Como usar MVVM no React usando Hooks e TypeScript

I. Visão geral do MVVM
II. Visão geral do aplicativo
III. Implementação View
IV. Implementação ViewModel
V. Implementação Model
VI. Perguntas frequentes

Os aplicativos de software geralmente começam simples e crescem rapidamente. Não importa quão pequeno se tente mantê-lo, o código sempre se transforma em um aplicativo completo. Em muitas linguagens, seguimos padrões estruturais como MVVM e Clean Architecture, o que é menos comum de ver no Velho Oeste da Web.

Neste artigo, daremos uma olhada em como um aplicativo React pode ser projetado para seguir o padrão de design arquitetônico MVVM para criar um aplicativo React que seja escalonável e de fácil manutenção. Usaremos componentes TypeScript e React funcionais e escreveremos hooks personalizados para ViewModel e Model. É claro que o mesmo código funcionará com React Native com algumas pequenas modificações.

Se você tiver alguma dúvida ao ler o artigo, você terá um extenso FAQ no final do artigo.

Visão geral do MVVM

MVVM é um padrão de design que fornece uma separação clara de interesses, facilitando o desenvolvimento, teste e manutenção de aplicativos de software.

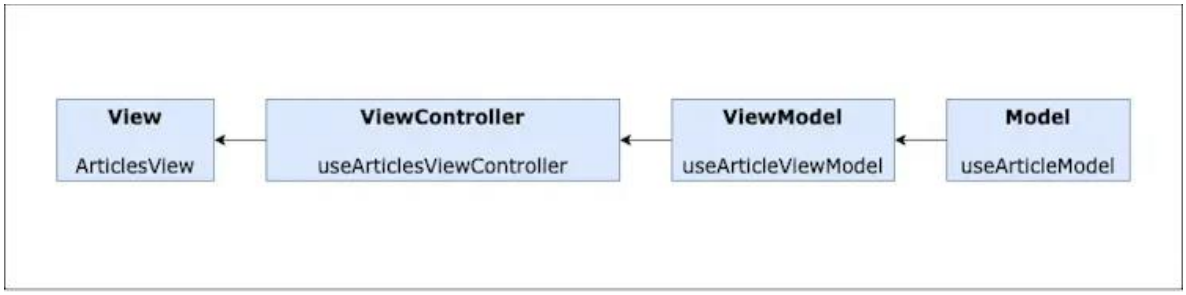
O que o MVVM faz é separar uma aplicação em Views, ViewModels e Models. Enquanto as Views são responsáveis por apresentar dados ao usuário e capturar a entrada do usuário, os ViewModels atuam como mediadores entre as Views e os Modelos, e os Modelos gerenciam os dados do aplicativo.

Visão geral do aplicativo

O aplicativo que construiremos neste artigo é pequeno e que você pode ver aqui. Ele é mantido pequeno para que possamos nos concentrar na arquitetura MVVM. O aplicativo consiste em uma única visualização listando artigos, com um campo de entrada para adicionar um novo artigo com um determinado nome.

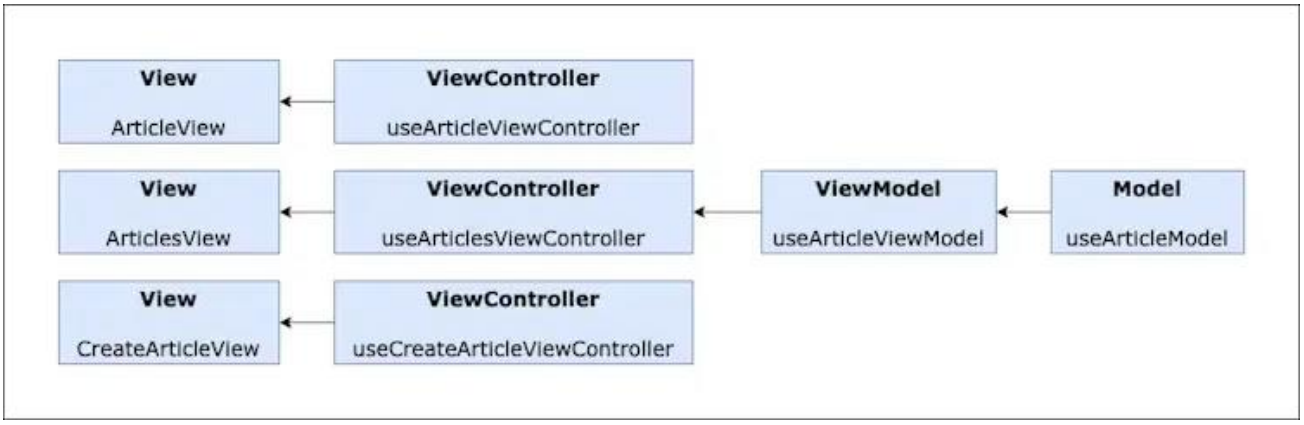
O aplicativo a ser construído possui uma única visualização para mantê-lo simples.

