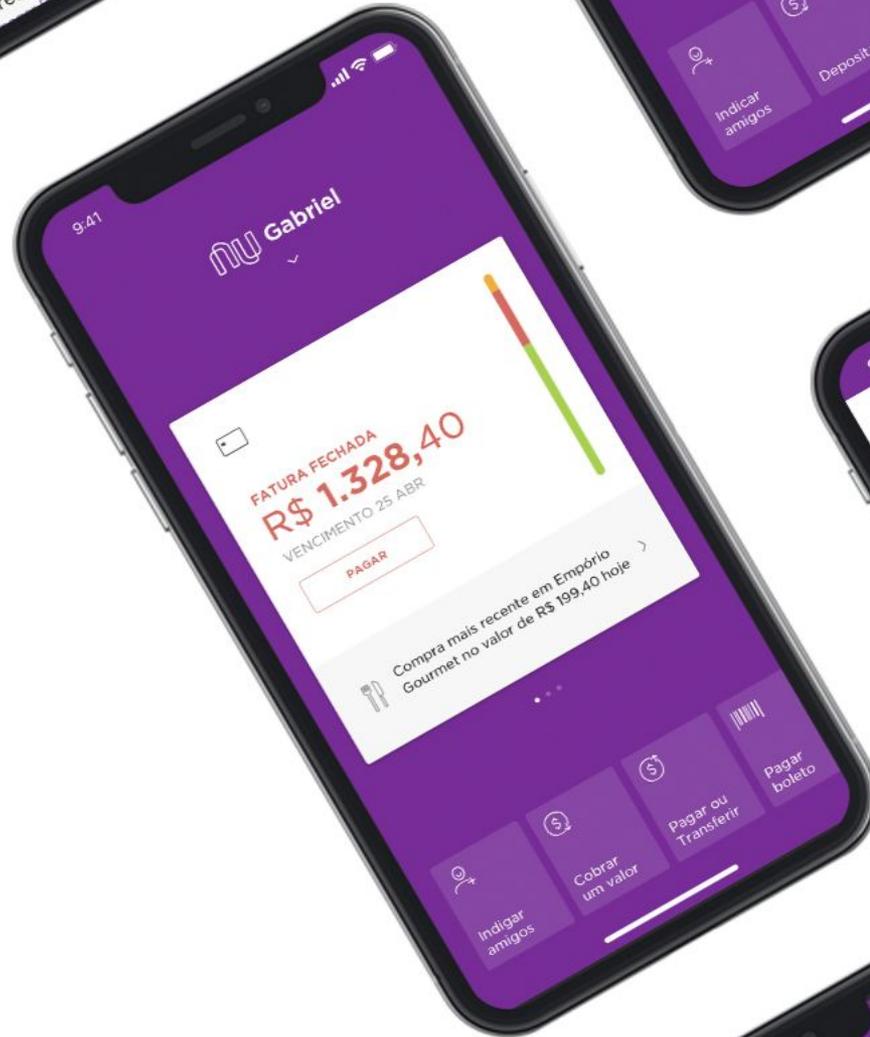


# Como testamos React Native no Nubank



# Quem somos?



**Newton Angelini**  
**@newtonbeck**

Engenheiro de  
Software

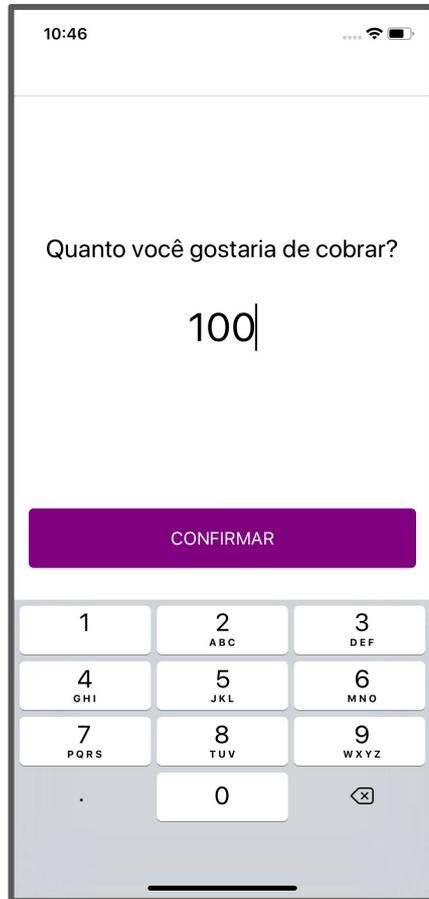


**Fellipe Chagas**  
**@chagasaway**

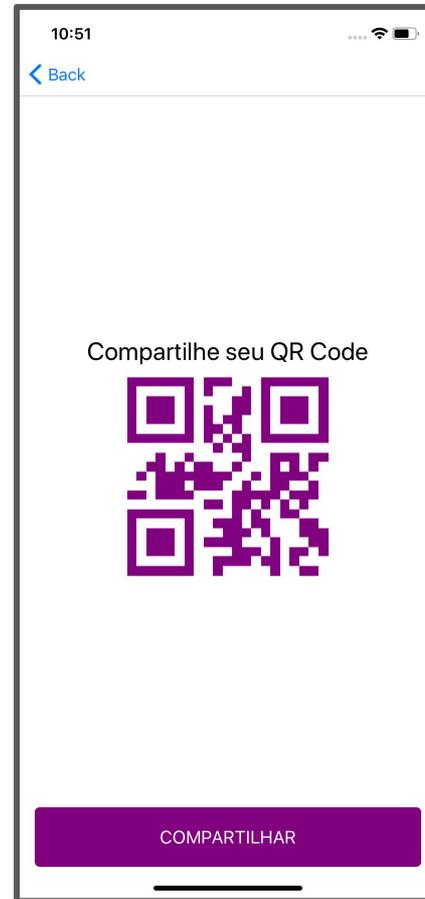
Engenheiro de  
Software

ny bank

# Exemplo



**AmountInput**



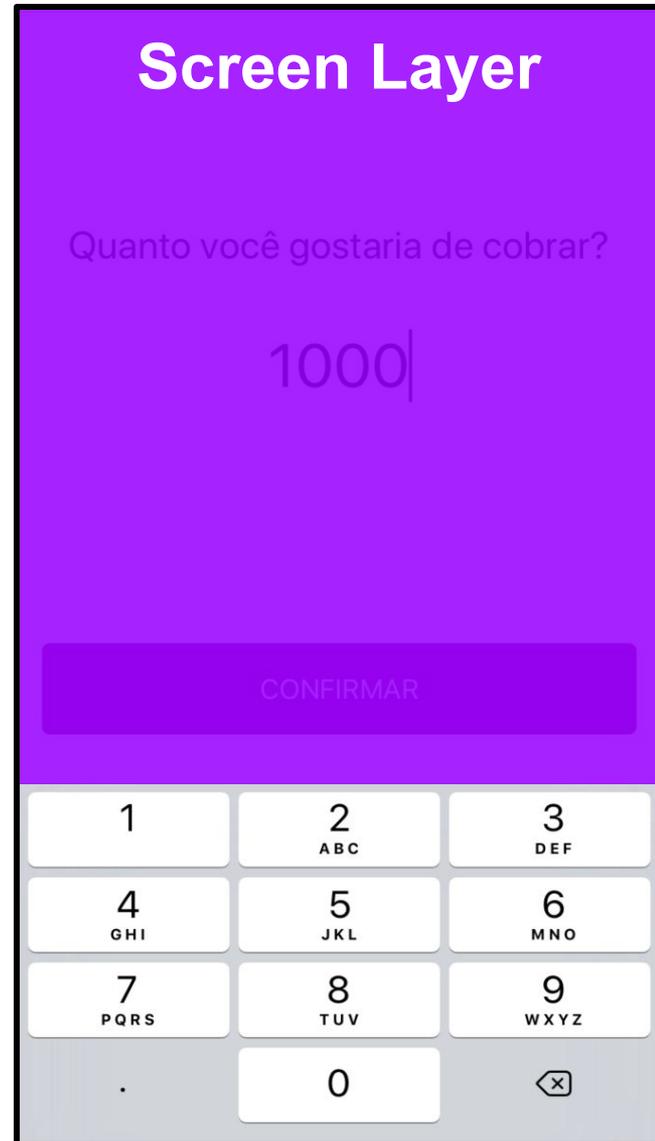
**QRCode**

Como estruturamos o código do nosso app?

# Camada de Screen

## Responsabilidades

- Navegação;
- Tratamento para devices específicos;





```
export class AmountInputScreen extends Component<NavigationInjectedProps> {
```

```
}
```



```
export class AmountInputScreen extends Component<NavigationInjectedProps> {
  private navigateBack = () => {
    this.props.navigation.pop();
    return true;
  }

  private navigateToQRCode = (requestMoney: RequestMoney) => {
    this.props.navigation.navigate(RequestMoneyScreens.QRCode, requestMoney);
  }

  private navigateToError = () => {
    this.props.navigation.navigate(RequestMoneyScreens.Error);
  }

  // ...
}
```

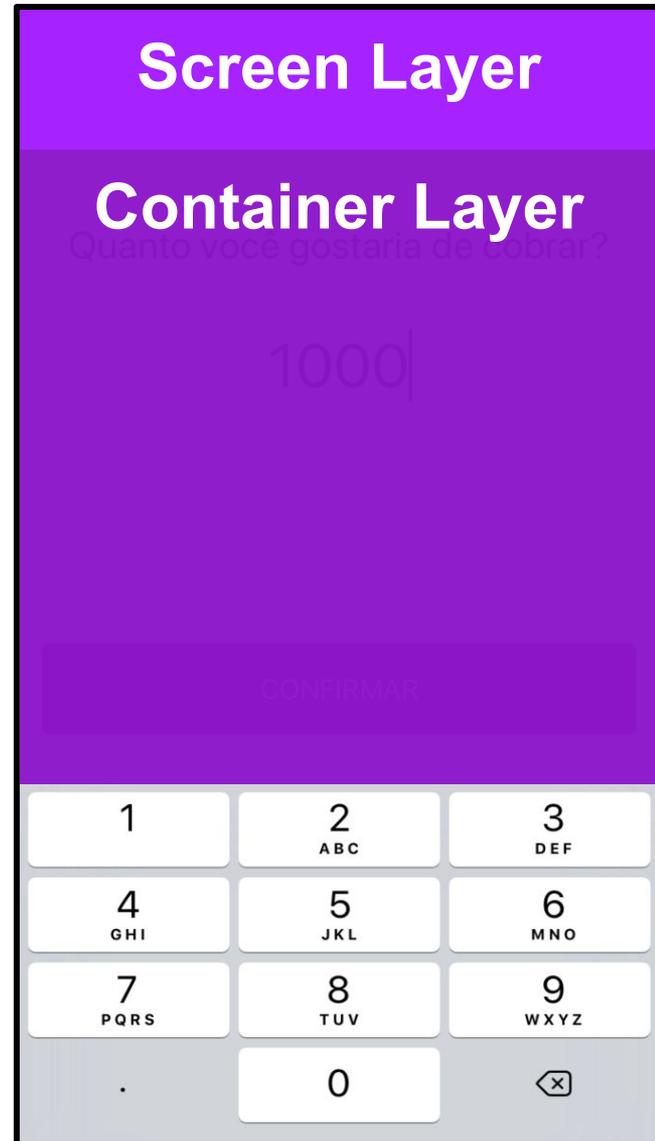


```
export class AmountInputScreen extends Component<NavigationInjectedProps> {  
  // ...  
  
  public render() {  
    return (  
      <BaseScreen testID='AmountInputScreen'>  
