Arquitetura

Architecture is about the important stuff. Whatever that is. – Ralph Johnson

Os padrões de Arquitetura de Desenvolvimento Web mais utilizados hoje são a:

> Arquitetura Cliente-Servidor

> Arquitetura MVC (Model-View-Controller)

> Arquitetura SPA (Single-Page Application)

> Arquitetura Serverless

> Arquitetura Microserviços

> Arquitetura RESTful (Representational State Transfer)

E os padrões de Arquiteturas de Desenvolvimento Mobile mais utilizados são a:

> Arquitetura MVC (Model-View-Controller)

> Arquitetura MVVM (Model-View-ViewModel)

> Arquitetura MVP (Model-View-Presenter)

> Clean Architecture

> Flux/Redux

> Arquitetura VIPER

> Arquitetura Flutter BLoC

> Arquitetura SwiftUI

___

Fundamentos detalhados

O padrão MVC (Model-View-Controller) no desenvolvimento web e mobile segue a mesma ideia fundamental, mas existem algumas nuances em sua implementação específica para cada plataforma. Vamos detalhar o padrão MVC em cada contexto:

Desenvolvimento web e mobile
Arquitetura de Software Moderna - Prof. Ingrid Chaves Carneiro Greco

Desenvolvimento Web:

Model (Modelo): O Modelo contém a lógica de negócios e gerencia os dados do aplicativo. Ele representa as regras e operações relacionadas aos dados, como validação, manipulação, consulta e persistência. No desenvolvimento web, o Modelo geralmente inclui a camada de acesso a banco de dados ou serviços de API.

View (Visualização): A View é responsável por renderizar a interface do usuário e exibir as informações do Modelo. Ela é a camada responsável pela apresentação dos dados ao usuário. No desenvolvimento web, a View geralmente consiste em arquivos HTML, CSS e, em alguns casos, templates ou componentes.

Controller (Controlador): O Controlador é responsável por receber as requisições do usuário, interagir com o Modelo apropriado e atualizar a View correspondente. Ele lida com a lógica de controle e coordena as ações do usuário. No desenvolvimento web, o Controlador é implementado como uma classe ou função que recebe as requisições HTTP, processa-as, realiza as operações necessárias no Modelo e seleciona a View adequada para renderizar a resposta.

O fluxo típico de interação no MVC para desenvolvimento web ocorre da seguinte maneira: o usuário interage com a interface do usuário (View), como enviar um formulário ou clicar em um link. A View captura essa interação e envia uma requisição para o Controlador. O Controlador processa a requisição, atualiza o Modelo, realiza qualquer lógica necessária e seleciona a View apropriada para renderizar a resposta. A View recebe os dados atualizados do Modelo e os apresenta ao usuário.

Desenvolvimento Mobile:

O padrão MVC no desenvolvimento mobile segue uma estrutura similar ao desenvolvimento web, mas com algumas adaptações para a plataforma mobile. Vamos detalhar a abordagem mais comum:

Model (Modelo): O Modelo é responsável pela lógica de negócios, manipulação de dados e gerenciamento do estado do aplicativo. Ele representa as regras de negócio e os dados subjacentes. No desenvolvimento mobile, o Modelo geralmente inclui acesso a dados, chamadas

de API e manipulação de armazenamento local.

View (Visualização): A View no contexto mobile é responsável pela interface do usuário e pela exibição dos dados do Modelo. Ela é a camada que renderiza os elementos da interface, como botões, campos de texto, listas e outros componentes visuais. No desenvolvimento mobile, a View é implementada usando elementos de interface nativos ou componentes específicos do framework utilizado.

Controller (Controlador): O Controlador é responsável por receber as interações do usuário na View, processá-las e atualizar o Modelo conforme necessário. Ele coordena as ações do usuário e lida com a lógica de controle. No desenvolvimento mobile, o Controlador é implementado como uma classe ou componente que gerencia as interações da View e atualiza o Modelo.

O fluxo de interação no MVC para desenvolvimento mobile ocorre de maneira semelhante ao desenvolvimento web: o usuário interage com a interface do usuário (View), a View captura a interação e envia-a para o Controlador. O Controlador processa a interação, atualiza o Modelo e notifica a View para exibir os dados atualizados.

