Exploring the Prototype Pattern in Web APIs using 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/50d98d
out 07 2024

Abstract:
The article explores the application of the Prototype pattern in Web APIs using C#. The Prototype allows the creation of new objects from existing instances, avoiding the complexity and cost of creating from scratch. The implementation includes the definition of an interface IProduct and classes that implement it, demonstrating the cloning of products and categories. The use of this pattern offers benefits such as increased performance, simplicity, and ease of maintenance.

Key words:
Prototype Pattern, Web APIs, C#, programming, design patterns, object cloning, software development, code reuse, software architecture, instance, performance, flexibility, abstraction, implementation, design patterns, serialization, deserialization, agile development, testing, code maintenance.

Exploring the Prototype Pattern in Web APIs using C#

The Prototype pattern is one of the creational design patterns that allows the creation of new objects from an existing object, without the need to depend on concrete classes. This pattern is especially useful in scenarios where creating new objects is complex or costly. In web API development, applying the Prototype pattern can help optimize the creation of object instances that share common characteristics, facilitating code maintenance and scalability.

Why use the Prototype Pattern?

Using the Prototype pattern offers several advantages, such as:

Performance: Avoids the cost of creating new objects from scratch.
Flexibility: Allows the creation of new objects through prototype cloning.
Simplicity: Can simplify the code by allowing objects to be created uniformly.

These advantages make the Prototype pattern an attractive choice, especially in environments where performance and resource efficiency are crucial, such as in applications dealing with large volumes of data or requests.

Implementing the Prototype in C#

To illustrate the implementation of the Prototype pattern in C#, let’s create a simple API that manages a collection of products. We will use ASP.NET Core to set up the API and implement the Prototype pattern.

Defining the IProduct Interface

First, we will define an interface IProduct that will have a Clone method:

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

```
}
```

Implementing the Product Class

Now, let’s implement a Product class that represents a product and implements the IProduct interface:

```
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
        };
    }
}
```

Using ICloneable

It is possible to use the ICloneable interface, which is already designed for this kind of situation. However, it is important to be aware that the interface does not define how cloning will be done. In the example below, MemberwiseClone is being used.

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

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

ProductController Controller

Now that we have our prototype implemented, we can create an API that utilizes this pattern. Let’s create a ProductController controller:

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

    public ProductController()
    {
        _products = new List
        {
```

```
            new Product { Name = "Product A", Price = 10.0m },
            new Product { Name = "Product 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);
    }
}
```

In this controller, we have a Clone method that accepts a product ID, clones the corresponding product, and adds the new product to the list.

Handling Objects with the Prototype Pattern

One of the main applications of the Prototype pattern is the manipulation of complex objects. Consider a scenario where we have products with various properties and relationships. We can expand our implementation to include a Category object, which has a list of products.

```
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()
        };
    }
}
```

In this example, when we clone a category, we also clone all the associated products. This ensures that we are not

just referencing the same objects, but creating independent copies.

Complete Example of API with Prototype

Now, let's look at a complete example of an API that uses the Prototype pattern to manage products and categories.

```
[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);
    }
}
```

In this controller, we have the ability to create and clone categories, using the Prototype pattern to ensure that the cloned instances are independent.

Benefits of the Prototype Pattern in Web APIs

Applying the Prototype pattern in Web APIs brings a number of benefits, including:

Reduction of Complexity: By allowing objects to be cloned instead of created from scratch, the code becomes cleaner and easier to understand.

Increased Performance: Cloning objects can be more efficient than creating new objects, especially when the object construction is costly.

Ease of Maintenance: Centralized cloning logic makes it easier to maintain and update the object creation logic.

Extending the Prototype Pattern

Although the above example already covers the fundamental concepts of the Prototype pattern, it is interesting to explore how we can extend this implementation to handle more complex scenarios. For example, we can introduce a configuration system where different types of products can have unique characteristics but still benefit from cloning.

Adding Dynamic Properties

Suppose we want to add the capability to set dynamic properties for our products. We can do this by using a dictionary to store these properties.

```
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)
        };
    }
}
```

In this way, when cloning a product, we also clone its dynamic properties, ensuring that each instance is unique.

Integration with Other APIs

Another important consideration when working with the Prototype pattern in Web APIs is how to integrate these prototypes with other APIs. For instance, we could have a service that provides product data from an external system, and when cloning a product, we might want to automatically fill some of its properties with data from that service.

Testing the Implementation

It is important to test the implementation of the Prototype pattern to ensure that cloning works as expected. Using unit tests can help validate that cloned objects are truly independent and that their properties are being copied correctly.

```
using Xunit;

public class ProductTests
{
    [Fact]
    public void Clone_ShouldCreateIndependentInstance()
    {
        var originalProduct = new Product
        {
            Name = "Original Product",
            Price = 30.0m,
            Attributes = new Dictionary { { "Color", "Blue" } }
        };
```

```
        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);
    }
}
```

This test ensures that cloning is functioning correctly and that cloned instances are truly independent of the originals.

Performance Considerations

While the Prototype pattern can improve performance in object creation, it is important to consider that cloning complex objects can still be a costly process, depending on the depth and complexity of the objects being cloned. Therefore, it is essential to monitor performance and optimize the implementation as necessary.

Conclusion

The Prototype pattern provides an effective way to manage object creation in software applications, especially in web API environments. It not only simplifies object creation but also enhances performance and facilitates code maintenance. By applying this pattern, developers can create more efficient and scalable systems that respond quickly to user needs.

References

GOF. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
Microsoft. ASP.NET Core Documentation. Available at: 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 is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs at MIT and the University of Chicago. Other achievements include the authorship of a peer-reviewed article on chatbots, presented at the University of Barcelona.