README.md

SystemInfo

SystemInfo é uma ferramenta em .NET para coletar, exibir e manipular informações do sistema Windows utilizando **WMI (Windows Management Instrumentation)**.

Funcionalidades

- Exibir fingerprint do sistema (CPU, memória, disco, rede, etc.).
- Consultar classes WMI e visualizar suas propriedades.
- Parar serviços desnecessários do Windows.
- Ajustar prioridades de processos com base em uso.
- Testar adaptadores de rede (execução real e modo dry-run).
- Otimizar partições de disco.

Estrutura do Projeto

- **Core** → Abstrações (`IWmiQueryService`, `IWmiMethodInvoker`).
- **Configuration** → Mapas de classes/propriedades WMI.
- **Features** → Funcionalidades (serviços, disco, fingerprint).
- **Infrastructure** → Implementações WMI, fila de invocações, helpers.
- **Presentation** → Menus interativos.
- **Program.cs** → Ponto de entrada.

Documentação

- [`USER_MANUAL.md`](USER_MANUAL.md) → Manual do usuário.
- [`INSTALL.md`](INSTALL.md) → Instalação e execução.
- [`MANUAL_DEV.md`](MANUAL_DEV.md) → Guia do desenvolvedor.
- [`API_GUIDE.md`](API_GUIDE.md) → Referência da API.

■ Nota Importante

Este projeto já fornece uma API pronta para:

- Consultas (`GetScalar`, `QuerySelect`, `QueryAll`).
- Invocação de métodos WMI (`Invoke`).

- Manipulação de serviços/processos/discos.

Alguns exemplos mostrados na documentação (ex.: exportar JSON, integração com API REST, jobs em agendador, dashboards) **não são métodos nativos**, mas **cenários de uso/integração possíveis** com a API já existente.

USER_MANUAL.md

Manual do Usuário – SystemInfo

1. Introdução

SystemInfo é uma ferramenta em linha de comando para visualizar e gerenciar informações do sistema operacional Windows.

Através de menus interativos, você pode inspecionar classes WMI, exibir fingerprints do sistema, controlar serviços e processos, e otimizar recursos.

---

2. Como Navegar

Ao executar o programa (`dotnet run`), o usuário verá um **menu principal** com opções numeradas.

Use o teclado para digitar o número correspondente à funcionalidade desejada.

---

3. Funcionalidades

3.1 Fingerprint

- Lista classes de hardware e software (CPU, memória, disco, rede etc.).
- Permite selecionar quais classes e propriedades visualizar.
- Pergunta ao usuário se deseja ver todas as propriedades disponíveis.

3.2 Exibir Classes WMI

- Apresenta um menu interativo com todas as classes do `WmiClassMap`.
- Permite escolher uma classe e listar valores de propriedades.
- Pergunta se o usuário deseja ver todas as propriedades disponíveis.

3.3 Serviços

- Detecta serviços desnecessários do Windows.
- Pergunta se o usuário deseja pará-los.
- Exemplo: Fax, XPS, Xbox Services etc.

3.4 Processos

- Mostra processos em execução.
- Pergunta se o usuário deseja **reduzir prioridade** de processos pouco usados.
- Pergunta se deseja **aumentar prioridade** de processos críticos.

3.5 Disco

- Lista partições e espaço disponível.
- Oferece ferramenta básica de otimização.

3.6 Rede

- Permite testar adaptadores de rede.
- Dois modos:
- **Execução real** (`NetworkAdapterTesterReal`)
- **Dry-run** para testes sem efeito (`NetworkAdapterTesterDryRun`)

---

4. Requisitos

- Windows 10 ou superior.
- .NET 6 SDK ou runtime instalado.
- Permissões de administrador para gerenciar serviços/discos.

