VTEX

The Enterprise
Digital Commerce
Platform

indeva BY VTEX

Diretrizes para o desenvolvimento de Apps em VTEX IO

Sumário

Sumário

Sumário

01

O que é VTEX IO?

O que é VTEX IO?

VTEX IO é um sistema para o desenvolvimento de aplicativos web, principalmente para e-commerces. Esse sistema, além de facilitar o trabalho do desenvolvedor, também facilita a gestão do negócio pelo lado do administrador, já que não é necessário dominar desenvolvimento web para manter e atualizar sua loja.

Com o VTEX IO podemos criar e usar vários componentes configuráveis e reutilizáveis, o que significa que podemos evitar o retrabalho de muitas etapas do desenvolvimento e utilizar o mesmo componente em situações diferentes.

Portanto, com o uso desse sistema, podemos criar aplicativos novos e integrá-los na nossa loja, instalar aplicativos existentes e testar mudanças em tempo real, o que resulta em uma total liberdade para desenvolver e personalizar as aplicações com VTEX IO.

O que é um app?

Em VTEX IO não há limites que determinem o que um app pode englobar.
Pensando em *React atomic design*, um app pode ser por exemplo, um botão de compra, um “átomo”. Um app pode ser uma feature de localizador de lojas, com sua própria página ou um “resolver”, responsável por tratar as informações provenientes de uma API de busca.

Todos os exemplos de apps acima possuem suas semelhanças e só de olhar a raiz de um app e ver sua estrutura de pastas é possível saber o tipo de app que ele é.

Todo o ecossistema de um e-commerce que utiliza VTEX IO tem seu front-end construído por apps. Não há um código HTML com upload manual associado a um template, o que temos lá são apps tema com templates que chamam componentes React muitas vezes provenientes de outros apps.

Neste livro descreveremos alguns modelos de apps, para que servem e como são estruturados, mas para isso o conhecimento de alguns conceitos de VTEX IO é essencial.

02

Conceitos de VTEX IO

Manifest

O arquivo manifest.json é um arquivo essencial em todo app VTEX IO, sendo este encarregado de armazenar importantes informações de metadata do app como o seu name, vendor , versão, description, suas dependencies, os builders utilizados e etc. Alguns exemplos de campos simples presentes no manifest são:

name:
É o próprio nome do app. Ele deve ser conciso e expressar claramente a funcionalidade do app. Os apps publicados pela VTEX utilizam a padronização kebab case, onde todas as letras do nome app são minúsculas, os espaços são substituídos por hífen e caracteres especiais são evitados. Seguindo essas recomendações, por exemplo, um app com a funcionalidade de renderizar um selo de promoção no produto pode ser chamado de promotion-badge.

vendor:
Em VTEX IO a relação de propriedade de um app é atrelado a uma account VTEX, consequentemente, o vendor é o nome da conta proprietária que publicou o app. Cada app pode ter apenas um vendor, mas dependendo de suas billingOptions ele pode ser instalado em várias contas.

version:
Assim como todo software, os apps VTEX IO devem ser versionados. A VTEX utiliza como padrão de versionamento Versionamento Semântico (SemVer) que separa as versões em Majors, Minors e Patchs. Conforme uma nova versão estável é publicada e é realizado o seu deploy, contanto que ela não seja uma nova Major, essa nova versão será atualizada automaticamente em todas as contas que utilizem esse app.

title:
É o nome de distribuição do app que ficará visível no painel administrativo ou na App Store. No caso acima usamos o name: promotion-badge, um title adequado poderia ser Promotion Badge.

description:
Utilizado para ser uma descrição concisa do app. Deve em poucas linhas explicar a funcionalidade realizada pelo app. No caso acima poderíamos ter como por exemplo a seguinte description:

Renders labels in top of the Product image in case there's a promotion.

```
"vendor": "vtex",
"name": "facebook-pixel",
"version": "2.2.2",
"title": "Facebook Pixel",
"description": "Integrate with Facebook advertising tools.",
```

Billing Options

“billingOption” é um campo presente no arquivo manifest.json do app VTEX IO. Ele é utilizado para definir como o seu app será distribuído no ecossistema VTEX IO e, consequentemente, se ele possui algum custo agregado. Caso este campo não seja incluído no manifest.json do seu app, seu app será considerado privado.

Apps privados não possuem custo, mas também não podem ser encontrados na VTEX app store. Vale ressaltar também que sua instalação só pode ser realizada na account VTEX na qual ela foi publicada, ou seja, seu *vendor*.

Ao tentar pelo CLI a instalação de um app privado em uma conta diferente você receberá o seguinte erro:

```
08:08:07.467 - error: Installation failed!
08:08:07.468 - error: Private app cannot be accessed
08:08:07.468 - warn: The following app was not installed: acctglobal.regionalized-content@0.1.1
PS C:\Users\Admin\Documents\general-components\acct-regionalized-content>
```

Temos as seguintes propriedades dentro de billingOptions:

type:
usada para definir como o app será cobrado. Alguns possíveis valores são free, o que indica um app gratuito e billable que indica uma cobrança conforme seus planos definidos na propriedade plans.

support:
usado para listar os canais de contato caso seja necessário obter suporte para o app. Dentro dele é possível passar um objeto contendo email, url e phone.

availableCountries:
usado para definir em quais países o app pode ser instalado e em quais países há suporte. Os países devem ser passados em formato de array com o seu id no formato ISO.

plans:
usado para definir o plano de assinatura do app. Nele deve ser passado um array de objetos planos com id, moeda e preço.

```
"billingOptions": {
  "termsURL": "https://compliance.vtex.com/gdpr/policies/vtex-privacy-policy",
  "support": {
    "url": "https://support.vtex.com/hc/requests"
  },
  "free": true,
  "type": "free",
  "availableCountries": [
    "*"
  ]
},
```

Builders

O papel de um Builder é processar, validar e encaminhar um determinado bloco de código para o runtime ou framework capaz de executá-lo. Por exemplo, para utilizar React dentro do seu app é necessário que o builder de react seja listado dentro do manifest.

Para que o builder identifique o código correspondente é necessário que uma pasta com o nome dele exista na raiz do projeto. Frequentemente novos builders são adicionados pelo time de produto da VTEX, mas alguns necessitam de uma liberação feita pelo formulário de VTEX IO Closed Beta para sua utilização.

Neste formulário é imperioso que informe o vendor, nome do app a ser publicado, sua major, os builders desejados e motivo de suas utilizações.

A lista atualizada com todos os builders disponíveis pode ser encontrada dentro do developer docs

Dependencies

Dependecies é um campo no manifest.
json em formato de objeto JSON usado
para sinalizar quais são os outros apps
necessários para o funcionamento do
seu app. Na prática, quando um app é
instalado em uma conta VTEX, todos os
apps presentes no campo dependencies do
manifest são instalados automaticamente
para garantir seu funcionamento.

Sua estrutura semântica segue o padrão
{vendor}.{appName}: {majorVersion}.x`

Um exemplo claro de sua utilização pode
ser vista no próprio app Store-theme, onde
todos os outros apps necessários para
criar as interfaces usadas no front-end são
chamados através de dependências.

Peer Dependencies

PeerDependencies é também um
campo presente no arquivo manifest.
json . Assim como em dependencies,
em peerDependencies temos uma lista
de apps que são essenciais para o
funcionamento do app em questão. O que
separa peerDependencies de dependencies
é que os apps presentes neste campo
não são automaticamente instalados
junto com o app, o principal caso de uso
para a utilização desse campo é para a
listagem de apps que podem ter custo ou
assinatura. Por conta da sua delicadeza,
alterar peerDependencies de um app
requer que você lance uma nova versão
Major, garantindo assim que não haverá
atualização automática de um app que
pode não ter sido adquirido por outra
conta.

Interfaces

Interfaces.json é um arquivo que pode ser utilizado pelo Builder Store que estabelece um relacionamento entre um bloco e um componente React. Apps que criam novos componentes de front-end usam o interfaces para que um app tema possa chamá-lo.