Em termos de código, o aplicativo consiste em um componente React para os artigos View e, em seguida, alguns hooks para ViewController, ViewModel e Model. Abaixo está um diagrama arquitetônico do código que examinaremos. Desconsidere o ViewController por enquanto, que não é uma parte oficial do MVVM, discutiremos isso mais tarde.



Visão geral dos componentes React para o aplicativo neste artigo

É claro que uma única visualização não é suficiente para uma aplicação útil. Se o exemplo fosse maior, cada View teria seu próprio ViewController, conforme ilustrado no diagrama abaixo. Todos os ViewControllers que funcionam no mesmo modelo usariam um ViewModel comum.



Como seria o aplicativo com mais visualizações

Quando a aplicação cresce ainda mais ela pode incluir mais Models e ViewModels. Eu acho que você entendeu. De qualquer forma, vamos começar a analisar a implementação do aplicativo.

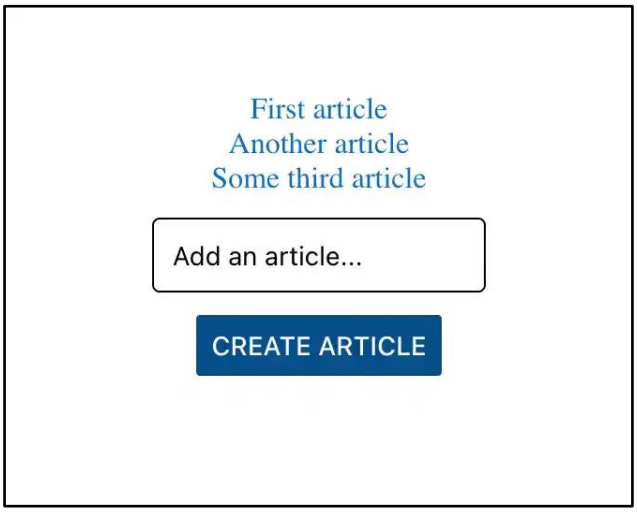
Implementação View

Uma Visualização no aplicativo pode ser qualquer tela ou componente visual da tela. Se o site a ser construído for uma plataforma de blog, poderíamos ter telas para listar artigos, visualizar um único artigo e escrever um novo artigo. Essas telas podem ser implementadas em Views como ArticlesView, ArticleView e CreateArticleView

A Visualização é responsável por exibir a UI e disparar ações com base nas interações do usuário. Embora tradicionalmente não exista um ViewController no MVVM, este guia do MVVM dividiu a parte View do MVVM em duas partes, um View e um ViewController, onde a única responsabilidade do componente View é renderizar a UI enquanto o ViewController lida com todos os lógica para a visualização.