Vale ressaltar que existem variações e adaptações do padrão MVC específicas para cada plataforma e framework utilizado no desenvolvimento web e mobile. Essas adaptações podem ter diferentes nomenclaturas, como MVP (Model-View-Presenter) ou MVVM (Model-View-ViewModel), mas a ideia central de separar responsabilidades em Model, View e Controller permanece consistente.

O padrão MVVM (Model-View-ViewModel) é um padrão de arquitetura de software que se tornou popular no desenvolvimento web e mobile, especialmente com o surgimento de frameworks como Angular, React e Vue.js. Ele é uma variação do padrão MVC que visa separar ainda mais as responsabilidades e melhorar a testabilidade e a manutenibilidade do código. Vamos detalhar o MVVM no contexto de desenvolvimento web e mobile:

Desenvolvimento Web:

Model (Modelo): O Modelo no MVVM é responsável por representar os dados e a lógica de negócios do aplicativo. Ele é independente da View e do ViewModel e geralmente inclui acesso a

dados, validação e manipulação de dados. O Modelo pode ser compartilhado entre várias ViewModels.

View (Visualização): A View no MVVM é responsável pela renderização da interface do usuário e pela exibição dos elementos visuais. Ela é passiva e não contém lógica de negócios. A View é vinculada aos dados fornecidos pelo ViewModel e reflete as alterações nesses dados. Ela também envia eventos de interação do usuário para o ViewModel.

ViewModel: O ViewModel é o componente-chave do MVVM. Ele atua como um intermediário entre a View e o Modelo. O ViewModel contém a lógica de apresentação, manipulação de dados e gerenciamento de estados da View. Ele expõe propriedades e comandos que a View pode vincular e manipular. O ViewModel também pode realizar operações no Modelo, como buscar dados ou atualizar o estado.

No MVVM, a View e o ViewModel estão fortemente ligados por meio de um mecanismo de binding de dados. Quando ocorrem alterações nos dados do ViewModel, a View é atualizada automaticamente, e quando ocorrem interações do usuário na View, essas interações são propagadas para o ViewModel para processamento.

Desenvolvimento Mobile:

No desenvolvimento mobile, o padrão MVVM é aplicado de forma semelhante ao desenvolvimento web, mas com algumas adaptações específicas para as plataformas e frameworks móveis, como Android e iOS. Aqui estão os componentes do MVVM no desenvolvimento mobile:

Model (Modelo): O Modelo no desenvolvimento mobile é responsável por representar os dados e a lógica de negócios. Ele pode incluir acesso a dados, chamadas de API, manipulação de armazenamento local e qualquer outra operação relacionada a dados.

View (Visualização): A View no desenvolvimento mobile é responsável por renderizar a interface do usuário e exibir os elementos visuais. Assim como no desenvolvimento web, a View é passiva e não contém lógica de negócios.

ViewModel: O ViewModel no desenvolvimento mobile é semelhante à sua contraparte no desenvolvimento web. Ele atua como intermediário entre a View e o Modelo, contendo a lógica de apresentação, manipulação de dados e gerenciamento de estados da View. O ViewModel também pode realizar operações no Modelo, como busca de dados ou atualização de estados.

Assim como no desenvolvimento web, a View e o ViewModel no desenvolvimento mobile são conectados por meio de um mecanismo de binding de dados, permitindo que as alterações nos dados do ViewModel sejam refletidas automaticamente na View e vice-versa.

O padrão MVVM no desenvolvimento web e mobile ajuda a separar as responsabilidades, facilitando a manutenção, a testabilidade e a reutilização de código. Ele promove uma melhor organização do código, permitindo que as equipes trabalhem em paralelo e colaborativamente em diferentes partes do aplicativo.

A Arquitetura MVP (Model-View-Presenter) é um padrão de arquitetura de software utilizado no desenvolvimento web e mobile. Ela é uma variação do padrão MVC que visa separar as responsabilidades de forma mais clara, principalmente no que diz respeito à lógica de apresentação e à interação com o usuário. Vamos detalhar a Arquitetura MVP no contexto de desenvolvimento web e mobile:

Desenvolvimento Web:

Model (Modelo): O Modelo no MVP é responsável por representar os dados e a lógica de negócios. Ele encapsula os dados, realiza operações relacionadas aos dados e pode se comunicar com a camada de persistência de dados, como um banco de dados ou serviços de API.