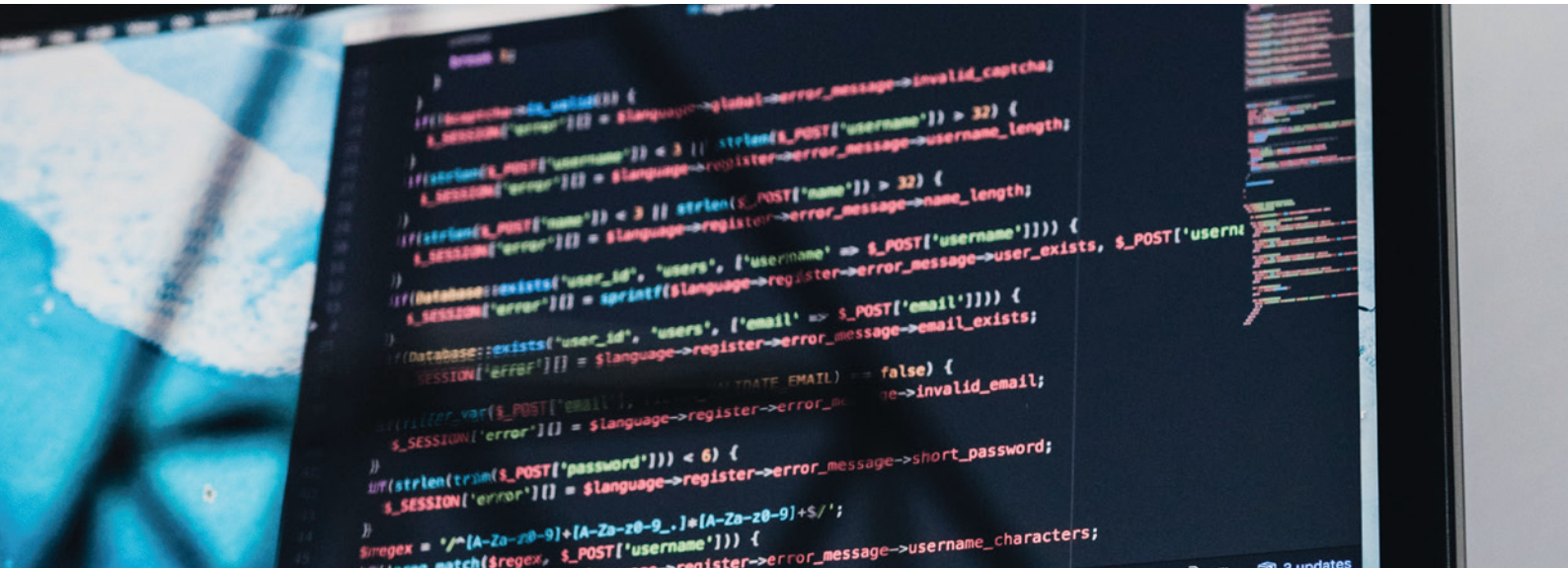
```
{
  "store-badges": {
    "component": "BadgesStore"
  }
}
```

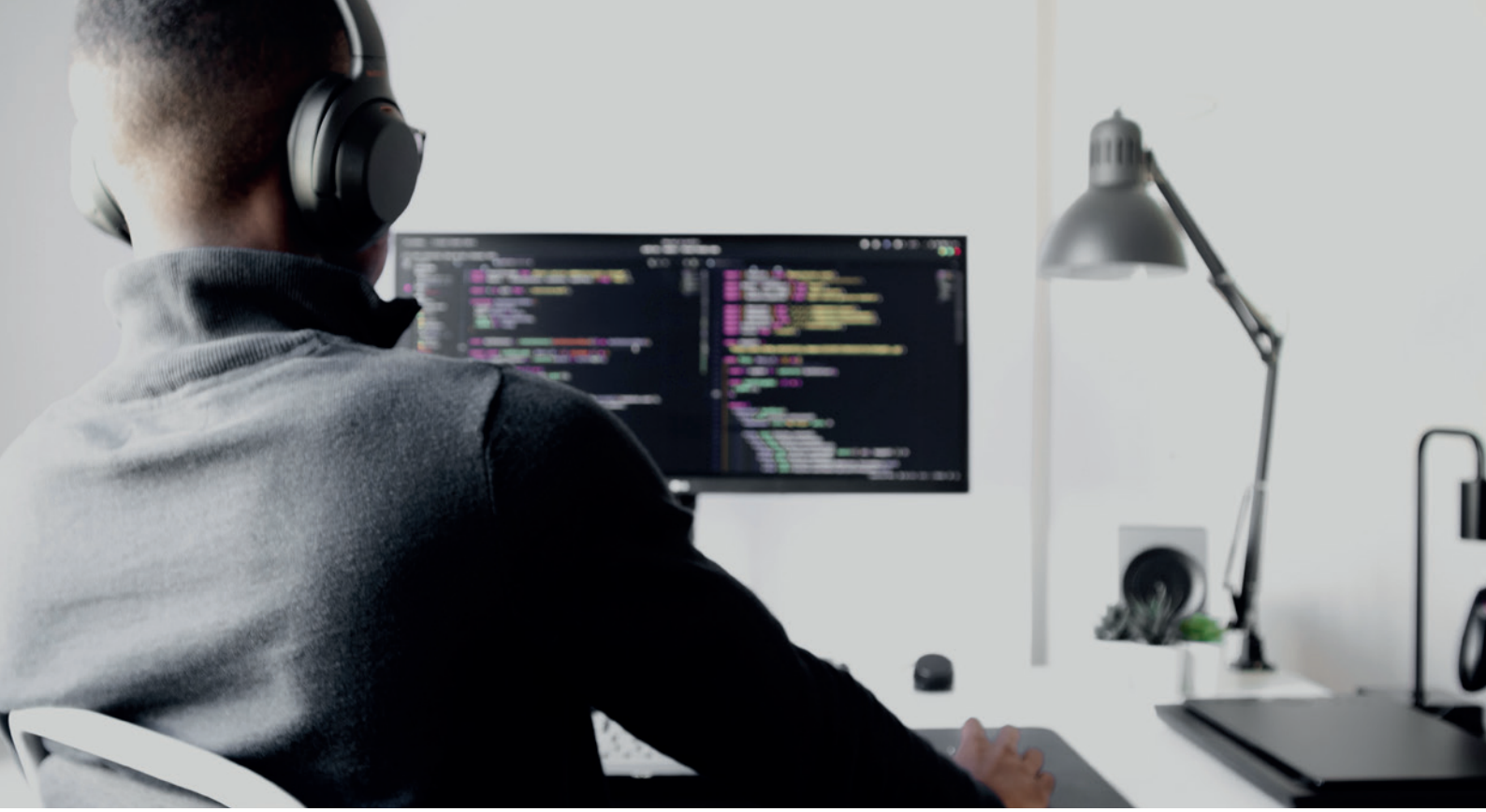
Dentro do interfaces.json temos também composition, que é onde o comportamento do componente react é declarado podendo ser blocks , children ou slots.

Blocks:
é a composição padrão para os componentes definidos no seu app tema. Nela uma lista de blocos é passada e, uma vez chamada, todos os seus blocos são renderizados dentro dela. Um exemplo que facilita sua ilustração é a própria home do store-theme.

```
"store.home": {
  "blocks": [
    "list-context.image-list#demo",
    "__fold__.experimentalLazyAssets",
    "flex-layout.row#deals",
    "rich-text#shelf-title",
    "flex-layout.row#shelf",
    "info-card#home",
    "rich-text#question",
    "rich-text#link",
    "newsletter"
  ]
},
```

Blocos passados desta forma tem sua ordenação determinada pela ordem em que são chamados





Children:
oriunda da própria definição de React children, aqui temos uma estrutura que também recebe uma lista de blocos, mas esses são os blocos necessários para estruturar o componente pai:

```
"flex-layout.row#list-price-savings": {
  "props": {
    "colGap": 2,
    "preserveLayoutOnMobile": true,
    "preventHorizontalStretch": true,
    "marginBottom": 2,
    "marginTop": 5
  },
  "children": [
    "product-list-price",
    "product-price-savings"
  ]
}
```

Slots:
é a sugestão da VTEX sempre que aplicável devida a sua maior flexibilidade. Permite que você declare blocos filhos usando propriedades React ao invés de um array de blocos. Não é necessário usar o atributo allowed com essa composição. Qualquer bloco pode ser chamado como prop de outro.

```
"hello-world": {
"props": {
  "Icon": "icon-caret#point-up"
```

Vale ressaltar que a sintaxe de nomeação utilizada nas props é PascalCase. Atualmente também não é possível ter slots aninhados.

Routes

Routes associam um padrão de URL e um request HTTP a uma ação. Em VTEX IO não é necessário se preocupar com o roteamento de requests HTTP uma vez que estes são gerenciados pelo builder de store com o auxílio do App Rewriter, que cuida das rotas de Produto, Busca e Navegação.

Existem também as Custom routes que são Rotas criadas pela interface do CMS ou definidas no próprio app.

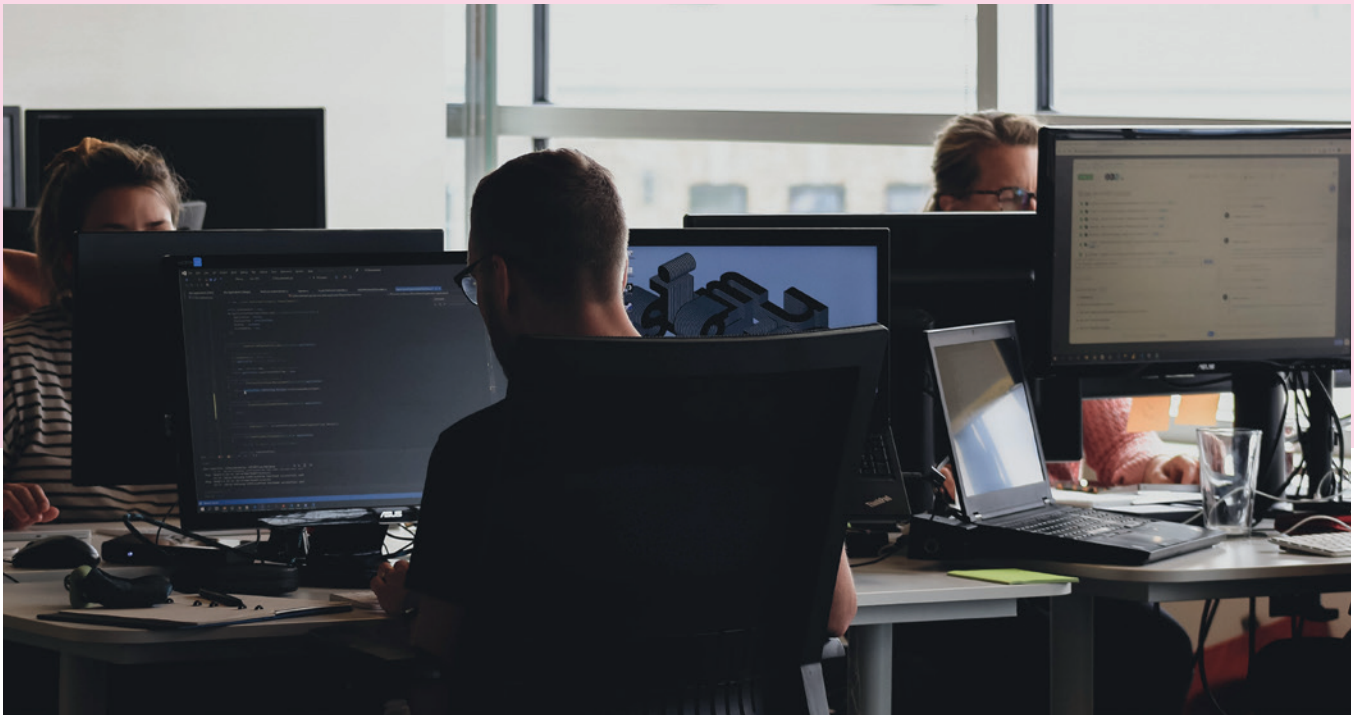
Para criar uma custom route no seu app, deve ser criado um arquivo routes.json dentro da pasta store e nele deve ser incluído uma identificação do template que será renderizado e o path que ele responderá.