Componente View

Como vimos antes, a View ficará mais ou menos assim com algum CSS:



Componente ArticleView

```
// view/article/articles-view/ArticlesView.tsx

import React from 'react'
import useArticlesViewController from 'view/article/articles-view/useArticlesViewController'

const ArticlesView: React.FC = () => {
  const {
    articleName,
    articles,
    navigateToArticle,
    onArticleNameChange,
    onCreateArticleClick,
  } = useArticlesViewController()

  return (
    <div>
      {!!articles &&
        articles.map(({ id, title }) => (
          <div key={id} onClick={() => navigateToArticle(id)}>
            {title}
          </div>
        ))}
      <div>
        <input
          type="text"
          onChange={onArticleNameChange}
          value={articleName}
          placeholder="Add an article..."
        />
        <button onClick={onCreateArticleClick}>Create article</button>
      </div>
    </div>
  )
}

export default ArticlesView
```

Para implementar o View, usamos um componente React funcional regular que renderiza uma UI. Para evitar o tratamento da lógica de visualização no componente View, usamos um ViewController que exporta funções para todas as ações possíveis do usuário.

O componente ArticlesView acima renderiza uma visualização com artigos. Quando o usuário interage com a IU, este componente View simplesmente encaminha a ação para uma função no hook useArticlesViewController.

Como esse arquivo contém apenas componentes de UI puros e apenas reage às interações do usuário encaminhando-as para um hook, é muito trivial testar e ler. Os testes de unidade para este componente seriam apenas para testar se todos os artigos são renderizados e se as funções exportadas de useArticlesViewController são chamadas. O próprio useArticlesViewController seria preferencialmente ridicularizado, pois testaremos esse hook separadamente.

Para testar as interações com a visualização, eu recomendaria estruturas de teste interacionais como React Testing Library . Você pode ler mais sobre os benefícios e como isso se compara aos testes unitários baseados em código em meu outro artigo sobre testes unitários.

Hook ViewController

O hook ViewController usado no componente View lida com a lógica de visualização dessa View.

Cada componente View possui seu próprio hook ViewController. Existem vários tipos de lógica de visualização no ViewController.

- Lógica de visualização que atualiza o estado interno dos componentes de visualização (normalmente useState, useReducer ou useRef)
- Lógica de visualização que atualiza o estado da aplicação (por exemplo, navegando para outras telas)
- Lógica de visualização que interage com o ViewModel

Em geral, a lógica View é responsável por gerenciar a interface do usuário e lidar com as interações do usuário. Isso inclui a manutenção do estado interno do componente, bem como o tratamento das condições para interagir com o ViewModel. Ao manter todo esse tipo de lógica de visualização no ViewController, não precisamos de nenhuma lógica de visualização no ViewModel.

```
// view/article/articles-view/useArticlesViewController.tsx

import { useCallback, useEffect, useState } from 'react'
import { useHistory } from 'react-router-dom'
import { HOME_SCREEN, ROOM_SCREEN } from 'routing/screens'
import useArticleViewModel from 'viewmodel/useArticleViewModel'

const useArticlesViewController = () => {
  const history = useHistory()
  const [articleName, setArticleName] = useState('')
  const { articles, createArticle, getArticles } = useArticleViewModel()

  const onCreateArticleClick = useCallback(async () => {
    await createArticle({ name: articleName })
  }, [createArticle, articleName])

  const navigateToArticle = useCallback((articleId: number) => {
    history.push(`${ROOM_SCREEN}/${articleId}`)
  }, [history])

  useEffect(() => {
    getArticles()
  }, [getArticles])

  return {
    articleName,
    articles,
    navigateToHome,
    navigateToArticle,
    onCreateArticleClick,
    onArticleNameChange: setArticleName
  }
}

export default useArticlesViewController
```

useArticlesViewController é um pouco mais complexo que seu componente View correspondente. Ele contém código para navegação entre telas usando React Router. Ele também mantém o estado interno da visualização, que pode ser usado tanto para lidar com condições lógicas internas quanto para ser passado de volta ao componente View para ser usado na renderização.

Por último, usamos um hook useArticleViewModel que é o ViewModel que esta View usa. Somente as funções necessárias para serem usadas nesta View precisam ser desestruturadas desse ViewModel.

Este arquivo inclui mais lógica para testar do que o componente View. O bom é que não há necessidade de testar nenhum elemento da UI. Você pode testar o hook puramente usando React Hooks Testing Library , não há necessidade de usar nenhuma correspondência de elemento da UI ou verificar se os dados são renderizados. Simplesmente invoque as funções retornadas do hook e afirme o resultado esperado.