View (Visualização): A View no MVP é responsável pela interface do usuário e pela exibição dos elementos visuais. Ela é passiva e não contém lógica de negócios. A View é responsável por exibir os dados do Modelo e transmitir as interações do usuário para o Presenter.

Presenter: O Presenter é o componente-chave do MVP. Ele atua como intermediário entre a View e

o Modelo, gerenciando a lógica de apresentação e a interação com o usuário. O Presenter recebe as interações do usuário da View, processa essas interações e atualiza o Modelo conforme necessário. Ele também atualiza a View com os dados atualizados do Modelo.

A comunicação entre os componentes do MVP ocorre da seguinte forma: quando o usuário interage com a interface do usuário (View), como clicar em um botão, a View notifica o Presenter. O Presenter processa a interação, atualiza o Modelo e notifica a View para exibir os dados atualizados.

Desenvolvimento Mobile:

No desenvolvimento mobile, a Arquitetura MVP segue uma estrutura semelhante à do desenvolvimento web, mas com algumas adaptações específicas para as plataformas e frameworks móveis, como Android e iOS. Aqui estão os componentes do MVP no desenvolvimento mobile:

Model (Modelo): O Modelo no desenvolvimento mobile é responsável por representar os dados e a lógica de negócios, assim como no desenvolvimento web.

View (Visualização): A View no desenvolvimento mobile é responsável por renderizar a interface do usuário e exibir os elementos visuais. Ela é passiva e não contém lógica de negócios.

Presenter: O Presenter no desenvolvimento mobile é semelhante ao desenvolvimento web. Ele atua como intermediário entre a View e o Modelo, gerenciando a lógica de apresentação e a interação com o usuário. O Presenter recebe as interações do usuário da View, processa essas interações e atualiza o Modelo conforme necessário. Ele também atualiza a View com os dados atualizados do Modelo.

No MVP, a View e o Presenter estão fortemente acoplados, com a View notificando o Presenter sobre as interações do usuário e o Presenter atualizando a View com os dados do Modelo.

A Arquitetura MVP no desenvolvimento web e mobile ajuda a separar as responsabilidades, facilitando a testabilidade, a manutenção e a reutilização de código. Ela promove uma melhor organização e clareza no fluxo de dados e eventos, permitindo uma melhor divisão do trabalho em

equipes e uma implementação mais escalável e modular.

A Arquitetura Clean (ou Clean Architecture) é uma abordagem de arquitetura de software proposta por Robert C. Martin, também conhecido como Uncle Bob. Essa arquitetura busca criar sistemas altamente testáveis, escaláveis e independentes de frameworks e tecnologias específicas. Ela promove a separação de responsabilidades e a implementação de um código limpo e organizado. Vamos detalhar a Arquitetura Clean no contexto de desenvolvimento web e mobile:

A Arquitetura Clean é baseada em camadas e princípios de inversão de dependência, tornando o código modular, desacoplado e altamente testável. Ela é composta por camadas com diferentes responsabilidades, onde as dependências fluem de dentro para fora, mantendo as regras de negócio no centro da arquitetura. Aqui estão as principais camadas da Arquitetura Clean:

Domínio (ou Camada de Regras de Negócio): Essa camada contém as regras de negócio do sistema. Ela representa o núcleo da aplicação, onde as entidades, objetos de valor, interfaces e serviços de domínio são definidos. Essa camada não possui dependências de outras camadas, tornando-a independente de qualquer tecnologia ou framework específico.

Camada de Casos de Uso (ou Aplicação): Essa camada orquestra as interações entre as entidades do domínio e as camadas externas, como a interface do usuário e a persistência de dados. Ela contém a lógica de aplicação, implementando os casos de uso específicos do sistema. Essa camada depende do domínio, mas não possui dependências externas.

Camada de Interface do Usuário: Essa camada é responsável pela interação com o usuário. Ela pode ser implementada como uma interface gráfica, uma API ou qualquer outro meio de comunicação entre o sistema e o usuário. Essa camada se comunica com a camada de Casos de Uso para obter os dados necessários e atualizar o estado da aplicação.

Camada de Infraestrutura: Essa camada lida com as preocupações técnicas e de infraestrutura, como acesso a banco de dados, serviços externos, frameworks, etc. Ela implementa os detalhes técnicos e depende das outras camadas, mas as camadas internas não têm conhecimento dessa camada.

