**C# 技术管理文档**

**一、强制（Mandatory）**

这些条目必须严格遵守，以保证项目的稳定性、安全性和可维护性。

**1.项目结构规范**

项目必须遵循标准的目录结构，确保模块化，功能清晰分层，便于扩展维护。 如：

Controllers：API控制器

Models：数据模型

Views：视图文件

Services：业务逻辑服务层

Repositories：数据访问层

**2.依赖注入（DI）**

①必须使用依赖注入容器（如ASP.NET Core内置的DI）管理所有服务和依赖项，避免手动实例化对象。

②使用构造函数注入作为默认依赖注入方式，避免使用静态类或全局状态。

**3.跨平台支持**

①必须使用.NET Core或更高版本（如.NET 6/7）进行开发，确保应用可以在Windows、Linux和macOS上运行。

②避免使用特定平台依赖的API或库，确保代码的移植性。

**4.版本控制规范**

①必须使用Git进行版本控制。每次提交时必须提供清晰且有意义的提交信息，符合语义化版本控制规范。

②采用Git Flow或类似的分支管理策略，确保开发、测试、生产环境的分离。

**5.代码审查**

每次提交代码必须经过代码审查，审核的重点包括：代码风格一致性、函数的单一职责、性能瓶颈、潜在的bug或异常处理问题

**6.单元测试覆盖率**

①必须为每个公共方法编写单元测试，确保至少90%的代码路径被覆盖。

②测试用例必须独立，避免与外部服务或数据库产生依赖。

**7.命名规范**

所有命名必须符合C#语言的标准命名约定：

类名、方法名使用PascalCase（首字母大写）。

变量名、私有字段使用camelCase（首字母小写）。

常量名使用全大写字母并用下划线分隔。

**8.错误处理与日志**

必须使用try-catch-finally结构处理已知异常，并记录详细的错误日志。

采用统一的日志库（如Serilog、NLog）进行日志记录，日志级别应包括：Info、Warn、Error、Fatal。

**9.异步编程**

所有I/O密集型操作（如文件读取、数据库访问、网络请求等）必须使用async和await进行异步处理，避免阻塞线程。

应用程序的UI线程必须与后台线程解耦，避免因同步阻塞导致应用无响应。

**10.输入验证**

必须验证所有外部输入，包括用户输入、URL参数、请求体等，以防止SQL注入、XSS等安全漏洞。

**11.身份验证与授权**

必须使用JWT（Json Web Token）进行身份验证，确保API请求通过身份验证后可访问。

采用基于角色或权限的授权策略，避免硬编码权限，确保灵活的权限管理。

**12.API设计**

所有API必须遵循RESTful设计原则，统一使用HTTP动词（GET、POST、PUT、DELETE等）和标准的URL路径命名。

使用版本控制（如/api/v1）来管理API的不同版本。

**13.持续集成（CI）**

必须配置自动化构建和测试工具（如Jenkins、Azure DevOps）进行CI集成，确保代码提交后自动进行构建、单元测试和静态分析。

CI过程中需要自动检查代码的风格、测试覆盖率等。

**14.内存管理**

必须避免内存泄漏，尤其在处理大对象、长生命周期对象时，确保及时释放资源。

**15.回滚机制**

在生产环境中必须配置自动回滚机制，确保部署后出现问题时能够快速恢复到稳定版本。

**16.配置文件分离**

所有配置（数据库连接、环境标识等）必须集中在appsettings.{Environment}.json 或 .env 文件中，禁止硬编码配置。

项目必须支持多环境配置切换（开发/测试/生产）。

**17.禁止敏感信息提交**

API密钥、密码、私钥、数据库凭证等敏感信息严禁提交至 Git 仓库，必须使用密钥托管系统（如 Azure Key Vault、Vault）管理。

必须在 .gitignore 中忽略含敏感信息的本地配置文件。

**18.使用枚举替代魔法数字**

项目中所有标识类常量（如状态码、用户等级、配置选项等）必须使用 enum 定义，禁止使用魔法数字或字符串。

**19.接口文档自动生成**

项目必须集成 Swagger（Swashbuckle）或类似工具，自动生成 RESTful 接口文档，确保接口变更可同步对外。

**20.数据模型注释完整**

所有数据模型（DTO/Entity/ViewModel）必须写明注释，说明字段含义、单位、默认值、是否可空，便于团队协作与维护。

**二、推荐（Recommended）**

这些条目可以帮助提升代码质量、可维护性和系统性能，尽量遵循这些规范。

**1.模块化设计**

推荐将应用拆分为多个模块，每个模块负责单一功能，便于团队分工协作和系统的扩展。

使用领域驱动设计（DDD）或者分层架构（如MVC、MVVM）来进行系统架构划分。

**2.单一职责原则**

推荐每个类和方法遵循单一职责原则，避免类和方法承担过多功能，从而降低耦合度。

