CovidShot
Processo de Design Arquitetural

1. Introdução

Neste documento serão apresentadas diretrizes, normas e boas práticas para o engajamento e desenvolvimento da arquitetura do CovidShot, assim como sua implementação.

2. Representação Arquitetural

A arquitetura será composta pelos estilos arquitetônicos client-server e layers focado no atendimento dos requisitos arquiteturais e estimativas de qualidade levantados, como já apontado anteriormente, cada camada possui suas respectivas responsabilidades, funcionalidades e escopo.

2.1. Client

Esse componente é responsável por todas as páginas que serão apresentadas, contendo serviços e modelos que irão fornecer as funcionalidades através da interface, aqui é feita às requisições ao servidor por meio de chamadas HTTPS para executar o processamento e armazenamento dos dados.

2.2. Server

O módulo de “Server” será construído utilizando o estilo arquitetônico em camadas, dividido em quatro camadas: Aplicação, Domínio, Serviços e Infraestrutura.

2.3. Server - Aplicação

Camada responsável por controlar e gerenciar as entradas e saídas do servidor, repassando os dados formatados em entidades de apresentação ao usuário, estando no mais alto nível do negócio.

2.4. Server - Domínio

Camada responsável por manter os modelos de entidades do negócio, assim como as interfaces utilizadas em todas as camadas por meio da injeção de dependência, de certa forma, o domínio é visto como um Core compartilhado pelo projeto,

oferecendo extensões e classes auxiliares para a execução dos serviços das demais camadas.

2.5. Server - Serviços

Camada responsável por implementar todas as regras de negócio do sistema, com as funções que a aplicação deve fornecer, além de realizar as validações necessárias para o funcionamento do serviço, fornecer e consumir os modelos na visão do negócio e realizar integração com sistemas externos.

2.6. Server - Infraestrutura

Camada responsável por realizar a integração do serviço com sistemas gerenciadores de banco de dados e garantir a persistência de dados, consumindo os modelos de negócio da camada de domínio, além de fornecer serviços de segurança e persistência.

2.7. Azure Functions Package

Neste módulo estão contidos os códigos das Azure Functions que serão chamadas pelo Serviço da camada Serviço, em conjunto com suas configurações e outros serviços exclusivos para cada função

2.8. Diagrama de Pacotes

Seguindo o diagrama da figura 1, pode ser observado cada componente interno dentro das camadas, assim como as relações entre eles.