Na Arquitetura Clean, o fluxo de dependências segue a direção de dentro para fora, com as

camadas internas não tendo conhecimento das camadas externas. Isso promove um acoplamento fraco e facilita a substituição de componentes e testes unitários.

Essa abordagem permite que a lógica de negócios seja completamente isolada e testada independentemente das tecnologias utilizadas para a interface do usuário, banco de dados ou outras partes do sistema. Ela também facilita a evolução do sistema ao longo do tempo, pois as mudanças nas tecnologias e frameworks externos têm um impacto mínimo nas regras de negócio e nos casos de uso do sistema.

A Arquitetura Clean é aplicável tanto no desenvolvimento web quanto no mobile, permitindo a construção de sistemas robustos, escaláveis e de fácil manutenção em ambas as plataformas. A adaptação da Arquitetura Clean para cada plataforma específica pode exigir algumas adaptações, mas os princípios fundamentais permanecem os mesmos.

___

A Arquitetura Flux/Redux é um padrão de arquitetura de software utilizado no desenvolvimento web e mobile para gerenciar o estado da aplicação de forma previsível e centralizada. Ela se baseia nos conceitos de fluxo unidirecional de dados e imutabilidade do estado. Vamos detalhar a Arquitetura Flux/Redux no contexto de desenvolvimento web e mobile:

A Arquitetura Flux/Redux é composta por quatro principais componentes:

Actions (Ações): As Actions representam eventos que ocorrem na aplicação, como uma interação do usuário ou uma resposta de uma API. Elas são objetos simples que descrevem o tipo de ação que está ocorrendo e podem conter dados associados a essa ação.

Store (Armazenamento): A Store é responsável por armazenar e gerenciar o estado da aplicação. Ela é um repositório centralizado onde todo o estado da aplicação é mantido. A Store é imutável, o que significa que o estado não pode ser alterado diretamente. Em vez disso, ele é atualizado por meio de funções puras chamadas de reducers.

Reducers: Os Reducers são funções puras que especificam como o estado da aplicação deve ser atualizado em resposta às Actions. Eles recebem o estado atual e uma Action como entrada e retornam um novo estado atualizado. Os Reducers são responsáveis por garantir que o estado seja imutável, criando uma cópia atualizada do estado em vez de modificá-lo diretamente.

View (Visualização): A View é a interface do usuário da aplicação, responsável por exibir o estado atual da aplicação e enviar Actions para a Store. Ela pode ser implementada como uma página web, uma tela mobile ou qualquer outro componente visual. A View é atualizada automaticamente sempre que o estado da Store muda.

O fluxo de dados na Arquitetura Flux/Redux é unidirecional, ou seja, segue uma direção específica. Ele começa com as Actions sendo despachadas pela View e enviadas para os Reducers. Os Reducers atualizam o estado da Store com base nas Actions recebidas e retornam um novo estado atualizado. A View é notificada sobre as mudanças no estado e atualiza a interface do usuário conforme necessário.

A principal vantagem da Arquitetura Flux/Redux é que ela facilita o gerenciamento do estado da aplicação, tornando-o previsível e facilitando o rastreamento de como o estado é alterado ao longo do tempo. Além disso, como o estado é centralizado na Store e os Reducers são funções puras, a depuração e o teste do código se tornam mais simples.

A Arquitetura Flux/Redux é amplamente utilizada em frameworks e bibliotecas populares, como Redux (JavaScript), NgRx (Angular), Vuex (Vue.js) e MobX-State-Tree (React Native), entre outros. Ela pode ser aplicada tanto no desenvolvimento web quanto no mobile, fornecendo uma abordagem consistente e escalável para o gerenciamento de estado em aplicações complexas.

—

A Arquitetura VIPER é um padrão de arquitetura de software utilizado no desenvolvimento web e mobile, especialmente em aplicativos móveis, com o objetivo de facilitar a escalabilidade, a modularidade e a testabilidade do código. Ela foi introduzida pela primeira vez pela equipe da Uber Technologies em 2014. Vamos detalhar a Arquitetura VIPER no contexto de desenvolvimento web e mobile:

A Arquitetura VIPER é baseada em componentes que representam diferentes responsabilidades dentro do aplicativo. Cada letra em VIPER representa um desses componentes:

View (Visualização): A View é responsável pela interface do usuário e pela exibição dos elementos visuais. Ela recebe as interações do usuário e envia eventos para o Presenter. A View é

passiva e não possui lógica de negócios.

