing library

https://testing-library.com/docs/react-testing-library/cheatsheet

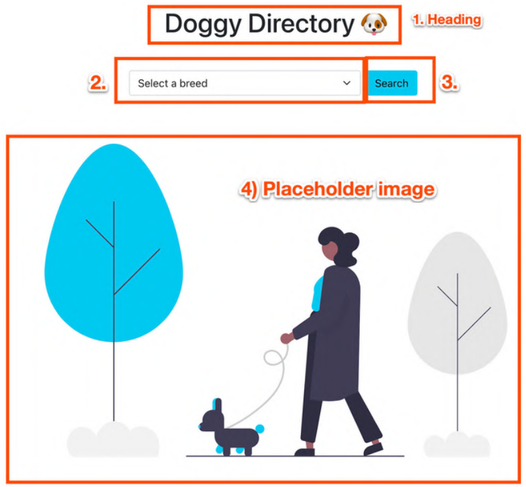
https://testing-playground.com/

Jest e RTL: Exemplo


```
<div className="d-flex justify-content-center flex-column text-center">
  <header>
    <h1 className="mt-4 mb-5">Doggy Directory 🐶</h1>
  </header>
  <main role="main">
    <div className="d-flex justify-content-center">
      <select
        className="form-select w-25"
        aria-label="Select a breed of dog to display results"
        value={selectedBreed}
        onChange={(event) => setSelectedBreed(event.target.value)}
      >
        <option value="" disabled>
          Select a breed
        </option>
        {breeds.map((breed) => (
          <option key={breed} value={breed}>
            {breed}
          </option>
        ))}
      </select>
      <button
        type="button"
        className="btn btn-info mx-2"
        disabled={!selectedBreed}
        onClick={searchByBreed}
      >
        Search
      </button>
    </div>
```

https://github.com/julialuiza/doggy-directory

Jest e RTL: Exemplo



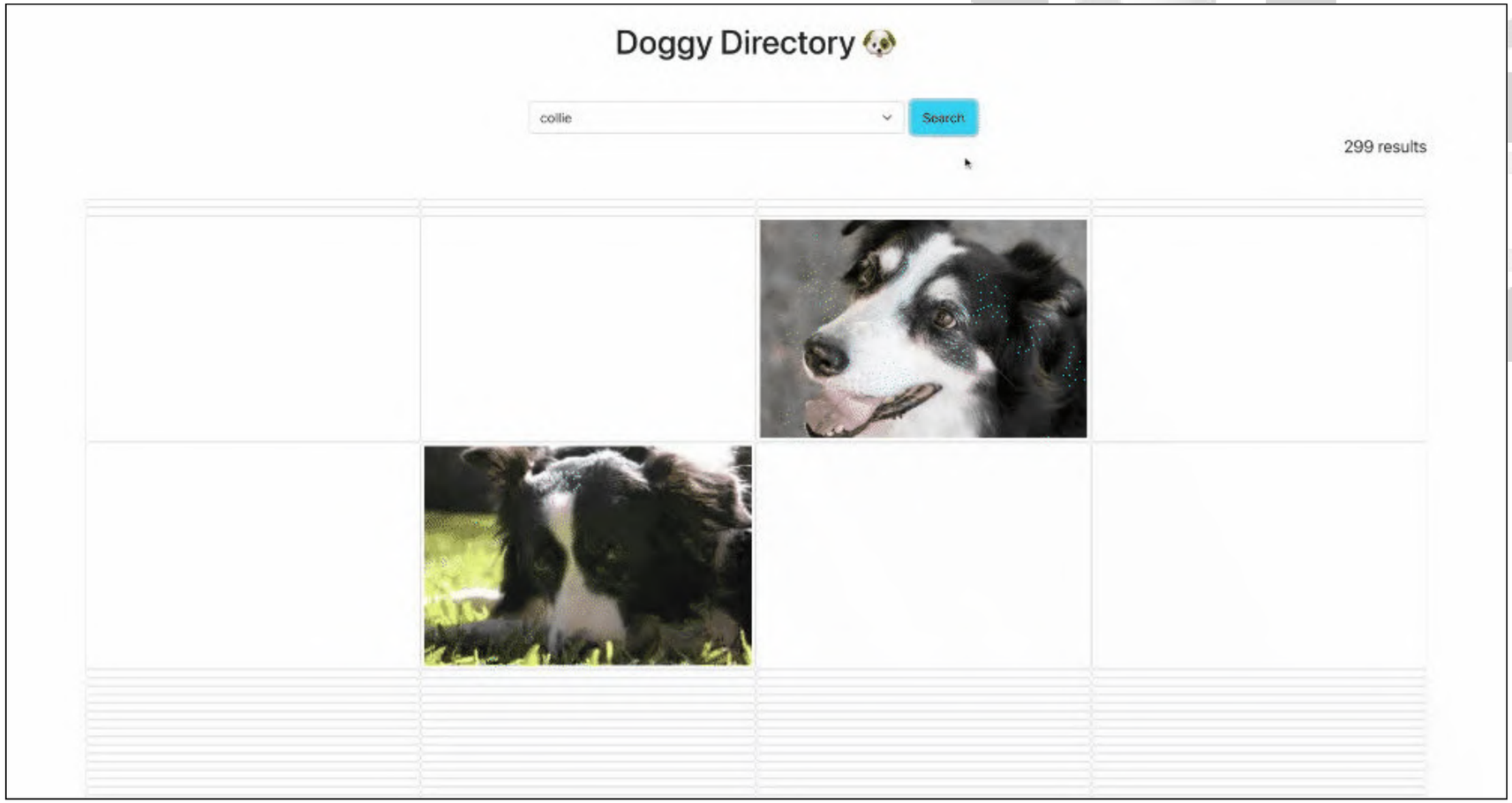

```
JS App.test.js > ...
import {
  render,
  screen,
} from "@testing-library/react";
import App from "./App";

test("renders the landing page", () => {
  render(<App />);

  expect(screen.getByRole("heading")).toHaveTextContent(/Doggy Directory/);
  expect(screen.getByRole("combobox")).toHaveDisplayValue("Select a breed");
  expect(screen.getByRole("button", { name: "Search" })).toBeDisabled();
  expect(screen.getByRole("img")).toBeInTheDocument();
});
```