---

5. Exemplos de Uso

- **Exibir versão do Windows:**
- Menu → Fingerprint → OperatingSystem → Property: Version
- **Parar o serviço de spooler:**
- Menu → Serviços → Selecionar “Spooler”
- **Otimizar disco:**

- Menu → Disco → Otimizar

■ Nota Importante

Este manual mostra:

- **Funcionalidades reais** já disponíveis no menu (Fingerprint, Serviços, Processos, Disco, Rede).

- **Cenários opcionais de uso** que dependem da confirmação do usuário (parar serviços, mudar prioridade de processos etc.).

A API também pode ser usada diretamente em outros projetos para cenários avançados (como exportar fingerprint em JSON ou integrar a uma API REST). Esses casos **não estão no menu padrão**, mas podem ser implementados usando a API já fornecida pelo código.

INSTALL.md

Guia de Instalação – SystemInfo

1. Requisitos

- **Sistema Operacional**: Windows 10 ou superior.

- **.NET**: SDK ou Runtime do .NET 6 instalado.

- **Permissões**: privilégios de administrador são necessários para:

- Parar/alterar serviços.

- Alterar prioridade de processos.

- Otimizar partições de disco.

---

2. Instalação

1. Clone ou baixe o repositório:

```bash
git clone https://github.com/seu-usuario/SystemInfo.git
cd SystemInfo
```

2. Compile o projeto:

```bash
dotnet build
```

```

3. Execute:

```bash
dotnet run
```

---

3. Estrutura de Pastas

- **/Core** → Interfaces principais (`IWmiQueryService`, `IWmiMethodInvoker`).
- **/Configuration** → Mapas de classes/propriedades WMI.
- **/Features** → Funcionalidades (serviços, processos, disco, fingerprint).
- **/Infrastructure** → Implementações concretas (WMI, helpers, filas).
- **/Presentation** → Menus interativos.
- **Program.cs** → Entrada do programa.

---

4. Executando como Administrador

Muitas funcionalidades (ex.: parar serviços, mudar prioridades, otimizar discos) **exigem execução como administrador**.

Para garantir:

- No PowerShell:

```powershell
Start-Process "dotnet" "run" -Verb RunAs
```

---

5. Problemas Comuns

- **Erro de acesso negado** → Execute como administrador.
- **Classe WMI não encontrada** → Certifique-se de estar no Windows 10+.
- **Menu não responde** → Confira se o terminal aceita entrada interativa.

■ Nota Importante

Este guia cobre:
```

- **Passos reais de instalação e execução** do projeto.

- **Permissões necessárias** para funcionalidades críticas.

Outros exemplos da documentação (como integração em APIs REST, exportação de JSON ou execução via agendadores) **não fazem parte da instalação padrão**, mas são cenários de uso possíveis aproveitando a API já fornecida pelo projeto.

MANUAL_DEV.md

Manual do Desenvolvedor – SystemInfo

1. Introdução

Este manual explica como **funciona internamente** o projeto SystemInfo e como você pode **estender, integrar e testar** suas funcionalidades.

O projeto foi desenhado para ser modular, escalável e fácil de integrar a outros sistemas.

Estrutura do Projeto

- **Core** → Interfaces, abstrações e utilitários.

- **Configuration** → Mapeamentos de classes/propriedades WMI.

- **Features** → Funcionalidades (Fingerprint, Processos, Serviços, Disco).

- **Infrastructure** → Implementações concretas (WMI, sistema, output).

- **Presentation** → Menus e interação com usuário.

- **Program.cs** → Ponto de entrada.

---

2. Consultas WMI

Buscar valor único (`GetScalar`)

```csharp
var wmi = new WmiQueryService();
string version = wmi.GetScalar("OperatingSystem", "Version");
Console.WriteLine($"Versão do Windows: {version}");
```

Buscar múltiplas propriedades (`QuerySelect`)

```csharp
var wmi = new WmiQueryService();
var cpus = wmi.QuerySelect("Processor", new[] { "Name", "NumberOfCores" });
foreach (var cpu in cpus)
Console.WriteLine($"{cpu["Name"]} - {cpu["NumberOfCores"]} cores");
```