        <AmountInputContainer  
          navigateBack={this.navigateBack}  
          navigateToQRCode={this.navigateToQRCode}  
          navigateToError={this.navigateToError}  
        />  
      </BaseScreen>  
    );  
  }  
}
```

# Camada de Container

## Responsabilidades

- Regras de Negócio;
- Busca de dados na API (REST, GraphQL, etc);
- Chamar as funções de navegação quando necessário;





```
export interface AmountInputContainerProps {
  navigateBack: () => boolean;
  navigateToQRCode: (requestMoney: RequestMoney) => void;
  navigateToError: () => void;
}

interface AmountInputContainerState {
  amount: number;
}

export class AmountInputContainer extends Component<AmountInputContainerProps, AmountInputContainerState> {
  public state = {
    amount: 0,
  };

  // ...
}
```



```
// ...
export class AmountInputContainer extends Component<AmountInputContainerProps, AmountInputContainerState> {
  // ...
  private handleAmountChange = (amount: number) => {
    this.setState({ amount });
  }

  private handleSubmit = async () => {
    try {
      const { amount } = this.state;
      const requestMoney = await createRequestMoney(amount);
      this.props.navigateToQRCode(requestMoney);
    } catch {
      this.props.navigateToError();
    }
  }

  // ...
}
```

