

3.1 System Architecture & Tech Stack

- **Core Technology:** C#.NET 8 [1]
- **Clients:** Discord.Net (Bot), Blazor/Web (Web UI) [1]
- **Engine:** ASP.NET Core Web API [1]
- **Database:** PostgreSQL (Production), EF Core 8 (ORM) [1]

Architecture: The "API-First" Model

This is the core strength of the project.[1] All game logic is centralized in the Nightstorm.API.

- **The "Dumb Client" Principle:** The Nightstorm.Bot and Nightstorm.Web clients contain zero game logic. They are "dumb terminals" [1] that only:
 1. Send authenticated API requests (e.g., POST /api/combat/action).
 2. Receive and display state changes (via SignalR for Web, via a new Notification Service for Discord).

Visual Studio Solution Structure

The project will be organized into a 5-project solution [1]:

1. **Nightstorm.Core:** Shared Class Library (Models, Enums, Interfaces).
2. **Nightstorm.Data:** EF Core RpgContext and Migrations.
3. **Nightstorm.API:** The ASP.NET Core Game Engine (Controllers, Services, SignalR Hubs).
4. **Nightstorm.Bot:** .NET Worker Service (Discord.Net client, Interaction Handlers, and the new NotificationService detailed in 3.2).
5. **Nightstorm.Web:** Blazor WebAssembly (UI Components).

3.2 CRITICAL FLAW #1: The Discord Client & Rate-Limit Fallacy

The v1.0 Flaw:

The "Combat Flow" [1] states: "To Discord: Edits the battle message. Unlocks buttons for the active player." This implies a direct POST /messages/{message.id} call from the game engine every time a combatant's turn begins.

The Technical Reality:

Discord's API rate limits [7] are strict and dynamic. A single busy 10v10 GvG fight could generate 20+ message edits in under a minute on the same channel. This will get the bot "choked" [8] and globally rate-limited, bringing all game functions to a halt.

The Architectural Solution: Decoupling via Queued Background Service

The API Engine must not talk to Discord directly. It should add an event to an in-memory queue, and the Nightstorm.Bot worker [1] should process this queue at a safe, self-limited

pace. This is a common and robust pattern for background processing in .NET.[9, 10] The implementation will use the `IHostedService` pattern with a `Channel<T>`-based background queue, which is the modern, high-performance approach.[11]

Step 1: Define the Queue (in `Nightstorm.Bot`) [11]

This interface and class will be registered as a Singleton, accessible to both the notification-receiving controllers and the background worker.

C#

```
// IDiscordNotificationQueue.cs
public interface IDiscordNotificationQueue
{
    // The work item is a delegate that performs the Discord API call
    void Enqueue(Func< CancellationToken, Task> workItem);
    Task<Func< CancellationToken, Task>> DequeueAsync(CancellationToken token);
}

// DiscordNotificationQueue.cs (Register as Singleton)
public class DiscordNotificationQueue : IDiscordNotificationQueue
{
    private readonly Channel<Func< CancellationToken, Task>> _queue;

    public DiscordNotificationQueue()
    {
        _queue = Channel.CreateUnbounded<Func< CancellationToken, Task>>();
    }

    public void Enqueue(Func< CancellationToken, Task> workItem)
    {
        // TryWrite is non-blocking and safe
        _queue.Writer.TryWrite(workItem);
    }

    public async Task<Func< CancellationToken, Task>> DequeueAsync(CancellationToken token)
    {
        // ReadAsync will wait until an item is available
        return await _queue.Reader.ReadAsync(token);
    }
}
```

Step 2: Define the Worker (in `Nightstorm.Bot`) [11]

This service will be registered using AddHostedService. It runs for the lifetime of the bot, processing the queue.

