String Interpolation in C#: Essential Tips to Avoid Issues

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/21a1d4
jul 29 2024

Abstract:
String interpolation in C# allows for inserting variables directly into strings in a readable and efficient manner. However, improper use can compromise important aspects, such as application security.

Key words:
string interpolation, C#, interpolated strings, interpolation tips, common issues, security, interpolation errors, string manipulation.

Understanding String Interpolation

String interpolation in C# is a technique used to insert variable values into strings in a clear and efficient manner. This approach has become popular due to its readability and simplicity compared to older methods, such as concatenation. With the introduction of C# 6.0, string interpolation was enhanced and became a recommended practice for text formatting.

How String Interpolation Works

String interpolation in C# uses the prefix $ before a string in quotes. When you insert a variable inside curly braces {}, C# evaluates that expression and replaces it with the corresponding value. This not only improves readability but can also prevent common errors that occur with concatenation.

```
var name = "Maria";
var age = 30;
var message = $"Hello, my name is {name} and I am {age} years old.";
Console.WriteLine(message); // Output: Hello, my name is Maria and I am 30 years old.
```

Avoid Common Issues with Interpolation

Although string interpolation is intuitive, there are pitfalls that can lead to problems in the code. Here are some essential tips to avoid issues:

Escaping Curly Braces: If you need to use literal curly braces within an interpolated string, you must double them. For example: var text = $"{{Curly}}";.

Check Data Types: Ensure that the interpolated data types are compatible. Non-convertible variables can cause runtime exceptions.

Performance: If interpolation is done within loops, consider using StringBuilder for better performance, especially with large amounts of data.

Security Risks in Interpolation

Improper use of string interpolation can introduce significant security risks to the application. It is important to be aware of the following vulnerabilities:

Code Injection: If unvalidated variables are directly inserted into command strings, such as in SQL queries or shell commands, code injection can occur. Always use command parameters or ORM to avoid this.

Exposure of Sensitive Data: Interpolated variables that contain sensitive information may be accidentally exposed in logs or error messages. Ensure that sensitive data is masked or protected before interpolation.

Format Manipulation: Malicious users may try to manipulate string formats to cause unexpected behaviors. Validate and sanitize user inputs before using them in interpolation.

Be Cautious of Code Injection through String Interpolation

One of the biggest vulnerabilities when using string interpolation improperly is code injection, especially in SQL queries. Here is an example illustrating how this can happen:

```
// This method is assumed to receive user input.
public void ExecuteQuery(string username)
{
    // String interpolation used directly in the SQL query.
    string query = $"SELECT * FROM Users WHERE Name = '{username}'";

    // Creates an SQL command with the interpolated string.
    using (SqlConnection connection = new SqlConnection("YourConnectionString"))
    {
        SqlCommand command = new SqlCommand(query, connection);

        try
        {
            connection.Open();
            SqlDataReader reader = command.ExecuteReader();

            while (reader.Read())
            {
                Console.WriteLine($"ID: {reader["ID"]}, Name: {reader["Name"]}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}

// Calling the method with malicious input.
ExecuteQuery("Maria' OR '1'='1");
```

In the example above, the SQL query is constructed directly with string interpolation, which can lead to code injection. A malicious user might provide input like "Maria' OR '1'='1", resulting in an SQL query that returns all records from the Users table:

```
SELECT * FROM Users WHERE Name = 'Maria' OR '1'='1'
```

This compromises the database security as it allows the user to bypass the query conditions and access unauthorized data.

Preventing Code Injection

One way to prevent code injection is to use query parameters instead of direct interpolation. Here is a safe version of the previous example:

```
public void ExecuteSafeQuery(string username)
{
    // SQL query with parameters.
    string query = "SELECT * FROM Users WHERE Name = @Name";

    using (SqlConnection connection = new SqlConnection("YourConnectionString"))
    {
        SqlCommand command = new SqlCommand(query, connection);
        command.Parameters.AddWithValue("@Name", username);

        try
        {
            connection.Open();
            SqlDataReader reader = command.ExecuteReader();

            while (reader.Read())
            {
                Console.WriteLine($"ID: {reader["ID"]}, Name: {reader["Name"]}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}

// Calling the method with safe input.
ExecuteSafeQuery("Maria");
```

By using query parameters, you ensure that the values provided by users are handled safely, preventing code injection and protecting your application from malicious attacks.

Another approach would be adopting the Entity Framework, as the latest version includes protection against this type of attack.

Advanced Examples of Interpolation

String interpolation also allows for complex expressions and formatting. Here are some practical examples:

```
var price = 19.99;
var quantity = 3;
var total = price * quantity;
var result = $"The unit price is {price:C} and the total is {total:C}.";
Console.WriteLine(result); // Output: The unit price is R$ 19.99 and the total is R$ 59.97.
```

Conditional Formatting

You can include conditional logic directly in the interpolation. This is useful for situations where you want to change

the output based on a condition:

```
var userActive = true;
var status = $"The user is {(userActive ? "active" : "inactive")}.";
Console.WriteLine(status); // Output: The user is active.
```

Conclusion

String interpolation in C# is a powerful technique that makes it easier to insert variables into strings, making the code more readable and less prone to errors. However, it is crucial to be aware of the security risks associated with this practice and take appropriate precautions to avoid vulnerabilities, such as code injection. Using query parameters and secure frameworks can help mitigate these risks, ensuring a more robust and secure application.

References and Resources

For a deeper understanding and updates on string interpolation in C#, refer to the following official sources:

MICROSOFT. Official C# Documentation - String Interpolation. Available at: https://docs.microsoft.com/dotnet/csharp/language-reference/tokens/interpolated. Accessed on: July 29, 2024.
MICROSOFT. Programming Guide - Interpolated Strings. Available at: https://docs.microsoft.com/dotnet/csharp/programming-guide/strings/interpolated-strings. Accessed on: July 29, 2024.
MICROSOFT. C# Keywords Reference. Available at: https://docs.microsoft.com/dotnet/csharp/language-reference/keywords. Accessed on: July 29, 2024.

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. With a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP, Nagib has also 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.