27 Tips for

Faster .NET & MAUI Apps

PreEmptive

INDEX

Overview..1

27 Tips for Faster .NET & MAUI Apps................2-14

How to Balance Speed With Security....................15

OVERVIEW

Speed and efficient performance are critical for applications to succeed. Even a one-second delay in response time can lead to an [11% decrease in conversions] and a 16% decrease in customer satisfaction. In a world where alternatives are but a click away, users demand and expect quick, responsive applications. This reality pushes developers to prioritize performance in app development, making technology choices that align with this priority.

.NET, with its efficient, just-in-time compilation, is the technology of choice for many developers because it offers a foundation for building fast, responsive applications. .NET MAUI (Multi-platform App UI), extending the .NET ecosystem, further empowers developers to create applications that run across multiple platforms with native performance.

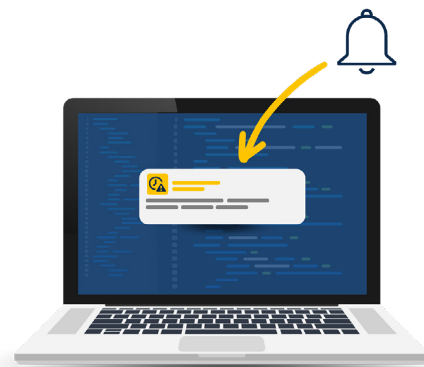
While .NET and .NET MAUI inherently prioritize performance, there are additional strategies and optimizations you can employ to really hone your app speed. This guide presents 27 essential tips to help you maximize the performance of your .NET and .NET MAUI applications to operate with the speed and responsiveness that your users expect.

27 Tips for Faster .NET & MAUI Apps

1 *Use Asynchronous Programming*

Incorporating asynchronous programming, facilitated by the async and await keywords, enables an app to remain productive during input/output-bound processes, enhancing system resource efficiency and preventing thread blocking. This approach is crucial for .NET applications, especially in web development, to handle concurrent requests more effectively and improve responsiveness. For MAUI apps, asynchronous programming is equally vital for smooth and responsive user experiences across different platforms and devices. By avoiding performance bottlenecks, you can create scalable .NET and MAUI applications that maintain high performance even under heavy loads.

2 *Manage Memory Efficiently*



Efficient memory management is critical in .NET and MAUI applications. Focus on creating only necessary resources and timely releasing them to optimize performance. Avoid unnecessary allocations, utilize value types (structs) to reduce heap allocations, and employ “using” statements for effective resource disposal.

In MAUI applications, where resource constraints are more pronounced, especially on mobile devices, managing memory efficiently is paramount to ensure responsiveness and prevent sluggishness. Strategies like avoiding large object heap allocations are crucial for maintaining a low memory footprint across platforms. Regularly profile memory usage to identify and fix memory leaks and excessive allocations to reduce garbage collection overhead and enhance application performance.

3 Profile Applications Regularly

Want insights into your apps' runtime behavior? Use profiling tools. They can help you pinpoint performance bottlenecks, understand memory allocation patterns—including leaks—optimize CPU-intensive tasks, resolve threading issues like deadlocks and thread contention, and identify inefficient database queries.

This practice is crucial not only for .NET applications but also for MAUI apps, where optimizing for diverse platforms and devices adds complexity. For MAUI, profiling becomes even more critical as it helps ensure that applications perform efficiently across all target platforms, maintaining a seamless user experience whether on mobile or desktop.

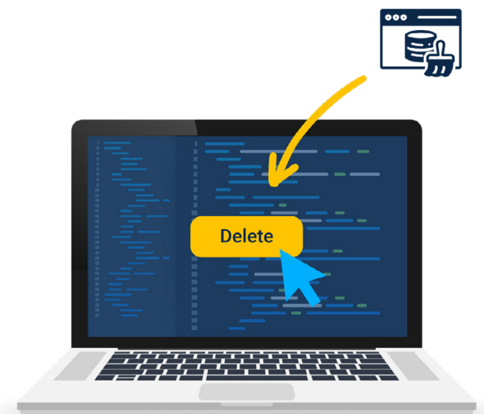
4 Speed Up Data Access

If database queries slow down your application, there are ways to streamline data access and enhance responsiveness. Object-relational mapping (ORM) tools, selective queries, and caching of frequently unchanged data minimize database round trips. Strategies such as eager, explicit, or select loading help fetch related data more efficiently, addressing the N+1 query issue. Additionally, reusing database connections and batching operations can significantly reduce overhead. Proper indexing and thoughtful database schema design are also key to faster query execution, while NoSQL databases may offer performance advantages in certain scenarios.