Interactor (Interator): O Interactor contém a lógica de negócios do aplicativo. Ele é responsável por recuperar e processar dados, aplicar regras de negócios e coordenar as operações relacionadas a uma funcionalidade específica. O Interactor não possui conhecimento direto da View ou do Presenter.

Presenter: O Presenter atua como intermediário entre a View e o Interactor. Ele recebe os eventos da View, processa-os e solicita ações ao Interactor. O Presenter também é responsável por atualizar a View com os dados recebidos do Interactor.

Entity (Entidade): A Entidade representa os objetos de negócio do aplicativo. Ela encapsula os dados e as operações relacionadas a esses objetos. As Entidades são independentes da camada de apresentação e não têm conhecimento direto dos outros componentes.

Router (Roteador): O Roteador gerencia a navegação entre as diferentes telas do aplicativo. Ele lida com a lógica de roteamento e é responsável por criar e apresentar as ViewControllers (no desenvolvimento mobile) ou as páginas (no desenvolvimento web) correspondentes.

A comunicação entre os componentes do VIPER ocorre de forma bidirecional, com o fluxo de dados seguindo uma direção específica. A View envia eventos para o Presenter, que os processa e solicita ações ao Interactor. O Interactor realiza as operações necessárias e retorna os resultados ao Presenter, que atualiza a View com os dados recebidos. O Roteador também pode ser acionado pelo Presenter para navegar para outras telas ou páginas.

A principal vantagem da Arquitetura VIPER é a clara separação de responsabilidades e a alta modularidade. Cada componente tem uma função bem definida e pode ser testado de forma independente. Isso facilita a manutenção do código, a reutilização de componentes e a colaboração entre equipes de desenvolvimento.

Embora a Arquitetura VIPER seja mais comumente utilizada no desenvolvimento de aplicativos móveis, ela também pode ser aplicada no desenvolvimento web, adaptando-se aos frameworks e tecnologias específicas utilizadas na plataforma.

No entanto, é importante mencionar que a adoção da Arquitetura VIPER pode adicionar complexidade ao projeto, especialmente em aplicativos de menor escala. Portanto, é recomendado avaliar cuidadosamente as necessidades e requisitos do projeto antes de optar pela utilização do VIPER.

—

A Arquitetura Flutter BLoC (Business Logic Component) é uma abordagem para gerenciamento de estado e separação de lógica de negócios no desenvolvimento de aplicativos Flutter, que abrange tanto o desenvolvimento web quanto mobile. BLoC é uma camada intermediária que atua como uma ponte entre os Widgets e os dados, ajudando a manter a separação de preocupações. Vamos detalhar a Arquitetura Flutter BLoC:

A Arquitetura Flutter BLoC é baseada nos seguintes componentes principais:

Widgets: Os Widgets são os elementos de interface do usuário no Flutter. Eles representam os componentes visuais e podem ser tanto Stateless (sem estado) quanto Stateful (com estado). Os Widgets podem emitir eventos (ações) para a camada BLoC e atualizar sua aparência com base nas alterações de estado.

BLoC: O BLoC é o componente central da arquitetura. Ele é responsável por gerenciar o estado da aplicação e a lógica de negócios. O BLoC recebe eventos dos Widgets, processa-os e emite alterações de estado de volta para os Widgets. Ele atua como um controlador que facilita a comunicação entre os Widgets e a fonte de dados, como serviços externos ou repositórios.

Streams: As Streams são fluxos de eventos assíncronos. No contexto do Flutter BLoC, elas são usadas para transmitir eventos dos Widgets para o BLoC e para notificar os Widgets sobre as alterações de estado. Os Streams são usados para estabelecer uma comunicação unidirecional e reativa entre os componentes.

Sink: O Sink é uma interface que permite adicionar eventos ao fluxo de eventos (Stream) do BLoC. É usado pelos Widgets para enviar eventos ao BLoC.

StreamBuilder: O StreamBuilder é um Widget fornecido pelo Flutter que facilita a vinculação de

Widgets à transmissão (Stream) de eventos do BLoC. Ele reconstrói automaticamente os Widgets quando novos dados são emitidos pelo BLoC, permitindo que a interface do usuário seja atualizada de forma reativa.