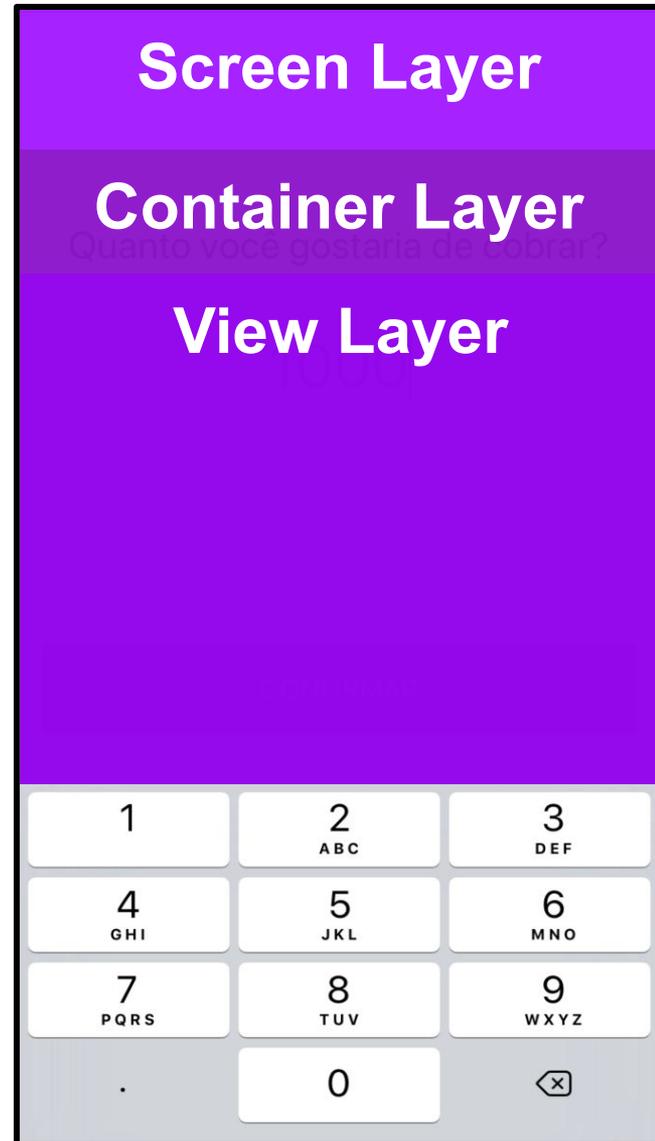


```
// ...
export class AmountInputContainer extends Component<AmountInputContainerProps, AmountInputContainerState> {
  // ...
  public render() {
    return (
      <AmountInputView
        amount={this.state.amount}
        onAmountChange={this.handleAmountChange}
        onSubmit={this.handleSubmit}
      />
    );
  }
}
```

# Camada de View

## Responsabilidades

- Apresentar os dados na tela utilizando os componentes do nosso Design System;
- Capturar interações dos usuários;





```
export interface AmountInputViewProps {
  onSubmit: () => void;
  amount: number;
  onAmountChange: (amount: number) => void;
}
```

```
export const AmountInputView: React.SFC<AmountInputViewProps> = ({ onSubmit, amount, onAmountChange }) => (
  <Fragment>
    <SpaceFiller />
    <Title>Quanto você gostaria de cobrar?</Title>
    <MoneyInput testID='amountInput' amount={amount} onAmountChange={onAmountChange} />
    <SpaceFiller />
    <Button testID='confirmButton' onPress={onSubmit} label='CONFIRMAR' />
  </Fragment>
);
```

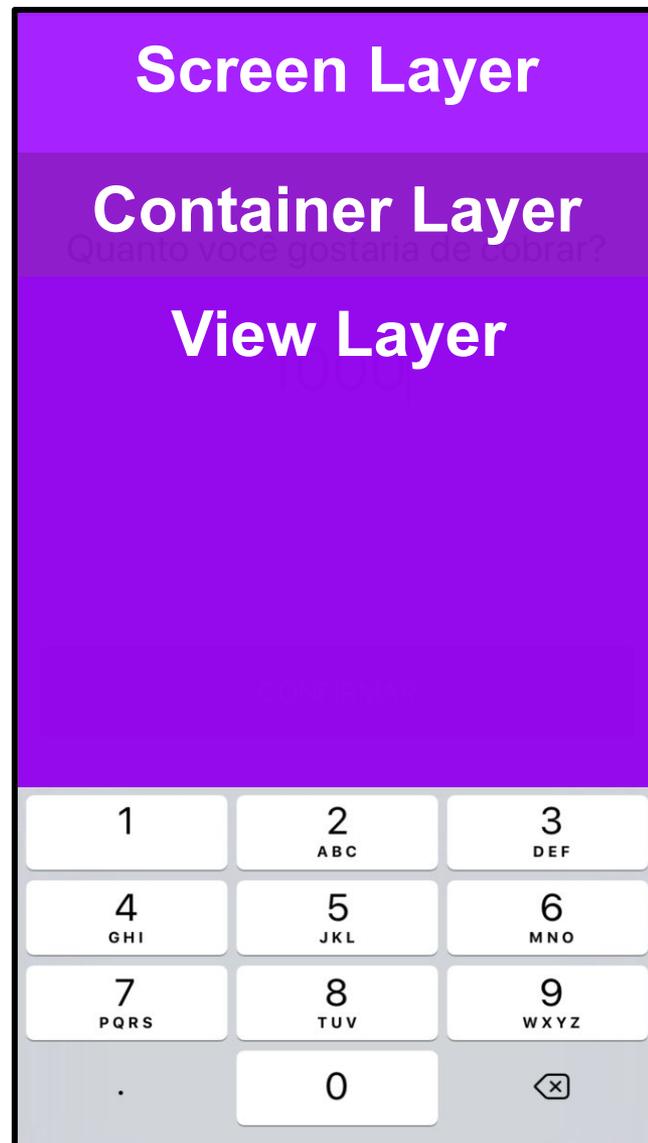
Como testamos o  
nosso app?

