Modelo Relacional UML da Aplicação CAMAAR

Introdução

A aplicação é um sistema voltado para a avaliação de atividades acadêmicas remotas, permitindo que instituições de ensino organizem turmas, atribuam formulários de avaliação a essas turmas e recebam respostas dos alunos. Este relatório descreve o modelo relacional UML do sistema, destacando suas principais entidades, atributos e relacionamentos.

Entidades Principais

1. Usuario

Atributos:

- **matricula**: Identificador único.
- **email**: Endereço eletrônico do usuário.
- **nome**: Nome completo.
- **hashDaSenha**: Senha armazenada de forma segura.
- **perfil**: Define o tipo de usuário (Aluno, Professor ou Administrador).
- **dataDeCriacao**: Data de registro do usuário no sistema.

Relacionamentos:

- Pode criar formulários, se tiver o perfil de Administrador..
- Pode lecionar em turmas, se tiver o perfil de Professor ou Administrador.
- Pode estar matriculado em turmas, se tiver o perfil de Aluno.
- Qualquer usuário, independentemente do perfil, pode responder a um formulário.

2. Turma

Atributos:

- **idDaTurma**: Identificador único da turma.
- **nomeDaTurma**: Nome ou código identificador da turma.
- **semestre**: Semestre letivo correspondente.
- **matriculaDoProfessor**: Referência ao professor responsável.
- **ativo**: Indica se a turma está ativa ou inativa.
- **descricao**: Detalhes adicionais sobre a turma.

Relacionamentos:

- É associada a um Usuário com perfil de Professor ou Administrador.
- Pode ter múltiplos Usuários com perfil de Aluno matriculados.
- Pode receber múltiplos Formulários por meio de uma relação de muitos-para-muitos.

3. Formulario

Atributos:

- **idDoFormulario**: Identificador único do formulário.
- **matriculaDoCriador**: Referência ao usuário criador.
- **titulo**: Título do formulário.
- **ehTemplate**: Indica se é um modelo reutilizável.
- **status**: Situação do formulário (ativo, rascunho, etc.).
- **estruturaJSON**: Estrutura de perguntas e campos no formato JSON.
- **dataDeExpiracao**: Prazo final para resposta.

Relacionamentos:

- É criado por um Usuário com perfil de Administrador.
- Pode ser atribuído a múltiplas Turmas.
- Recebe Respostas dos usuários.

4. Resposta

Atributos:

- **idDaResposta**: Identificador único da resposta.
- **idDoFormulario**: Formulário respondido.
- **matriculaDoAvaliador**: Aluno que respondeu.
- **dataDeSubmissao**: Data da entrega da resposta.
- **respostasJSON**: Conteúdo preenchido, armazenado em JSON.

Relacionamento

- Representa a resposta de um Usuário a um Formulário específico.

Modelo Relacional UML

Abaixo está o modelo relacional UML que descreve a estrutura de dados da aplicação **CAMAAR**:

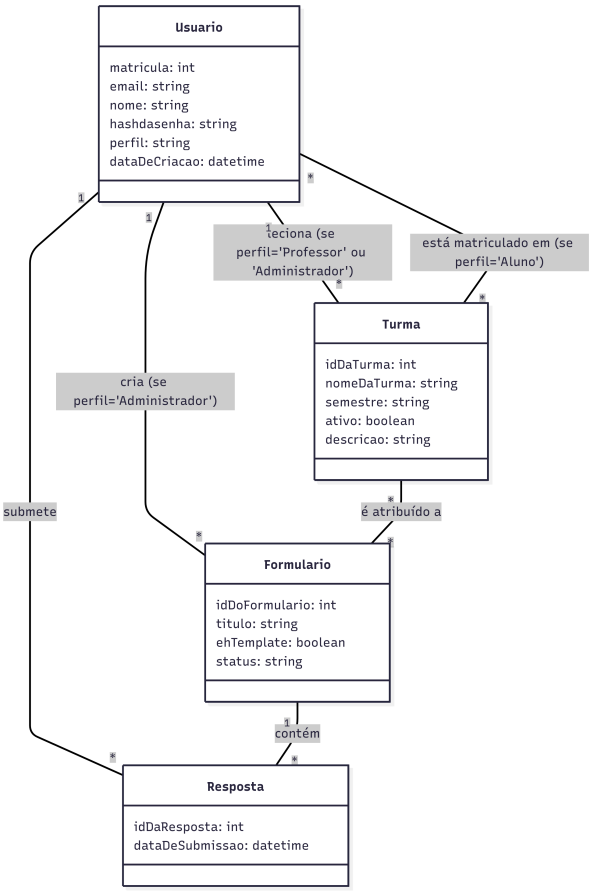

Figure 1: Modelo Relacional UML da aplicação CAMAAR.

Considerações Finais

O modelo apresentado reflete a organização e a clareza necessárias para sistemas de avaliação acadêmica remota. Ele permite facilidade de manutenção, escalabilidade e reaproveitamento de formulários por meio de templates. A estrutura pode ser expandida futuramente para incluir funcionalidades como análise estatística de desempenho, envio de notificações e exportação de relatórios.

Investigação Técnica: Implementação da Camada de Visualização para o Sistema CAMAAR em Ruby on Rails

Este relatório apresenta uma investigação técnica aprofundada (spike) sobre as melhores práticas e tecnologias para a implementação da camada de visualização (views) do sistema CAMAAR, utilizando Ruby on Rails. A análise baseia-se no protótipo de design fornecido no Figma e nas histórias de usuário detalhadas, com o objetivo de definir uma arquitetura de front-end que seja robusta, escalável, manutenível e produtiva.

Seção 1: Conectando Design e Desenvolvimento: Um Fluxo de Trabalho de Figma para Rails

Esta seção estabelece o processo fundamental para traduzir a linguagem visual do protótipo Figma em uma base de código front-end estruturada e sustentável. O objetivo é criar uma "única fonte de verdade" que minimize a divergência entre o design e a implementação final.

1.1 Otimizando o Figma para a Entrega ao Desenvolvedor

A qualidade da implementação final começa com um arquivo de design bem-estruturado. Uma organização deficiente no Figma leva a um código desordenado e ineficiente, aumentando o tempo de desenvolvimento e a complexidade da manutenção. Para garantir uma transição suave, as seguintes práticas devem ser adotadas no Figma:

- **Nomenclatura e Limpeza de Camadas:** É de vital importância nomear cada camada (layer) de forma descritiva. Esses nomes frequentemente se traduzem diretamente em nomes de classes ou identificadores no código. Camadas não utilizadas ou ocultas devem ser excluídas para evitar a exportação de código desnecessário. Além disso, as camadas de texto devem ser configuradas para "Auto-width" ou "Auto-Height" em vez de "Fixed" para garantir que o design se adapte a diferentes comprimentos de texto, um requisito essencial para a responsividade.
- **Organização Estrutural com Frames:** O uso de `Frames` no Figma é recomendado em detrimento de `Groups` para organizar camadas relacionadas. Os Frames oferecem um controle superior sobre a lógica de layout, incluindo `Auto Layout` e `Constraints`, que são essenciais para a construção de designs responsivos. O aninhamento excessivo de grupos ou frames deve ser evitado, pois complica tanto o design quanto o código resultante.

- **Design Baseado em Componentes:** O protótipo do Figma deve ser construído utilizando uma abordagem baseada em componentes, espelhando a estrutura de uma aplicação front-end moderna. Elementos como barras de navegação, botões, modais e cartões devem ser definidos como componentes reutilizáveis no Figma. Esta abordagem alinha-se com os princípios do "design atômico" e é a base de sistemas de design robustos como o Material Design e o Flowbite.

A adoção de uma estrutura de componentes no Figma não é apenas uma boa prática; é um pré-requisito para a implementação bem-sucedida de uma arquitetura baseada em ViewComponents no Rails, como será recomendado na Seção 2. Quando os designers criam layouts não estruturados, os desenvolvedores são forçados a construir views não reutilizáveis ou a fazer a engenharia reversa dos componentes, um processo ineficiente e propenso a erros. Ao garantir que a estrutura do arquivo Figma (por exemplo, um componente "App Bar") mapeie diretamente para um `AppBarComponent` correspondente no Rails, cria-se uma linguagem e um modelo mental compartilhados entre designers e desenvolvedores, reduzindo drasticamente o atrito e acelerando o ciclo de desenvolvimento.

