A arquitetura de um software é um dos pontos principais e de maior relevância na hora de iniciar o desenvolvimento de qualquer tipo de aplicação. Ela deve levar em consideração os principais tópicos, funcionalidades, requisitos e comportamentos do sistema, pois, uma vez definida, a menor das mudanças pode ocasionar um refatoramento que será, muitas vezes, considerado inviável devido à sua alta complexidade. Escolhas como o banco de dados, a linguagem de programação ou os padrões de projeto podem ter impactos profundos no futuro do software.

Além disso, um problema frequentemente negligenciado pelos desenvolvedores é a adoção e manutenção de padrões de código. Ainda que a definição de padrões seja uma decisão coletiva, em que a equipe estabelece métricas e diretrizes para o desenvolvimento, a quebra desses padrões pode tornar o código muito mais complexo e dispendioso para manter. Em contrapartida, seguir um padrão bem definido pode simplificar significativamente a manutenção e evolução do sistema. Vale ressaltar que não existe um padrão universalmente correto, mas sim aquele que melhor se adapta à equipe e ao projeto em questão.

A discussão sobre diferentes padrões de projeto é antiga e levou a diversas conclusões ao longo dos anos, culminando em um ponto central para a criação de um software sustentável em termos de longevidade e manutenção. Esse debate é bem exemplificado na famosa discussão entre Andrew Tanenbaum e Linus Torvalds, relacionada à modularidade de um programa. Tanenbaum defendia a abordagem de microkernel, argumentando que um sistema altamente modular, com processos independentes e isolados, facilitaria a manutenção e aumentaria a confiabilidade. Já Torvalds era favorável a um kernel monolítico, alegando que a integração das funções dentro de um único núcleo garantiria maior desempenho e simplicidade no desenvolvimento. Assim sendo, ainda que a arquitetura de microkernel tenha se provado a mais adequada, é válido ressaltar que até certo momento, não havia a possibilidade de uma aplicação utilizando da mesma, uma vez que demandaria tempo de refatoração/ desenvolvimento, e pensando que o software bom, é aquele que dá retorno e cumpre com aquilo que o usuário espera, mesmo com uma arquitetura monolítica, o kernel monolítico possibilitou muitas aplicações, e foi usado por muito tempo.

Dessa forma, o capítulo aborda diferentes arquiteturas, suas histórias e utilização, inicialmente, a arquitetura em camadas é uma maneira de organizar o desenvolvimento de software para que tudo fique mais estruturado e fácil de entender. A ideia é dividir o sistema em partes, onde cada uma tem sua função e só se comunica com a camada abaixo dela. Isso evita bagunça no código e facilita tanto a manutenção quanto futuras melhorias. Um exemplo abordado é a arquitetura de 3 camadas: a interface do usuário, a lógica de negócio e o banco de dados. A interface é a parte que a gente vê e interage, como um site ou aplicativo. A lógica de negócio é onde as regras do sistema acontecem, como validar uma compra ou calcular um imposto. Já o banco de dados guarda todas as informações, garantindo que nada se perca.

No dia a dia, esse conceito faz diferença porque ajuda a deixar o código mais organizado e flexível. Se precisar mudar a interface do meu sistema, por exemplo, não preciso mexer na lógica por trás dele. Ou, se quiser trocar o banco de dados, pode fazer isso sem afetar as outras partes do

sistema. Isso também facilita o trabalho em equipe, já que diferentes desenvolvedores podem atuar em partes separadas sem interferir no que os outros estão fazendo. Mesmo em projetos menores, vale a pena aplicar essa ideia. Pensar na separação entre interface, lógica e armazenamento desde o começo ajuda a evitar dores de cabeça no futuro.

MVC é uma forma inteligente de organizar um sistema, separando a interface gráfica, a lógica de negócio e o controle das interações do usuário. Isso torna o código mais limpo, fácil de entender e de modificar. Em vez de misturar tudo em um único bloco confuso, cada parte tem um papel bem definido. Na prática, isso significa que um desenvolvedor pode alterar a interface sem mexer no funcionamento do sistema, ou modificar regras de negócio sem precisar refazer toda a parte visual. Essa separação facilita tanto o trabalho em equipe quanto a manutenção ao longo do tempo. Com o tempo, essa ideia foi adaptada para a web, ajudando a estruturar sistemas modernos. Ao desenvolver trabalhos na faculdade utilizando de SpringBoot (Java), o MVC foi um grande aliado, tanto na construção, quanto na refatoração do código. No fim das contas, é um conceito que ajuda a manter o código organizado e preparado para crescer sem virar uma bagunça.

A arquitetura de microsserviços tem ganhado cada vez mais atenção nos últimos anos como uma solução eficaz para sistemas de grande escala. Ela se baseia na ideia de dividir uma aplicação monolítica em pequenos serviços independentes que podem ser desenvolvidos, implementados e escalados separadamente. Cada microsserviço é responsável por uma funcionalidade específica do sistema e se comunica com outros serviços por meio de APIs. Essa abordagem oferece vantagens como maior flexibilidade, escalabilidade e facilidade de manutenção, pois permite que diferentes equipes trabalhem em diferentes serviços simultaneamente. No entanto, adotar microsserviços também pode trazer desafios, como a complexidade na gestão de múltiplos serviços, a necessidade de uma infraestrutura de rede robusta e o controle sobre transações distribuídas. Assim, a decisão de adotar a arquitetura de microsserviços deve ser feita com cautela, levando em consideração as necessidades do projeto e a capacidade da equipe de gerenciar sua complexidade, não fazendo justo a adoção de uma arquitetura tão complexa e estruturada pra projetos simples, ou quando a latência é algo importante à levar-se em consideração, uma vez que em um sistema mais robusto e menos modular, o atraso na transmissão de informações é mínimo.

A arquitetura orientada a mensagens é um estilo arquitetural que foca na troca de mensagens entre os componentes de um sistema, ao invés de utilizar chamadas diretas de funções ou métodos. Nesse modelo, os componentes se comunicam por meio de filas de mensagens, tópicos ou brokers, que garantem que as mensagens sejam enviadas de forma assíncrona e desacoplada. Essa abordagem facilita a escalabilidade, pois permite que os componentes sejam distribuídos e independentes, sem necessidade de conhecer diretamente uns aos outros. A arquitetura orientada a mensagens é especialmente útil em sistemas distribuídos, onde a comunicação eficiente e a garantia de entrega de mensagens são essenciais. Embora ofereça benefícios como maior flexibilidade e resiliência, ela também exige uma gestão eficiente de mensagens e a garantia de que o sistema lida bem com falhas e a ordem de processamento das mensagens.

No padrão 'publish/subscribe', a ideia é que o publicador (componente) crie e envie eventos, enquanto outros componentes (os assinantes) se inscrevem para receber esses eventos de acordo com seus interesses. Isso cria uma comunicação mais flexível e desacoplada, porque o publicador não precisa saber quem são os assinantes, e os assinantes podem escolher quais eventos querem receber. Esse modelo é útil quando temos sistemas que precisam reagir a mudanças ou notificações sem estarem diretamente ligados, como um sistema de e-commerce avisando clientes sobre promoções ou status de pedidos. A vantagem é que ele facilita a escalabilidade e mantém os componentes independentes, mas pode exigir cuidados com a entrega dos eventos e o gerenciamento das assinaturas.