# Testes de unidade

São testes que garantem o funcionamento de cada unidade do app isoladamente. Para isolar cada unidade do app nós criamos dublês que interagem com a unidade sendo testada.

# Testes de unidade na camada de view

Os testes garantem que o componente visual existe, possui propriedades relevantes e consegue responder às interações do usuário





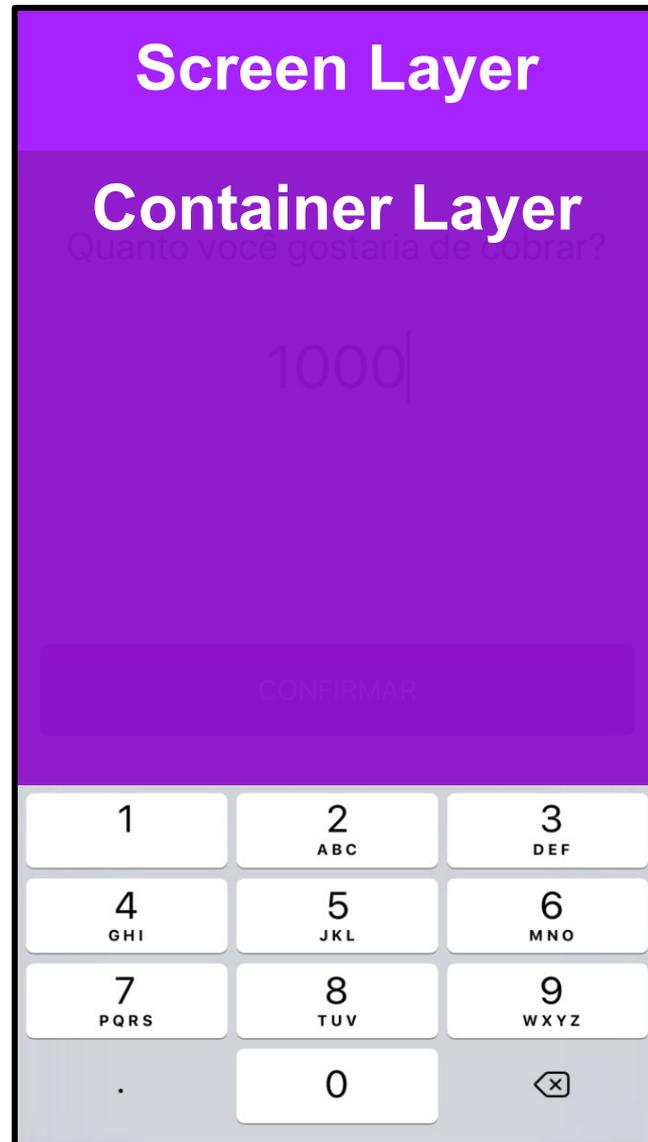
```
describe('AmountInput View', () => {
  it('should display a title, a input and a submit button', () => {
    // given
    const props = {
      amount: 20,
      onSubmit: jest.fn(),
      onAmountChange: jest.fn(),
    };

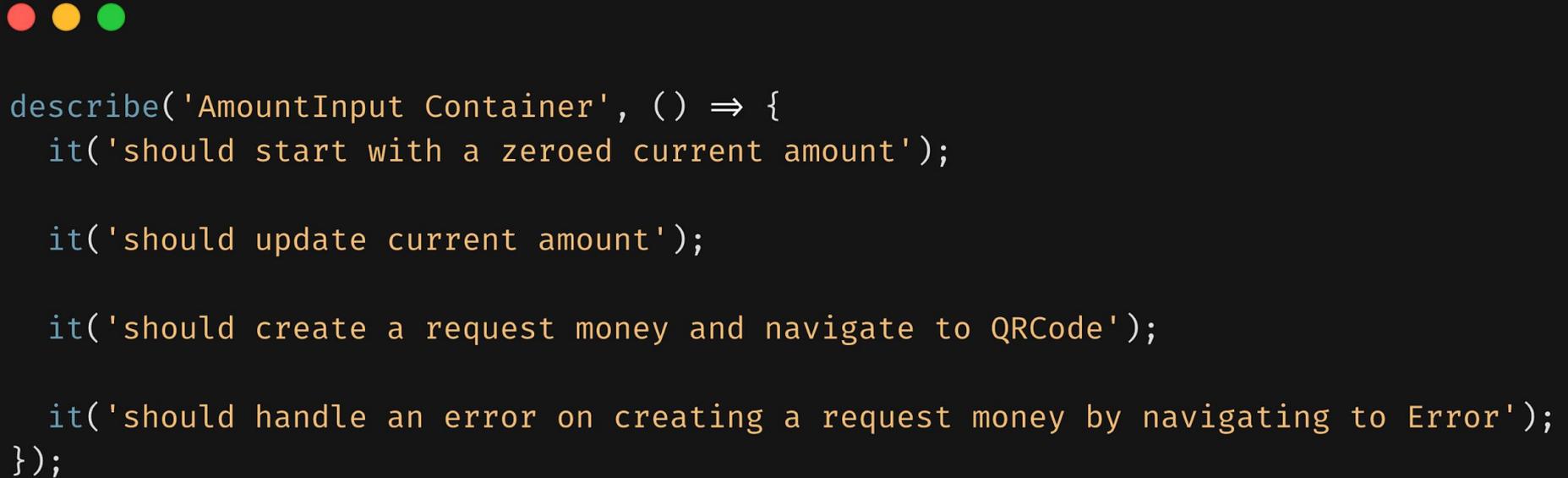
    // when
    const { title, input, button } = render(props);

    // then
    expect(title.props.children).toBe('Quanto você gostaria de cobrar?');
    expect(input.props.onAmountChange).toBe(props.onAmountChange);
    expect(button.props.onPress).toBe(props.onSubmit);
    expect(button.props.label).toBe('CONFIRMAR');
  });
});
```

# Testes de unidade na camada de container

Os testes garantem que o container executa as regras de negócio disparadas pela camada de visualização e faz a navegação para outras telas quando necessário





```
describe('AmountInput Container', () => {  
  it('should start with a zeroed current amount');  
  
  it('should update current amount');  
  
  it('should create a request money and navigate to QRCode');  
  
  it('should handle an error on creating a request money by navigating to Error');  
});
```