For .NET MAUI applications, these data access optimizations are especially important due to the varied network conditions and hardware limitations of mobile devices. Efficient data access ensures that MAUI apps remain responsive and performant, providing a seamless user experience across all platforms. Implementing these strategies effectively reduces latency, improves scalability, and ensures that your .NET and MAUI apps perform as expected.

5 Leverage Caching

Effective caching is pivotal in reducing redundant processing and alleviating database load, offering significant performance improvements. Caching data within an app's memory or using distributed caching solutions for large-scale applications, can make for quick data retrieval. While in-memory caching suits frequently accessed, non-shared data, distributed caching excels in environments with shared data across multiple instances.



6 *Optimize LINQ Queries*

LINQ (Language Integrated Query) plays a vital role in data manipulation within .NET, executed lazily to ensure that the operation is performed at the most efficient moment. You can enhance performance by using "select" to fetch only necessary data fields, reducing overhead. Placing "Where" clauses early in the query sequence minimizes the data processed.

While LINQ simplifies data querying with its integrated syntax, there are scenarios, especially involving complex joins and calculations, where raw SQL might outperform LINQ queries in efficiency. Analyzing the SQL generated by LINQ queries is a critical step in identifying and implementing optimizations.

For applications developed with .NET MAUI, efficient data querying directly influences the application's performance and user experience across various devices. In some cases, leveraging raw SQL for data access in MAUI apps can provide significant performance benefits, particularly for complex data operations. Keep an eye on the efficiency of LINQ queries, considering both the generated SQL and the performance characteristics of mobile and desktop.

7 *Use Efficient Serialization*

Efficient serialization can improve performance when transferring data over networks or saving it to storage. The choice of serializer can significantly impact both speed and memory usage. For JSON serialization, "System.Text.Json" in .NET Core offers superior performance and efficiency over "Newtonsoft.Json." For binary serialization, consider using "Protobuf" or "MessagePack," which are both compact and quick, making them ideal for high-performance scenarios.

In .NET MAUI applications, where data transfer efficiency may have to contend with potentially limited bandwidth and diverse device capabilities, choosing the right serialization method becomes even more important. Focus on lean data contracts to avoid serializing unnecessary properties, and attributes should be used wisely to control serialization processes.

A tip that may offer substantial performance gains is to reuse serializer instances because the overhead of instantiating a new serializer for each operation can accumulate (especially for mobile).

8 *Implement HTTP/2 Where Applicable*

Adopting HTTP/2 offers significant advantages over HTTP/1.1, including header compression, multiple requests over a single TCP connection, and server push capabilities. These features enhance web application performance by reducing latency and improving resource loading times. Specifically, HTTP/2's server push can proactively deliver resources to the client, potentially speeding up the user's experience. But use this feature wisely to avoid wasting bandwidth through unnecessary resource pushes.

For .NET MAUI applications, which often interact with services across various network conditions and devices, implementing HTTP/2 can lead to substantial performance improvements. Just continue to maintain HTTP/1.1 support as a fallback to accommodate clients or environments where HTTP/2 is not available or supported.

9 *Minimize the Use of Reflection*

Reflection is a .NET feature that enables runtime type inspection and dynamic code execution. While it provides a flexible approach to working with types, its performance implications cannot be overlooked as long as you save it for scenarios where dynamic type access is essential, such as dynamically loading assemblies or dealing with members unknown at compile time. For more routine tasks, such as accessing property getters and setters, compiled expressions via Expression<T> offer a more performant alternative. These expressions incur a compilation cost upfront but execute significantly faster than reflective operations, making them suitable for high-frequency use.

Minimizing reflection is smart with MAUI apps because it can promote a smooth and responsive user experience across various devices and operating systems. By opting for compiled expressions over reflection for common operations, you can reduce the runtime overhead associated with dynamic type manipulation and have .NET and MAUI applications that are both efficient and responsive, regardless of the platform.

