# Архитектурная Трансформация: Неделя 18 — Конфигурация и CI/CD Пайплайны

## 1. Введение: От Статических Ассетов к Динамическим Средам

Переход от разработки на Unity к Enterprise.NET Backend инженерии достигает критической точки на восемнадцатой неделе обучения. До этого момента фокус обучения, согласно плану 1, был сосредоточен на внутренней логике приложения: управлении памятью (SOH/LOH), асинхронности и работе с базами данных. Однако, написание кода — это лишь часть ответственности backend-инженера. В отличие от игрового клиента, который компилируется в монолитный бинарный файл (APK, EXE) и распространяется через магазины приложений, backend-сервис живет в динамичной, враждебной и постоянно меняющейся среде.

В экосистеме Unity конфигурация часто жестко "запекается" в ScriptableObjects или префабы, а секреты (ключи API) нередко прячутся внутри кода или сериализованных полей редактора. В мире Enterprise.NET такой подход неприемлем. Приложение должно быть **stateless** (без сохранения состояния) и **environment-agnostic** (независимым от среды). Один и тот же Docker-образ должен корректно работать и на локальной машине разработчика, и в тестовом контуре (QA), и в продакшене, изменяя свое поведение исключительно за счет внешней конфигурации.1

Эта неделя посвящена фундаментальному сдвигу парадигмы: внедрению методологии **12-Factor App** в процессы конфигурации и автоматизации доставки кода. Мы переходим от ручного копирования файлов и нажатия кнопки "Build" в IDE к строгим, воспроизводимым пайплайнам CI/CD (Continuous Integration / Continuous Deployment). Мы научимся управлять конфигурацией через **Options Pattern**, обеспечивая строгую типизацию настроек 2, и использовать утилиты типа envdir для безопасной инъекции секретов в контейнеры.1

### 1.1. Сравнительный анализ ментальных моделей

Для успешной трансформации необходимо четко разграничить подходы к управлению жизненным циклом приложения в GameDev и Enterprise Backend.

| **Характеристика** | **Подход Unity (GameDev)** | **Подход Enterprise.NET (Backend)** | **Цель обучения Недели 18** |
| --- | --- | --- | --- |
| **Источник конфигурации** | ScriptableObject, JSON в StreamingAssets, PlayerPrefs. Данные часто упакованы внутри билда. | Иерархия провайдеров: Переменные окружения, Аргументы CLI, Файлы (Secrets Mounts), Azure Key Vault.3 | Освоение иерархии IConfiguration и приоритетов перезаписи настроек. |
| **Доступ к настройкам** | Прямой доступ к полям синглтонов (например, GameManager.Instance.Speed). | Внедрение зависимостей (DI) через IOptions<T>, IOptionsSnapshot<T>.2 | Реализация **Options Pattern** для типизированного и валидируемого доступа. |
| **Управление секретами** | Часто игнорируется или используется кастомное шифрование. Риск утечки через декомпиляцию клиента. | Строгое разделение: User Secrets (Dev) и Environment/Vault (Prod). Использование envdir паттерна.5 | Исключение секретов из Git. Настройка безопасной инъекции через Docker Secrets. |
| **Сборка (Build)** | Ручной процесс внутри Unity Editor. Длительная компиляция шейдеров и ассетов. | Автоматизированные CI пайплайны (GitHub Actions), запускаемые по триггеру git push.7 | Написание детерминированных скриптов сборки и тестирования в YAML. |
| **Доставка (Deploy)** | Загрузка билда в Steam/AppStore. Редкие релизы. | Пуш Docker-образов в реестр (GHCR/DockerHub). Непрерывная доставка микросервисов.9 | Создание CD пайплайна с версионированием и автоматической публикацией. |

## 2. Протокол взаимодействия с AI-ментором (GEM-Bot)

На этой неделе роль AI-ассистента трансформируется из "Наставника по C#" в "DevOps-архитектора". Студент должен использовать следующий системный промт для настройки контекста всех диалогов в течение недели. Это обеспечит получение ответов, соответствующих лучшим практикам облачной инфраструктуры и безопасности, а не просто "рабочего кода".

### 2.1. Системный промт на Неделю 18

**System Instruction for GEM-Bot:**

"Ты выступаешь в роли Senior DevOps Architect и Tech Lead платформы.NET Core. Твоя специализация — пересечение программной инженерии и инфраструктуры (Infrastructure as Code). Ты строго придерживаешься методологии '12-Factor App' и обладаешь глубокими знаниями внутреннего устройства ASP.NET Core Host Builder, стратегий кэширования слоев Docker и оптимизации workflows в GitHub Actions.

