Explorando o Prototype Pattern em Web APIs usando Csharp

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/explorando-o-prototype-pattern-em-web-apis-usando-csharp
out 07 2024

Abstract:
O artigo explora a aplicação do padrão Prototype em Web APIs utilizando C#. O Prototype permite a criação de novos objetos a partir de instâncias existentes, evitando a complexidade e o custo da criação do zero. A implementação inclui a definição de uma interface IProduct e classes que a implementam, demonstrando a clonagem de produtos e categorias. O uso desse padrão oferece benefícios como aumento de performance, simplicidade e facilidade de manutenção.

Key words:
Prototype Pattern, Web APIs, C#, programação, design patterns, clonagem de objetos, desenvolvimento de software, reutilização de código, arquitetura de software, instância, performance, flexibilidade, abstração, implementação, padrões de projeto, serialização, deserialização, desenvolvimento ágil, testes, manutenção de código.

Explorando o Prototype Pattern em Web APIs usando C#

O padrão Prototype é um dos padrões de design criacionais que permite a criação de novos objetos a partir de um objeto existente, sem a necessidade de depender de classes concretas. Este padrão é especialmente útil em cenários onde a criação de novos objetos é complexa ou custosa. Em desenvolvimento de Web APIs, a aplicação do padrão Prototype pode ajudar a otimizar a criação de instâncias de objetos que compartilham características comuns, facilitando a manutenção e a escalabilidade do código.

Por que usar o Prototype Pattern?

Usar o padrão Prototype traz várias vantagens, como:

Desempenho: Evita o custo de criação de novos objetos a partir do zero.
Flexibilidade: Permite a criação de novos objetos através da clonagem de protótipos.
Simplicidade: Pode simplificar o código ao permitir que objetos sejam criados de maneira uniforme.

Essas vantagens tornam o padrão Prototype uma escolha atraente, especialmente em ambientes onde a performance e a eficiência de recursos são cruciais, como em aplicações que lidam com grandes volumes de dados ou requisições.

Implementando o Prototype em C#

Para ilustrar a implementação do padrão Prototype em C#, vamos criar uma simples API que gerencia uma coleção de produtos. Usaremos o ASP.NET Core para configurar a API e implementar o padrão Prototype.

Definindo a Interface IProduct

Primeiro, vamos definir uma interface IProduct que terá um método Clone:

```
public interface IProduct
{
    IProduct Clone();
}
```

Implementando a Classe Product

Agora, vamos implementar uma classe Product que representa um produto e implementa a interface IProduct:

```
public class Product : IProduct
{
    public string Name { get; set; }
    public decimal Price { get; set; }

    public IProduct Clone()
    {
        return new Product
        {
            Name = this.Name,
            Price = this.Price
        };
    }
}
```

Utilização do ICloneable

É possível usar a interface ICloneable, que já é projetada para este tipo de situação. No entando, é preciso estar ciente de que a interface não define como será feito a clonagem. No exemplo abaixo, está sendo utilizado MemberwiseClone.

```
public class Product : ICloneable
{
    public string Name { get; set; }
    public decimal Price { get; set; }

    public object Clone()
    {
        return MemberwiseClone();
    }
}
```

Controlador ProductController

Agora que temos nosso protótipo implementado, podemos criar uma API que utilize esse padrão. Vamos criar um controlador ProductController:

```
[ApiController]
[Route("api/[controller]")]
public class ProductController : ControllerBase
{
    private readonly List _products;

    public ProductController()
```

```
    {
        _products = new List
        {
            new Product { Name = "Produto A", Price = 10.0m },
            new Product { Name = "Produto B", Price = 20.0m }
        };
    }

    [HttpGet]
    public ActionResult> Get()
    {
        return Ok(_products);
    }

    [HttpPost("clone/{id}")]
    public ActionResult Clone(int id)
    {
        var productToClone = _products.FirstOrDefault(p => p.Id == id);
        if (productToClone == null)
        {
            return NotFound();
        }

        var clonedProduct = (Product)productToClone.Clone();
        _products.Add(clonedProduct);

        return Ok(clonedProduct);
    }
}
```

Nesse controlador, temos um método Clone que aceita um ID de produto, clona o produto correspondente e adiciona o novo produto à lista.

Manipulando Objetos com o Prototype Pattern

Uma das principais aplicações do padrão Prototype é a manipulação de objetos complexos. Considere um cenário em que temos produtos com várias propriedades e relacionamentos. Podemos expandir nossa implementação para incluir um objeto Category, que possui uma lista de produtos.

```
public class Category : ICategory
{
    public string Name { get; set; }
    public List Products { get; set; }

    public ICategory Clone()
    {
        return new Category
        {
            Name = this.Name,
            Products = this.Products.Select(p => (Product)p.Clone()).ToList()
        };
    }
```

```
}
```

Nesse exemplo, quando clonamos uma categoria, também clonamos todos os produtos associados a ela. Isso garante que não estamos apenas referenciando os mesmos objetos, mas criando cópias independentes.

