- ## Modelo em camadas

  Divide as camadas de forma hierárquica, trazendo uma função específica para cada uma, como a apresentação de dados, manipulação de dados e armazenamento. Assim trazendo um isolamento de camadas mais importantes e possuindo um alto nível de abstração.
  Útil para se ter uma alta modularidade, escalabilidade, elasticidade e testes pois alterando uma das camadas tende a não afetar a outra.
  Exemplos de modelo de camadas:
  1. Interface do usuário: Toda a interface que o usuário vai interagir, sejam telas da aplicação ou do website.
  2. Lógica de negócio: Camada responsável pela manipulação do dado e a sua validação, assim mantendo um padrão na camada seguinte
  3. Acesso aos dados: Como o próprio nome impõe, esta camada é a responsável pelo acesso aos dados no banco de dados para as outras camadas.
  4. Infraestrutura: Por a camada onde a aplicação reside, Esta camada controla os níveis de acesso de cada usuário e também mantendo logs e erros

  As camadas se comunicam entre si através de diversas formas, tais como:
  1. Funções
  2. APIs
  3. Rotas
  4. Eventos (irei falar mais para frente sobre esta)

- ## Cliente servidor

  Sendo uma arquitetura simples, permite uma comunicação eficiente entre o cliente e o servidor pois normalmente é uma conversa mais direta, através de requisições e respostas, utilizando o protocolo HTTP para se ter GET, POST, PUT e DELETE.
  Com esta arquitetura é necessário focar mais recursos em segurança, pois o lado do cliente consegue chegar nas informações mais facilmente, assim um mau ator sem restrições consegue fazer um estrago considerável. Porém a simplicidade desta arquitetura mostra uma fácil escalabilidade, pois basta ter mais recursos no lado do servidor que a aplicação tende a funcionar melhor. Claro que o lado do cliente necessita ser otimizado para funcionar nos mais diversos dispositivos.
- ## Orientada a serviço

  Nesta metodologia, cada serviço fica responsável por uma tarefa, onde ao juntar vários em sequência conseguem realizar tarefas complexas, sendo que cada serviço pode funcionar em uma linguagem diferente. Um exemplo seria escrever dados em um banco, ao invés de criar uma função diferente para cada aplicação, basta escrever um serviço que cuidaria disto.
  Isto divide a carga de cada serviço e facilita a escalabilidade, manutenção e comunicação de serviço de terceiros dependentes. O problema fica na segurança, onde é necessário criar diversas regras para que recursos críticos não fiquem expostos, outro problema pode ser problemas de comunicação entre linguagens, isto acaba colocando uma carga maior nos desenvolvedores.
- ## Em micro serviços

Bastante similar a de serviços, porém estes micro serviços como o próprio nome diz, são serviços menores e sua principal diferença da arquitetura de serviços são independentes, onde cada um se comunica por uma chamada de API. Onde cada um fica de forma bem distinta, focado em apenas uma tarefa. A alta segregação destes serviços facilita a escalabilidade. A independência dos outros serviços ajuda na hora de caso um destes micro serviços caem, a aplicação não é afetada totalmente e também ajuda bastante para criar novas funcionalidades. Porém deve se prestar atenção em alguns tópicos importantes para não criar problemas,

- Usar funções assíncronas para que eles não fiquem penduradas esperando outros microsserviços
- Normalmente possuem sua própria base de dados.
- Usar protocolos OAuth e JWT para autorização.
- Devem possuir um health check para verificar a integridade do micro serviço
- Se comunicam através de API REST com sua documentação em um lugar compartilhado.
- Seus logs devem ficar em um local um único repositório centralizado.

- Baseada em eventos

Funciona baseado em um sistema de eventos,trabalham de forma assíncrona enquanto eles trocam informações, com consumidores e produtores desacoplados e acoplados de forma leve. Isso significa que eles funcionam de forma independente porém não totalmente separados. O produtor é responsável por perceber e detectar um evento e respondê-lo com uma mensagem. Após isto ele é enviado para o consumidor através de uma plataforma de processamento, onde então o consumidor é informado sobre o evento e pode processá-lo ou afetado. Devido ao desacoplamento, o produtor não conhece o consumidor e nem o resultado.
Esta arquitetura é muito utilizada para IOTs, devido a sua capacidade de receber informações de diversos sensores de forma assíncrona e em tempo real. Devido ao fato de serem desacoplados, os consumidores não dependem dos produtores funcionarem assim caso um falhe não afete o sistema como um todo.

- Hexagonal

A arquitetura hexagonal é bem parecida com a em camadas, porém ele é focado em isolar o núcleo de processamento do mundo exterior, apenas expondo as entradas e saídas na fronteira do projeto. Para isto existem portas primárias e portas secundárias, onde as primárias são a comunicação entre o núcleo e o núcleo externo. Já as secundárias são utilizadas para upstream de dados ou serviços externos. Devido a isto, o projeto fica simbolizado com um hexágono, cada lado dele é diferentes métodos que o sistema se comunica. Com a metodologia da saída, podemos alterar um dos hexágonos sem afetar o exterior ou o núcleo. Devido a todas as dependências do hexágono apontarem para o interior, o núcleo não depende do mundo exterior.
O Exterior do hexágono conversa com outras plataformas através de APIs REST. Para isto são abertas portas específicas no hexágono que levam até o centro.
Com isto esta arquitetura tem alguns princípios,

- responsabilidade única. Onde uma parte deve ter apenas um motivo para alterar, assim não se preocupando com essa parte caso alteremos outra.