При ревью кода или ответе на вопросы студента (бывшего Unity-разработчика):

1. **Приоритет безопасности:** Немедленно блокируй любые попытки хардкода учетных данных, паролей или ключей API. Настаивай на использовании User Secrets или переменных окружения.
2. **Идемпотентность и Детерминизм:** Твои скрипты сборки и деплоя должны давать одинаковый результат при повторном запуске.
3. **Оптимизация пропускной способности (Throughput):** Всегда предлагай оптимизации для ускорения сборки Docker (кэширование слоев) и выполнения CI тестов. Время обратной связи критично.
4. **Контекстуализация:** Используй аналогии с Unity (например, сравнивай Docker Layers с Asset Bundles или Prefab Overrides), чтобы объяснить концепции наследования конфигурации, но жестко пресекай привычки 'stateful' разработки.
5. **Архитектурная чистота:** Отвергай использование индексаторов конфигурации (Configuration["Key"]) в бизнес-логике. Требуй использования **Options Pattern** (IOptions<T>) с валидацией.

Твоя цель — трансформировать мышление студента от 'локального билда' к распределенной, автоматизированной доставке надежных сервисов."

## 3. Детальный учебный план: День 1 — Архитектура Конфигурации

**Тема дня:** Переход от "магических строк" к строго типизированной конфигурации. Валидация настроек при старте приложения.

### 3.1. Теоретический модуль: Host Builder и Иерархия Провайдеров

В Unity разработчик привык, что настройки загружаются из одного места. В ASP.NET Core WebApplication.CreateBuilder(args) инициализирует хост, который автоматически настраивает иерархию источников конфигурации.3 Понимание порядка приоритетов критически важно для backend-инженера, так как это позволяет переопределять поведение приложения без перекомпиляции.

Порядок загрузки конфигурации по умолчанию (от низшего к высшему приоритету):

1. **appsettings.json**: Базовая конфигурация.
2. **appsettings.{Environment}.json**: Специфичные настройки среды (Development, Staging, Production). Файл appsettings.Development.json переопределит значения из базового файла.11
3. **User Secrets**: Локальное хранилище секретов разработчика (только в среде Development).
4. **Переменные окружения (Environment Variables)**: Значения, установленные ОС или Docker-контейнером. Обычно имеют префикс ASPNETCORE\_ или DOTNET\_.5
5. **Аргументы командной строки (CLI arguments)**: Наивысший приоритет.

Этот механизм слоев позволяет разработчику иметь безопасные дефолтные настройки в репозитории, но полностью менять их в продакшене через инфраструктуру, не меняя код.12

#### Паттерн Options (Options Pattern)

Использование \_configuration является анти-паттерном в крупных системах из-за отсутствия типизации и риска опечаток ("Magic Strings"). **Options Pattern** решает эту проблему, связывая секции конфигурации с C# классами (POCO).2

Интерфейсы внедрения зависимостей для опций имеют разные жизненные циклы:

* **IOptions<T>**: Singleton. Регистрируется один раз при старте. Не видит изменений в файле appsettings.json после запуска приложения. Идеально для статичных настроек.
* **IOptionsSnapshot<T>**: Scoped. Пересчитывается на каждый HTTP-запрос. Позволяет приложению реагировать на изменения конфигурации "на лету" без перезагрузки (Hot Reload). Это критично для систем с "Zero Downtime" требованиями.2
* **IOptionsMonitor<T>**: Singleton. Предоставляет механизм подписки на изменения (OnChange) для синглтон-сервисов.

#### Валидация (Fail Fast)

Backend-сервис должен падать сразу при старте, если его конфигурация некорректна. Недопустимо, чтобы сервис запустился, но начал выдавать 500-е ошибки при попытке обращения к БД из-за отсутствующей строки подключения. Механизм ValidateOnStart() в сочетании с Data Annotations или FluentValidation позволяет гарантировать целостность настроек до начала приема трафика.16

### 3.2. Практическое задание: "Типизированный Календарь"

**Контекст:** В предыдущих неделях сервис "Календарь" (Calendar Service) использовал прямое чтение настроек для подключения к RabbitMQ и PostgreSQL. Это делает код хрупким.