No app store-locator, por exemplo, temos a rota /stores que renderiza o conteúdo do localizador de lojas de modo que não seja necessário chamar as interfaces definidas no seu app tema:

```JSON
{
  "store.storelocator": {
    "path": "/stores",
    "isSitemapEntry": true
  }
}
```

O campo path pode aceitar parâmetros de URL como /path/:param(/:optional-param) por exemplo, mas é recomendado cautela na sua utilização. Um exemplo prático de sua aplicação também está presente no app store-locator, onde os parâmetros da URL são usados para identificar e exibir os detalhes da loja selecionada:

```
{
  "store.storedetail": {
    "path": "/store/:slug/:store_id",
    "isSitemapEntry": false
  }
}
```



Workspace

Workspaces são ambientes de desenvolvimentos isolados um do outro. Eles consomem as mesmas informações provenientes dos módulos de catálogo, preço, estoque, logística, promoção e pedidos. Consequentemente qualquer alteração nesses módulos, mesmo que feito em workspace, será refletido na produção. O objetivo de um workspace é, principalmente, ser independente a nível de instalação e linkagem de apps e alterações no site-editor. Isso permite que o desenvolvimento seja validado no ambiente myVTEX sem que afete produção ou outros desenvolvimentos que aconteçam simultaneamente.

Ideologicamente elas são bem semelhantes às branchs utilizadas em Git.

Existem três tipos de workspaces: os de desenvolvimento, que podem ser utilizados para validar o desenvolvimento, tendo seu código linkado e refletido no ambiente myVTEX em tempo real. Os de produção que não permitem que o código seja linkado, mas que podem ser usados em teste AB e podem ser promovidos para se tornarem o novo ambiente de produção e, por fim, temos também o workspace master, que é o próprio ambiente de produção.

VTEX IO CLI

CLI, command-line interface, também conhecido como “Interface de linha de comando”, é a ferramenta utilizada para desenvolver utilizando a plataforma VTEX IO. Ela possibilita a utilização de diversos comandos essenciais e obrigatórios na hora do desenvolvimento de um app e de uma loja.

Como instalar:
Com o gerenciador de pacotes yarn instalado em sua máquina, basta rodar o comando yarn add global VTEX. Após a finalização da instalação, você pode rodar o comando VTEX -v para verificar se a instalação foi concluída com sucesso e este comando retornará em seu terminal a versão instalada do VTEX IO CLI.

VTEX IO CLI

Comandos essenciais:

VTEX:

este comando lista todos os outros comandos disponíveis que podem ser utilizados no CLI. O comando `VTEX help` também realiza a mesma ação.

```
PS C:\Users\Admin\Documents\store-theme> vtex
The platform for e-commerce apps

VERSION
  vtex/2.121.3 win32-x64 node-v14.18.0

USAGE
  $ vtex [COMMAND]

COMMANDS
  add          Add app(s) to the manifest dependencies
  autoupdate   update the vtex CLI
  browse       Open endpoint in browser window
  config       Config commands. Run 'vtex config' to see all subcommands
  debug        Debug commands. Run 'vtex debug' to see all subcommands
  deploy       Deploy a release of an app
  deprecate    Deprecate an app
  deps         Dependencies commands. Run 'vtex deps' to see all subcommands
  edition      Edition commands. Run 'vtex edition' to see all subcommands
  help         display help for vtex
  infra        Infra commands. Run 'vtex infra' to see all subcommands
  init         Create basic files and folders for your VTEX app
  install      Install an app (defaults to the app in the current directory)
  lighthouse   Run lighthouse audit over a specific url
  link         Start a development session for this app
  list         List your installed VTEX apps
  local        Local commands. Run 'vtex local' to see all subcommands
  login        Log into a VTEX account
  logout       Logout of the current VTEX account
  logs         Show apps production logs
```

VTEX login:

este comando é utilizado para efetuar o login em uma account VTEX. Ao utilizá-lo uma janela do navegador será aberta para realizar a autenticação naquela account. É necessário primeiro estar logado em uma account para que seja possível criar uma workspace e linkar seu app

```
PS C:\Users\Admin\Documents\store-theme> vtex login acctglobal
10:42:03.957 - info: Logged into acctglobal as luiz.priolli@acct.global at workspace master
10:42:05.304 - info: Welcome to VTEX IO!
```

VTEX use:

este comando é utilizado para trocar a workspace ativa e caso uma workspace ainda não existente seja selecionada, ele lhe dá a possibilidade de criá-la. Também é possível utilizá-lo para criar workspaces de produção adicionando -p após o nome da workspace

```
PS C:\Users\Admin\Documents\store-theme> vtex use novaworkspace
! Workspace novaworkspace doesn't exist
? Do you wish to create it? » (Y/n)
```

VTEX whoami:

este comando mostra qual é o account em que você está logado e o workspace que está sendo utilizado. Ele também mostra o tipo do workspace, que pode ser production ou development.

VTEX init:

este comando é utilizado para inicializar um repositório, adicionando os arquivos mínimos necessários para você iniciar o desenvolvimento de um app. Ao utilizá-lo é possível definir o tipo de app que será desenvolvido:

```
PS C:\Users\Admin\Documents\store-theme> vtex init
10:47:55.170 - info: Hello! I will help you generate basic files and folders for your app.
? Choose where do you want to start from ...
> store
  edition app
  checkout-ui-settings
  Cancel
```

VTEX link: este comando estabelece um link entre o app que está sendo desenvolvido, pegando o código da sua máquina, realizando o build e deixando-o visível na sua workspace. Qualquer alteração feita e salva no seu código enquanto o link estiver ativo gerará uma nova build do seu código sendo possível validar o que está sendo desenvolvido em tempo real.

VTEX unlink: interrompe o link estabelecido com o comando VTEX link desconsiderando todas as alterações de código que não estejam instaladas na account.

VTEX ls: este comando lista os apps que estão linkados e instalados na workspace que está em uso. Apps provenientes de um edition app também são exibidos desta forma.

VTEX install: este comando instala o app especificado na account e workspace ativo. Apps instalados em workspace não afetam a master. Caso o app que você deseja instalar esteja sendo desenvolvido e você esteja com o terminal no próprio repositório não é necessário incluir seu vendor e app, pois os dados serão puxados diretamente do manifest.json.

```
PS C:\Users\Admin\Documents\store-theme> vtex install
10:52:11.866 - info: Installed app vtex.store-theme@4.3.0 successfully
```

VTEX uninstall: desinstala o app especificado, da mesma forma que o comando acima, essa ação é específica de workspaces.

VTEX publish: publica o app que está sendo desenvolvido. Uma vez que um app esteja publicado é possível instalá-lo manualmente em uma account.

VTEX deploy: lança o app que está sendo desenvolvido em produção, só é possível executar este comando após publicar um app. Uma vez que o deploy tenha sido feito ele estará apto a ser atualizado automaticamente caso uma versão deste app seja utilizado por alguma account.

VTEX release: este comando é utilizado para lançar uma nova versão do seu app sem que seja necessário mudar a versão presente no manifest manualmente. Para utilizá-lo é necessário especificar o tipo de versão a ser lançada (major, minor ou patch) e se ela é uma versão stable ou beta. Exemplo: VTEX release minor beta.

VTEX update: este comando exibe a lista de todos as atualizações disponíveis para apps que estejam instalados. Havendo alguma atualização disponível é possível também atualizá-los manualmente com esse comando.

VTEX promote: este comando só pode ser executado em workspaces do modo produção. Ele faz com que o workspace ativo se torne o novo workspace master, ou seja, vá para produção. Todos os apps instalados e toda configuração realizada no site-editor também será migrada para a produção.