Buscar todos os objetos (`QueryAll`)

```csharp
var wmi = new WmiQueryService();
var disks = wmi.QueryAll("LogicalDisk");
foreach (var d in disks)
Console.WriteLine($"{d["DeviceID"]} - {d["FreeSpace"]}");
```

Usando WHERE

```csharp
var wmi = new WmiQueryService();
var c = wmi.GetScalar("Service", "Name", "Name='Spooler'");
Console.WriteLine($"Serviço encontrado: {c}");
```

---

3. Invocando Métodos WMI

Método sem parâmetros (`StopService`)

```csharp
var invoker = new WmiMethodInvoker();
var result = invoker.Invoke("Service", "StopService", "Name='Fax'");
Console.WriteLine($"Código de retorno: {result}");
```

Método com parâmetros (`SetPowerState`)

```csharp
var invoker = new WmiMethodInvoker();
var result = invoker.Invoke("DiskDrive", "SetPowerState", null, new Dictionary
{
{ "PowerState", 4 }, // Hibernate
{ "Time", 0 }
});
```

Interpretando códigos

```csharp
string msg = WmiReturnCodeHelper.Describe(result);
Console.WriteLine(msg);
```

---

4. Integração no Projeto

- O **menu principal** usa `WmiQueryService` e `WmiMethodInvoker` para executar ações.
- O **fingerprint** usa `WmiClassMap` e `WmiPropertiesMap` para saber o que exibir.
- As **features** ficam separadas em `Features/`, e podem ser adicionadas sem alterar o core.

---

5. Extensibilidade

Nova classe no `WmiClassMap`

```csharp
["Battery"] = "Win32_Battery"
```

Novas propriedades no `WmiPropertiesMap`

```csharp
["Battery"] = new[] { "Name", "EstimatedChargeRemaining" }
```

Criando novo feature

- Criar classe em `Features/`.
- Usar `IWmiQueryService` para consultas.
- Adicionar ao menu principal.

---

6. Boas práticas

- Evite consultas que retornam milhares de objetos.
- Sempre valide se a propriedade existe.
- Rode como administrador quando mexer com serviços/discos.
- Nunca pare serviços críticos.

---

7. Integração Externa

- Referencie `SystemInfo.dll` em outro projeto.
- Exemplo em API REST (ASP.NET Core):

```csharp
[HttpGet("cpu")]
public IActionResult GetCpu([FromServices] IWmiQueryService wmi)
{
var cpu = wmi.GetScalar("Processor", "Name");
return Ok(cpu);
}
```

- Em agendadores (Quartz.NET, Task Scheduler) → chame métodos do `WmiQueryService` em jobs.

---

8. Persistência

Salvar fingerprint em JSON:

```csharp
var wmi = new WmiQueryService();
var sys = wmi.QuerySelect("OperatingSystem", new[] { "Caption", "Version" });
File.WriteAllText("fingerprint.json", JsonSerializer.Serialize(sys));
```

```

---

## 9. Testabilidade

- `NetworkAdapterTesterDryRun` é exemplo de mock.

- Você pode implementar `IWmiQueryService` fake para testes:

```csharp
class FakeWmi : IWmiQueryService
{
public string GetScalar(string alias, string property, string whereClause = null)
=> "FakeValue";
// ...
}
```

■ Nota Importante

Este manual mostra tanto:

- Funcionalidades **nativas** já implementadas no projeto (`GetScalar`, `QuerySelect`, `QueryAll`, `Invoke`, etc.).

- Quanto **cenários de uso/integração** que podem ser implementados em cima da API existente (ex: exportar fingerprint em JSON, integrar em API REST, usar em agendador).

Ou seja: nada aqui requer mudar o core do código — tudo já é possível com o que está implementado.

API_GUIDE.md

API Guide – SystemInfo

Este documento descreve a API interna do projeto.

---

1. Visão Geral

Interfaces:
```