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Jest e RTL: Exemplo com mock

No fluxo ao lado, temos os seguintes comportamentos para serem testados:

- Barra de busca listando as raças disponíveis;
- Botão de pesquisar ficando com estado 'habilitado' após selecionar uma raça;
- Estado de 'carregando';
- Listagem das imagens de cachorrinhos em si.



https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Jest e RTL: Exemplo com mock

Como vimos anteriormente, **ás vezes será necessário 'mockar' comportamentos que não são o foco dos nossos testes,** como por exemplo requisições à APIs:

- Esse cenário torna-se mais frequente ainda quando estamos tratando de testes para o front-end;
- No nosso exemplo, há vários comportamentos de front-end que queremos testar no fluxo de busca por raça;
- Por outro lado, esses comportamentos dependem de respostas de requisições feitas à uma API;
- Portanto, esse é um cenário ideal para o uso de mocks, como veremos a seguir.

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Jest e RTL: Exemplo com mock

Dando uma olhada no código, temos o seguinte cenário de requisições:

```
useEffect(() => {
  fetch("https://dog.ceo/api/breeds/list/all")
    .then((response) => {
      if (response.status === 200 || response.ok) {
        return response.json();
      } else {
        throw new Error(`HTTP error status: ${response.status}`);
      }
    })
    .then((json) => {
      setBreeds(Object.keys(json.message));
    });
}, []);
```

```
const searchByBreed = () => {
  setIsLoading(true);
  fetch(`https://dog.ceo/api/breed/${selectedBreed}/images`)
    .then((response) => {
      if (response.status === 200 || response.ok) {
        return response.json();
      } else {
        setIsLoading(false);
        throw new Error(`HTTP error status: ${response.status}`);
      }
    })
    .then((json) => {
      setIsLoading(false);
      setDogImages(json.message);
    });
};
```

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Em nosso exemplo, com o uso do **mockImplementation** do Jest, a implementação do **fetch()** foi sobreescrita, da seguinte forma:

```
beforeEach(() => {
  jest.spyOn(window, "fetch").mockImplementation(mockFetch);
});

afterEach(() => {
  jest.restoreAllMocks();
});
```

App.test.js

"jest.spyOn() é útil quando queremos testar uma função que já existe em um módulo, mas precisamos de mais controle sobre como ela é chamada ou seu comportamento."

https://www.dio.me/articles/diferenca-entre-fn-mock-e-spyon

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

mockFetch.js

```
const breedsListResponse = {
    message: {
        boxer: [],
        cattledog: [],
        dalmatian: [],
        husky: [],
    },
};

const dogImagesResponse = {
    message: [
        "https://images.dog.ceo/breeds/cattledog-australian/IMG_1042.jpg ",
        "https://images.dog.ceo/breeds/cattledog-australian/IMG_5177.jpg",
    ],
};

export default async function mockFetch(url) {
    switch (url) {
        case "https://dog.ceo/api/breeds/list/all": {
            return {
                ok: true,
                status: 200,
                json: async () => breedsListResponse,
            };
        }
        case "https://dog.ceo/api/breed/husky/images" :
        case "https://dog.ceo/api/breed/cattledog/images": {
            return {
                ok: true,
                status: 200,
                json: async () => dogImagesResponse,
            };
        }
        default: {
            throw new Error(`Unhandled request: ${url}`);
        }
    }
}
```