C#

```
// QueuedDiscordUpdateService.cs (Register as AddHostedService)
public class QueuedDiscordUpdateService : BackgroundService
{
    private readonly ILogger<QueuedDiscordUpdateService> _logger;
    private readonly IDiscordNotificationQueue _taskQueue;
    // Inject your DiscordSocketClient or a service that wraps it
    // private readonly DiscordSocketClient _discordClient;

    public QueuedDiscordUpdateService(ILogger<QueuedDiscordUpdateService> logger,
        IDiscordNotificationQueue queue)
        // DiscordSocketClient discordClient)
    {
        _logger = logger;
        _taskQueue = queue;
        // _discordClient = discordClient;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var workItem = await _taskQueue.DequeueAsync(stoppingToken);

            try
            {
                // Execute the delegate, which contains the logic
                // to edit the Discord message.
                await workItem(stoppingToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error executing Discord notification work item.");
            }
            finally
            {
                // CRITICAL: This self-imposes a rate limit.
            }
        }
    }
}
```

```

    // This delay smooths out API call spikes.
    // A more advanced solution would be a token bucket,
    // but a simple delay is the first, essential step.
    await Task.Delay(TimeSpan.FromSeconds(1.5), stoppingToken);
}
}
}
}

```

Step 3: The New Combat Flow

1. The Nightstorm.API's CombatService completes an action.
2. The API's "Notification Service" [1] is now a simple HttpClient. It sends a request to a *new, internal-only* API endpoint on the Nightstorm.Bot worker (e.g., POST /notify/battle-update).
3. The Bot's controller receives this, generates the Func<CancellationToken, Task> (which contains the "edit message" logic), and Enqueues it.[11]
4. The QueuedDiscordUpdateService dequeues and executes the task, respecting its own 1.5-second delay, thus smoothing out API-call spikes and saving the bot from rate-limiting.

3.3 CRITICAL FLAW #2: The Database Concurrency Catastrophe

The v1.0 Flaw:

The proposed solution to the "Double-Click Exploit" [1] is Optimistic Concurrency, using a ``row version.[1, 12, 13] The v1.0 doc states the API will catch the DbUpdateConcurrencyException and tell the user "Action already processed."

The Technical Reality (The Catastrophe):

This works for one user on two clients but fails catastrophically for a party.

- **Scenario:** A 5-player party attacks a boss.
 1. It is the party's "turn" (e.g., all 5 players can act).
 2. Player 1 (Web) clicks Attack. The API reads the BattleSession (Version v1).
 3. Player 2 (Discord) clicks Heal. The API reads the same BattleSession (Version v1).
 4. Player 1's request finishes. It saves. The BattleSession is now Version v2.
 5. Player 2's request tries to save. It fails, throwing a DbUpdateConcurrencyException [1] because the database Version (v2) no longer matches the original Version (v1).
 6. Player 2 is told "Action already processed," even though their action never happened.

This design *breaks party-based combat*. Optimistic concurrency [14] is for *low-conflict* scenarios (e.g., updating a user's profile). Combat is a *high-conflict* [14] scenario and requires

pessimistic locking.

The Architectural Solution: Hybrid Concurrency Model

- **Keep Optimistic Concurrency (`)** for *low-conflict* data: Player stats, Inventory, Guild management.
- **Use Pessimistic Concurrency (Database-Level Locking)** for *all high-conflict* BattleSession modifications.

Implementation (EF Core 8 + PostgreSQL):

EF Core does not natively support pessimistic locking.[15] We must use raw SQL [12, 16] to issue a SELECT... FOR UPDATE command, which PostgreSQL supports and will lock the row until a transaction is committed or rolled back.[16]

Step 1: The New CombatService Logic

The entire ProcessPlayerAction logic must be wrapped in a database transaction that explicitly acquires a pessimistic lock.