- `IWmiQueryService`
- `IWmiMethodInvoker`

---

2. IWmiQueryService

`GetScalar`

```csharp
string version = wmi.GetScalar("OperatingSystem", "Version");
```

`QuerySelect`

```csharp
var cpu = wmi.QuerySelect("Processor", new[] { "Name", "NumberOfCores" });
```

`QueryAll`

```csharp
var disks = wmi.QueryAll("LogicalDisk");
```

Exemplos práticos

- Versão do Windows:

```csharp
wmi.GetScalar("OperatingSystem", "Version");
```

- Nome da CPU:

```csharp
wmi.GetScalar("Processor", "Name");
```

- Espaço livre em disco:

```csharp
var c = wmi.GetScalar("LogicalDisk", "FreeSpace", "DeviceID='C:'");
```

---

3. IWmiMethodInvoker

Invoke sem parâmetros

```csharp
invoker.Invoke("Service", "StopService", "Name='Fax'");
```

Invoke com parâmetros

```csharp
invoker.Invoke("DiskDrive", "SetPowerState", null, new Dictionary
{
{ "PowerState", 4 },
{ "Time", 0 }
});
```

---

4. Tratamento de Erros

- Códigos → `WmiReturnCodeHelper.Describe(code)`
- Classe não existe → exceção
- Propriedade inválida → exceção
- Serviço não encontrado → retorno != 0

---

5. Casos Avançados

- Integrar fingerprint:

```csharp
var props = WmiPropertiesMap.Map["Processor"];
var cpu = wmi.QuerySelect("Processor", props);
```

- Filtros dinâmicos:

```csharp
```

wmi.QuerySelect("Service", new[] { "Name" }, "State='Running'");
```

- Consultas em fila:
```csharp
var queue = new WmiMethodInvokerQueue(invoker);
queue.Enqueue("Service", "StopService", "Name='Fax'");
queue.Enqueue("Service", "StopService", "Name='XboxGipSvc'");
queue.ExecuteAll();
```

---

6. Snippets Rápidos

- Valor único:
```csharp
wmi.GetScalar("OperatingSystem", "Caption");
```

- Várias propriedades:
```csharp
wmi.QuerySelect("Processor", new[] { "Name", "NumberOfCores" });
```

- Invocar método:
```csharp
invoker.Invoke("Service", "StopService", "Name='Spooler'");
```

---

7. Extensão da API

Adicionar novo método em `IWmiMethodInvoker`:
```csharp
public uint RestartService(string name)
{
Invoke("Service", "StopService", $"Name='{name}'");
return Invoke("Service", "StartService", $"Name='{name}'");
}
```
```

---

8. Batch Operations

```csharp
var queue = new WmiMethodInvokerQueue(invoker);
queue.Enqueue("DiskDrive", "SetPowerState", null, new { PowerState = 3, Time = 0 });
queue.Enqueue("Service", "StopService", "Name='Fax'");
queue.ExecuteAll();
```

---

9. Performance & Async

- Consultas pesadas → rodar em threads separadas.
- Versão async:

```csharp
await Task.Run(() => wmi.QueryAll("Process"));
```

- Em UI → sempre `await`.

---

10. Exemplos Avançados

Monitor em tempo real

```csharp
while (true)
{
var usage = wmi.GetScalar("Processor", "LoadPercentage");
Console.WriteLine($"CPU: {usage}%");
Thread.Sleep(1000);
}
```

Dashboard de fingerprint

- Salvar fingerprint em JSON

- Consumir em frontend (React, Angular)

Automação com PowerShell

```powershell
dotnet run --project SystemInfo -- get-scalar OperatingSystem Version
```

■ Nota Importante

Os exemplos deste guia incluem:

- **Chamadas reais já suportadas** (`GetScalar`, `QueryAll`, `Invoke`, etc.).

- **Extensões possíveis** que o desenvolvedor pode implementar sobre a API (ex: `RestartService`, exportar dados, dashboards).

Essas extensões são apenas **aplicações práticas** da API já existente.