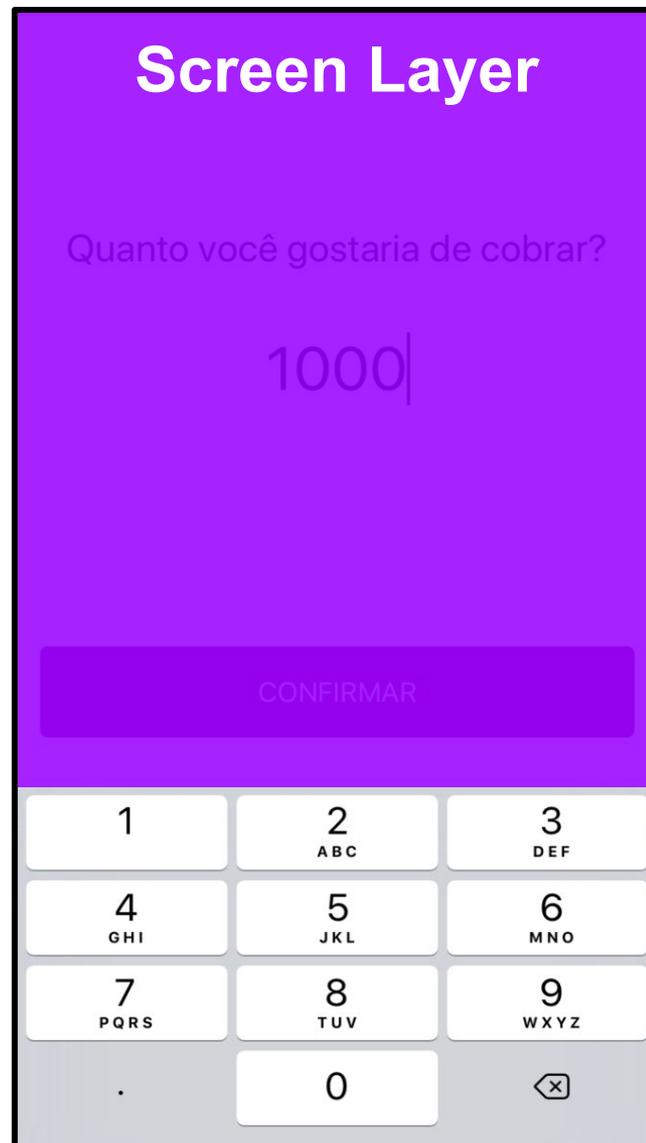
```
it('should create a request money and navigate to QRCode', async () => {
  // given
  const requestMoney = {
    id: 123,
    amount: 100,
  };
  const mock = jest.spyOn(RequestMoneyAPI, 'createRequestMoney');
  mock.mockReturnValue(Promise.resolve(requestMoney));
  const props = {
    navigateToQRCode: jest.fn(),
  };

  // when
  const { testInstance } = render(props as any);
  testInstance.instance.handleAmountChange(100);
  await testInstance.instance.handleSubmit();

  // then
  expect(mock).toBeCalledWith(100);
  expect(props.navigateToQRCode).toBeCalled();
});
```

# Testes de unidade na camada de Screen

Os testes garantem que a tela consegue navegar corretamente utilizando o navegador da aplicação





```
describe('AmountInput Screen', () => {  
  it('should navigate back');  
  
  it('should navigate to QRCode screen');  
  
  it('should navigate to Error screen');  
});
```



```
it('should navigate to QRCode screen', () => {  
  // given  
  const props = {  
    navigation: {  
      navigate: jest.fn(),  
    },  
  };  
  const requestMoney = {  
    id: 12,  
    amount: 1000,  
  };  
  
  // when  
  const { testInstance } = render(props as any);  
  testInstance.instance.navigateToQRCode(requestMoney);  
  
  // then  
  expect(props.navigation.navigate).toBeCalledWith(RequestMoneyScreens.QRCode, requestMoney);  
});
```

# Testes de integração

São testes que garantem o funcionamento de fluxos, exercitando cada unidade do fluxo e suas interações. Para isso, renderizamos o app em memória de forma a que consigamos navegar por ele - abstraindo a complexidade da camada nativa de plataformas.

# Testes de Integração

O objetivo do teste é garantir o funcionamento de um fluxo por completo, dessa forma garantimos que todos os caminhos possíveis do fluxo estão funcionando corretamente.



Inserir Valor

Verificamos a existência da tela, o preenchimento do campo e a submissão do formulário para um servidor mockado que retorna sucesso



QR Code

Verificamos a existência da tela de erro e exercitamos o botão de voltar que deveria voltar para a tela anterior



Inserir Valor

Verificamos a existência da tela inicial e assim finalizamos o teste desse fluxo



```
describe('RequestMoney', () => {
  describe('success QR code creation & share', () => {
    let app: RenderAPI;

    beforeEach(() => {
      fetchMock.resetMocks();
      fetchMock.mockResponseOnce(JSON.stringify(requestMoney));
      app = renderApp();
    });

    it('should open amount input screen');

    it('should fill the amount input');

    it('should confirm the amount & go to QR code screen');

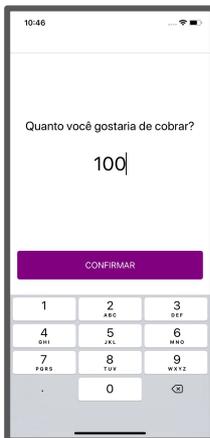
    it('should share the QR Code & go back to amount screen');
  });
});
```



```
it('should confirm the amount & go to QR code screen', async () => {  
  // given  
  const amountPage = new AmountInputPageObject(app);  
  
  // when  
  const qrCodePage = await amountPage.confirm();  
  
  await waitForExpect(async () => {  
    const isVisible = qrCodePage.isVisible();  
    expect(isVisible).toBe(true);  
  });  
});
```

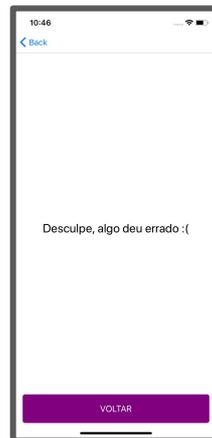
# Testes de Integração

O objetivo do teste é garantir o funcionamento de um fluxo por completo, dessa forma garantimos que todos os caminhos possíveis do fluxo estão funcionando corretamente.



