Relatório de Qualidade - Visitas+

1. Aplicação de Código Limpo

Durante todo o desenvolvimento, foram aplicados os princípios de Código Limpo para garantir a legibilidade, manutenibilidade e qualidade do software.

Nomenclatura

- **Exemplos de bons nomes utilizados**: Foram escolhidos nomes intencionais e que revelam o propósito.
  - **Modelos**: `Cliente`, `Visita`. Nomes claros que representam as entidades do domínio do problema.
  - **Views**: `listar_clientes`, `detalhe_cliente`, `nova_visita`. Nomes que descrevem precisamente a ação que a função executa.
  - **Variáveis**: `clientes_do_banco`, `anotacoes_completas`. Nomes explícitos que evitam ambiguidade e abreviações.

Estrutura de Funções

- **Tamanho médio das funções**: A ferramenta `radon` apontou uma **Complexidade Ciclomática Média de 'A'**, indicando que as funções são pequenas, diretas e com poucos caminhos lógicos, atendendo ao critério de serem idealmente menores que 20-30 linhas, com média de 19.1 linhas por método.
- **Exemplos de funções bem estruturadas**: A função `listar_clientes` é um bom exemplo de responsabilidade única, focada apenas em buscar os dados e renderizar o template, delegando a apresentação para o HTML.

```
# app_visitas/views.py

@login_required
def listar_clientes(request: HttpRequest) -> HttpResponse:
    """Exibe a página com a lista de todos os clientes cadastrados."'
    clientes = Cliente.objects.all().order_by("nome")
    contexto = {"clientes": clientes}
    return render(request, "clientes/listar_clientes.html", contexto)
```

Formatação

- **Padrões de indentação**: O código segue o padrão de 4 espaços para indentação, garantindo consistência visual.
- **Organização visual do código**: Foi aplicada uma formatação automática para padronizar o espaçamento vertical entre funções, a organização dos imports e o comprimento das linhas.

2. Code Smells Identificados

A tabela a seguir resume os principais problemas de qualidade identificados pela ferramenta `pylint` e o status de cada um após a refatoração.

Code Smell	Localização	Severidade	Status
Poor Naming	Pasta `App_Visitas/`	Baixa	Corrigido
Missing Docstring	Múltiplos arquivos `.py`	Média	Corrigido
Broad Exception Caught	`app_visitas/views.py`	Média	Corrigido
Inconsistent Formatting	Múltiplos arquivos `.py`	Baixa	Corrigido

Com certeza! Aqui está a seção "Refatorações Realizadas" do seu relatório, preenchida com todos os detalhes do `Log de Refatorações` que você forneceu.

Usei a **Refatoração #3 (Refinar Tratamento de Exceções)** como o exemplo principal com o bloco de código "Antes e Depois", pois é a mudança mais impactante na lógica e segurança do código, o que a torna um excelente caso de estudo para o relatório.

---

3. Refatorações Realizadas

Refatoração 1: Rename Module (Renomear Módulo)

- **Data**: 2025-10-02
- **Code Smell**: Poor Naming (`invalid-name`)
- **Técnica Aplicada**: Rename Method/Module
- **Arquivos Afetados**: Pasta `App_Visitas/` para `app_visitas/` , `visitas/settings.py` , `visitas/urls.py` .
- **Justificativa**: O nome do módulo não seguia a convenção `snake_case` do Python. A mudança garante consistência e aderência às boas práticas da linguagem, melhorando a legibilidade.

- **Impacto**: O código se torna mais idiomático e profissional, eliminando o aviso do `pylint` e facilitando a identificação do módulo.

Refatoração 2: Add Docstrings (Adicionar Documentação)

- **Data**: 2025-10-07
- **Code Smell**: Código não documentado (Missing Docstring).
- **Técnica Aplicada**: Documentação de Código.
- **Arquivos Afetados**: `app_visitas/views.py`, `app_visitas/models.py`, `app_visitas/forms.py`, etc.
- **Justificativa**: A ausência de `docstrings` dificultava o entendimento rápido do propósito de cada componente do sistema. Adicionar documentação é um pilar do Código Limpo.
- **Impacto**: Melhora drástica na legibilidade e manutenibilidade. Outro desenvolvedor pode entender a função de cada parte do código sem precisar ler toda a sua implementação.

Refatoração 3: Refine Exception Handling (Refinar Tratamento de Exceções)

- **Data**: 2025-10-03
- **Code Smell**: Broad Exception Caught (Captura de Exceção Genérica).
- **Técnica Aplicada**: Ser específico com as exceções.
- **Arquivos Afetados**: `app_visitas/views.py`.

Antes:
O código capturava a exceção genérica `Exception`, o que poderia mascarar bugs inesperados e dificultar a depuração.

```
# app_visitas/views.py (versão antiga)

try:
    # ... lógica para salvar a visita ...
except Exception as e:
    return JsonResponse({'sucesso': False, 'mensagem': f'Ocorreu um erro
```

Depois:
O tratamento de erro foi refinado para capturar apenas as exceções esperadas (`JSONDecodeError` e `Cliente.DoesNotExist`), tornando o comportamento do sistema mais previsível e robusto.

```
# app_visitas/views.py (versão refatorada)
```

```
    try:
        # ... lógica para salvar a visita ...
    except (json.JSONDecodeError, Cliente.DoesNotExist):
        return JsonResponse({
            "sucesso": False,
            "mensagem": "Erro nos dados ou cliente não encontrado.",
        })
```

Justificativa: Capturar `Exception` de forma genérica é uma má prática que pode ocultar falhas. A refatoração para exceções específicas torna o código mais seguro e confiável, pois garante que apenas os erros previstos estão sendo tratados, permitindo que falhas inesperadas sejam identificadas e corrigidas durante o desenvolvimento.

Refatoração 4: Apply Code Formatter (Aplicar Formatador de Código)

- **Data**: 2025-10-10
- **Code Smell**: Inconsistent Formatting (Formatação Inconsistente).
- **Técnica Aplicada**: Formatação Automática de Código.
- **Arquivos Afetados**: Múltiplos arquivos `.py`.
- **Justificativa**: Inconsistências na formatação visual do código dificultam a leitura e a colaboração. A aplicação de um formatador automático padroniza o estilo em todo o projeto.
- **Impacto**: Garante um padrão visual consistente em todo o código, melhorando a legibilidade e permitindo que os desenvolvedores se concentrem na lógica, e não no estilo.

4. Ferramentas Utilizadas

- **Análise estática**: `pylint`, para identificar violações de estilo, erros de programação e *code smells*.
- **Métricas de qualidade**: `radon`, para medir a Complexidade Ciclomática e a média de linhas por método.

5. Próximos Passos

- **Code smells ainda a corrigir**: Todos os *code smells* relevantes identificados pelo `pylint` foram corrigidos. Os avisos restantes são "falsos positivos" relacionados à forma como o Django gera atributos dinamicamente em seus modelos.
- **Melhorias planejadas para U3**:
    - Implementar testes unitários para as views e modelos, aumentando a confiabilidade do código.

- Refatorar o código JavaScript, separando a lógica de coleta de dados da interface da lógica de envio para a API, aplicando o Princípio da Responsabilidade Única.
- Adicionar `type hints` em mais partes do código para melhorar a clareza e permitir uma verificação estática mais robusta.
- Melhoria visual de todo o App, visando uma identidade visual definida.
- Adição de novos parâmetros relevantes tanto para a criação de clientes no banco de dados, como também na realização de visitas.