10 *Choose the Right Collection*

When selecting the appropriate collection type, consider the specific requirements of their application, such as the need for ordering, fast lookups, or efficient modifications, as well as whether the size of the collection is expected to change. This analysis aids in narrowing down potential candidates. Subsequent benchmarking and profiling can then identify the most suitable option for their specific use case.

11 *Avoid Blocking Calls*

Blocking calls can slow an application's responsiveness by causing it to wait and tie up resources, and asynchronous programming patterns generally mitigate this issue. But be advised that with large or complex codebases with third-party or open-source components may encounter challenges and deadlocks with mixed synchronous and asynchronous code.

To address blocking calls from third-party APIs or libraries, encapsulate these calls within asynchronous methods using Task.Run. Additionally, using ConfigureAwait(False) in library code can prevent blocking by avoiding the need to marshal the execution back to the original context, especially in scenarios where returning to the original context is not required.

For .NET MAUI applications, minimizing blocking calls helps to accommodate environments ranging from resource-constrained mobile devices to more powerful desktops. The impact of blocking calls is more pronounced on mobile devices, where resources are limited, and user experience expectations are high. Using asynchronous methods effectively can significantly improve performance, making the app more agile and responsive to user inputs.

12 *Optimize String Operations*

Inefficient string handling increases memory use and slows processing times. If your app performs frequent string concatenations, utilizing StringBuilder is a more efficient alternative to traditional concatenation methods, as it minimizes the number of new string objects created. Furthermore, the use of String.Intern can prevent the creation of multiple identical string literals, conserving memory. In .NET Core and later versions, ReadOnlySpan<char> is recommended for string-slicing operations because it does not generate new string instances, offering a more memory-efficient approach.

In .NET MAUI applications, efficient string management improves performance, especially on mobile devices where memory and processing power are more limited compared to desktop environments. For scenarios requiring high performance and frequent string manipulation, adopting ArrayPool<char> or custom pooling mechanisms can further reduce memory overhead and enhance application responsiveness.

13 Use Value Types Judiciously

Value types are allocated on the stack and automatically deallocated when out of scope, reducing garbage collection pressure. This makes structs particularly suitable for small, immutable data structures where their stack allocation and subsequent lack of garbage collection can lead to performance gains. However, due to structs being passed by value, larger structs can be inefficient as they require copying upon each method call or assignment. When in doubt, use structs for data structures under 16 bytes to maintain efficiency.

14 Implement Proper Exception Handling

Exceptions should be reserved for exceptional conditions, not used as a control flow mechanism. Pre-validating data and conditions to prevent exceptions can significantly reduce unnecessary overhead. Catch only the exceptions that the application can handle, focusing on specific exception typesrather than general or system exceptions, to avoid masking errors or wasted resources.

Asynchronous operations, common in MAUI apps for non-blocking UI interactions, introduce additional complexity to exception handling. Utilizing try-catch blocks or the exception-handling capabilities of tasks can help manage exceptions in asynchronous code effectively. This approach ensures apps remain responsive, stable, and user-friendly by minimizing disruptions caused by unhandled exceptions or inappropriate exception-handling strategies.

15 Minimize Cross-Process Communication

The overhead associated with network latency, serialization, and synchronization can be a big drain on the responsiveness and speed of an application. You can reduce this by limiting the volume of data transferred between processes to only the essentials. When data transfer is necessary, leveraging efficient serialization formats such as Protocol Buffers or MessagePack can provide advantages over formats like XML or JSON due to their compactness and speed.

For .NET MAUI applications, which may interact with various services and processes, especially in a distributed environment, look to communication protocols such as gRPC for its efficient binary...

... serialization and HTTP/2 support. Scenarios requiring local interprocess communication options like shared memory or named pipes present a more efficient alternative to network calls, reducing latency and overhead.

16 *Use Dependency Injection Effectively*

Dependency injection (DI) enhances modularity and testability by promoting loose coupling, but be advised that improper use can detrimentally affect performance. Optimizing DI involves selecting appropriate service lifetimes—transient for lightweight, short-lived services; scoped for request-duration services in web apps; and singleton for stateless, thread-safe services requiring long-term persistence. Given their application-lifespan scope, avoid overusing singletons to prevent memory leaks. Additionally, minimizing constructor injection can alleviate performance and maintainability concerns, as excessive dependencies may indicate an overly burdensome class responsibility.