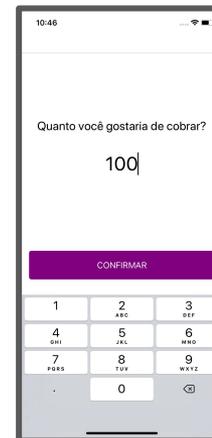
Inserir Valor

Verificamos a existência da tela, o preenchimento do campo e a submissão do formulário para um servidor mockado que retorna um erro



Exibição do Erro

Verificamos a existência da tela de erro e exercitamos o botão de voltar que deveria voltar para a tela anterior



Inserir Valor

Verificamos a existência da tela inicial e assim finalizamos o teste desse fluxo



```
describe('RequestMoney', () => {
  describe('failure when creating a QR code', () => {
    let app: RenderAPI;

    beforeEach(() => {
      fetchMock.resetMocks();
      fetchMock.mockReject(new Error());
      app = renderApp();
    });

    it('should open amount input screen');

    it('should fill the amount input');

    it('should confirm the amount & go to error screen');

    it('should close the error & go back to amount screen');
  });
});
```

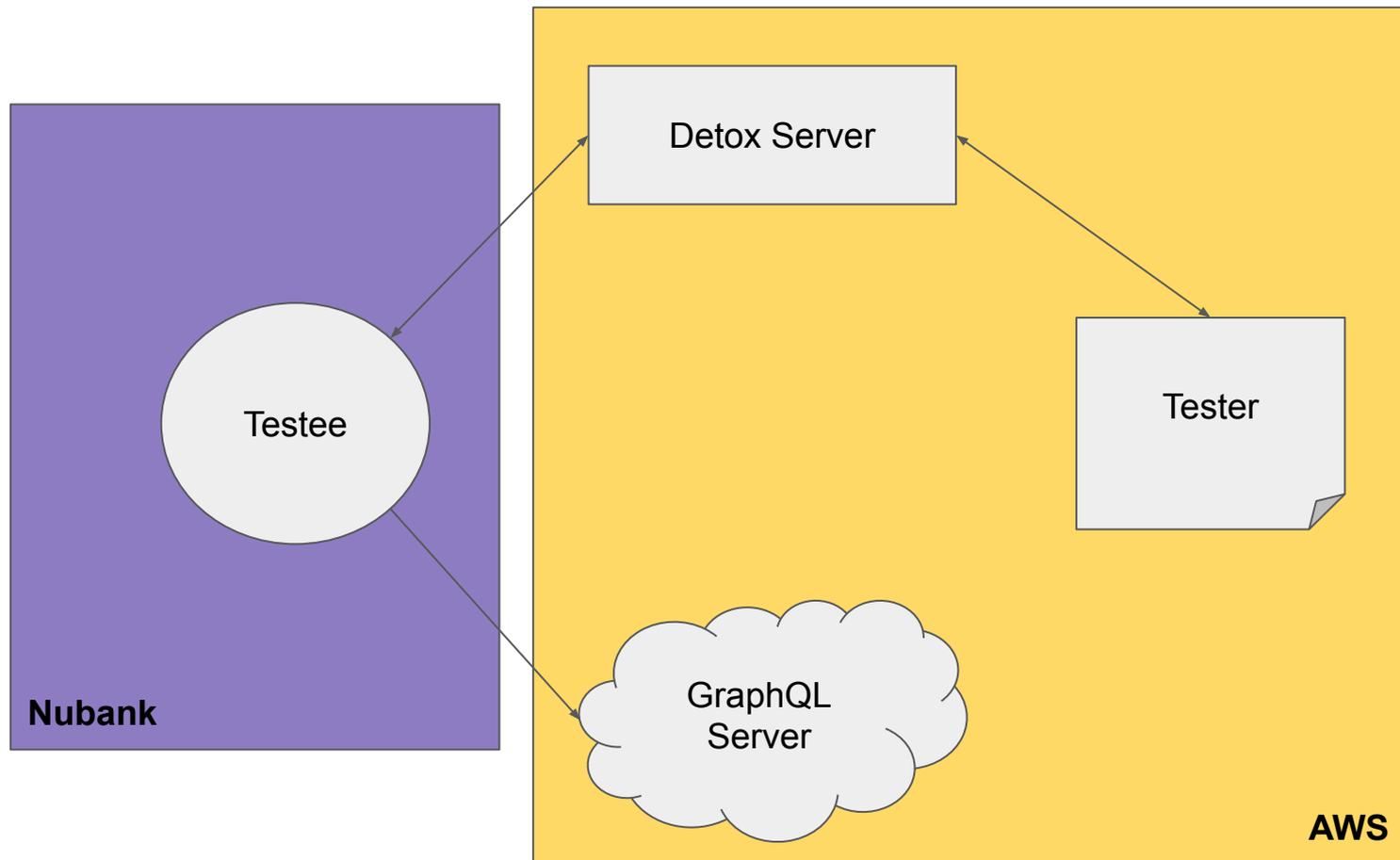
# Testes E2E

São testes que garantem o funcionamento do app como um todo, não se importando com a implementação mas sim com as interações entre o usuário e o app.

Esse tipo de teste também garante o funcionamento da integração das plataformas Nativas e React Native.