**Задача:** Рефакторинг слоя конфигурации с внедрением валидации.

1. **Создание моделей опций:**
   * Создать класс RabbitMqOptions со свойствами: Host, VirtualHost, Username, Password, Port.
   * Создать класс DatabaseOptions со свойствами: ConnectionString, CommandTimeout.
2. **Связывание (Binding):**
   * В Program.cs использовать builder.Services.Configure<RabbitMqOptions>(builder.Configuration.GetSection("RabbitMq")).
3. **Реализация валидации:**
   * Подключить System.ComponentModel.DataAnnotations.
   * Добавить атрибуты `` для хостов и паролей.
   * Добавить атрибут `` для порта.
   * Вызвать .ValidateDataAnnotations() и .ValidateOnStart() в цепочке регистрации сервисов.17
4. **Инъекция:**
   * Обновить конструктор RabbitMqService, заменив IConfiguration на IOptions<RabbitMqOptions>.
5. **Тест Hot Reload:**
   * Запустить приложение. Изменить значение в appsettings.json во время работы.
   * Внедрить IOptionsSnapshot<RabbitMqOptions> в контроллер (для теста) и проверить, подхватывается ли новое значение без перезапуска.

### 3.3. Промт для генерации теории с GEM-Bot

**Задание для студента:** Используй этот промт, чтобы углубить понимание работы Options Pattern.

**Промт:** "Я провожу рефакторинг сервиса на.NET 8. Объясни мне разницу в управлении памятью между IOptions<T> и IOptionsSnapshot<T>.

1. Если я использую IOptionsSnapshot, создается ли объект настроек заново для каждого запроса? Как это влияет на GC (Garbage Collector) при высокой нагрузке (10k RPS)?
2. Покажи пример кода, как использовать библиотеку FluentValidation для реализации сложного правила валидации настроек: 'Если SslEnabled равно true, то Port должен быть 5671, иначе 5672'.
3. Как мне протестировать (Unit Test) класс, который зависит от IOptions<MySettings>, используя Moq?"

## 4. Детальный учебный план: День 2 — Управление Секретами и envdir

**Тема дня:** Безопасность конфигурации. Как не слить пароли в Git и использовать файловую систему как источник конфигурации в стиле envdir.

### 4.1. Теоретический модуль: Принцип 12-Factor Config

Методология **12-Factor App** гласит: "Сохраняйте конфигурацию в среде выполнения".1 Это означает строгое табу на хранение паролей, токенов и ключей шифрования в репозитории кода, даже в приватном.

#### Проблема appsettings.json

Многие новички совершают ошибку, коммитя appsettings.json с реальными паролями. В Unity часто создается класс Keys.cs, который исключается из Git, но в.NET ecosystem есть более надежные инструменты.

#### Инструменты инъекции секретов

1. **User Secrets (Для разработки):** Инструмент CLI, который хранит секреты в системной папке профиля пользователя (%APPDATA% на Windows, ~/.microsoft/usersecrets на Linux/macOS).13 Это гарантирует, что секреты физически находятся вне папки проекта и не могут быть случайно добавлены в коммит.
2. **Docker Secrets и envdir (Для продакшена):** В оркестраторах (Docker Swarm, Kubernetes) секреты монтируются как *файлы* в файловую систему контейнера (обычно в /run/secrets/). Имя файла соответствует ключу, содержимое — значению. Это концептуально идентично утилите envdir.1

#### Реализация envdir паттерна в.NET

Хотя.NET умеет читать переменные окружения, чтение из файлов (Docker Secrets) требует специального провайдера конфигурации KeyPerFileConfigurationProvider.6 Это позволяет приложению прозрачно получать настройки:

* В Dev-среде: из User Secrets (через механизм.NET).
* В Prod-среде: из файлов в /run/secrets (через AddKeyPerFile).  
  Для кода приложения источник прозрачен — оно просто запрашивает Configuration.