1.2 Implementando um Sistema de Design Tokens

A prática de codificar valores fixos como cores, fontes e espaçamentos (`hard-coding`) torna as interfaces de usuário frágeis e difíceis de manter. Um sistema de "design tokens" externaliza esses valores, criando uma fonte única de verdade que governa a aparência da aplicação.

O processo de implementação de um sistema de tokens envolve a identificação de elementos de design fundamentais no Figma, atribuindo-lhes nomes padronizados (por exemplo, `$primary-color`, `$spacing-medium`) e exportando-os. Plugins como o "Figma Tokens" (também conhecido como "Tokens Studio") podem automatizar a exportação desses tokens para um arquivo JSON. Este arquivo JSON é então convertido programaticamente em propriedades personalizadas (variáveis) de CSS, geralmente dentro de um bloco

`:root` no arquivo de folha de estilos principal da aplicação.

Exemplo de conversão de JSON para CSS:

JSON exportado do Figma:

JSON
```
{
  "colors": {
    "primary": { "value": "#0065ff" }
  },
  "spacing": {
    "medium": { "value": "16px" }
  }
```

```
}
```

CSS gerado:

```
CSS
:root {
  --primary-color: #0065ff;
  --spacing-medium: 16px;
}
```

Essas variáveis CSS são então utilizadas em toda a aplicação, substituindo valores estáticos. Por exemplo, `background-color: #0065ff;` torna-se `background-color: var(--primary-color);`.

Este sistema transforma a natureza da manutenção da UI. O arquivo JSON exportado do Figma atua como um "contrato" formal ou uma API entre as equipes de design e desenvolvimento. Quando os designers publicam uma nova versão deste contrato (por exemplo, alterando uma cor primária), a equipe de desenvolvimento simplesmente executa um script para regenerar as variáveis CSS. A responsabilidade pelos *valores* do sistema de design retorna à equipe de design, onde pertence, garantindo que uma única alteração no Figma se propague de forma consistente por toda a aplicação.

1.3 Fluxo de Trabalho e Automação Recomendados

Processos manuais são propensos a erros. A automação da ponte entre o Figma e o pipeline de ativos do Rails é crucial para a eficiência e consistência.

Recomenda-se um fluxo de trabalho híbrido para o CAMAAR, que combina a rigorosidade de um sistema de design com a agilidade da prototipagem rápida:

1. **Fundação do Sistema de Design (Automação de Tokens):** Utilizar o plugin "Tokens Studio" para estabelecer os design tokens centrais (cores, tipografia, espaçamento). Um script (por exemplo, uma tarefa Rake no Rails) deve ser criado para buscar o JSON de tokens e gerar um arquivo
2. `app/assets/stylesheets/tokens.css`. Esta será a base imutável do sistema de design.
3. **Prototipagem de Componentes (Prototipagem Rápida):** Para componentes complexos, como os formulários de avaliação, a abordagem sugerida pela gem `templates-rails` pode ser adotada. Um desenvolvedor pode rapidamente criar uma versão estática da nova página (por exemplo, a view de criação de formulário) usando os tokens CSS já estabelecidos. Isso permite um feedback imediato sobre o layout e a

responsividade dentro do ambiente real da aplicação, sem a necessidade de construir toda a lógica de back-end.

4. **Componentização (Implementação Final):** Uma vez que o template estático é aprovado, o desenvolvedor refatora o ERB estático para um ViewComponent totalmente funcional e orientado a dados. Este fluxo de trabalho combina a disciplina de um sistema de design com a agilidade da prototipagem no aplicativo, criando um ciclo de desenvolvimento altamente eficaz.