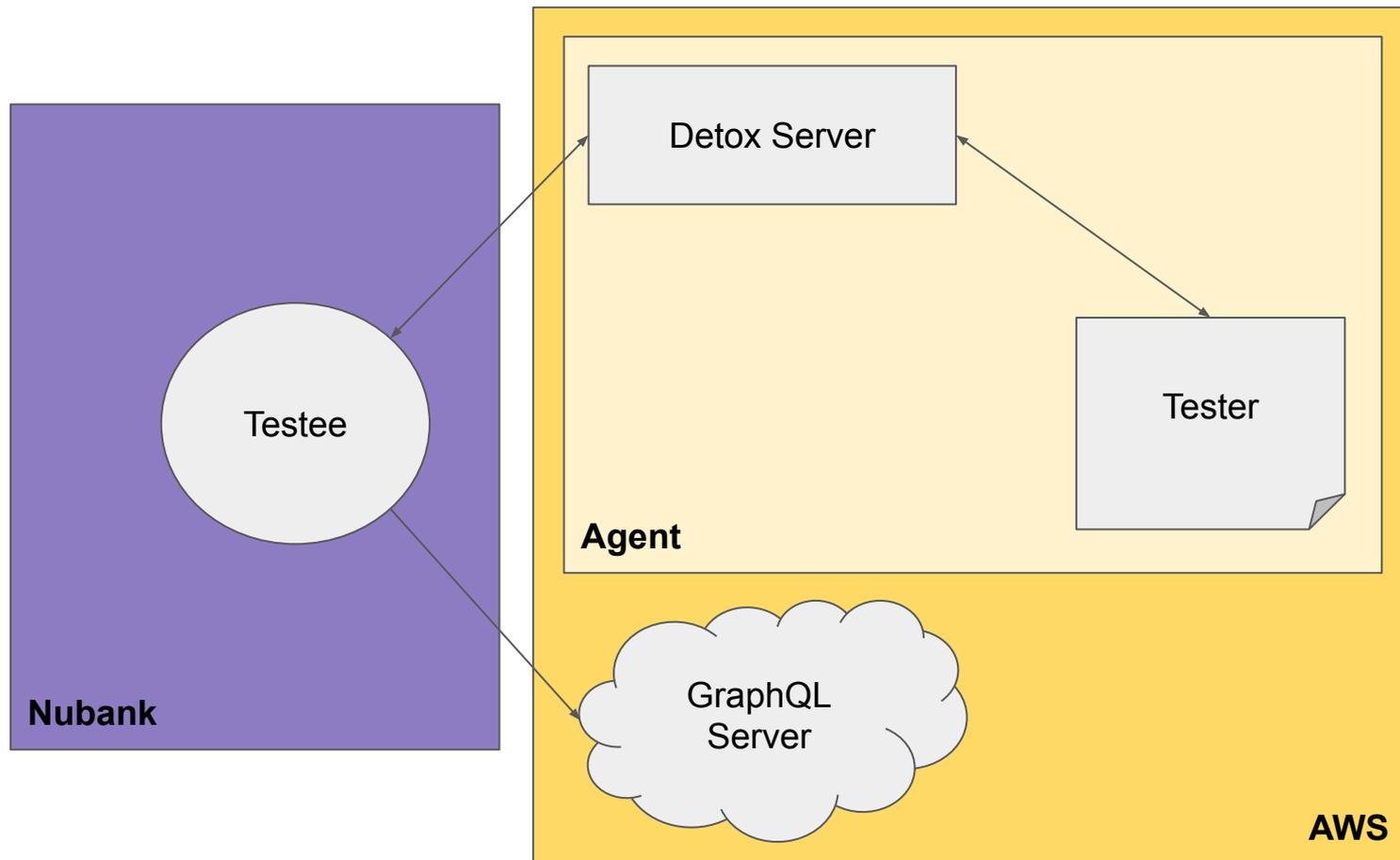
# Testes E2E com Detox

O Detox é uma biblioteca para testes E2E para Android & iOS, onde você escreve seu teste uma vez em Javascript e ele é executado nativamente usando Espresso (Android) e EarlGrey (iOS).



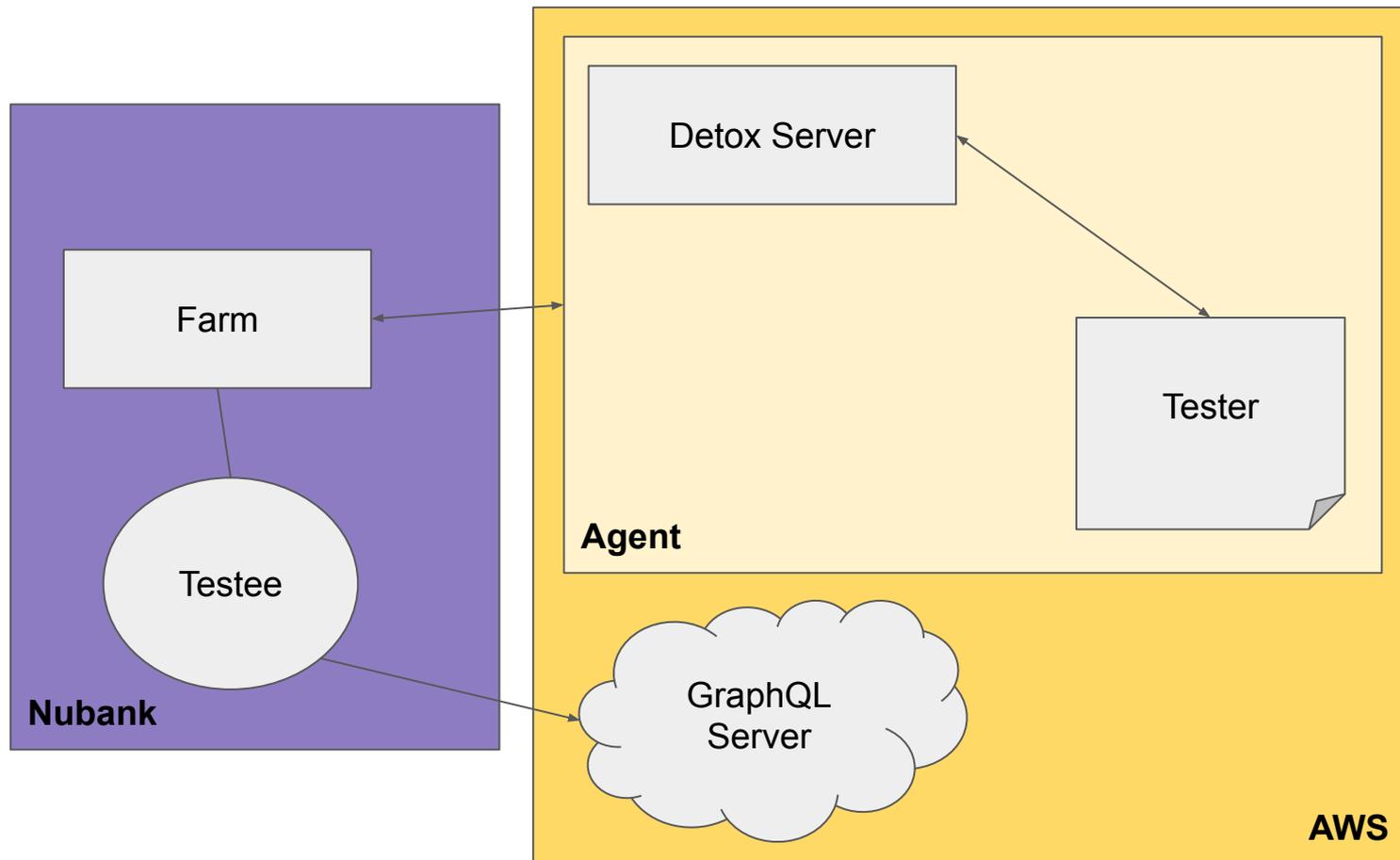
# Testes E2E com Detox

O Detox é uma biblioteca para testes E2E para Android & iOS, onde você escreve seu teste uma vez em Javascript e ele é executado nativamente usando Espresso (Android) e EarlGrey (iOS).