### 4.2. Практическое задание: "Секретный Контур"

**Контекст:** Сервис уведомлений (Notification Service) требует доступа к SMTP-серверу и RabbitMQ. Эти данные критичны.

**Задача:** Полностью исключить чувствительные данные из исходного кода.

1. **Инициализация User Secrets:**
   * Выполнить dotnet user-secrets init в папке проекта.
   * Перенести пароль RabbitMQ и SMTP-креденшалы из appsettings.json в User Secrets:  
     dotnet user-secrets set "RabbitMq:Password" "super\_secure\_pass".13
   * Убедиться, что приложение запускается локально.
2. **Симуляция Docker Secrets (envdir):**
   * Создать локальную папку .secrets\_mock.
   * Создать файл .secrets\_mock/RabbitMq\_\_Password (обратите внимание на двойное подчеркивание для вложенности секций) с содержимым docker\_prod\_pass.
3. **Настройка конфигурации:**
   * В Program.cs добавить код, который подключает KeyPerFile, если директория существует:  
     C#  
     var secretsPath = "/run/secrets";  
     if (Directory.Exists(secretsPath))  
     {  
      builder.Configuration.AddKeyPerFile(directoryPath: secretsPath, optional: true);  
     }
4. **Docker Compose интеграция:**
   * Обновить docker-compose.yml. Использовать секцию secrets для монтирования файла с паролем в /run/secrets/RabbitMq\_\_Password внутри контейнера.6
   * Настроить override переменных окружения через env\_file для несекретных настроек.5

### 4.3. Промт для генерации теории с GEM-Bot

**Задание для студента:**

**Промт:** "Я настраиваю управление секретами для контейнеризированного.NET приложения.

1. Проанализируй риски безопасности при передаче паролей через обычные переменные окружения в Docker (ENV MY\_PASS=...) по сравнению с использованием Docker Secrets (файловые маунты). Что увидит злоумышленник, выполнив docker inspect?
2. Помоги мне настроить docker-compose.yml так, чтобы он брал пароль из локального файла на моем компьютере и монтировал его как секрет в контейнер.
3. Напиши C# код для Program.cs, который использует AddKeyPerFile для чтения конфигурации из папки /run/secrets, правильно обрабатывая иерархию ключей (например, файл Database\_\_Password должен мапиться в секцию Database:Password)."

## 5. Детальный учебный план: День 3 — Контейнеризация и Оптимизация Сборки

**Тема дня:** Написание идеального Dockerfile. Многоэтапные сборки (Multi-stage builds) и кэширование слоев для ускорения CI.

### 5.1. Теоретический модуль: Анатомия Dockerfile для.NET

Прямолинейный подход к созданию Docker-образа ("скопировать всё и скомпилировать") приводит к огромным образам и медленной сборке. В CI/CD скорость обратной связи критична. Если билд идет 10 минут, разработчик теряет контекст.

#### Многоэтапные сборки (Multi-Stage Builds)

Мы разделяем среду сборки (Build Environment) и среду выполнения (Runtime Environment).

* **Build Stage:** Использует образ SDK (mcr.microsoft.com/dotnet/sdk), который весит сотни мегабайт. Содержит компилятор Roslyn, MSBuild и инструменты CLI.
* **Runtime Stage:** Использует образ Runtime (mcr.microsoft.com/dotnet/aspnet), который значительно легче и не содержит исходного кода. Мы просто копируем скомпилированные DLL из первого этапа во второй.9

#### Кэширование слоев (Layer Caching) и dotnet restore

Docker кэширует каждый шаг (инструкцию) в Dockerfile. Если входные данные шага не изменились, Docker берет результат из кэша.

Критическая оптимизация для.NET:

Если написать:

Dockerfile

COPY..  
RUN dotnet restore

То любое изменение в коде (даже пробел в комментарии) инвалидирует кэш для COPY.., и dotnet restore будет выполняться заново, скачивая мегабайты пакетов из Nuget.

Правильный подход:

Dockerfile

COPY  
RUN dotnet restore "MyService.csproj"  
COPY..  
RUN dotnet build

Здесь dotnet restore выполнится заново, только если изменится файл проекта (.csproj), то есть при добавлении новых зависимостей. В 99% сборок этот слой будет взят из кэша мгновенно.22

#### Безопасность контейнера

Запуск контейнеров от имени root — уязвимость. Современные образы.NET 8 поддерживают запуск от пользователя без привилегий (обычно $APP\_UID).9

### 5.2. Практическое задание: "Полиглот Dockerfile"

**Контекст:** Сервис "Календарь" нуждается в production-ready образе.

**Задача:** Написать и оптимизировать Dockerfile.

1. **Draft Multi-Stage Dockerfile:**
   * **Stage 1 (Build):** FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build.
   * **Stage 2 (Publish):** RUN dotnet publish -o /app/publish.
   * **Stage 3 (Final):** FROM mcr.microsoft.com/dotnet/aspnet:8.0. Копирование артефактов из Stage 2.
2. **Оптимизация слоев:**
   * Рефакторинг команд COPY. Сначала копировать только .csproj.
   * Выполнить dotnet restore.
   * Только потом копировать остальной исходный код (COPY..).
3. **Настройка пользователя:**
   * Добавить инструкцию USER $APP\_UID в финальном стейдже для запуска без прав root.
4. **.dockerignore:**
   * Создать файл .dockerignore. Исключить папки bin, obj, .git, .vs, .idea. Это предотвращает попадание локального мусора в контекст сборки, уменьшая его размер и предотвращая ложноположительные инвалидации кэша.
5. **Бенчмарк:**
   * Замерить время первой сборки (docker build --no-cache).
   * Изменить одну строку в .cs файле.
   * Замерить время повторной сборки. Убедиться, что шаг Restore занял 0 секунд (Cached).