Seção 2: Arquitetando a Camada de Visualização: Uma Análise Comparativa

Esta seção aborda a decisão arquitetônica mais crítica para o front-end: a tecnologia usada para renderizar o HTML. A escolha terá implicações de longo alcance na manutenibilidade, testabilidade e produtividade do desenvolvedor.

2.1 O Espectro de Opções no Rails: ERB, ViewComponent e Phlex

O Rails oferece um espectro de tecnologias para a camada de visualização, desde templates tradicionais até objetos Ruby puros.

- **ERB (Embedded Ruby) / Partials:** O motor de templates padrão e testado pelo tempo no Rails. A lógica é embutida diretamente no HTML, e a reutilização é alcançada através de `partials`. É familiar para todos os desenvolvedores Rails e excelente para prototipagem rápida. No entanto, pode se tornar inchado e difícil de manter à medida que as aplicações crescem, misturando lógica de apresentação com lógica de negócios.
- **ViewComponent:** Um framework para construir componentes de visualização reutilizáveis, testáveis e encapsulados. Cada componente é uma classe Ruby com seu próprio template, encapsulando sua lógica e apresentação. Isso promove a modularidade, melhora drasticamente a testabilidade e clarifica o fluxo de dados. É notavelmente mais rápido que os partials e é usado extensivamente por empresas como o GitHub. Ele suporta padrões avançados como "slots" para composição de componentes.
- **Phlex:** Uma abordagem mais recente e radical onde as views são escritas como objetos Ruby puros usando uma DSL para gerar HTML, eliminando arquivos de template separados. Isso oferece máxima clareza na separação da lógica e uma estrutura de componentes semelhante ao React, mas vem com uma curva de aprendizado mais acentuada e um ecossistema menos maduro.

2.2 Análise de Trade-offs

Uma análise comparativa das tecnologias de visualização é essencial para tomar uma decisão informada. A tabela a seguir resume os principais trade-offs.

Tabela 2.1: Análise Comparativa das Tecnologias da Camada de Visualização do Rails

Critério	ERB / Partials	ViewComponent	Phlex
Desempenho	Linha de base. Pode ser lento com lógica complexa.	Mais rápido que partials (~2.5-10x). Templates são pré-compilados.	Potencialmente o mais rápido, por ser geração de objetos Ruby puros.
Testabilidade	Difícil de testar isoladamente; requer testes de integração/controlador.	Excelente. Componentes são objetos Ruby que podem ser testados unitariamente de forma rápida e isolada.	Excelente. Classes Ruby puras são inerentemente fáceis de testar unitariamente.
Escalabilidade & Manutenibilidade	Baixa. Propenso a "código espaguete" em templates grandes; a lógica se torna dispersa.	Alta. Impõe separação de preocupações e encapsulamento, levando a uma biblioteca de componentes limpa e manutenível.	Alta. Impõe separação de lógica limpa, mas a própria DSL pode ser um novo desafio de manutenção.
Curva de Aprendizado	Nenhuma para desenvolvedores Rails.	Moderada. Requer a compreensão do ciclo de vida do componente e do design de views orientado a objetos.	Acentuada. Requer o aprendizado de uma nova DSL para escrever HTML em Ruby.
Ecossistema & Maturidade	Central para o Rails há mais de uma década.	Maduro e amplamente adotado por grandes empresas como	Ecossistema mais jovem com menos

		GitHub, GitLab e Shopify.	práticas estabelecidas.
Recomendado para o CAMAAR?	Não. A complexidade dos formulários e das views baseadas em papéis levaria a templates insustentáveis.	**Sim.** Oferece o melhor equilíbrio de estrutura, desempenho e testabilidade para uma aplicação complexa e de longo prazo.	Não. A curva de aprendizado e o risco do ecossistema superam os benefícios potenciais para este projeto.