# Testes E2E com Detox

O Detox é uma biblioteca para testes E2E para Android & iOS, onde você escreve seu teste uma vez em Javascript e ele é executado nativamente usando Espresso (Android) e EarlGrey (iOS).





```
describe('RequestMoney', () => {  
  beforeAll(async () => {  
    await reloadApp();  
  });  
  
  it('should show amount input screen');  
  
  it('should fill the requested amount');  
  
  it('should confirm the requested amount');  
  
  it('should share the QR code');  
  
  it('should should show amount input screen again');  
});
```



```
it('should confirm the requested amount', async () => {  
  // given  
  const amountInputPage = new AmountInputPageObject();  
  
  // when  
  const qrCodePage = await amountInputPage.confirm();  
  
  // then  
  await qrCodePage.assertIsVisible();  
});
```



```
export class AmountInputPageObject {
    public async assertIsVisible() {
        await expect(element(by.id(screenId))).toBeVisible();
    }

    public async fillAmount(amount: number) {
        await element(by.id(amountInputId)).tap();
        await element(by.id(amountInputId)).typeText(`${amount}`);
    }

    public async assertHasAmountOf(amount: number) {
        await expect(element(by.text(`${amount}`))).toBeVisible();
    }

    public async confirm(): Promise<QRCodePageObject> {
        await element(by.id(confirmButtonId)).tap();
        return new QRCodePageObject();
    }
}
```

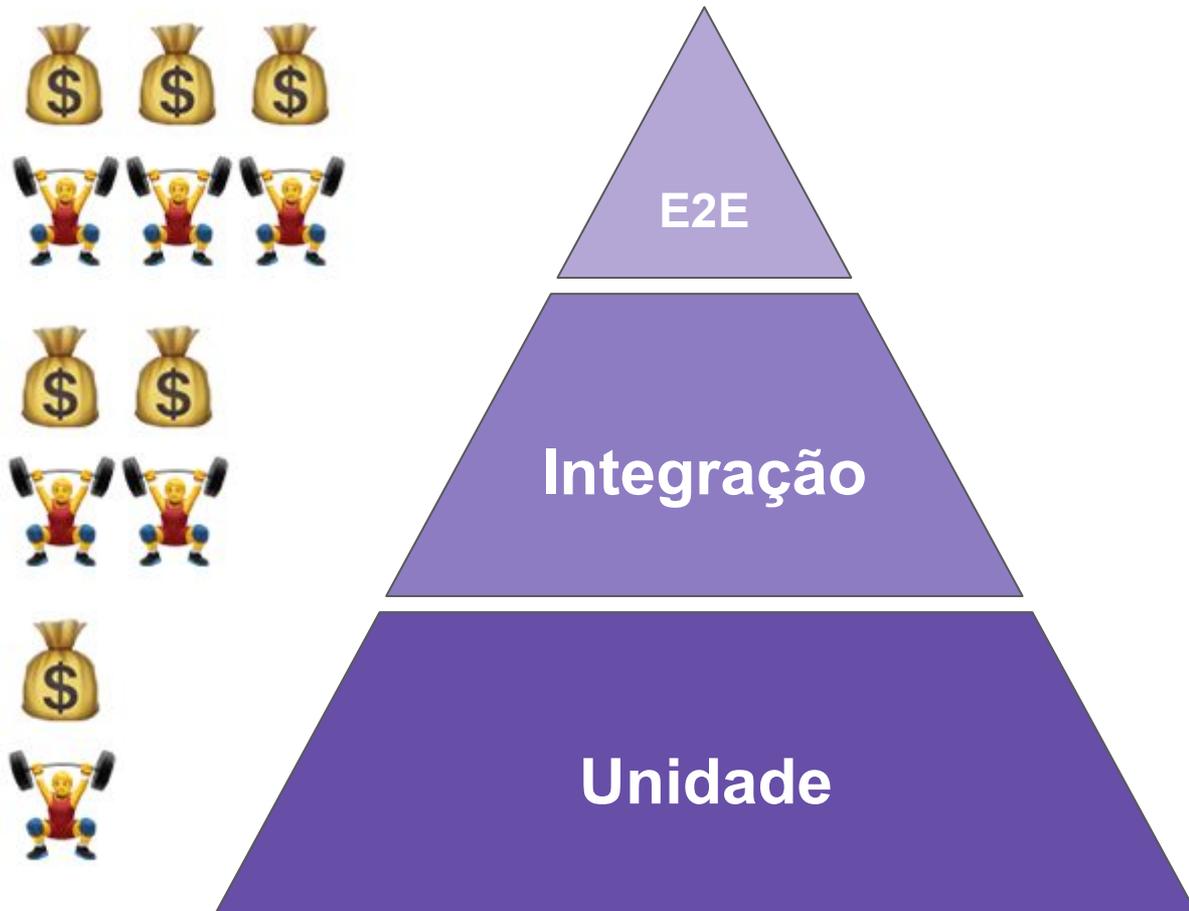
# Pirâmide de Testes

Analogia a distribuição dos testes entre os diferentes tipos de teste: unidade, integração e E2E.

Utilizar essa analogia ajuda a decidir qual tipo de teste deveríamos fazer para que nossas garantias sejam sanadas da melhor forma.

# Pirâmide de Testes

A pirâmide de testes nos ajuda a enxergar a proporção que devemos ter de cada tipo de teste.



# Perguntas?



**GitHub do projeto**

<https://github.com/chagasaway/tdc-2019>



**Estamos contratando!**

<https://nubank.workable.com>