Implementação ViewModel

O ViewModel é responsável por tratar as interações entre a View e o Model, servindo como ponte entre eles. Normalmente, o View interage diretamente com o ViewModel, mas como já vimos, dividimos o View em um componente View e um ViewController. Isso significa que o ViewController é aquele que usa o ViewModel.

O ViewModel é, assim como o ViewController, implementado como um hook. Não inclui muito código em nosso exemplo de artigo. Isso ocorre porque nosso exemplo de aplicação simplesmente não possui nenhuma lógica de negócios. Mas se tivéssemos, teríamos colocado aqui.

```
// viewmodel/useArticleViewModel.tsx
import useArticleModel from 'model/useArticleModel'

const useArticleViewModel = () => {
  const { article, articles, createArticle, getArticles } = useArticleModel()

  return {
    article,
    articles,
    createArticle,
    getArticles
  }
}

export default useArticleViewModel
```

Sim, é isso. Ele apenas agrupa as funções no hook useArticleModel reexportando-as. Se algumas dessas ações exigissem alguma lógica de negócios, teríamos criado uma função wrapper para realizar os cálculos esperados dentro deste hook.

E quanto aos testes? Bem, ao olhar para isso, eu diria que não há necessidade de testar este arquivo. Com mais lógica, você também pode usar a biblioteca de testes React Hooks aqui.

Implementação Model

O modelo é responsável por obter dados de fontes de dados e servi-los ao ViewModel. O ViewModel, por sua vez, usa o Model para buscar e modificar os dados.

O modelo pode ser escrito de várias maneiras. Redux, Apollo, hooks como useSWR e assim por diante podem ser usados. Além disso, pode não haver apenas uma única fonte de dados; aplicativos complexos podem buscar dados de diversas fontes diferentes. Nesse caso, pode ser uma boa ideia colocar um repositório entre o ViewModel e o Model. Os repositórios não fazem parte do padrão de design MVVM e, portanto, não serão usados aqui.

Como a implementação do modelo pode variar significativamente entre projetos, este guia não se aprofundará em como implementá-lo. Em vez disso, forneceremos um exemplo simples que utiliza funções da API REST para recuperar e postar artigos.

```
// model/useArticleModel.tsx

import { useCallback, useState } from 'react'
import { getAllArticles, postArticle } from 'model/api/article'
import { ArticleDTO, CreateArticleDTO } from 'model/api/article'

const useArticleModel = () => {
  const [articlesData, setArticlesData] = useState<ArticleDTO[] | null>(null)

  const getArticles = useCallback(async () => {
    const articles = await getAllArticles()
    setArticlesData(articles)
  }, [])

  const createArticle = useCallback(async (createData: CreateArticleDTO) => {
    if (Array.isArray(articlesData)) {
      const response = await postArticle(createData)

      if (response !== null) {
        setArticlesData([...articlesData, { id: response.id, name: response.name }])
      }
    }
  }, [articlesData])

  return {
    articles: articlesData,
    createArticle,
    getArticles
  }
}

export default useArticleModel
```

Como você pode ver, há mais lógica nisso. Ainda assim, fomos breves. A lógica não é muito inteligente ou robusta, mas podemos perceber que temos funcionalidades básicas para buscar e criar artigos. O código deve ser facilmente testado de maneira semelhante aos hooks anteriores.

REST API

No exemplo do modelo acima, não vimos como as funções da API REST foram implementadas. Mesmo que não seja necessário para o exemplo MVVM, ainda veremos alguns exemplos de código triviais para ver como ele pode ser.

Este código provavelmente não terá a mesma aparência em seu projeto, ele está aqui simplesmente para servir de exemplo. Esperamos que seu código real seja mais bem escrito do que isso; normalmente, você precisaria lidar com coisas como erros, retiradas, armazenamento em cache e tornar o código idempotente.

```
// model/api/article.ts

export interface ArticleDTO {
  id: number
  name: string
}

export interface CreateArticleDTO {
  name: string
}

const API_URL = 'https://example.com/api'

export const getAllArticles = async (): Promise<ArticleDTO[] | null> => {
  const response = await fetch(`${API_URL}/articles`)
  if (!response.ok) {
    throw new Error('Failed to get all articles')
  }
  const data = await response.json()
  return data
}

export const postArticle = async (createData: CreateArticleDTO): Promise<ArticleDTO | null> => {
  const response = await fetch(`${API_URL}/articles`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(createData)
  })
  if (!response.ok) {
    throw new Error('Failed to create article')
  }
  const data = await response.json()
  return data
}
```

Perguntas frequentes

P : O que é uma visualização?