2.3 Recomendação: Adotando uma Arquitetura Orientada a Componentes com ViewComponent

Com base na análise da Tabela 2.1, **ViewComponent é a escolha ideal para o CAMAAR**. As histórias de usuário descrevem um sistema com elementos de UI distintos e reutilizáveis (formulários, listas de templates, views de relatórios) que são candidatos perfeitos para componentes. A necessidade de views diferentes para Administradores e Participantes reforça ainda mais a necessidade de uma abordagem estruturada, testável e manutenível que os partials de ERB não podem fornecer em escala. Embora o Phlex seja interessante, sua curva de aprendizado apresenta um risco desnecessário ao projeto. O ViewComponent oferece 80% do benefício de uma abordagem puramente Ruby com apenas 20% da curva de aprendizado, tornando-o a escolha pragmática e poderosa.

2.4 Estruturando a Aplicação: Layouts, Blocos e Componentes Atômicos

Adotar o ViewComponent não é suficiente; uma estratégia clara para organizar os componentes é necessária. Uma estrutura de três níveis, inspirada em sistemas de design do mundo real, é recomendada :

1. **Layouts:** A estrutura de mais alto nível (por exemplo, `IndexViewLayoutComponent`, `FormViewLayoutComponent`). Eles definem preocupações no nível da página, como barras laterais e comportamento responsivo.
2. **Blocos:** Composições de múltiplos componentes para um propósito específico (por exemplo, um `DataTableBlockComponent` contendo componentes de tabela, paginação e filtro).
3. **Componentes:** Os menores blocos de construção, atômicos (por exemplo, `ButtonComponent`, `IconComponent`).

Uma questão importante é se os próprios layouts devem ser ViewComponents ou permanecer como layouts tradicionais do Rails (`application.html.erb`) que utilizam `yield`. Uma abordagem equilibrada é a melhor. Recomenda-se o uso de layouts padrão do Rails (

`app/views/layouts/`) para o invólucro de nível superior absoluto (as tags `<html>`, `<head>`, `<body>` e a inclusão de ativos JS/CSS). No entanto, o conteúdo *dentro* da tag `<body>` deve ser renderizado por meio de componentes de layout de alto nível. Por exemplo, `application.html.erb` conteria `<%= render(PageLayoutComponent.new) do %> <%= yield %> <% end %>`. Este `PageLayoutComponent` gerenciaria a renderização da navegação, barras laterais e áreas de conteúdo principal, fornecendo estrutura e ao mesmo tempo aproveitando o sistema de layout nativo do Rails.

Seção 3: Criando uma Experiência de Usuário Moderna com Hotwire e Tailwind CSS

Esta seção detalha as escolhas tecnológicas para estilização e interatividade, construindo sobre a arquitetura de componentes definida na Seção 2.

3.1 Aproveitando o Hotwire para Interatividade Semelhante a SPAs

As histórias de usuário implicam uma UI dinâmica (por exemplo, criação de formulários, gerenciamento de templates) que não deve exigir recarregamentos de página completos. Hotwire é a solução padrão do Rails para isso.

Hotwire é um termo guarda-chuva para Turbo e Stimulus, que proporciona uma experiência rápida e interativa enviando HTML pela rede, em vez de JSON.

- **Turbo Drive:** Acelera a navegação transformando cliques em links e submissões de formulários em requisições AJAX.
- **Turbo Frames:** Permite que seções independentes de uma página sejam atualizadas sem afetar o restante.
- **Turbo Streams:** Permite que o servidor envie atualizações parciais da página em resposta a ações ou via WebSockets, habilitando recursos em tempo real. Esta é a tecnologia chave para os formulários aninhados dinâmicos.

A alternativa ao Hotwire seria um framework front-end JavaScript completo como o React, o que introduziria uma complexidade significativa: uma API separada, gerenciamento de estado do lado do cliente e uma segunda pilha de tecnologia para manter. As histórias de usuário do CAMAAR, embora exijam UIs dinâmicas, são todas centradas em recursos do lado do servidor. Portanto, Hotwire não é apenas um "padrão", é uma escolha estratégica para

evitar complexidade desnecessária, permitindo que a equipe entregue a UX moderna necessária, mantendo toda a lógica dentro do ecossistema familiar do Rails.