17 *Tailor the Garbage Collection Settings*

The .NET framework provides two primary GC modes: workstation and server. Workstation GC, the default mode, suits most applications well, especially those running on single-core processors or requiring fast user interface responsiveness. On the other hand, server GC is optimized for applications running on multicore processors, offering more efficient heap management and throughput, making it ideal for backend services.

MAUI apps run across a variety of devices and computational scenarios so choosing the appropriate GC mode and adjusting the GCSettings.LatencyMode is crucial. The LowLatency mode, for instance, can minimize pause times, enhancing the responsiveness of UI-driven applications at the cost of higher memory usage.

Also consider the impact of memory allocations on GC performance; reducing allocations in performance-critical paths to decrease the frequency of GC interruptions. Use GC.RegisterForFullGCNotification so applications can anticipate and efficiently manage memory usage around GC events, particularly in memory-intensive scenarios.

18 *Disable Runtime Compilation in ASP.NET Core*

Runtime compilation in ASP.NET Core can facilitate rapid iteration and debugging by allowing changes to be viewed without restarting the application. However, while this feature promotes agility in development, it comes with a performance cost that is not suitable for production environments. Runtime compilation in production can lead to unnecessary overhead and slower response times, so disable runtime compilation in production settings for better performance and speed.

For .NET MAUI applications, although runtime compilation is not directly related, the underlying principle is still relevant. Differentiate between features and settings that are beneficial for development versus those suitable for production. This could involve decisions around linking, AOT compilation, and resource management to ensure that applications are as efficient as possible when deployed.

19 *Minify and Bundle Resources*

Minifying and bundling are key in reducing the size and number of client-side files (like JavaScript and CSS) that speed up page load times by minimizing HTTP requests and enhancing the user's experience. ASP.NET Core supports these processes natively, with the "Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" package facilitating setup for dynamic compilation scenarios. For ASP.NET MVC 5 and versions prior, the "System.Web.Optimization" namespace provides the necessary tools for effective resource optimization.

Configuring these involves setting up bundle configurations within the "Startup.cs" file or a dedicated configuration file. For ASP.NET MVC 5 and earlier versions, this setup is typically done in the "BundleConfig.cs" file located in the App_Start directory. By bundling, multiple CSS or JavaScript files are combined into a single file, reducing the overall number of HTTP requests during page loading. Minification further reduces file sizes by eliminating unnecessary characters, such as whitespace and comments, from CSS, JavaScript, and HTML files.

Although .NET MAUI apps primarily target mobile and desktop applications where the concept of HTTP requests for resources is less of an issue, you can still optimize for performance by compressing images, using efficient data serialization, and ensuring that embedded resources are optimally managed.

20 *Implement Proper Logging*

Effective logging helps to troubleshoot, understand app behavior, and ensure security. But if it's not done properly, logging itself can be a performance chokepoint. Utilize logging frameworks that support asynchronous operations, like NLog, Serilog, or .NET Core's ILogger, to reduce the impact on application performance by not blocking the main execution flow.

Apply different logging levels (Debug, Info, Warning, Error, Critical) judiciously. This filters logs that are most relevant to the current application state or environment. For instance, limiting logs to Warning or above in production can reduce the volume of logged information, focusing on what's crucial.

Use structured logging over plain text. Structured logs organize data into a consistent format or schema is more easily queryable. Keep log configurations outside the application codebase so operational teams can adjust log levels or targets without the need for code changes, recompilation, or redeployment.

21 Optimize Network Calls

If your app frequently interacts with web services, APIs, or other remote resources, make sure to optimize network efficiency. One key strategy is the effective use of HttpClient, which is optimized for reusability across an application's lifetime rather than being recreated for each request. This minimizes the overhead and latency associated with establishing network connections, thereby improving application performance.

- **Reuse HttpClient:** Instantiate HttpClient once and reuse it throughout the application to leverage connection pooling and reduce the cost of network operations.
- **Caching and Asynchronicity:** Implement caching strategies for network responses to avoid redundant requests and utilize asynchronous network calls.
- **Timeouts and Retry Logic:** Define timeout policies and retry mechanisms for handling transient network errors even in the face of network instability.
- **Content Delivery Networks:** For static resources like images, CSS, and JavaScript, CDNs can drastically reduce load times.
- **Connection Pooling:** Reduce the time and resources required for network communications.