Exemplo Completo de API com Prototype

Agora, vamos ver um exemplo completo de uma API que utiliza o padrão Prototype para gerenciar produtos e categorias.

```
[ApiController]
[Route("api/[controller]")]
public class CategoryController : ControllerBase
{
    private readonly List _categories;

    public CategoryController()
    {
        _categories = new List();
    }

    [HttpPost]
    public ActionResult Create(Category category)
    {
        _categories.Add(category);
        return Ok(category);
    }

    [HttpPost("clone/{id}")]
    public ActionResult Clone(int id)
    {
        var categoryToClone = _categories.FirstOrDefault(c => c.Id == id);
        if (categoryToClone == null)
        {
            return NotFound();
        }

        var clonedCategory = (Category)categoryToClone.Clone();
        _categories.Add(clonedCategory);

        return Ok(clonedCategory);
    }
}
```

Neste controller, temos a capacidade de criar e clonar categorias, utilizando o padrão Prototype para garantir que as instâncias clonadas sejam independentes.

Benefícios do Prototype Pattern em Web APIs

A aplicação do padrão Prototype em Web APIs traz uma série de benefícios, incluindo:

Redução da Complexidade: Ao permitir que objetos sejam clonados em vez de criados do zero, o código se torna mais limpo e fácil de entender.

Aumento da Performance: A clonagem de objetos pode ser mais eficiente do que a criação de novos objetos, especialmente quando a construção do objeto é custosa.

Facilidade de Manutenção: A lógica de clonagem centralizada facilita a manutenção e a atualização da lógica de criação de objetos.

Estendendo o Padrão Prototype

Embora o exemplo acima já aborde os conceitos fundamentais do padrão Prototype, é interessante explorar como podemos estender essa implementação para lidar com cenários mais complexos. Por exemplo, podemos introduzir um sistema de configuração onde diferentes tipos de produtos podem ter características únicas, mas ainda assim se beneficiar da clonagem.

Adicionando Propriedades Dinâmicas

Suponha que desejamos adicionar a capacidade de definir propriedades dinâmicas para nossos produtos. Podemos fazer isso usando um dicionário para armazenar essas propriedades.

```
public class Product : IProduct
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public Dictionary Attributes { get; set; } = new Dictionary();

    public IProduct Clone()
    {
        return new Product
        {
            Name = this.Name,
            Price = this.Price,
            Attributes = new Dictionary(this.Attributes)
        };
    }
}
```

Dessa forma, ao clonar um produto, também clonamos suas propriedades dinâmicas, garantindo que cada instância seja única.

Integração com Outras APIs

Outra consideração importante ao trabalhar com o padrão Prototype em Web APIs é como integrar esses protótipos com outras APIs. Por exemplo, poderíamos ter um serviço que fornece dados de produtos de um sistema externo, e ao clonar um produto, poderíamos querer preencher automaticamente algumas de suas propriedades com dados desse serviço.

Testando a Implementação

É importante testar a implementação do padrão Prototype para garantir que a clonagem funcione como esperado. O uso de testes unitários pode ajudar a validar que os objetos clonados são verdadeiramente independentes e que suas propriedades estão sendo copiadas corretamente.

```
using Xunit;

public class ProductTests
{
```

```
    [Fact]
    public void Clone_ShouldCreateIndependentInstance()
    {
        var originalProduct = new Product
        {
            Name = "Produto Original",
            Price = 30.0m,
            Attributes = new Dictionary { { "Cor", "Azul" } }
        };

        var clonedProduct = (Product)originalProduct.Clone();

        // Assert that the cloned product is not the same instance
        Assert.NotSame(originalProduct, clonedProduct);
        // Assert that the properties are copied correctly
        Assert.Equal(originalProduct.Name, clonedProduct.Name);
        Assert.Equal(originalProduct.Price, clonedProduct.Price);
        // Assert that the attributes are also independent
        Assert.NotSame(originalProduct.Attributes, clonedProduct.Attributes);
    }
}
```

Este teste garante que a clonagem está funcionando corretamente e que as instâncias clonadas são verdadeiramente independentes das originais.

Considerações sobre Desempenho

Embora o padrão Prototype possa melhorar o desempenho na criação de objetos, é importante considerar que a clonagem de objetos complexos ainda pode ser um processo custoso, dependendo da profundidade e da complexidade dos objetos que estão sendo clonados. Portanto, é essencial monitorar o desempenho e otimizar a implementação conforme necessário.

Conclusão

O padrão Prototype fornece uma maneira eficaz de gerenciar a criação de objetos em aplicações de software, especialmente em ambientes de Web APIs. Ele não apenas simplifica a criação de objetos, mas também melhora a performance e facilita a manutenção do código. Ao aplicar esse padrão, os desenvolvedores podem criar sistemas mais eficientes e escaláveis, que respondem rapidamente às necessidades dos usuários.

Referências

GOF. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
Microsoft. ASP.NET Core Documentation. Disponível em: https://docs.microsoft.com/en-us/aspnet/core/.
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.
Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Dissertation, University of California, Irvine.

Nagib é Professor Universitário e Tech Manager. Possui uma trajetória de conquistas em certificações técnicas e ágeis, incluindo MCSD, MCSA e PSM1. PG em Gestão de TI pelo SENAC e MBA em Tecnologia de Software pela USP, Nagib cursou programas de extensão do MIT e Universidade de Chicago. Outras conquistas incluem a autoria de um artigo sobre chatbots, revisado por pares e apresentado na Universidade de Barcelona.