3.2 Aprimorando a Interatividade com Stimulus

Enquanto o Turbo lida com a maioria das interações, pequenos comportamentos do lado do cliente (por exemplo, alternar um menu suspenso, limpar um campo de formulário) ainda são necessários. O Stimulus é um framework JavaScript "modesto" que conecta objetos JS (controladores) a elementos HTML usando atributos `data-*`. Ele não assume a renderização; apenas adiciona comportamento. É o companheiro perfeito para o Turbo, ideal para lidar com interações pequenas e direcionadas que não justificam uma viagem de ida e volta ao servidor. Um exemplo perfeito é um controlador Stimulus para adicionar/remover dinamicamente campos de formulário aninhados no lado do cliente.

3.3 Estratégia de Estilização: Por que o Tailwind CSS é a Escolha Ideal

Um framework CSS é necessário para estilizar a aplicação. A escolha principal é entre os dois líderes de mercado: Bootstrap e Tailwind CSS.

- **Bootstrap:** Um framework baseado em componentes que fornece componentes pré-construídos e pré-estilizados (`.card`, `.btn-primary`). É ótimo para desenvolvimento rápido, mas pode levar a UIs de aparência genérica e requer a substituição de estilos para personalização, o que pode ser complicado.
- **Tailwind CSS:** Um framework "utility-first" que fornece classes de utilidade de baixo nível (`.bg-blue-500`, `.p-4`) que são compostas diretamente no HTML para construir designs personalizados. Oferece flexibilidade incomparável e evita a necessidade de escrever CSS personalizado.

A escolha do ViewComponent na Seção 2 cria uma forte inclinação para o Tailwind CSS. O Bootstrap fornece tanto a estrutura (HTML) quanto o estilo (CSS) em seus componentes. O ViewComponent é responsável pela estrutura. Usar Bootstrap com ViewComponent cria um conflito: você está usando o componente Bootstrap ou construindo seu próprio ViewComponent que apenas envolve as classes Bootstrap? Isso leva a abstrações desajeitadas. O Tailwind, por ser "utility-first", não tem opiniões sobre a estrutura do componente; ele apenas fornece primitivas de estilo. Esta é uma combinação perfeita para o ViewComponent. O template ERB de um `ButtonComponent` pode ser composto por classes de utilidade do Tailwind, encapsulando totalmente a estrutura *e* a aparência do componente. Esta combinação, alimentada pelos design tokens da Seção 1, cria um verdadeiro sistema de design em código que é muito mais coeso e manutenível.

3.4 Integrando Tailwind CSS com ViewComponent para um Sistema de Design Coeso

A prática recomendada é aplicar as classes do Tailwind diretamente dentro do template do componente (`.html.erb`). A classe Ruby do componente (`.rb`) pode conter lógica para aplicar classes condicionalmente.

Padrão de Implementação:

1. Um `ButtonComponent` é definido no Figma usando auto-layout e vinculado aos design tokens.
2. O arquivo `tokens.css` é gerado a partir desses tokens.

Um arquivo `button_component.rb` é criado:
Ruby
```
class ButtonComponent < ViewComponent::Base
  def initialize(label:, type: :primary)
    @label = label
    @type = type
  end

  private

  def type_classes
    case @type
    when :secondary
      "bg-gray-500 hover:bg-gray-700 text-white"
    else
      "bg-blue-500 hover:bg-blue-700 text-white"
    end
  end
end
```

3.

Um arquivo `button_component.html.erb` é criado:
Snippet de código
```
<button class="font-bold py-2 px-4 rounded <%= type_classes %>">
  <%= @label %>
</button>
```

4.
5. Este componente é então renderizado em outras views: `<%= render(ButtonComponent.new(label: "Criar Template")) %>`.

Seção 4: Implementando Funcionalidades Centrais da Aplicação

Esta seção traduz as decisões arquitetônicas em padrões de implementação concretos para as principais histórias de usuário.