Uma visualização é uma parte substancial da IU. Pode ser a viewport completa ou pode ser um componente dentro da viewport que deve lidar com seu próprio estado e seus próprios dados. Não sei de que maneira você trabalhou com contêineres e componentes no React, ou componentes inteligentes e burros ou como você os chama. Mas se você pensar em um componente como um componente burro, isso não é uma Visualização, é apenas uma parte de uma Visualização. Botões, campo de entrada e pequenos componentes são todos componentes, não visualizações.

P : Por que existe um ViewController na implementação do MVVM?

O ViewController é adicionado como parte do View para separar a implementação da interface do usuário de sua lógica. Isso permite que a Visualização seja testada mais facilmente, pois a lógica da visualização e a UI podem ser testadas separadamente. A desvantagem é que temos mais componentes para testar, renderizar e manter.

Outro motivo para adicionar essa camada extra é servir como uma ponte entre o View e o ViewModel. Ao fazer isso, podemos manter mais lógica de visualização dentro da camada View, em vez de espalhá-la para o ViewModel, que é compartilhado por vários Views. Não há necessidade de um ViewModel saber sobre o estado interno do View que o utiliza.

P : Eu realmente preciso de um ViewController para cada componente do View?

Isso depende do que você pensa quando diz Visualizar. Leia a pergunta O que é uma visão? . Se o componente que você está escrevendo não for considerado uma View, você não precisará, e nem deveria, escrever um ViewController para ele. Definitivamente, é um exagero adicionar um ViewController para um componente de botão trivial. Se o componente que você implementa realmente for uma Visualização, sugiro adicionar sua lógica de visualização a um ViewController para fins de teste e para ser consistente com a forma como você implementa suas Visualizações.

P : Como implementar MVVM no React sem um ViewController?

Se você não quiser dividir o View em View e ViewController, simplesmente coloque toda a lógica do View no ViewController no próprio componente View. A outra opção seria colocar parte ou toda essa lógica no ViewModel, mas o ViewModel conteria muita lógica de visualização específica para determinadas visualizações, o que o tornaria confuso e bastante grande.

P : Por que usamos hooks para implementar o ViewController?

O ViewController pode ser implementado de várias maneiras. Poderíamos, por exemplo, implementar View e ViewController como dois componentes, onde o componente ViewController seria o componente principal que simplesmente passa acessórios para o componente View. Essa solução aumentaria o DOM virtual e possivelmente até físico, pois exigiria dois componentes em vez de um.

P : Podemos implementar MVVM no React usando componentes de classe antigos?

Sim claro. Mas neste artigo usamos componentes funcionais. Os componentes da classe são antigos e não devem mais ser usados.

P : Vi o MVVM sendo implementado usando injeção de dependência no React.

Bom para você. Já vi componentes de classe usando essa abordagem, que se adapta bem a esse caso. A implementação do MVVM no React com componentes funcionais vem com outras soluções. Os hooks são mais familiares aos desenvolvedores do React do que os padrões de injeção de dependência. Por esse motivo, este exemplo usa hooks React para implementar MVVM.

P : Meu aplicativo React MVVM deve incluir apenas um único modelo?

Provavelmente não. Você provavelmente terá vários modelos em seu aplicativo. Cada modelo também virá com um ViewModel que utiliza o modelo.

P : Meu aplicativo React MVVM deve incluir apenas um único ViewModel?

Não. Cada modelo terá um ViewModel. Se você tiver vários modelos, também poderá ter vários ViewModels.

P : Devo ter um ou vários ViewController para cada ViewModel?