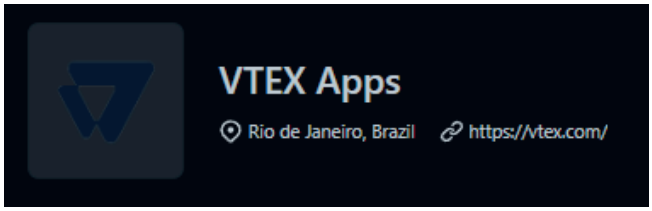
Entendendo as funcionalidades desejadas

Um passo essencial para o desenvolvimento é o entendimento completo do que tem que ser desenvolvido. Um épico ou feature pode precisar do desenvolvimento de múltiplos apps diferentes. Refletir sobre a estrutura de apps necessária é um ponto de atenção em VTEX IO principalmente por vários builders necessitarem de whitelist. Um app de Node.js, por exemplo, não pode ter seu desenvolvimento iniciado assim que a necessidade dele é identificada, portanto o planejamento com antecedência é fundamental.

Também evite ter apps genéricos que são responsáveis por múltiplas features e componentes. A granularidade provê uma estabilidade e facilidade de manutenção significativamente maior.

Verificando se já não existe um app que resolva o problema

Antes de iniciar qualquer desenvolvimento customizado, é importante checar se a VTEX já não nos disponibiliza um componente nativo com uma funcionalidade e layout parecida ou igual com a que precisamos. Para isso, podemos verificar o repositório de componentes nativos dela, onde contém todos os componentes usáveis, com documentações amigáveis para integrarmos na nossa loja. finalização da instalação, você pode rodar o comando VTEX -v para verificar se a instalação foi concluída com sucesso e este comando retornará em seu terminal a versão instalada do VTEX IO CLI.



Podemos encontrar esse repositório no GitHub, ele se chama VTEX Apps e atualmente se encontra com mais de 400 repositórios para explorarmos e estudarmos, a fim de encontrar o componente que resolva o problema do nosso projeto.

Evoluindo apps nativos

Uma outra opção para que não precisemos criar um aplicativo customizado, é evoluir um já existente, pois a maioria dos componentes da VTEX são de código aberto, ou seja, podemos contribuir para a aplicação e a funcionalidade implantada estará disponível após a aprovação deles, lembrando que caso eles aprovem, seu perfil do GitHub ficará presente na documentação do aplicativo.

Para contribuir, você irá precisar entrar no repositório do aplicativo, forkar ele para a sua conta, criar uma nova branch, e após o término do desenvolvimento e testes, enviar um pull request para o mesmo repositório, assim seu código será analisado e você terá um retorno se foi aceito, negado ou se precisa mudar algo.

É muito importante ter uma visão bem ampla dessa sua nova feature no aplicativo deles, pois tem que ser uma evolução que não impacte negativamente nenhuma loja. Não é recomendado evoluir um aplicativo nativo, caso sua feature seja bem específica para um projeto, e também, não esqueça de atualizar a documentação no final do seu desenvolvimento, explicando o que foi adicionado ou alterado.

03

Desenvolvendo um novo App

Primeiros passos para o desenvolvimento

Inicializando o ambiente

Ao efetuar o login pelo CLI da VTEX será exibida qual edition é utilizada por aquela conta:

```
PS C:\Users\Admin\Documents\store-theme> vtex login accttest
14:34:32.473 - info: Logged into accttest as luiz.priolli@acct.global at workspace master
14:34:34.128 - info: Welcome to VTEX IO!
General
Account             accttest
Workspace           master
Edition             Business Edition
Edition id          vtex.edition-business@0.21.0
Edition activated   Wed Oct 27 19:54:29 UTC 2021

Installed Apps
vtex.order-placed   1.7.3
```

A VTEX possui duas editions: Store Edition e Business. Caso o seu ambiente não esteja utilizando a Store Edition, você não irá conseguir utilizar os comandos supracitados para realizar o seu desenvolvimento. O próprio ato de tentar criar uma workspace irá lhe mostrar um erro:

```
PS C:\Users\Admin\Documents\store-theme> vtex use teste
ℹ Workspace teste doesn't exist
√ Do you wish to create it? ... yes
14:37:39.400 - info: Workspace teste created successfully with production=false
14:37:40.103 - warn: This account is using the edition vtex.edition-business@0.21.0.
14:37:40.104 - warn: If you are developing your store in IO, it is strongly recommended that you switch to the vtex.edition-store@2.x.
14:37:40.104 - warn: For more information about editions, check https://vtex.io/docs/concepts/edition-app/
? Would you like to change the edition to vtex.edition-store@2.x now? » (y/N)
```

É necessário, portanto primeiro instalar o Store edition antes de iniciar o desenvolvimento. Para isso é necessário enviar um ticket pelo central de atendimento da VTEX, uma vez que apenas as contas VTEX possuem o acesso necessário para realizar essa instalação. Caso o seu ambiente utilize o motor de busca inteligente da VTEX, o Intelligent Search é também muito importante sempre revisar se o catálogo foi corretamente indexado para que os produtos apareçam na loja. Se a aba busca/search não estiver disponível no menu lateral do painel administrativo da VTEX pode ser necessário realizar a instalação do app vtex.admin-search.

A versão do app `vtex.search-resolver` também deve ser revisada, uma vez que o Intelligent Search utiliza a versão na Major 1.

Boilerplate a ser usado

Tendo em vista as principais tecnologias que usamos para os nossos aplicativos, é importante que comecemos o desenvolvimento a partir de algum boilerplate, pois assim já temos as configurações iniciais necessárias para iniciar nosso desenvolvimento, o que facilita o processo como um todo. Dentro os boilerplates que podemos encontrar no repositório de componentes do GitHub da VTEX, os principais são:

Boilerplate para Apps React
Boilerplate para Apps tema do store framework
Boilerplate para Apps Node.js
Boilerplate para Apps Node.js com GraphQL

04

Etapas do desenvolvimento

Desenvolvendo e testando seu app com React Builder e Store Builder

Podemos começar clonando um template através deste link.

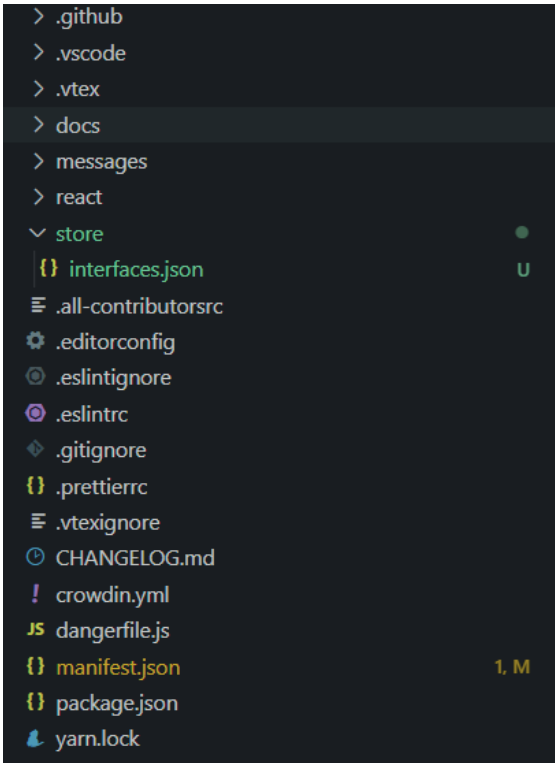
Após tendo isso feito, é importante mudarmos as informações referente ao app nos seguintes arquivos: manifest.json, package.json, CHANGELOG.md e README.md. Outro passo importante é rodar o yarn na raiz do projeto e na pasta react.

Dos arquivos citados acima, é importante se atentar às mudanças feitas no manifest.json, pois estas serão as informações pelas quais seu app será identificado, segue o exemplo criado para este guia:

```
{} manifest.json > ...
You, seconds ago | 4 authors (You and others)
{
  "vendor": "acctglobal",
  "name": "vtexbook",
  "version": "0.1.0",
  "title": "React app template for VTEX Book Example",
  "description": "This app has the objective to give development example to the VTEX Book",
  "builders": {
    "react": "3.x",
    "messages": "1.x",
    "docs": "0.x",
    "store": "0.x"
  },
  "dependencies": {},
  "registries": [
    "smartcheckout"
  ],
  "policies": [],
  "$schema": "https://raw.githubusercontent.com/vtex/node-vtex-api/master/gen/manifest.schema"
}
```

Em seguida, devemos adicionar o builder de store aos builders do manifest.json na versão 0.x e criar a pasta store na raiz do projeto, e o arquivo interfaces.json dentro da mesma.

Segue demonstração de como deverá ficar a estrutura das suas pastas:

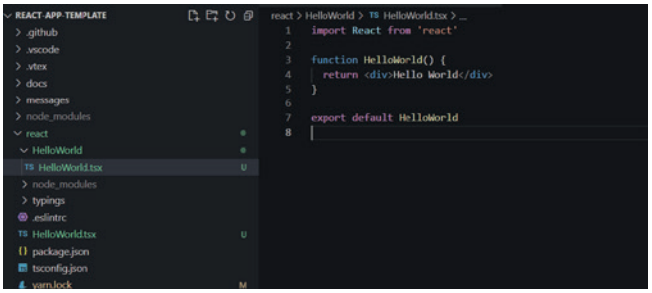


Agora você poderá começar a desenvolver o seu primeiro componente utilizando os builders de React e Store.