4.1 Autenticação e Autorização de Usuários

4.1.1 Personalizando as Views do Devise

As histórias de usuário para login, redefinição de senha e configuração de novo usuário exigem views de autenticação. Devise é a gem padrão para isso, mas suas views padrão precisam ser personalizadas para corresponder ao protótipo do Figma. O processo envolve a execução de `rails generate devise:views`. Para criar views separadas para diferentes modelos de usuário (por exemplo, `Admin` vs. `User`), é necessário definir `config.scoped_views = true` em `config/initializers/devise.rb` e, em seguida, executar o gerador com o nome do modelo, por exemplo, `rails generate devise:views users`. Isso copia os templates ERB para

`app/views/users/`, onde podem ser estilizados com as classes Tailwind CSS do projeto.

4.1.2 Implementando Controle de Acesso Baseado em Papéis com Pundit

A aplicação possui papéis distintos ("Administrador", "Participante") com permissões diferentes. Pundit é uma gem de autorização flexível e baseada em políticas, mais adequada para este cenário do que o CanCanCan, que é mais centralizado. O CanCanCan centraliza todas as regras em um único arquivo `Ability`, o que é simples para papéis básicos, mas pode se tornar difícil de gerenciar. O Pundit usa objetos `Policy` dedicados para cada modelo (por exemplo, `TemplatePolicy`, `FormPolicy`), o que representa uma abordagem mais limpa e orientada a objetos que escala melhor com permissões complexas. A configuração envolve adicionar a gem, executar

`rails g pundit:install` e incluir `Pundit::Authorization` no `ApplicationController`.

4.1.3 Padrão para Renderização Condicional nas Views

A UI deve mudar com base no papel do usuário. Por exemplo, um Administrador vê os botões "Editar" e "Deletar" para um template, enquanto um Participante não. O Pundit fornece um método auxiliar `policy` para uso nas views. O padrão é envolver elementos de UI condicionais em um bloco `if` que verifica a política.

Padrão de Implementação:

Definir uma `TemplatePolicy` para o modelo `Template`:
Ruby
```
# app/policies/template_policy.rb
class TemplatePolicy < ApplicationPolicy
  def update?
    user.admin? # Apenas administradores podem atualizar
  end

  def destroy?
    user.admin? # Apenas administradores podem deletar
  end
end
```

1.

Na view (ou, mais provavelmente, em um `TemplateComponent`), usar o auxiliar `policy`:
Snippet de código
```
<%# app/views/templates/show.html.erb ou um componente %>
<h2><%= @template.name %></h2>

<% if policy(@template).update? %>
  <%= link_to "Editar", edit_template_path(@template) %>
<% end %>

<% if policy(@template).destroy? %>
  <%= link_to "Deletar", @template, data: { turbo_method: :delete, turbo_confirm: "Tem
certeza?" } %>
<% end %>
```

2.

Este padrão é muito mais limpo e manutenível do que verificar papéis diretamente (por exemplo, `if current_user.admin?`), pois a lógica de autorização está encapsulada no objeto de política.

4.2 Manipulação Avançada de Formulários para Avaliações

A funcionalidade de UI mais complexa é a criação de um template de formulário, que envolve um modelo pai `Template` e muitos modelos aninhados `Question`. Isso requer um formulário aninhado dinâmico.

A base para isso é o `accepts_nested_attributes_for` do Rails no modelo pai e o `fields_for` na view. Historicamente, a adição e remoção dinâmica de campos era feita com

bibliotecas JavaScript como Cocoon. A abordagem moderna e nativa do Hotwire é usar Turbo Streams.

Uma solução puramente do lado do servidor usando Turbo Streams é mais simples e alinhada com a filosofia Hotwire. O padrão, sintetizado a partir de várias fontes, é o seguinte :

1. O botão "Adicionar Pergunta" é um `link_to` ou `button_tag` que faz uma requisição GET para uma ação `new` dedicada (por exemplo, `questions#new`) com `data: { turbo_stream: true }`.
2. A ação `QuestionsController#new` apenas responde ao formato turbo stream.
3. Uma view `new.turbo_stream.erb` correspondente é criada. Esta view contém uma tag `<turbo-stream>` com `action="append"` direcionada a um `div` container no formulário principal (por exemplo, `<div id="questions">`).
4. Dentro da tag `turbo-stream`, `fields_for` é usado para renderizar um novo conjunto de campos para um objeto `Question.new`. Um índice único é gerado no servidor (por exemplo, `index: Time.current.to_i`) para evitar a complexidade do lado do cliente.
5. Este stream é enviado de volta ao navegador, e o Turbo anexa automaticamente os novos campos ao formulário.