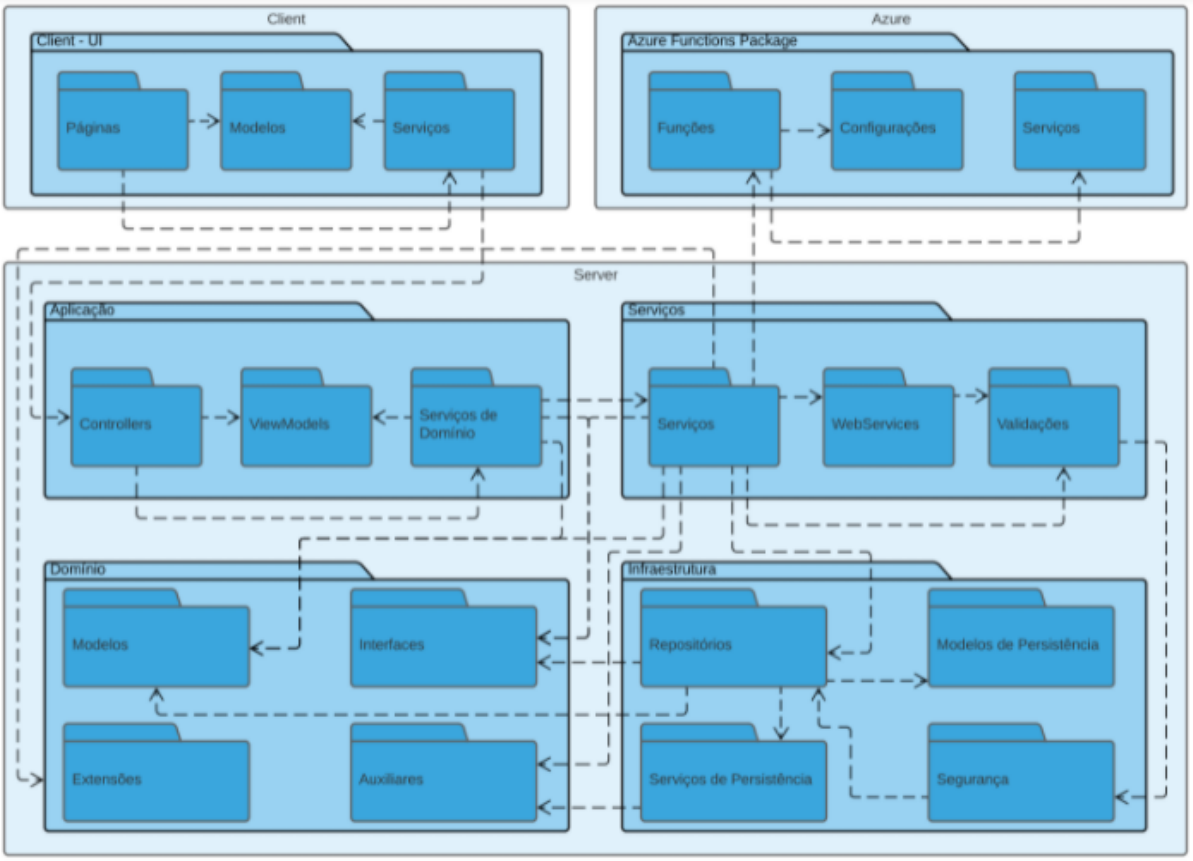

Figura 1 - Diagrama de Pacotes

2.9. Interligação dos Componentes

No Client, as interações entre os componentes Páginas, Modelos e Serviços, são feitos de forma peculiar, seguindo o padrão MVP, as páginas suportam apenas as estruturas básicas de design e estilização, na qual não realiza nenhum funcionalidade, os modelos, por sua vez são responsáveis por manter as entidades de visualização e seus dados, garantindo a integridade de comportamentos oferecidos pelo escopo dela, e consequentemente sendo apresentados na tela, já os serviços são responsáveis por realizar as chamadas HTTPS para o servidor e realizar processamentos de validações, sua chamada é feita de forma concreta, ou seja, dependente de implementações.

No Server, existem quatro camadas (Aplicação, Serviços, Domínio e Infraestrutura), além do projeto complementar de funções, contudo, as camadas de responsabilidade conhecem apenas o domínio, as demais são inexistentes em seu contexto, mantendo o baixo acoplamento, portanto, todas as relações intercamadas

são feitos através de contratos de interfaces presentes no domínio, que possuem implementações concretas apenas em seu respectivo escopo.

2.9.1. Exemplo do Relacionamento Entre Camadas

Como apresentado na figura 2, a camada de aplicação desconhece a existência do serviço, sendo abstraída apenas a responsabilidade de autenticação pela interface, que está implementada em outro lugar, logo, a camada de domínio é a ligação entre as chamadas das camadas externas, sendo conhecidas por todas as demais.