C#

```
// In Nightstorm.API's CombatService.cs
// Inject IDbContextFactory<RpgContext> to create short-lived contexts
private readonly IDbContextFactory<RpgContext> _contextFactory;
private readonly ILogger<CombatService> _logger;
// private readonly NotificationService _notificationService; // (From 3.2)

public async Task<bool> ProcessPlayerAction(Guid battleId, Guid playerId, string skillId)
{
    // Use a factory to create a new context for this specific operation
    await using var context = await _contextFactory.CreateDbContextAsync();

    // 1. Begin a transaction. This is essential for the lock.
    await using var transaction = await context.Database.BeginTransactionAsync();

    try
    {
        // 2. ACQUIRE PESSIMISTIC LOCK
        // This is the most important line of code in the combat engine.
        // It locks the BattleSession row. All other threads/requests
        // for this battleId will wait here until this lock is released.
        // [16]
        var battle = await context.BattleSessions
            .FromSql($@"
                SELECT * FROM battlesessions
                WHERE battleid = @battleId
                FOR UPDATE
            ") as BattleSession;
        if (battle == null)
        {
            _logger.LogWarning("No battle found for ID {battleId}.");
            return false;
        }
        battle.lastUsedSkillId = skillId;
        await context.SaveChangesAsync();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error processing player action for battle {battleId}." );
        return false;
    }
}
```

```

        SELECT * FROM ""BattleSessions"""
        WHERE ""Id"" = {battleId}
        FOR UPDATE")
.SingleOrDefaultAsync();

    if (battle == null) throw new Exception("Battle not found.");

    // 3. Engine Validation [1]
    // (Is it player's turn? Enough AP? Enough MP?)
    var validation = ValidateAction(battle, playerId, skillId);
    if (!validation.IsSuccess) throw new Exception(validation.Error);

    // 4. Action Executed [1]
    // (Calculate damage, update HP, consume AP/MP)
    ExecuteAction(battle, playerId, skillId);

    // 5. Save changes (still inside the transaction)
    context.Update(battle);
    await context.SaveChangesAsync();

    // 6. Commit the transaction (releases the lock)
    await transaction.CommitAsync();

    // 7. Enqueue notification (the NEW way, see 3.2)
    // await _notificationService.EnqueueBattleUpdate(battle.Id, battle.GetState());

    return true;
}
catch (Exception ex)
{
    // 8. Rollback (releases the lock) if anything fails
    await transaction.RollbackAsync();
    _logger.LogError(ex, "Failed to process combat action due to: {Message}", ex.Message);
    return false;
}
}

```

This hybrid model provides the performance of optimistic concurrency for 90% of the application, and the high-contention safety of pessimistic locking for the 10% (combat) that would otherwise break the entire game.

3.4 API & Real-Time Engine

Table TDD-1: Core API Endpoints (REST)

This table defines the primary contract between the Dumb Clients (Bot, Web) and the API Engine, noting the concurrency model for each.

Method	Endpoint	Auth	Concurrency Model	Description
POST	/api/player/start	Yes	Optimistic	Creates a new player character.
GET	/api/player/{id}	Yes	N/A (Read)	Gets a player's full state (stats, inventory).
POST	/api/combat/attack/{npcId}	Yes	Pessimistic	Initiates a new PvE BattleSession (locks session).
POST	/api/combat/action	Yes	Pessimistic (High Contention) Submits an action to the CombatService.	Submits an action to the CombatService.
GET	/api/market/listings	Yes	N/A (Read)	Gets all active Marketplace listings.
POST	/api/market/list	Yes	Optimistic	Lists a new item for sale (low contention).

POST	/api/guild/declare-war/{territoryId}	Yes	Pessimistic	Declares war (locks the Territory row).
------	--------------------------------------	-----	-------------	---

Real-Time Service (SignalR):

The Nightstorm.Web (Blazor) client will connect to a Nightstorm.API SignalR Hub (e.g., CombatHub).[17, 18, 19]

- When the CombatService (3.3) successfully commits a transaction, the _notificationService will also push an event to the SignalR Hub.
- Example: await _hubContext.Clients.Group(battle.Id).SendAsync("BattleStateUpdated", battle.NewState);
- This creates the desired *asymmetric, client-aware* notification:
 - **Web Client:** Receives an *instant* update via SignalR.[1]
 - **Discord Client:** Receives a *queued, rate-limited* update via the QueuedDiscordUpdateService (3.2).

3.5 Infrastructure & Deployment

The infrastructure plan from the v1.0 TDD is sound.[1]

- **Server Specs:** 4 VCPU, 16 GB RAM, 200 GB NVMe.[1] This is "overkill for 100 concurrent users" [1], which is ideal, providing significant headroom for growth.
- **Platform:** Ubuntu 24.04 LTS.[1]
- **Deployment:** Containerization via Docker and a single docker-compose.yml file [1] is the correct, modern, and maintainable approach for a 5-project stack.
- **PostgreSQL Tuning:** The plan to tune PostgreSQL is excellent [1]:
 - shared_buffers to ~4GB
 - effective_cache_size to ~10GB
 - The analysis that this will allow the active dataset to be "served directly from RAM" [1] is correct and demonstrates a high level of competence.
 - **Recommendation:** Post-launch, monitoring tools (e.g., pg_stat_statements and the pg_buffcache extension) must be used to validate the cache hit ratio and tune these values further.