22 Use Task.Run Sparingly

Task.Run can offload work to background threads and facilitate CPU-bound tasks without blocking the UI. However, using it in scenarios not suited for background threading can lead to decreased performance and inefficiencies. So, for I/O-bound operations, which naturally support asynchronous execution, Task.Run is unnecessary and can introduce overhead without any benefit. Asynchronous I/O operations should leverage the inherent async patterns available in .NET, avoiding the unnecessary allocation of threads.

In server-side applications, excessive use of Task.Run can lead to thread pool exhaustion. In high-load environments, this can significantly degrade performance, as the thread pool struggles to manage an increasing number of concurrent tasks, leading to delays in task execution and an overall slowdown of the application. So you should reserve Task.Run for truly CPU-bound operations and ensure that its use is justified, particularly considering the context and the expected load on the application.

23 *Enable Tiered Compilation*

Tiered compilation enhances performance by enabling the runtime to compile and optimize code adaptively. This can strike a balance between rapid startup times and high-throughput execution by initially compiling methods using a faster, albeit less optimized, compiler. Over time, frequently invoked methods ("hot" methods) are identified and recompiled with more aggressive optimizations to boost performance.

Enabled by default in .NET Core 3.x and later versions, developers working with earlier versions of .NET Core can activate tiered compilation by including a specific property in their project file:

```
1  <PropertyGroup>
2
3  <TieredCompilation>true</TieredCompilation>
4
5  </PropertyGroup>
6
7
```

While tiered compilation offers many benefits, avoid it in situations where the recompilation of hot methods consumes additional CPU resources, as that could impact performance temporarily. That means applications with very short run times or those that are highly sensitive to CPU usage spikes during their startup phase might not benefit as much from this feature.

Experiment with different runtime configuration flags. These flags allow for customization of the recompilation process, such as adjusting the thresholds for identifying hot methods or controlling the aggressiveness of the optimizations applied during recompilation. Fine-tuning these settings can tailor the tiered compilation behavior to align with their application's specific performance characteristics and requirements.

24 *Optimize Docker Images*

For applications deployed with Docker, including those built with .NET and potentially packaged for deployment in a .NET MAUI context, optimizing Docker images enhances the performance and efficiency of containerized applications. It not only improves startup times and reduces resource consumption but also quickens deployments and scalability, especially in cloud-based environments.

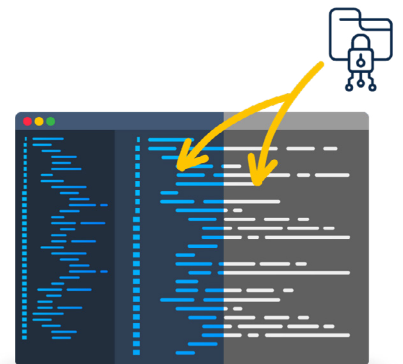
- **Choose the Smallest Base Image:** Start with the smallest base image that meets your application's needs. Alpine Linux is often recommended for its minimal footprint compared to standard Debian-based images.

- **Multi-Stage Builds**: This technique involves using a larger image with all the necessary build tools to compile and build your application and then transferring the final build to a smaller, runtime-specific image. This method excludes unnecessary build dependencies and intermediate files from the final image, considerably reducing its size.

- **Minimize Image Layers:** Aim to reduce the number of layers in your Docker image by consolidating RUN commands where practical. Since each command in a Dockerfile introduces a new layer, combining commands can help minimize the total number of layers. For example, chaining commands using && and cleaning up in the same RUN statement can prevent the addition of temporary files to the image.

- **Use .dockerignore:** Implement a .dockerignore file to exclude files and directories that are not needed in your Docker context, such as local build artifacts, logs, or temporary files. This step ensures they are not unnecessarily added to the build context, streamlining the build process and resulting in lighter images.

25 *Optimize Entity Framework Performance*



Entity Framework (EF) is a bridge between object-oriented programming and relational databases that enables you to work with data using domain-specific objects. This is especially true for .NET MAUI applications that leverage Entity Framework for data access—whether accessing local SQLite databases on mobile devices or connecting to remote databases over network calls.