Você deve ter um ViewController para cada View. Vários ViewControllers usarão o mesmo ViewModel. ViewControllers como useArticleViewController, useArticlesViewController e useCreateArticleViewController compartilharão o mesmo ArticleViewModel.

P : Posso usar apenas um único ViewModel em um ViewController?

Não, você pode usar muitos. A relação ViewController-ViewModel é uma relação muitos para muitos, não uma relação muitos para um. Uma View pode usar dados de vários modelos e, nesse caso, usará vários ViewModels. Por exemplo, um useArticleViewController pode utilizar um ArticleViewModel e um AuthorViewModel.

P : Por que você agrupa a função createArticle em onCreateArticleClick no ViewController?

Isso é feito por dois motivos. O primeiro motivo é apenas para convenções de nomenclatura, para manter a lógica de visualização, como manipulação de cliques no ViewController e não no ViewModel. A segunda razão é porque o ViewController tem acesso ao estado interno da View. Isso torna possível atualizar onCreateArticleClick com lógica da visualização, como texto de campos de entrada.

P : Tenho um componente que não precisa do Model, ainda preciso usar um ViewModel no ViewController?

Se eu respondesse sim, o que você escreveria aí? É totalmente normal e até normal ter componentes sem ViewModels.

P : Ainda preciso de bibliotecas de teste como React Test Renderer, Enzyme e React Testing Library?

Sim e não. Você ainda tem UI para testar. Mas o único lugar onde você precisa de bibliotecas de teste de UI como essas são nos arquivos View, e todos consistem em elementos puros de UI, que você provavelmente não precisa testar. Os hooks para ViewController, ViewModel e Model podem ser testados com React Hooks Testing Library. O que há de tão incrível nisso? Bem, olhe a pequena documentação ! É tudo que você precisa, honestamente.

P : Qual estrutura de pastas você recomendaria?

Você pode ver a estrutura de pastas que usei neste exemplo observando as importações e os nomes dos arquivos. Em geral, eu diria que existem várias estruturas de pastas que funcionam, use qualquer abordagem que você (e pelo menos alguns de seus colegas...) considere lógica.

P : E se eu não quiser usar TypeScript?

Bem, isso é com você. Basta remover os tipos. Eles estão lá apenas para ajudar.

P : E se eu usar o React Native?

Você também pode usar o mesmo código para React Native com apenas algumas pequenas modificações.

P : Como o MVVM ajuda nos testes?

Duas coisas decidem quanto os testes irão sobrecarregar ou economizar você. Uma coisa é a arquitetura do código. O MVVM irá, por design, tornar o código mais fácil de testar, separando as preocupações de cada camada do aplicativo.

A outra coisa que ajuda você é como você estrutura seus testes e quais bibliotecas você usa para escrevê-los .

P : Devo usar MVVM no React?

A resposta curta seria: com a arquitetura baseada em componentes e ferramentas como Redux e useSWR não temos grande necessidade de uma estrutura MVVM no React.

Com isso dito, muitos desenvolvedores de backend perguntam sobre como implementar MVVM com React. O benefício que vejo com o MVVM é principalmente nos testes. O teste muitas vezes representa metade do tempo de desenvolvimento e, para novos desenvolvedores, pode ser muito mais do que isso.

O problema de testar componentes com muita lógica mista é que isso requer muitos simulados e asserções diferentes para um único componente. O que geralmente acontece é que os desenvolvedores pesquisam o código na esperança de encontrar simulações semelhantes que possam reutilizar.

Usando a configuração MVVM que descrevi neste artigo, o teste é muito fácil. Cada arquivo terá apenas alguns tipos de testes que precisam ser escritos. Para aprender como testar o código, os desenvolvedores só terão que ver como isso é feito em um arquivo adjacente e poderão escrever rapidamente os testes para todo o código que desenvolverem.

No final das contas é só uma questão de gosto. MVVM não é o método principal do React, mas com este artigo eu queria mostrar uma maneira de fazer isso.

Referência

Artigo traduzido e adaptado: How To Use MVVM in React Using Hooks and TypeScript, Dennis Persson, em https://www.perssondennis.com/articles/how-to-use-mvvm-in-react-using-hooks-and-typescript, acesso em 22/05/2024.