A comunicação na Arquitetura Flutter BLoC segue um fluxo unidirecional, onde os eventos são enviados pelos Widgets ao BLoC por meio de Sinks, o BLoC processa esses eventos e emite alterações de estado por meio de Streams, e os Widgets atualizam a interface do usuário com base nessas alterações.

A principal vantagem da Arquitetura Flutter BLoC é a separação clara de responsabilidades e a reatividade dos Widgets em relação às alterações de estado. Isso torna o código mais organizado, facilita o teste e melhora a escalabilidade. Além disso, a abordagem reativa permite um melhor controle do fluxo de dados e evita problemas comuns, como a propagação de estados inconsistentes.

Para implementar a Arquitetura Flutter BLoC, é comum utilizar bibliotecas como o bloc, provider ou rxDart, que fornecem ferramentas adicionais para facilitar a implementação e a integração com o Flutter.

Em resumo, a Arquitetura Flutter BLoC é uma abordagem eficaz para o gerenciamento de estado e a separação de lógica de negócios no desenvolvimento de aplicativos Flutter, permitindo uma construção reativa e escalável de interfaces do usuário tanto no desenvolvimento web quanto mobile.

—

A Arquitetura SwiftUI é uma abordagem de desenvolvimento de interface do usuário introduzida pela Apple para criação de aplicativos iOS, iPadOS, macOS, watchOS e tvOS. Diferentemente de outras arquiteturas tradicionais, como o MVC (Model-View-Controller), a Arquitetura SwiftUI adota um paradigma declarativo e reativo, oferecendo uma maneira simplificada e eficiente de construir interfaces do usuário. Vamos detalhar a Arquitetura SwiftUI no contexto de desenvolvimento web e mobile:

Views (Visualizações): As Views são os blocos de construção fundamentais na Arquitetura SwiftUI. Elas representam componentes visuais da interface do usuário, como botões, campos de texto, listas e outros elementos. No SwiftUI, as Views são construídas usando uma sintaxe declarativa, onde você descreve como a interface do usuário deve ser renderizada com base no

estado atual. As Views são reativas, o que significa que elas são atualizadas automaticamente sempre que o estado subjacente muda.

State (Estado): O State representa o estado atual da interface do usuário. No SwiftUI, você define propriedades de estado dentro das Views, que podem ser atualizadas e refletidas automaticamente nas Views correspondentes. Isso permite que a interface do usuário seja atualizada dinamicamente à medida que o estado muda.

Bindings: Bindings são usados para criar uma conexão bidirecional entre uma propriedade de estado e uma View. Eles permitem que você atualize o estado por meio da interação do usuário e, ao mesmo tempo, atualize a View automaticamente quando o estado muda.

View Models: Embora a Arquitetura SwiftUI não exija explicitamente o uso de um View Model separado, é comum utilizar essa abordagem para lidar com a lógica de negócios e a manipulação de dados. O View Model é responsável por fornecer os dados necessários para as Views, processar eventos e realizar operações relacionadas à interface do usuário.

Observables: Observables são usados para criar objetos observáveis no SwiftUI. Eles permitem que as Views sejam notificadas automaticamente quando os dados subjacentes mudam. Isso é especialmente útil para manter as Views sincronizadas com o estado atualizado.

A Arquitetura SwiftUI é altamente integrada ao ambiente de desenvolvimento da Apple e aproveita recursos nativos, como o Xcode e o Swift. Ela fornece uma experiência de desenvolvimento fluida e eficiente, permitindo que os desenvolvedores criem interfaces do usuário complexas e interativas com menos código e em menos tempo.

No contexto do desenvolvimento web, a Arquitetura SwiftUI é mais aplicável para desenvolvimento específico da plataforma Apple, como sites otimizados para os navegadores Safari no iOS e macOS. Embora o SwiftUI seja principalmente voltado para o desenvolvimento de aplicativos nativos, é possível utilizar algumas partes dos conceitos e abordagens do SwiftUI para criar interfaces do usuário web declarativas e reativas em frameworks como o React ou Vue.js.

Em resumo, a Arquitetura SwiftUI é uma abordagem moderna e reativa para o desenvolvimento de interfaces do usuário em plataformas Apple. Ela permite a construção de interfaces do usuário

declarativas e atualizadas automaticamente, proporcionando uma experiência de desenvolvimento ágil e eficiente.

—