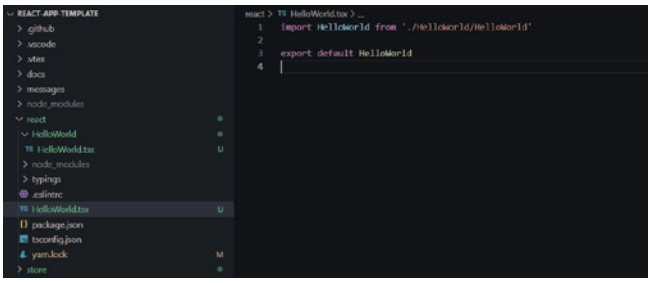
Você pode excluir os arquivos Greetings.tsx e Greetings.test.tsx , pois não precisaremos deles. Vamos iniciar criando uma pasta para o nosso componente, chamada HelloWorld e, dentro desta pasta, o arquivo HelloWorld.tsx.

Após o desenvolvimento do seu componente, você precisará criar também um arquivo de mesmo nome do seu componente na root da pasta react, você irá utilizar este arquivo somente para importar e exportar o seu componente a partir da mesma.

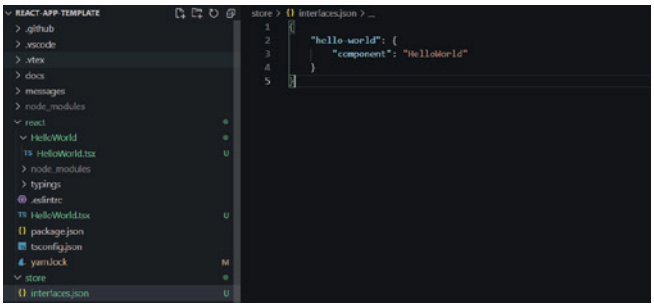
Abaixo segue exemplo de como deverá ficar o seu componente e a estrutura das pastas:



E este é o arquivo que deverá importar e exportar seu componente a partir da pasta raiz do react:



Agora, no arquivo interfaces.json, precisamos criar uma interface para o nosso componente, da seguinte maneira:



É importante saber que hello-world é o nome da nossa nova interface, e que esta interface irá renderizar o componente HelloWorld, que foi exportado a partir da root do react.

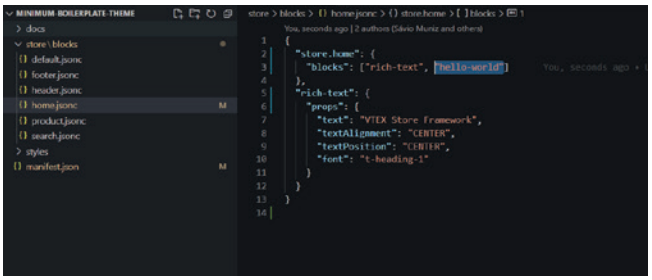
Conforme forem criadas novas interfaces, elas devem ser adicionadas no mesmo arquivo, seguindo o exemplo prévio. Agora, estando logado em seu ambiente myVTEX e em seu workspace criado previamente, basta rodar o comando VTEX link no terminal, na pasta raiz do projeto, para que possamos utilizar a interface hello-world no tema da loja.

No próximo passo, precisaremos clonar o minimum-boilerplate-theme para chamarmos nossa nova interface em uma loja e testarmos seu funcionamento. Neste projeto, podemos seguir os mesmos passos que realizamos em react-app-template, como mudar as informações dos arquivos manifest.json, README.md, e também é importante criarmos um CHANGELOG.md, para manter os releases documentados.

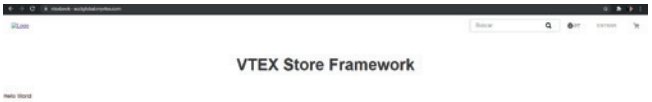
Em seguida, iremos adicionar o app com builder react no manifest.json de seu app tema, para que seja possível chamar a interface que foi desenvolvida a partir do app que criamos, no seguinte formato:

vendor.name: major.x

Feito isso, podemos abrir o arquivo home.json (ou qualquer outro de sua preferência), e adicionar a interface, hello-world, criada previamente:

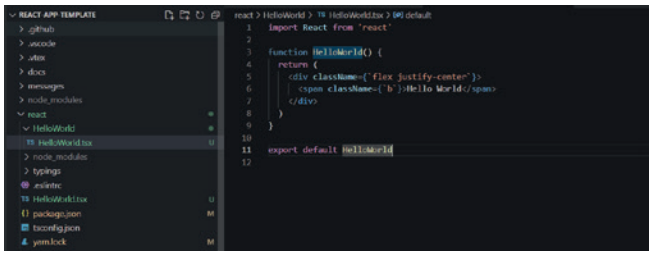


Podemos agora acessar o link da workspace que estamos utilizando e vermos a renderização da nova interface chamada hello-world:



Para finalizar, podemos adicionar algumas classes de tachyons em nosso componente, no app com builder react, para deixarmos este componente mais apresentável.

No exemplo abaixo adicionei um container para o elemento de texto, e adicionei algumas classes de tachyons para centralizar o elemento e deixar a fonte em negrito:



Agora, basta salvar o arquivo, e caso seu app ainda esteja linkado, você verá as alterações em sua workspace em tempo real. Se não estiver linkado, basta rodar o comando VTEX link no terminal, na pasta raiz do seu projeto.

Existem muitas propriedades que podemos utilizar do tachyons, para melhorar a aparência dos componentes, você pode conferi-las na documentação oficial: TACHYONS Docs.

Publicação e instalação

Nesta seção utilizaremos exemplos desenvolvidos acima para exemplificar a publicação e instalação de um app.

Com o nosso app react criado e configurado, precisamos realizar as seguintes alterações antes de publicarmos o app:

- É preciso documentar o CHANGELOG. md com as informações sobre a sua publicação. Você pode conferir mais informações sobre como fazer isto e boas práticas, na seção de Changelog deste livro.

- Precisamos definir a versão que será publicada no manifest.json, você também pode conferir na seção de manifest para saber como atualizar o versionamento do seu app.

Segue um exemplo após o conclusão dos passos acima:

```
CHANGELOG.md > # Changelog > ## [0.1.0] - 2021-11-16 > ### Added
You, seconds ago | 3 authors (Felipe Sales and others)
1  # Changelog
2
3  All notable changes to this project will be documented in this file.
4
5  The format is based on [Keep a Changelog](https://keepachangelog.com/en/1.0.0/),
6  and this project adheres to [Semantic Versioning](https://semver.org/spec/v2.0.0.html).
7
Felipe Sales, 2 years ago | 1 author (Felipe Sales)
8  ## [Unreleased]
9
You, seconds ago | 2 authors (You and others)
10 ## [0.1.0] - 2021-11-16
You, seconds ago | 2 authors (You and others)
11 ### Added
12 - `hello-world` component and interface.
13
```

Neste caso, foi possível manter o versionamento em 0.1.0, por se tratar de um app novo, esta versão ainda não havia sido publicada. Então não houveram alterações feitas no manifest.json.

Tendo isto feito, podemos rodar o comando VTEX publish em nosso terminal, estando na pasta raiz do projeto.

```
$ vtex publish
√ Are you sure that you want to release version 0.1.0 of acctglobal.vtexbook? ... yes
23:12:50.315 - info: Running yarn in react
yarn install v1.22.10
warning package.json: No license field
warning CHANGE_ME@0.0.0: No license field
[1/4] Resolving packages...
success Already up-to-date.
Done in 0.55s.
23:12:50.153 - info: Finished running yarn
23:12:50.729 - info: Project size: 0.45MB
23:12:50.730 - info: Compressing project files...
23:12:50.806 - info: Compressed project size: 0.19MB
23:13:01.925 - info: Sending files: 100% - 0.19MB/0.19MB
23:13:01.990 - info: Build accepted for acctglobal.vtexbook@0.1.0 at acctglobal/master vtex.builder-hub@0.281.1
23:13:02.120 - info: Starting build for app acctglobal.vtexbook@0.1.0 vtex.builder-hub@0.281.1
23:13:02.697 - warn: Failed to generate graphql operation types vtex.builder-hub@0.281.1
23:13:13.646 - warn: You do not have permission to run tests vtex.builder-hub@0.281.1
23:13:18.516 - info: acctglobal.vtexbook@0.1.0 was published successfully!
23:13:18.517 - info: You can deploy it with: vtex deploy acctglobal.vtexbook@0.1.0
23:13:18.518 - info: Your app documentation will be available at: https://vtex.io/docs/app/acctglobal.vtexbook@0.1.0
```

Após uma solicitação de confirmação e um breve instante, seu app terá sido publicado e você receberá uma mensagem de feedback.

Tendo feito a publicação do app, agora precisamos lançá-lo em produção com o comando `VTEX deploy`, para isto você precisa aguardar um período mínimo obrigatório de testes de sete minutos após a publicação do app.

```
$ vtex deploy
√ Are you sure you want to deploy app acctglobal.vtexbook@0.1.0 ... yes
23:22:38.439 - info: Successfully deployed acctglobal.vtexbook@0.1.0
```

Ao rodar o comando VTEX deploy você também precisará confirmar o comando, e após um pequeno instante, seu app estará em produção.

Feito isto, podemos utilizar o comando VTEX install <vendor.name@major.minor.patch>` para instalar o app no workspace desejado.

```
$ vtex install acctglobal.vtexbook@0.1.0
23:26:12.182 - info: Installed app acctglobal.vtexbook@0.1.0 successfully
```

Caso você esteja com o app publicado aberto em seu terminal, você poderá utilizar o comando VTEX install, sem a necessidade de especificar o app e sua versão. Estas informações serão puxadas automaticamente a partir do manifest.json.

Após publicar um app, e lançá-lo em produção com VTEX deploy, você também pode utilizar o comando VTEX update para que o app seja atualizado para a sua versão mais recente no workspace que está sendo utilizado.