### 5.3. Промт для генерации теории с GEM-Bot

**Задание для студента:**

**Промт:** "Я оптимизирую Dockerfile для.NET Core приложения.

1. Объясни концепцию 'Layer Invalidation' в Docker. Почему инструкция COPY.. перед dotnet restore считается ошибкой производительности?
2. Я хочу использовать Alpine Linux (образ 8.0-alpine) для уменьшения размера образа. Какие подводные камни это несет для.NET приложения (например, касательно musl libc vs glibc и работы с Globalization/ICU)?
3. Проверь мой файл .dockerignore. Не забыл ли я исключить какие-то файлы, специфичные для Unity-разработчика (например, .meta файлы или папки Library), если бы этот проект лежал в смешанном репозитории?"

## 6. Детальный учебный план: День 4 — CI Пайплайны и Интеграционное Тестирование

**Тема дня:** Автоматизация проверок качества. GitHub Actions, Service Containers и решение проблемы "Works on my machine".

### 6.1. Теоретический модуль: Анатомия GitHub Actions

GitHub Actions (GHA) — это стандарт де-факто для CI в экосистеме GitHub. Пайплайн описывается в YAML-файле и состоит из триггеров, джобов (Jobs) и шагов (Steps).7

#### Service Containers vs Testcontainers

Для запуска интеграционных тестов (изученных на 15-й неделе) нужна живая база данных. В CI у нас есть два пути:

1. **Service Containers:** Мы описываем зависимость (Postgres, Redis) прямо в YAML файле workflow. GHA запускает контейнер и "линкует" его сетевые порты к хосту, где бегут тесты.8 Это декларативный подход, удобный для простых конфигураций.
2. **Testcontainers:** Библиотека C#, которая поднимает контейнеры программно из кода тестов.25 Это дает большую изоляцию, но требует наличия Docker-сокета, доступного для раннера тестов.

*Стратегия на сегодня:* Мы будем использовать **Service Containers**, так как это нагляднее демонстрирует структуру YAML пайплайна.

#### Кэширование зависимостей в CI

В отличие от локального Docker, раннеры GitHub Actions каждый раз запускаются "чистыми". dotnet restore будет каждый раз скачивать гигабайты пакетов. Чтобы этого избежать, используется экшен actions/cache. Мы кэшируем папку ~/.nuget/packages используя хэш всех файлов \*.csproj в качестве ключа кэша.7

### 6.2. Практическое задание: "Железный Пайплайн"

**Контекст:** Настроить CI процесс, который блокирует Pull Request, если код не собирается или тесты падают.

**Задача:** Создать файл .github/workflows/dotnet-ci.yml.

1. **Триггеры:** Настроить запуск на push в ветку main и на pull\_request в main.
2. **Service Containers:**
   * Добавить сервис postgres (образ postgres:15-alpine).
   * Добавить сервис rabbitmq (образ rabbitmq:3-management).
   * Пробросить порты 5432:5432 и 5672:5672.
   * Настроить health-check для Postgres (команда pg\_isready), чтобы тесты не запускались, пока база не готова принимать соединения (избегание Race Condition).24
3. **Шаги сборки (Steps):**
   * Checkout: Получение кода.
   * Setup.NET: Установка SDK 8.0.
   * **Cache Nuget:** Использовать actions/cache для папки ~/.nuget/packages.
   * dotnet restore: Восстановление зависимостей.
   * dotnet build --no-restore: Сборка.
   * dotnet test --no-build --verbosity normal: Запуск тестов.
4. **Конфигурация тестов:**
   * Через блок env в шаге тестирования передать строку подключения к БД, указывающую на localhost (так как Service Containers пробрасывают порты на хост раннера).23
5. **Артефакты:**
   * Настроить загрузку отчетов о тестировании (Test Results) как артефактов пайплайна в случае падения тестов.8

### 6.3. Промт для генерации теории с GEM-Bot

**Задание для студента:**

**Промт:** "Я настраиваю GitHub Actions workflow с использованием Service Containers (Postgres) для интеграционных тестов.