- Inversão de dependência, onde podemos inverter a dependência, porém somente se ambos os lados da dependência pertencerem ao sistema e não de terceiros pois não controlamos o código dela.

- MVC

A Arquitetura MVC é uma das mais antigas, sendo uma das maneiras de demonstrar orientação a objetos. Trazendo consigo as vantagens de separar o código em partes, assim facilitando a mudança de UI sem alterar o código inteiro, facilitam a manutenção e escalabilidade. Por fim a arquitetura é composta por 3 diferentes partes, Model, View, Controller, onde estas são 3 diferentes camadas da aplicação:

- A camada de controller é responsável pelas regras de negócio e comunicações com outros sistemas, como o banco de dados, normalmente ela fica isolada. Ela também é a responsável por enviar as informações para a Model/View
- A camada Model é a estrutura do projeto, ela sabe o que fazer sobre as mudanças de estados e quais instruções precisam ser informadas.
- A camada View é a responsável por apresentar os dados recebido da model e do controller, ela não sabe exatamente o que a aplicação ta fazendo, ela é apenas responsável por demonstrar, assim o usuário não consegue manipular a aplicação de forma arbitrária.Por fim a camada view retornar as informações e seu estado para a view e a model.

- MVP

Bastante similar ao MVC temos a evolução da arquitetura para o “MVP” de Model, View e Presenter. Onde a camada

- Model é a responsável por manter os padrões de negócios, diferente do MVC que a camada Model é apenas de estrutura, no MVP ela tem duas funções, a de estrutura e de manter os dados.
- View continua bem similar ao MVC, continua possuindo a interação com o usuário e trazendo as informações solicitadas. Assim como no MVC a view não tem ideia do estado do sistema e por isto não tem contato com a model, ela apenas apresenta na tela as informações passadas pelo presenter e devolve seu estado.
- Presenter é onde a maior mudança acontece, mais simplificada lida com menos trabalho, deixando todo o esforço de manter os dados com a model. Sendo assim esta camada é responsável apenas por transmitir os dados da view para a model. Seu principal papel nesta arquitetura é de filtrar e manejar os dados.

Sendo assim é possível ver que o padrão MVP é uma melhoria do padrão MVC, distribuindo melhor as tarefas para cada uma das camadas, sem a necessidade de sobrecarregar a camada de controle em sistemas maiores. Além de que é mais fácil realizar testes unitários e reutilizar códigos, outro benefício é o fato da view ser desconhecida pela model e viceversa, assim nenhum má ator consegue chegar na model sem passar pelo presentar para ser filtrado. Porém entre suas desvantagens estão, sua alta taxa de código para pouco resultado, vários presenters e views diferentes para cada tela e a conversa entre as camadas é mais trabalhosa.

- MVVM

Por fim temos o padrão MVVVM um dos mais utilizados para desenvolvimento mobile, suas siglas significam:

- Model, similar ao MVP é onde as regras de negócios e de mantimento de dados estão, vale ressaltar que assim como no MVP esta camada não tem interações com a view.
- View continua simples e burra porém com um gosto a mais, basicamente a interface que o usuário vai interagir e não possui nenhum contato com a model. Sua grande diferença é que ela possui um “data bind” com a camada Viewmodel, onde devido ao vínculo a sincronização de dados acontece de forma automática.
- ViewModel além de receber os dados da camada view através do databind e tratá-los, a camada viewmodel também tem funções de gerir os dados que vão para a model e quais ações a view irá executar/exibir.

Além dos benefícios que as arquiteturas mvc e mvp trazem como reutilização de codigo, separamento de funções, facilidade de testes unitários, escalabilidade e elasticidade, o principal benefício fica por conta do databinding, onde facilita e simplifica a comunicação entre a view e a viewmodel, que antes era um problema para o esquema MVP. Já para desvantagens o databind acaba sendo um problema, pois é mais difícil e requer um profissional mais qualificado para se implementar já que precisam estar em sincronia, isto acaba causando alguns problemas de performance e caso não seja implementado de forma correta. E um dos seus principais problemas é o teste de aplicação, onde devido a diversos databinds um testes pode acabar tendo dificuldades de testar.

https://blog.infnet.com.br/arquitetura_software/o-que-e-e-como-fazer-arquitetura-de-software-em-camadas/
https://awari.com.br/arquitetura-de-software-em-camadas-entendendo-a-organizacao-e-comunicacao-dos-componentes/
https://www.alura.com.br/artigos/padroes-arquiteturais-arquitetura-software-descomplicada
https://awari.com.br/arquitetura-de-software-cliente-servidor-estrutura-e-interacao-entre-os-componentes/
https://aws.amazon.com/pt/what-is/service-oriented-architecture/
https://medium.com/xp-inc/entendendo-a-arquitetura-de-microservices-cdab6b52d6ed
https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture
https://medium.com/@marcio.kgr/arquitetura-hexagonal-8958fb3e5507
https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308
https://www.devmedia.com.br/mvp-model-view-presenter-revista-net-magazine-100/26318
https://www.zup.com.br/blog/mvp-vs-mvvm
https://coodesh.com/blog/dicionario/o-que-e-arquitetura-mvvm/