05

Boas práticas

Styleguide VTEX

Dentre as opções que a VTEX nos disponibiliza para facilitar o desenvolvimento, temos o Styleguide VTEX, que seria basicamente uma biblioteca de componentes React, que auxilia muito na hora de construir o layout dos nossos componentes customizados, principalmente pelos componentes seguirem com o padrão de estilização definido no seu arquivo style.json do store-theme.

Com o auxílio dessa biblioteca, podemos manipular inputs, botões, modals, tabs, accordions, entre outros, de uma forma muito simples, pois eles disponibilizam todos esses componentes com uma documentação bem explicada e cheia de exemplos.

Changelog

Para mantermos um histórico das atualizações dos nossos apps, é essencial criarmos um Changelog, pois é onde iremos deixar documentado todas as features feitas e bugs arrumados durante o versionamento da aplicação.

Em relação ao versionamento da aplicação, é bom fazermos ela seguindo o método de versionamento semântico, onde consideramos o número da versão do nosso app, seguindo o seguinte modelo: MAJOR.MINOR.PATCH, onde:

1. versão Maior(MAJOR): quando fizermos mudanças incompatíveis na API,
2. versão Menor(MINOR): quando adicionarmos funcionalidades, mas mantemos a compatibilidade, e
3. versão de Correção(PATCH): quando corrigimos algumas falhas e mantemos compatibilidade.

Já para a construção do Changelog em si, podemos considerar esse exemplo:

Changelog
All notable changes to this project will be documented in this file.
The format is based on [Keep a Changelog](https://keepachangelog.com/en/1.0.0/), and this project adheres to [Semantic Versioning](https://semver.org/spec/v2.0.0.html).

[Unreleased]
[0.4.0] - 2021-08-03
Changed
- Feature XXX
[0.3.0] - 2021-07-26
Added
- Feature XXX
[0.2.0] - 2021-07-12
Added
- Feature XXX
[0.1.0] - 2021-06-23
Added
- First release

Documentação

Com o uso dos apps que a VTEX nos disponibiliza, percebemos que a grande maioria possui uma documentação, pois é muito importante que consigamos passar para o próximo, o principal ponto que nosso aplicativo entrega, sem que ele tenha que ler o código e entender por conta própria, já que você não estará disponível 100% do seu tempo.

É muito comum que os projetos VTEX evoluam constantemente e provavelmente você não ficará nele do começo até o final de toda a fase de implantação, evolução e suporte. E nesse meio tempo, pessoas entrarão e sairão do projeto e do mesmo jeito que você pode entrar no meio de um projeto em curso uma documentação bem feita, facilita muito esse processo.

Como base, podemos pegar as documentações da VTEX, onde possui vários exemplos de como podemos construir uma, então podemos considerar os principais pontos para uma boa documentação:

- Título
- Descrição
- Imagem
- Exemplo de configuração na loja
- Exemplo de uso
- Exemplo de todos os componentes do aplicativo
- Explicação de cada propriedade de cada componente
- Citação de todas as classes estilizáveis por CSS

Não esqueça de colocar o builder de docs no seu manifest, assim, quando você fizer a publicação do seu app, você terá uma documentação online no site da VTEX, do que foi escrito.

06

Exemplos práticos

Apps de Front-end usando React, Typescript, GraphQL e Jest

Para esse exemplo, iremos construir uma aplicação que seja capaz de guardar o endereço do usuário no OrderForm, assim essa informação poderá ser usada em outras aplicações que dependam da localização do usuário, além de já trazer ela preenchida no checkout.

Como já explicado o processo de iniciar o desenvolvimento de um app customizado, vamos começar pela criação de um contexto para ser consumido pelos componentes do aplicativo, primeiramente vamos criar uma pasta context dentro da pasta react e criar nosso primeiro arquivo para desenvolvimento.

Nesse arquivo, trataremos principalmente as informações do OrderForm, como pegar as informações atuais e também, atualizar as existentes, especificamente, o endereço do usuário. Para pegar as informações do OrderForm, usaremos o hook useOrderForm do vtex.order-manager/OrderForm. Exemplo:

```
const {
orderForm: { id: orderFormId },
orderForm,
loading
} = useOrderForm()
```

Para atualizarmos o endereço que o usuário nos disponibilizar por qualquer método escolhido, como um formulário pedindo os campos necessários, uma API que você passe o CEP e ele retorna os dados do endereço, o input de autocomplete da Google etc, usaremos uma mutation disponibilizada pela VTEX, chamada updateOrderFormShipping, que pode ser encontrada no app vtex.store-graphql.

```
GraphQL
mutation UPDATE_ORDER_FORM_SHIPPING(
  $orderFormId: String
  $address: OrderFormAddressInput
) {
  updateOrderFormShipping(orderFormId: $orderFormId, address: $address)
    @context(provider: "vtex.store-graphql") {
    cacheId
  }
}
```

Para realizarmos essa mutation, precisaremos passar o id do Order Form, que pegamos da desconstrução do useOrderForm() e as principais informações de um endereço, como: bairro, cidade, código postal, estado, país e rua, outras informações interessantes para enviar para essa mutation, seriam: um id para o endereço que está sendo enviado, como um número aleatório; o tipo desse endereço, como por exemplo, residencial e as coordenadas dessa localização, que podem ser obtidas através do uso de alguma API. Uma boa prática, é não passarmos o número da casa para o OrderForm, pois assim não pulamos a etapa do endereço do checkout, mesmo que ela já venha com a maior parte preenchida.

Tendo o controle do status do carregamento das informações presentes no OrderForm, podemos pegar as informações do endereço dele, que iremos exibir no front, e guardar em algum estado que criaremos. Levando em consideração que iremos usar apenas a cidade e o estado, podemos fazer isso da seguinte forma:

```
const [addressInfo, setAddressInfo] = useState<AddressInfo | null>(null)

useEffect(() => {
  if (loading) {
    return
  }

  const {
    shipping: { selectedAddress },
  } = orderForm

  if (selectedAddress) {
    setAddressInfo({
      city: selectedAddress.city,
      state: selectedAddress.state,
    })
  }
}, [loading])
```

Basicamente, criamos nosso estado e declaramos que ele pode ser do tipo AddressInfo ou vazio, e logo em seguida, como já comentado, temos nosso useEffect que só executa quando tivermos alteração no loading, e caso não esteja carregando, pega as informações do endereço atual, que seria o que está dentro do objeto selectedAddress e caso ele não esteja vazio, guardamos as informações que iremos usar no nosso estado criado.

Nosso estado declarado possui um tipo que não criamos, e para fazermos isso, precisamos declará-lo dentro de um arquivo da pasta typings, que se encontra dentro da pasta react. Criaremos um arquivo chamado, por exemplo, address.d.ts e dentro dele colocaremos a declaração, por exemplo:

```
interface AddressInfo {
  city: string
  state: string
}
```

Agora que já temos o nosso contexto criado, a query configurada e a tipagem feita, vamos criar os componentes referente ao que o usuário irá interagir. Pensando em uma aplicação que use o modal do Styleguide VTEX, no seu trigger para abrir o modal, teremos a cidade e o estado, junto com uma mensagem de alterar e no conteúdo do modal aberto, teremos as informações escolhidas para pegar o endereço do usuário e passar para a query que atualizará o OrderForm. Exemplo:

```
TypeScript
import GeolocatorTrigger from './components/GeolocatorTrigger'
import GeolocatorModal from './components/GeolocatorModal'
import { useGeolocatorContext } from '../context/GeolocatorContextProvider'

const Geolocator: FC = () => {
  const {
    addressInfo,
  } = useGeolocatorContext()

  return (
    <>
      <GeolocatorTrigger
        addressInfo={addressInfo}
      />
      <GeolocatorModal
        addressInfo={addressInfo}
      />
    </>
  )
}
```

Portanto, já temos uma base de como podemos fazer um componente que consiga controlar o campo de endereço do OrderForm, agora um exemplo de como podemos testar os principais componentes do nosso app.

Dentro da pasta react, crie uma pasta chamada __tests__ e dentro dela, um arquivo com o nome do componente que quer testar, por exemplo GeolocatorTrigger.test.tsx.

Dentro desse arquivo, iremos criar a lógica do nosso teste. O teste que iremos fazer, será o de validar se o endereço passado por props para o componente, é o que será exibido na tela do usuário, para isso, dentro do arquivo criado, coloque:

```javascript
import { render } from '@vtex/test-tools/react'
import React from 'react'

import GeolocatorTrigger from '../components/geolocator/components/GeolocatorTrig

describe('GeolocatorTrigger', () => {
  it('Should render the user address', () => {
    const { container, queryByText } = render(
      <GeolocatorTrigger
        addressInfo={{
          city: 'Atibaia',
          state: 'SP',
        }}
      />
    )

    expect(queryByText(`Atibaia - SP`)).toBeTruthy()
    expect(container.firstChild).toMatchSnapshot()
  })
})
```

Como visto, fizemos a importação do react, da ferramenta de testes da VTEX e do nosso componente, logo em seguida declaramos a seguinte função describe, ela serve para descrever qual componente estamos testando, uma boa prática é passar o nome do seu componente. Logo em seguida temos a função it, que usamos para explicar o que estamos testando do nosso componente, que no caso, é o endereço passado para ele.