Assim, o teste dos comportamentos desejados pode ser feito da seguinte forma:

```
test("should be able to search and display dog image results", async () => {
  render(<App />);

  //Simulate selecting an option and verifying its value
  const select = screen.getByRole("combobox");
  expect(
    await screen.findByRole("option", { name: "cattledog" })
  ).toBeInTheDocument();
  userEvent.selectOptions(select, "cattledog");
  expect(select).toHaveValue("cattledog");

  //Initiate the search request
  const searchBtn = screen.getByRole("button", { name: "Search" });
  expect(searchBtn).not.toBeDisabled();
  userEvent.click(searchBtn);

  //Loading state displays and gets removed once results are displayed
  await waitForElementToBeRemoved(() => screen.queryByText(/Loading/i));

  //Verify image display and results count
  const dogImages = screen.getAllByRole("img");
  expect(dogImages).toHaveLength(2);
  expect(screen.getByText(/2 Results/i)).toBeInTheDocument();
  expect(dogImages[0]).toHaveAccessibleName("cattledog 1 of 2");
  expect(dogImages[1]).toHaveAccessibleName("cattledog 2 of 2");
});
```

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Assim, o teste dos comportamentos desejados pode ser feito da seguinte forma:

```
test("should be able to search and display dog image results", async () => {
  render(<App />);

  //Simulate selecting an option and verifying its value
  const select = screen.getByRole("combobox");
  expect(
    await screen.findByRole("option", { name: "cattledog" })
  ).toBeInTheDocument();
  userEvent.selectOptions(select, "cattledog");
  expect(select).toHaveValue("cattledog");

  //Initiate the search request
  const searchBtn = screen.getByRole("button", { name: "Search" });
  expect(searchBtn).not.toBeDisabled();
  userEvent.click(searchBtn);

  //Loading state displays and gets removed once results are displayed
  await waitForElementToBeRemoved(() => screen.queryByText(/Loading/i));

  //Verify image display and results count
  const dogImages = screen.getAllByRole("img");
  expect(dogImages).toHaveLength(2);
  expect(screen.getByText(/2 Results/i)).toBeInTheDocument();
  expect(dogImages[0]).toHaveAccessibleName("cattledog 1 of 2");
  expect(dogImages[1]).toHaveAccessibleName("cattledog 2 of 2");
});
```

"**user-event** permite descrever uma interação do usuário em vez de um evento concreto. Ele adiciona verificações de visibilidade e interatividade ao longo do caminho e manipula o DOM exatamente como faria uma interação do usuário no navegador.
É por isso que você deve usar **userEvent** para testar a interação com seus componentes."

https://testing-library.com/docs/user-event/intro/

https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library

Boas práticas em Jest e React Testing Library para testes unitários

- **Nomear bem os testes:** é essencial nomear de forma semântica os blocos de testes e os testes em si; dessa forma, quando algum teste falhar será fácil identificar qual parte do código está com problemas.
- Da mesma forma que testes unitários no back-end, **escrever testes rápidos, curtos e objetivos;**
- Ao decidir qual utilitário do RTL utilizar para o teste, **considerar o que representa mais próximo o comportamento do usuário ou que seja mais semântico;**
- Levar em consideração aspectos de **acessibilidade nos testes;**
- **Não ignore os testes.**

Testes com Jest e RTL: **Atividade extra**

Vamos praticar?

1. Clonar repositório https://github.com/julialuiza/doggy-directory
2. Alterar para branch ***tests-complete***
3. Realizar o setup de acordo com o README
4. Explorar a aplicação em localhost:3000
5. Adicionar novo teste unitário em *App.test.js,* para cobrir a nova funcionalidade de informação aleatória sobre cachorros, utilizando Mock.
6. Obter 100% dos testes passando.

Dicas:

- Para parte de mock do endpoint que traz as informações aleatórias, é possível reutilizar a estrutura já existente no projeto, construída em *src/mocks/mockFetch.js:*
  - Observe a requisição realizada para buscar as infos aleatórias, por meio do *devtools*, ou acesse a documentação da API: note qual a url solicitada, qual o retorno da API e quais partes do retorno estão sendo mostradas em tela para o usuário.

Testes com Jest e RTL: **Trabalho Prático 02**

Vamos praticar?

1. Clonar repositório https://github.com/julialuiza/AulaReactTurma2
2. Realizar o install das dependências do package.json que está dentro de /frontend, com npm install (dentro de /frontend)
3. Adicionar novos testes unitários para no mínimo dois (2) componentes do projeto, sendo que as opções de componentes alvos são:
    a. **ProductListGrid**, em src/components/ListaProdutosGrid/index.tsx;
    b. **ConfirmationModal**, em src/components/Modals/Confirmacao/index.tsx;
    c. **CustomTable**, em src/components/Tabela/index.tsx
    d. Ou qualquer outro de sua preferência, desde que não contenha lógica interligada a Redux, pois não abordamos testes com Redux na disciplina.
4. Obter 100% dos testes passando.

Obrigada!

Dúvidas?

- **Slack**
- **Email**: jlslc@icomp.ufam.edu.br