Esta abordagem mantém toda a lógica, incluindo a geração de nomes de campos e índices únicos, no servidor, o que é mais simples e robusto. É o padrão recomendado para o CAMAAR.

4.3 Gerenciamento e Apresentação de Dados

4.3.1 Implementando Paginação Eficiente

As views que listam templates e formulários exigirão paginação. As duas principais gems são Kaminari e Pagy. Embora ambas funcionem, benchmarks mostram consistentemente que Pagy é significativamente mais rápido e usa muito menos memória do que Kaminari. Para uma nova aplicação como o CAMAAR, sem restrições de legado,

Pagy é a escolha técnica clara.

4.3.2 Um Padrão Robusto para Exportar Resultados de Formulários para CSV

Os administradores precisam baixar os resultados dos formulários como um arquivo CSV. O padrão padrão do Rails para isso é :

1. Adicionar um formato `.csv` ao bloco `respond_to` na ação `index` (ou `show`) do controlador.
2. No modelo, criar um método de classe `self.to_csv` que usa a biblioteca `CSV` embutida do Ruby para gerar a string CSV.

3. No controlador, chamar `send_data` com o resultado do método `to_csv` e um nome de arquivo especificado.
4. Na view, criar um link para o caminho atual com `format: :csv`.

Recomenda-se seguir este padrão. Para exportações muito grandes, este processo deve ser movido para um trabalho em segundo plano (`background job`) para evitar o tempo limite da requisição web, mas para a implementação inicial, a ação síncrona do controlador é suficiente.

Seção 5: Síntese e Recomendações Estratégicas

Esta seção final consolida as recomendações em uma pilha de tecnologia coesa e fornece um roteiro de alto nível.

5.1 Pilha de Tecnologia Consolidada e Justificativa

- **Arquitetura de View:** ViewComponent
- **Estilização:** Tailwind CSS, gerenciado por um sistema de design tokens sincronizado do Figma.
- **Interatividade:** Hotwire (Turbo e Stimulus).
- **Autenticação:** Devise, com views personalizadas.
- **Autorização:** Pundit.
- **Paginação:** Pagy.

Esta pilha representa uma abordagem moderna, coesa e "nativa do Rails". Cada escolha de tecnologia reforça as outras: ViewComponent funciona melhor com Tailwind, que é alimentado por design tokens. Hotwire aproveita a natureza renderizada no servidor dos componentes. Pundit se integra de forma limpa nas views e componentes para controlar a renderização. Este não é apenas um conjunto de ferramentas populares, mas um sistema sinérgico projetado para produtividade, manutenibilidade e escalabilidade.

5.2 Roteiro de Implementação em Fases e Considerações Chave

- **Fase 1: Fundação.**
    - Configurar o pipeline de tokens do Figma para o Rails.
    - Instalar e configurar o Rails com ViewComponent, Hotwire, Tailwind, Devise, Pundit e Pagy.
    - Implementar as views de autenticação básicas e os papéis de usuário.
    - Construir o layout principal da aplicação como um ViewComponent.
- **Fase 2: Funcionalidades Principais do Administrador.**
    - Implementar o formulário aninhado dinâmico para criar e gerenciar templates de formulário usando o padrão Turbo Stream.
    - Construir as views para listar e gerenciar templates, incorporando paginação (Pagy) e autorização (Pundit).
- **Fase 3: Funcionalidades do Participante e Relatórios.**

    - Implementar as views para os participantes verem e responderem aos formulários.
    - Construir a view de resultados do formulário para os administradores.
    - Implementar a funcionalidade de exportação para CSV.

Consideração Chave: A parte tecnicamente mais desafiadora é o formulário aninhado dinâmico. Recomenda-se abordá-lo no início (Fase 2) para mitigar os riscos do projeto. A equipe deve construir um protótipo simples dessa funcionalidade primeiro para validar a abordagem Turbo Stream escolhida.