Para testarmos nosso componente, iremos usar a função queryByText, que serve para procurarmos um texto em específico na renderização do nosso componente, e a variável container, que usaremos para guardar um snapshot do código executado. Logo após, temos as funções expect e toBeTruthy, respectivamente, pela própria tradução, são funções que passaremos o que esperamos e o retorno disso, tem que ser verdade, caso seja, o teste passará.

Portanto, após esse exemplo de como desenvolver um aplicativo usando React, GraphQl, Typescript e Jest, você poderá começar a se aventurar no desenvolvimento de aplicativos customizados com boa qualidade de código!

Apps de Back-end usando Node, Typescript e GraphQ

Neste módulo iremos desenvolver um app simples, utilizando os builders Node e GraphQl,para consumirmos uma API da ViaCEP, e que possamos pesquisar um endereço através do CEP fornecido.
Primeiramente, devemos levar em consideração que os builders de node e GraphQl, não são abertos para utilização, para utilizá-los você irá precisar enviar um pedido via google form para a VTEX neste link, a liberação é passível de aprovação e pode levar algum tempo, é importante que você descreva com exatidão a sua necessidade para a utilização dos builders.
Para seguir com este exercício, irei utilizar o vendor VTEX para desenvolver e linkar o app em ambiente de desenvolvimento.
Para começar, podemos clonar o boilerplate graphql-example. O próximo passo seria alterarmos as informações do app nos arquivos manifest.json, package.json e CHANGELOG.md.
Tendo feito isto, podemos começar criando um client para consumir a API que iremos utilizar neste exercício, dentro da pasta clients, que está dentro da pasta node, deverá ser criada uma pasta para o nosso novo client, neste exemplo sendo chamada de searchAddress, e dentro dela, iremos criar o arquivo index.ts. Neste arquivo iremos escrever o construtor de nosso client, com as informações da API que irá ser consumida, e a função responsável por fazer a requisição, com seus respectivos parâmetros:

```
import type { InstanceOptions, IOContext } from '@vtex/api'
import { ExternalClient } from '@vtex/api'

export class SearchAddressClient extends ExternalClient {
  constructor(context: IOContext, options?: InstanceOptions) {
    super('https://viacep.com.br', context,
      {...options,
        headers: {
          'X-Vtex-Use-Https': 'true',
          }
      });
  }

  public getAddress = async (postalCode: string) => {
    try {
      const data = await this.http.get(`/ws/${postalCode}/json/`);

      return data;
    } catch (error) {
      return error;
    }
  }
}
```

O próximo passo, é adicionar a rota da API que irá ser utilizada no manifest.json, na seção policies:

```
{} manifest.json > ...
  },
  "scripts": {
    "prereleasy": "bash lint.sh"
  },
  "credentialType": "absolute",
  "policies": [
    {
      "name": "colossus-fire-event"
    },
    {
      "name": "colossus-write-logs"
    },
    {
      "name": "graphql-query"
    },
    {
      "name": "outbound-access",
      "attrs": {
        "host": "viacep.com.br",
        "path": "/ws/*"
      }
    }
  ],
  "$schema": "https://raw.githubusercontent.com/vtex/node-vtex-api/master/gen/manifest.schema"
}
```

Feito isto, é necessário importar o client criado, no arquivo index.ts, da pasta clients e criar um método público para acessá-lo, isto fará que ele seja acessível através do parâmetro context do node:

```
import { IOClients } from '@vtex/api'

import { BookClient } from './book'
import { MarkdownClient } from './markdown'
import { SearchAddressClient } from './searchAddress'

export class Clients extends IOClients {
  public get book() {
    return this.getOrSet('book', BookClient)
  }

  public get markdown() {
    return this.getOrSet('markdown', MarkdownClient)
  }

  public get searchAddress() {
    return this.getOrSet('searcAddress', SearchAddressClient)
  }
}
```

Agora, com os passos anteriores feitos, podemos criar o resolver que irá ser utilizado para fazermos uma chamada utilizando o graphQL. Dentro da pasta resolvers, criaremos o arquivo searchAddress.ts, e criaremos a função que irá requisitar o client:

```
export const searchAddressByPostalCode = async (
  _: any,
  args: any,
  {clients: { searchAddress }}: Context
) => {
  const data = await searchAddress.getAddress(args.postalCode)
  return data
}
```

Perceba na imagem acima que o client searchAddress, é importado através do context do node.

Feito isto, iremos importar nosso resolver no arquivo index.ts da pasta node e declará-lo como uma query, se a sua API utiliza método de POST, PUT ou PATCH, você irá adicioná-la como uma mutation.

```
import { newbook } from './resolvers/newbook'
import { source } from './resolvers/source'
import { total } from './resolvers/total'
import { searchAddressByPostalCode } from './resolvers/searchAddress'

const MEDIUM_TIMEOUT_MS = 2 * 1000

// Export a service that defines resolvers and clients' options
export default new ServiceClients, RecorderState, ParamsContext>({
  clients: {
    implementation: Clients,
    options: {
      default: {
        timeout: MEDIUM_TIMEOUT_MS,
      },
    },
  },
  graphql: {
    resolvers: {
      Book: {
      },
      Mutation: {
      },
      Query: {
        book,
        books,
        source,
        total,
        searchAddressByPostalCode,
      },
    },
  },
})
```

Agora, na pasta GraphQL, na root do projeto, no arquivo schema.graphql, precisaremos declarar a query, seguindo o mesmo nome do resolver, e também declarar os tipos dos parâmetros e os retornos da query:

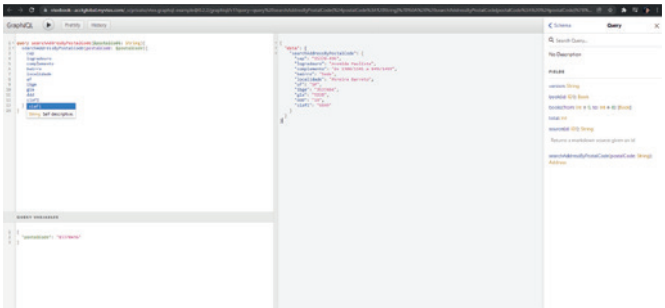
```
type Address {
  cep: String
  logradouro: String
  complemento: String
  bairro: String
  localidade: String
  uf: String
  ibge: String
  gia: String
  ddd: String
  siafi: String
}

type Query {
  # Cache control can only be set at root fields
  book(id: ID!): Book
  books(from: Int = 0, to: Int = 4): [Book] @cacheControl(scope: PUBLIC, maxAge: SHORT)
  total: Int

  ###
  #returns a markdown source given an id
  ###
  source(id: ID!): String
  searchAddressByPostalCode(postalCode: String): Address

type Mutation {
  # You should always return the whole object in a mutation, not only a boolean. This will help in browser caching
  delete(id: String): Boolean
  editBook(id: ID!, book: BookInput!): Book
  newBook(book: BookInput!): Book
}
```

Perceba que o type Address foi criado no mesmo arquivo, para tipar o retorno da query e os valores da mesma. Feito isto, você poderá linkar o seu app e no próprio terminal, a CLI irá gerar um link para que você possa testar sua query no ambiente GraphQL IDE:

```
Address was added.
Query.searchAddressByPostalCode was added. vtex.builder-hub@0.281.1
23:16:00.578 - info: Link updated service-node@6.34.6
23:16:00.586 - info: Fetching /apps/vtex.graphql-example@0.2.2/bundle/dist/service-node/app service-node@6.34.6
23:16:00.588 - info: Fetching /apps/vtex.graphql-example@0.2.2/files/dist/service-node/dependencies.tar.zst service-node@6.34.6
23:16:00.718 - info: OK: /apps/vtex.graphql-example@0.2.2/files/dist/service-node/dependencies.tar.zst service-node@6.34.6
23:16:00.720 - info: OK: /apps/vtex.graphql-example@0.2.2/bundle/dist/service-node/app service-node@6.34.6
23:16:00.753 - info: Extracted /apps/vtex.graphql-example@0.2.2/files/dist/service-node/dependencies.tar.zst service-node@6.34.6
23:16:00.782 - info: Extracted /apps/vtex.graphql-example@0.2.2/bundle/dist/service-node/app service-node@6.34.6
23:16:00.921 - info: Bundle: 53 files - 8.2MB vtex.builder-hub@0.281.1
23:16:00.934 - info: Sent bundle to Apps in 1169ms vtex.builder-hub@0.281.1
23:16:00.941 - info: App linked successfully vtex.builder-hub@0.281.1
          - info: You can try out the queries in this app using the GraphiQL IDE available at:
https://vtexbook--acctglobal.myvtex.com/_v/private/vtex.graphql-example@0.2.2/graphiql/v1
23:16:00.980 - info: Node app build finished successfully vtex.builder-hub@0.281.1
23:16:01.803 - info: Runtime @vtex/api is: /usr/local/app/node_modules/@vtex/api/lib/index.js service-node@6.34.6
23:16:01.804 - info: Using @vtex/api from: /usr/local/app/node_modules/@vtex/api/lib/index.js service-node@6.34.6
```

Se todos os passos foram seguidos corretamente, você poderá ver a sua query searchAddressByPostalCode, na guia Docs e em seguida em query, na lateral direita da página:

Feito isto, podemos montar nossa query e testá-la:



Agora você poderá utilizar as queries e mutations criadas utilizando os builders de node e graphql, no front-end, através dos hooks useQuery e useLazyQuery. Lembrando que para utilizar as queries em seu app de front-end, você irá precisar estar com o app em que as queries foram desenvolvidas linkado ou instalado e também como dependência em seu manifest.json.