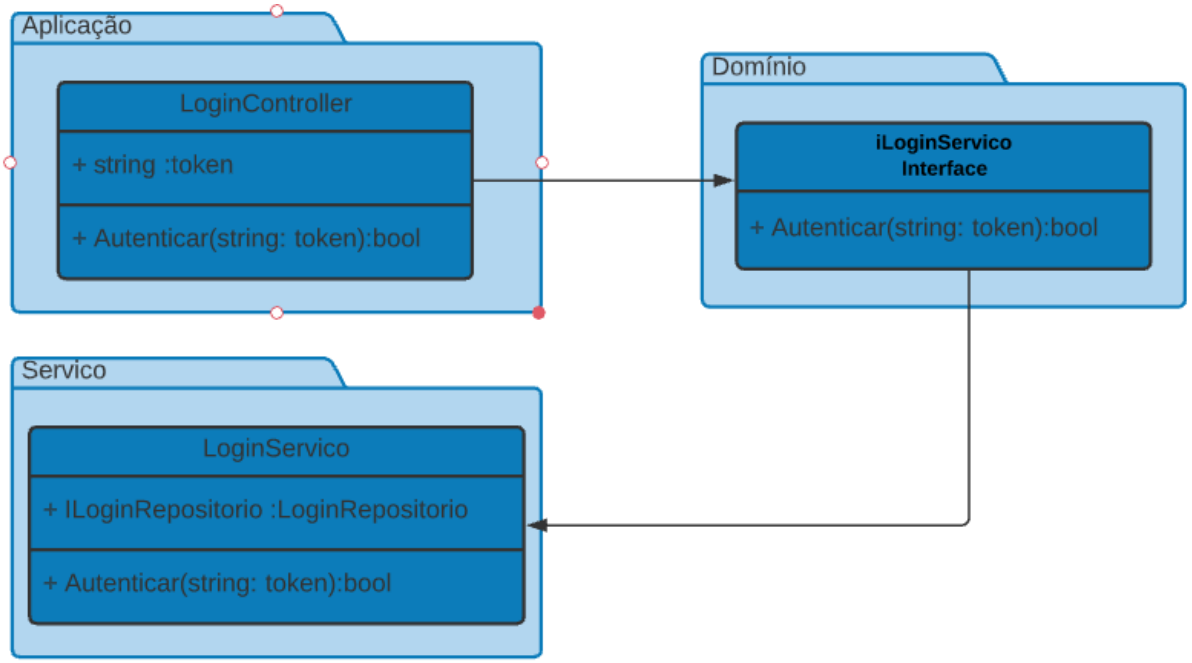

Figura 2 - Exemplo de Relacionamento Intercamadas

3. Design Patterns

Os design patterns são essenciais para a criação de uma boa arquitetura, portanto os seguintes patterns serão implementados de modo a garantir a manutenibilidade do sistema.

3.1. Singleton

O Singleton permite que exista apenas uma instância de determinada classe, durante toda a vida do sistema, independente do contexto de requisição, ou seja, funcionalidades interdependentes de escopo único, logo, esse padrão deve ser aplicado para todas as instâncias que realizaram os processos de requisições para chamadas externas, já que essa classe nunca irá mudar. Analisando a arquitetura

do CovidShot, as classes que realizaram as requisições para o Azure Cognitive Service, devem possuir apenas uma instância em toda a aplicação.

3.2. Adapter

O Adapter, permite que objetos e interfaces incompatíveis entre si, possuam uma interação e relacionamento. Será utilizado em conjunto com frameworks de mapeamento para realizar as conversões entre modelos externos e internos através de JSON e XML sendo traduzidos para os modelos de domínio e consequentemente ViewModels.

3.3. Abstract Factory

O Abstract Factory permite que sejam criados diversas instâncias de classes, sem que as classes concretas sejam especificadas, logo, em alinhamento com a inversão de controle através da injeção de dependências, a fábrica será responsável por criar, manipular e disponibilizar instâncias concretas as chamadas realizadas pelas interfaces. Logo, todo contrato utilizado no projeto, deve ser acrescentado na fábrica em conjunto com a classe concreta.

3.4. Observer

O Observer permite que haja uma notificação assíncrona para a notificação de eventos através de um estado de um entidade, portanto, sua implementação terá o foco de garantir que sejam acionados as chamadas as Azure Functions assim que as ações de gatilho do usuário forem ativadas, ou processamentos no servidor, sendo independentes do fluxo do processo da tarefa.

3.5. Bridge

O padrão Bridge em conjunto com Abstract Factory será responsável pela diferenciação das interfaces localizadas no domínio e as implementações concretas, que por sua vez podem possuir multiplicidades dependendo do contexto e responsabilidade das camadas.

3.6. Facade

Por fim, o Facade será responsável por abstrair todas as funcionalidades de serviços, frameworks e chamadas externas para a realização das tarefas em uma única interface de domínio,garantindo a integridade do escalonamento do sistema.

4. Padrões Arquiteturais

Os padrões arquiteturais estão acima dos design patterns, contudo fornecem padrões de comportamento que afetam todo o sistema, disponibilizando soluções confiáveis de arquitetura em um nível inferior.