**3.缓存机制**

推荐使用内存缓存（如MemoryCache）或分布式缓存（如Redis）来提高系统性能，减少频繁的数据库查询。

确保缓存的有效期和更新策略，以避免数据不一致。

**4.依赖版本管理**

推荐使用NuGet包管理外部依赖库，避免手动引用DLL文件。并通过版本锁定确保所有开发者和生产环境使用相同的版本。

**5.设计模式使用**

推荐使用常见的设计模式（如工厂模式、单例模式、观察者模式等）来解决常见的设计问题，确保代码的灵活性和扩展性。

**6.集成测试**

推荐为系统的各个模块之间编写集成测试，确保模块间的正确协作和集成后的功能无误。

**7.API版本控制**

推荐使用/api/v1、/api/v2等路径来版本化API，确保系统升级时能兼容旧版本的客户端。

**8.异步流（Async Streams）**

推荐使用C# 8.0引入的异步流（IAsyncEnumerable）来处理大数据集的异步操作，优化性能和响应性。

**9.日志管理**

推荐使用统一的日志框架（如Serilog、NLog）来记录系统的日志，便于后期故障排查。

使用不同的日志级别（Info、Warn、Error）进行日志分类，避免信息过载。

**10.UI与业务分离**

推荐使用MVVM或MVC模式分离UI层和业务逻辑层，提升代码可维护性和可测试性。

**11.缓存失效策略**

推荐实现合适的缓存失效策略（如LRU算法），确保缓存数据及时更新，避免使用过期数据。

**12.自动化测试工具**

推荐使用自动化测试框架（如NUnit、xUnit）来进行单元测试和集成测试，并与CI工具集成，确保测试的自动执行。

**13.代码重构**

推荐定期进行代码重构，清理不必要的代码，减少重复代码和过时的实现，保持代码的整洁和高效。

**14.数据库优化**

推荐对查询性能较差的数据库操作进行优化，使用索引、优化SQL查询等手段提升数据库性能。

**15.领域事件机制**

推荐使用领域事件（Domain Events）处理跨聚合或跨模块的业务通知逻辑，避免服务之间耦合。

**16.静态代码分析**

推荐使用 SonarQube、ReSharper、Roslyn Analyzer 等工具进行静态代码分析，发现潜在问题与技术债务。

**17.代码提交前自动检查**

推荐配置 Git Hook（如 pre-commit）或使用 Husky，对每次提交进行代码风格检查、自动格式化（如 dotnet-format）与单元测试校验。

**18.错误码集中管理**

推荐将所有系统错误码集中定义在统一类（如 ErrorCodes.cs），并绑定提示文案，便于统一处理错误信息及国际化。

**19.脚手架工具使用**

推荐使用 Yeoman、dotnet CLI 模板等脚手架工具快速创建标准结构项目，避免重复造轮子。

**20.领域层模型隔离**

推荐将数据库实体（Entity）与业务模型（DomainModel）分离，避免 Entity 直接暴露到业务或视图层。

**21.中间件统一处理异常**

推荐通过 ASP.NET Core 的 UseExceptionHandler 或自定义中间件统一拦截并格式化未处理异常。

**22.DTO转换使用自动映射工具**

推荐使用 AutoMapper 等工具统一进行 DTO ↔ Entity 转换，减少冗余的手动映射代码。

**23.合理使用结构体（struct）**

推荐将一些轻量、不可变、频繁创建的值类型（如坐标、颜色等）实现为 readonly struct，提升性能。

**三、允许（Allowed）**

这些条目提供了一定的灵活性，允许开发人员根据项目需求或实际情况作出选择。

**1.使用第三方库**

允许根据项目需求引入第三方开源库（如JSON解析库、数据库ORM工具等），但需经过安全审查，确保库的稳定性和安全性。

**2.并发模型选择**

允许使用不同的并发模型（如Task、Parallel、async/await等），但要注意线程安全和性能问题，避免资源竞争和死锁。

**3.设计架构选择**

允许选择合适的架构模式，如微服务架构、事件驱动架构等，但需经过团队讨论并评审，确保架构的可维护性和扩展性。

**4.数据库访问方式**

允许使用ORM（如Entity Framework、Dapper等）进行数据库操作，但应避免滥用ORM带来的性能问题。

**5.UI框架选择**

允许根据项目需求选择合适的UI框架（如WPF、Blazor等），但需考虑应用的可扩展性和开发效率。

**6.多语言支持**

允许项目支持多语言，特别是针对国际化的应用，确保用户体验的本地化。

**7.代码生成工具使用**

允许使用代码生成工具（如T4模板、Swashbuckle）来自动生成API文档或代码骨架，但需要确保生成的代码符合项目规范。

**8.项目文档格式**

允许根据团队或公司习惯选择合适的文档格式（如Markdown、Confluence等），但必须确保文档的可读性和维护性。