Edition App

Previamente falamos sobre os edition app da VTEX, mas também é possível desenvolver seu próprio edition app. Cada workspace pode ter apenas um edition app instalado por vez e seu objetivo é instalar uma lista de apps neste ambiente. O principal caso de uso para a utilização de um edition app é permitir a instalação de apps não públicos (e gratuito) em accounts diferentes, sem que seja necessário haver um custo de aquisição ou mensalidade do app. Seria possível neste cenário publicar o mesmo app com vendors diferentes, mas isso gera um esforço de manutenção significativamente maior.

Edition App

Para isso é necessário que uma conta do tipo sponsor seja a publicadora do app em questão e também a publicadora de edtion app. Isso permitirá que o app seja instalado em qualquer conta que utilize aquele edition app.

A solicitação para que uma conta VTEX se torne uma conta sponsor deverá ser realizada por meio de ticket, no canal de atendimento da VTEX.

É essencial que o novo edition app tenha o próprio app de edition store da VTEX como dependência, pois nele temos tudo o que é essencial para o funcionamento do VTEX IO. Uma vez que o edition app esteja publicado e seu deploy tenha sido realizado, será necessário abrir um novo ticket solicitando a instalação do edition app nas contas desejadas.

Estruturalmente um edition app é extremamente simples, tendo apenas um arquivo apps.json na pasta edition com a lista de todos os apps que fazem parte daquela edição. Lá os apps devem conter seu vendor, seu nome e default major a ser instalada.

```
{
    "apps": {
        "acctglobal.store-theme": {
```

Pixel App

Em VTEX IO não temos acesso a um arquivo HTML dos templates implementados e nem um módulo específico para a gestão ou adição de arquivos de script no formato javascript para realizar a integração com soluções de terceiros como o próprio tagueamento. Para isso temos os pixel apps, que possuem um builder adequado para realizar a inclusão de scripts na tag <head> das páginas.

Diversas soluções já possuem apps prontos que podem ser encontrados na app store, como o Google Tag Manager, o Facebook Pixel, o Zendesk Pixel e o Hotjar. Ao realizar a implantação de uma nova solução para o seu site, consulte a seção de Pixel Apps dos documentos de desenvolvedor da VTEX para verificar se um pixel app adequado já foi desenvolvido.

Caso um pixel app da solução desejada não seja encontrado será necessário desenvolver seu próprio pixel app. O primeiro passo para isso é solicitar o whitelist do builder de pixel no formulário de closed beta da VTEX, já citado previamente. Tendo a liberação do builder de pixel realizada, será possível iniciar o desenvolvimento do seu app.

O primeiro passo para o desenvolvimento é a utilização do pixel app template, disponibilizado no github vtex-apps, para servir de base para o seu repositório. Dentro da pasta pixel, você encontrará o arquivo head.html que é o responsável por incluir o script que será chamado:

```
<script>
  (function() {
    var appId = "{{settings.gtmId}}";
    if (!appId) {
      console.error('Warning: No APP NAME ID is defined. Please configure it in
      return
    }

    // Add script here
  })()
</script>
```

Basta alterar o conteúdo da tag script para a chamada do arquivo que deseja implementar.

Neste template, há também um exemplo de utilização do settingsSchema no arquivo manifest.json, onde na seção de apps do admin é possível incluir uma key do solução aplicada, de modo que não fique hard coded:

```
"settingsSchema": {
    "title": "YOUR APP NAME",
    "type": "object",
    "properties": {
      "gtmId": {
        "title": "SAMPLE FIELD",
        "description": "Enter the ID (GTM-XXXX) from your Google Tag Manager",
        "type": "string"
      }
    }
  }
```

Dentro da pasta store, é necessário também realizar alterações nos arquivos interfaces.json e plugins.json. Nestes arquivos é necessário incluir o nome do app que está sendo desenvolvido como plugin. Caso o nome do app fosse “test-pixel” ele ficaria da seguinte forma:

interfaces.json

```JSON
{
  "pixel.test-pixel": {
    "component": "index"
  }
}
```

plugins.json

```
{
  "pixels > pixel": "pixel.test-pixel"
}
```

Os apps de pixel também podem utilizar o builder de React, principalmente para taguear novos eventos, caso desejável. Dentro do pixel-app template, no arquivo events.d.ts há exemplos de sua aplicação, como por exemplo o evento de product impression:

```
export interface Impression {
  product: ProductSummary
  position: number
}
```

Consumindo informações no seu App

Componente editável pelo Site Editor

Quando construímos nossos apps, podemos ter a opção de deixá-los customizáveis pelo Site Editor, e para isso, precisamos adicionar alguns pontos no nosso aplicativo, como criar o nosso componente sendo do tipo StorefrontFC e configurar um schema para ser consumido por ele.

Para esse exemplo, faremos um aplicativo que exibirá uma mensagem definida pelo Site Editor, então podemos fazer isso considerando o seguinte trecho de código:

```JavaScript
import * as React from 'react'

export interface ExampleProps {
  text: string
}

const ExampleComponent: StorefrontFC<ExampleProps> = ({
  text,
}) => {
  return (
    <div>
      {text}
    </div>
  )
}

ExampleComponent.schema = {
  title: 'Example',
  type: 'object',
}

export default ExampleComponent
```

Após ter definido o tipo do componente e declarado o uso do schema nele, precisamos criar a interface StorefrontFC e as configurações do schema ExampleComponent. Dentro de um arquivo da pasta typings, a interface pode ser declarada da seguinte forma:

```
import { FunctionComponent, ReactElement } from 'react'

declare global {
  interface StorefrontFC<P = {}> extends FunctionComponent<P> {
    getSchema?(props: P): object
    schema?: object
  }

  interface StorefrontComponent<P = {}, S = {}> extends Component<P, S> {
    getSchema?(props: P): object
    schema: object
  }
}
```

Já o schema, é configurado dentro de um arquivo da pasta store, chamado contentSchemas.json, nesse arquivo declaramos qualquer schema que criaremos para os nossos componentes, nele passamos o nome do schema, e os campos que serão guardadas as informações para a exibição no nosso componente, exemplo:

```
{
  "definitions": {
    "Example": {
      "title": "Examplo de componente editável pelo Site Editor",
      "type": "object",
      "properties": {
        "monday": {
          "title": "Texto a ser renderizado",
          "type": "string",
          "default": ""
        }
      }
    }
  }
}
```

Dessa forma, ao linkar seu aplicativo e verificá-lo no Site Editor, você encontrará a aba dele e poderá definir uma mensagem que será editável a qualquer momento pelo admin da loja, sem a necessidade de uma release para mudar alguma informação. Lembrando que isso pode ser usado de várias maneiras, não apenas para mensagens.

Componente editável pela seção de gestão de apps

Podemos usar o mesmo exemplo anterior, mas dessa vez, configurando pela interface da seção de gestão de apps. Recomenda-se usar esse método, quando temos que passar algum campo variável que seja igual para a loja inteira, usado em qualquer componente do aplicativo, como por exemplo a chave de alguma API, o token etc.

Então para isso, você irá precisar executar uma query para recuperar as informações guardadas e configurar um schema, para definir os campos usados para guardar as informações necessárias. Começando pela query, usaremos a appSettings do VTEX. apps-graphql, nela passaremos o nome do aplicativo, junto com o vendor e também a versão da major dele, e teremos como retorno, os dados guardados, exemplo:

```
query AppSettings {
  appSettings(app: "vendor.app-name", version: "0.x")
    @context(provider: "vtex.apps-graphql") {
    message
  }
}
```

Já o schema, precisa ser configurado no manifest do aplicativo, seguindo a mesma lógica da configuração anterior, por exemplo:

```
JSON
{
  "vendor": "vendor",
  "name": "app-name",
  "version": "0.x",
  "title": "Componente Exemplo",
  "description": "Componente editável pela Loja de Aplicativo",
  "builders": {
    "react": "3.x",
    "messages": "1.x",
    "styles": "2.x",
    "store": "0.x",
    "sitemap": "0.x",
    "docs": "0.x"
  },
  "settingsSchema": {
    "title": "Exemplo de Schema",
    "properties": {
      "text": {
        "title": "Texto",
        "type": "string",
        "description": "Texto para ser renderizado na tela do usuário",
        "default": ""
      }
    }
  },
  "dependencies": {
    "vtex.apps-graphql": "2.x"
  },
  "registries": [
    "smartcheckout"
  ],
  "policies": [],
  "$schema": "https://raw.githubusercontent.com/vtex/node-vtex-api/master/gen/man
}
```

Com a query e o schema configurados, basta apenas você linkar seu app, que você verá o campo editável na seção de gestão de apps e o que for salvo nele, poderá ser recuperado pela execução da query e usado no seu componente da forma que você preferir.

VTEX

The Enterprise
Digital Commerce
Platform

Os autores da sexta edição foram:

Luiz Guilherme Priolli
Engineering Chapter Leader

Lucas Pacheco
Fullstack Developer

Renan Galan Guerra
Software Developer

VTEX

The Enterprise
Digital Commerce
Platform

VTEX is the first and only global, fully integrated, end-to-end commerce solution with native marketplace and OMS capabilities. We help companies in retail, manufacturing, wholesale, groceries, consumer packaged goods and other verticals to sell more, operate more efficiently, scale seamlessly and deliver remarkable customer experience. Our modern microservices based architecture and our powerful business and developer tools allow VTEX to future-proof our customers' businesses and free them from software updates.