## **Documentação - Decisões de Design**

Este documento detalha as principais decisões de arquitetura e design tomadas durante o desenvolvimento do projeto **AgendaPro**, explicando o "porquê" por trás de cada escolha tecnológica e de padrão.

### **1. Escolha da Arquitetura: Arquitetura Limpa (Clean Architecture)**

* **Decisão:** Adotar uma Arquitetura Limpa em quatro camadas (Dominio, Aplicacao, Infraestrutura, API) em vez de uma abordagem mais simples (como um projeto monolítico ou um modelo de 3 camadas tradicional).
* **Justificativa:**
  + **Separation of Concerns (SoC):** A principal vantagem é a separação clara de responsabilidades. A lógica de negócio (Dominio) não tem conhecimento do banco de dados (Infraestrutura) ou da interface do utilizador (API). Isto torna o sistema muito mais fácil de entender e manter.
  + **Testabilidade:** Esta arquitetura é projetada para ser testável. A camada de Dominio e Aplicacao podem ser testadas unitariamente sem a necessidade de um banco de dados ou de um servidor web, permitindo a criação de testes rápidos e fiáveis que validam o coração do sistema.
  + **Independência Tecnológica:** O núcleo do sistema (Dominio e Aplicacao) não depende de frameworks. Se no futuro decidirmos trocar o SQL Server por PostgreSQL, ou até mesmo a Web API por uma aplicação gRPC, apenas a camada de Infraestrutura e API precisariam de ser alteradas. O núcleo do negócio permanece intacto.
  + **Escalabilidade:** É muito mais simples escalar ou modificar uma aplicação com responsabilidades bem definidas. Adicionar uma nova funcionalidade significa criar novos Commands e Handlers, com impacto mínimo no código existente.

### **2. Padrão de Lógica de Aplicação: CQRS com MediatR**

* **Decisão:** Utilizar o padrão **CQRS (Command Query Responsibility Segregation)**, implementado através da biblioteca **MediatR**, em vez de usar uma camada de "Serviços" tradicional.
* **Justificativa:**
  + **Single Responsibility Principle (SRP):** Cada caso de uso tem a sua própria classe (Handler). Um CriarReservaComandoHandler tem apenas uma responsabilidade: criar uma reserva. Isto evita a criação de "God Classes" (serviços gigantes com dezenas de métodos), que são difíceis de manter.
  + **Desacoplamento:** O Controller na API não conhece o Handler que vai executar a ação. Ele apenas envia uma mensagem (Command ou Query) para o "mediador". Isto significa que podemos alterar a lógica do Handler sem nunca tocar no Controller.
  + **Controllers "Magros" (Thin Controllers):** Esta abordagem mantém os nossos controllers extremamente limpos e simples. A sua única função é receber a requisição HTTP e despachar a mensagem, tornando-os fáceis de ler e testar.
  + **Clareza de Intenção:** O código torna-se auto-documentado. Quando vemos um EditarSalaComando, sabemos imediatamente que esta ação modifica dados. Quando vemos um ListarSalasQuery, sabemos que é uma operação de leitura segura.

### **3. Acesso a Dados: Padrão Repository e Unit of Work**

* **Decisão:** Abstrair o acesso a dados através dos padrões **Repository** e **Unit of Work**.
* **Justificativa:**
  + **Abstração da Persistência:** O Repository atua como uma "coleção em memória" dos nossos objetos de domínio. A camada de Aplicacao usa o IReservaRepositorio para "adicionar" ou "buscar" uma reserva, sem nunca saber se os dados estão a vir do SQL Server, de um ficheiro de texto ou de outro sítio qualquer. Toda a complexidade do Entity Framework fica encapsulada na implementação do repositório, na camada de Infraestrutura.
  + **Testabilidade (Mocks):** Nos testes unitários dos nossos Handlers, podemos facilmente criar um "falso" repositório (um *mock*) que simula o comportamento do banco de dados, permitindo testar a lógica da aplicação de forma completamente isolada.
  + **Integridade Transacional (Unit of Work):** O Unit of Work garante que múltiplas operações (por exemplo, adicionar uma Reserva e atualizar o Status de uma Sala ao mesmo tempo) são salvas como uma única transação atómica. Ou tudo é salvo com sucesso, ou nada é salvo, prevenindo inconsistências nos dados.

### **4. Mapeamento Objeto-Relacional (ORM): Entity Framework Core (Code-First)**

* **Decisão:** Utilizar o Entity Framework Core com a abordagem **Code-First**.
* **Justificativa:**
  + **Produtividade:** O EF Core automatiza grande parte do código repetitivo de acesso a dados (CRUD), permitindo que o foco permaneça na lógica de negócio.
  + **Fonte Única da Verdade:** Com o Code-First, as nossas classes C# (Entidades e configurações) são a fonte única da verdade para a estrutura do banco de dados. O esquema do banco é gerado a partir do código, o que garante a consistência e facilita o versionamento através das **Migrations**.
  + **Independência de Banco de Dados:** Embora estejamos a usar o SQL Server, a abstração do EF Core facilita a migração para outros provedores de banco de dados no futuro com alterações mínimas no código da aplicação.

### **5. Transferência de Dados: DTOs (Data Transfer Objects)**

* **Decisão:** Utilizar DTOs específicos para as requisições da API (CriarSalaRequest) e para as respostas da camada de Aplicação (SalaDto).
* **Justificativa:**
  + **Segurança e Contrato de API:** Expor as entidades de domínio diretamente na API é uma má prática de segurança (risco de *over-posting*). Os Request DTOs garantem que a API só aceita os campos que ela espera.
  + **Estabilidade do Contrato:** A forma como a API se apresenta ao mundo não deve mudar só porque uma propriedade interna da nossa entidade de domínio mudou. Os DTOs desacoplam o "contrato público" da API da nossa "implementação interna".
  + **Modelagem da Resposta:** Os DTOs de resposta (SalaDto, ReservaDto) permitem-nos moldar os dados da forma mais conveniente para o cliente da API. Por exemplo, o ReservaDto inclui os nomes da sala e do utilizador, e não apenas os seus IDs, o que evita que o cliente tenha de fazer chamadas adicionais.