4.1. MVP

O MVP é uma evolução do MVC que se comunica bidirecionalmente com as outras camadas, evitando que o Modelo tenha que se comunicar diretamente com a *View* sem passar pelo *Controller* e este último é fundamental para a interação com o usuário. Será o principal pattern utilizado dentro do componente de Client, facilitando a abstração pela parte do front.

4.2. Repository Pattern

É um padrão de projeto que permite um encapsulamento da lógica de acesso a dados, impulsionando o uso da injeção de dependência (DI) e proporcionando uma visão mais orientada a objetos das interações com o banco de dados, portanto, na camada de Infraestrutura, os repositórios serão criados com base nos contextos de entidades fortes (Aggregate Roots).

4.3. Generic Pattern

É um padrão de projeto que define a criação de interfaces genéricas únicas, para qualquer tipo de dado ou entidade, associando-o com cada componente importante do sistema, separados por contextos, facilitando a injeção de dependência e reaproveitamento de código. Logo, todas as interfaces de contrato das camadas que implementam o Bridge, deverão ser genéricas, sendo diferenciadas apenas na tipagem das declarações de onde possui requisições.

4.4. Unit of Work

Unit of Work é o conceito relacionado à implementação efetiva do Padrão de Repositório, sendo responsável por separar em apenas uma transação várias

operações sequências destinadas do mesmo request, que podem ocasionar um rollback inteiro do fluxo já realizado em caso de falha, esse padrão é essencial para aplicações com alto nível de usuários, para garantir a consistência do sistema e resolução de falhas em caso de problemas de infraestrutura e armazenado, portanto, todos os repositórios devem ser separados em contextos únicos através de transações estagiadas.

4.5. Event Sourcing

Com Event Sourcing podemos armazenar todas as mudanças de estado dos modelos monitorados em eventos sequenciais, para a organização de uma tomada de decisão adequada, esse pattern será considerado na implementação de extração e manipulação de dados externos, sejam consultas em API ou Banco de dados, de forma assíncrona.

4.6. Aggregate Root Pattern

Por fim, o padrão Aggregate Root organiza todos os modelos para um domínio e define qual a entidade chave da operação, ou seja, os modelos deverão ser agregados de um modelo mais forte, garantindo a raiz de agregação, com isso, a separação de responsabilidade fica muito mais clara, independente do contexto.

5. Padronização e Boas Práticas

Os princípios do SOLID serão aplicados no desenvolvimento do software do CovidShot para atingir uma boa manutenibilidade, capacidade de adaptação e mudanças, um bom entendimento e as necessidades especificadas. São eles:

Single-Responsibility Principle: "uma classe deve ter apenas uma única responsabilidade (mudanças em apenas uma parte da especificação do software, devem ser capaz de afetar a especificação da classe)." Este princípio está diretamente relacionado com a coesão de uma classe, mensurada por suas responsabilidades e propósitos. Uma classe deve ter apenas uma responsabilidade, uma vez que classes que com muitas responsabilidades se tornam confusas, muito complexas e de difícil entendimento.

Open-Closed Principle: "entidades de software devem ser abertas para extensão, mas fechadas para modificação." Este princípio explica que a alteração de um

código deve ser feita sem alterar o que já estava implementado, uma vez que alterações causam impactos no software e devem ser evitadas para que erros não sejam criados. O código-fonte deve ser estruturado de modo que seja de fácil entendimento, flexível, simples de dar manutenção e que reflita em poucos impactos nas mudanças.

Liskov Substitution Principle: "objetos em um programa devem ser substituíveis por instâncias de seus subtipos, sem alterar a funcionalidade do programa." O terceiro princípio resume que classes derivadas devem poder ser substituídas por suas classes bases.

Interface Segregation Principle: "muitas interfaces de clientes específicas, são melhores que uma para todos os propósitos". Este princípio defende que uma classe não deve importar uma interface que contenha mais elementos do que irá utilizar. O objetivo é utilizar interfaces específicas e reduzir interfaces genéricas.

Dependency Inversion Principle: "deve-se depender de abstrações, não de objetos concretos." O último princípio diz respeito ao fato de que uma classe não deve se referir a itens concretos e sim a interfaces, classes abstratas e outros tipos de abstração, de modo a diminuir seu acoplamento. O alto acoplamento faz que as alterações sejam difíceis de ser executadas e afetem outros módulos do software.