However, its convenience can come with a cost to performance if not used judiciously. Here are ways to ensure you're leveraging EF efficiently:

- **Eager Loading with .Include()**: While .Include() is useful for loading related data in a single query, preventing the infamous N+1 query problem, use it selectively. Over-eager loading can cause unnecessarily large datasets to be retrieved and bog down performance. Evaluate the data needs and load only the necessary related entities.

- **Use .AsNoTracking() for Read-only Scenarios**: When retrieving data that won't be updated in the current context, .AsNoTracking() can significantly improve performance. This method bypasses the overhead of setting up change tracking for the entities, making the read operations faster and more resource-efficient.

- **Projected Queries**: Instead of retrieving entire entity models when only a few fields are needed, use projected queries to select only the specific data required and reduce the amount of data transferred, processed, and materialized.

- **Indexing:** Are your database tables properly indexed based on your query patterns? Proper indexing can dramatically improve query performance by reducing the amount of data the database engine needs to scan.

- **Batch Operations:** For bulk insert, update, or delete operations, extensions like Entity Framework Extensions or libraries support efficient batch operations to minimize database round-trips.

- **Avoid Multiple Saves:** Instead of calling .SaveChanges() after each individual data manipulation operation, aggregate your changes and save them in a single call where possible to reduce the number of transactions.

26 *Use Application Insights and Performance Monitoring Tools*

Whether you're developing web services, backend systems, or cross-platform applications with .NET MAUI, integrating APM tools into your workflow can significantly improve the quality and reliability of your software by identifying and diagnosing performance bottlenecks so that applications remain responsive and reliable under various conditions.

- **Telemetry Collection:** APM tools collect a range of telemetry data, like request rates, response times, failure rates, and dependencies. This is invaluable for understanding how an app behaves and can pinpoint areas needing optimization.

- **Dependency Tracking**: One of the key features of APM tools is their ability to track dependencies, such as database calls or external service requests. This can locate slow database queries or external calls that may be affecting application performance.

- **Integration with DevOps**: Integrating APM tools into your DevOps pipeline enables continuous monitoring of application performance across different stages of development and deployment so that performance considerations are addressed.

- **Alerting and Diagnostics**: Most APM tools provide alerting mechanisms that notify teams of potential issues before they impact users. Additionally, detailed diagnostics locate the root cause of performance issues.

27

Stay Updated with .NET Releases

Staying updated with the latest .NET releases is essential not only for addressing security vulnerabilities but also for harnessing performance improvements and new features that enhance application efficiency and capabilities. Updates often include critical patches that safeguard against security breaches, alongside optimizations that boost runtime efficiency and reduce resource consumption. Furthermore, each new release introduces innovative features and APIs, simplifying development tasks and enabling the creation of more robust applications.

How to Balance *Speed* With *Security*

Prioritizing speed without compromising security is a fine art. Embracing a “shift-left” approach, where security measures are integrated early in development, can significantly mitigate risks without slowing down progress. This not only streamlines the identification and resolution of security vulnerabilities but also ensures that subsequent development stages can proceed with confidence, minimizing the need for disruptive revisions or extensive rework.

Security is not just a checkpoint but an integral part of the development process, critical to safeguarding trade secrets, user data, and the application itself from potential threats. Utilizing automated tools like PreEmptive’s Dotfuscator offers a smart defense mechanism that uses advanced obfuscation techniques and runtime checks to protect your .NET and MAUI applications against reverse engineering and decompilation.

Contact us for a free trial and let’s make sure that your applications are not only fast and efficient but secure from the ground up.

.

***Smart App Protection** for an **Unsafe** World!*

GET IN TOUCH:

Headquarters in USA
10801 N Mopac Expressway
Building 1, Suite 100
Austin, TX, 78759
phone: **+1 (512) 226-8080**
email: **solutions@preemptive.com**

European Sales
140 bis rue de Rennes
75006 Paris
France
Tel: **+33 01.83.64.34.74**
email: **EuroSolutions@preemptive.com**

JAPAN
AG-Tech Corp
Tel: **+81-3-3293-5300**
Email: **info@agtech.co.jp**