1. Как мне избежать 'Race Condition', когда dotnet test запускается раньше, чем контейнер Postgres полностью инициализировал базу данных? Покажи, как использовать options и health-check в YAML конфигурации сервиса.
2. Сравни использование экшена actions/setup-dotnet (который имеет встроенное кэширование в новых версиях) и ручного использования actions/cache для глобальной папки nuget. Что предпочтительнее в 2025 году?
3. Мои тесты падают в CI с ошибкой 'Connection Refused', хотя локально работают. Проанализируй мои маппинги портов в YAML и строку подключения, которую я передаю через переменные окружения."

## 7. Детальный учебный план: День 5 — CD и Стратегии Развертывания

**Тема дня:** Непрерывная доставка (Continuous Deployment). Публикация артефактов и стратегии релизов.

### 7.1. Теоретический модуль: CD Пайплайн и Реестры

CD расширяет CI: после успешного прохождения тестов мы должны упаковать приложение и доставить его. В мире контейнеров это означает сборку Docker-образа и его пуш (Push) в реестр (Registry) — например, GitHub Container Registry (GHCR) или Docker Hub.

#### Версионирование и Тэгирование

Нельзя просто обновлять тэг latest. Хорошая практика — использовать "Commit Hash" или "Semantic Versioning" для тэгов.

* **SHA Tag (myapp:sha-a1b2c3d):** Позволяет точно знать, из какого коммита собран образ. Незаменимо для отладки.
* **SemVer (myapp:v1.0.1):** Для публичных релизов.
* **Стратегия:** При каждом пуше в main мы создаем образ с тэгом SHA и обновляем latest.9

#### Безопасность в CD

Для пуша в реестр нужны права. GitHub Actions предоставляет автоматический токен GITHUB\_TOKEN, который можно использовать для аутентификации в GHCR без создания долгоживущих паролей. Это безопаснее, чем хранить пароль от Docker Hub в секретах репозитория.30

#### Концепция Blue/Green Deployment

Для достижения "Zero Downtime" (нулевого времени простоя) используется стратегия Blue/Green. Мы разворачиваем новую версию (Green) рядом со старой (Blue), ждем прохождения Health Checks, и только потом переключаем трафик. Хотя полная реализация требует балансировщика нагрузки (Nginx/Traefik/K8s Ingress), понимание подготовки контейнера к этому (Graceful Shutdown) обязательно.31

### 7.2. Практическое задание: "Publish or Perish"

**Контекст:** Сервис готов к релизу. Нужно автоматизировать его публикацию.

**Задача:** Расширить пайплайн dotnet-ci.yml.

1. **Новый Job:** Создать джоб build-and-push, который зависит от успешного выполнения test (needs: test) и запускается только на ветке main.
2. **Docker Login:** Использовать экшен docker/login-action для входа в ghcr.io с использованием ${{ secrets.GITHUB\_TOKEN }}.
3. **Метаданные:** Использовать docker/metadata-action для автоматической генерации тэгов (включая latest, sha, и тэги на основе Git Tag, если есть).
4. **Build & Push:** Использовать docker/build-push-action 29:
   * Включить push: true.
   * Передать сгенерированные тэги.
   * **Docker Cache:** Настроить кэширование слоев Docker внутри GitHub Actions (cache-from: type=gha, cache-to: type=gha,mode=max). Это критически ускоряет сборку, используя кэш предыдущих запусков пайплайна.29
5. **Верификация:**
   * Запушить код. Убедиться, что пайплайн прошел.
   * Зайти в раздел "Packages" репозитория и найти там свежий Docker-образ.
   * Стянуть его локально (docker pull...) и запустить.

### 7.3. Промт для генерации теории с GEM-Bot

**Задание для студента:**

**Промт:** "Я хочу опубликовать Docker-образ моего сервиса в GitHub Container Registry (GHCR).

1. Напиши YAML шаг для аутентификации в GHCR используя автоматический GITHUB\_TOKEN. Какие права (permissions) я должен явно прописать в workflow файле для секции packages?
2. Объясни, как работает кэширование docker/build-push-action с параметром type=gha. Этот кэш общий с Nuget-кэшем, который мы настроили в джобе тестирования, или это отдельное хранилище?
3. Я хочу реализовать стратегию 'Nightly Builds'. Как мне изменить триггеры workflow, чтобы он запускался не только по пушу, но и по расписанию (cron) каждую ночь в 02:00?
4. Объясни концепцию 'Blue/Green Deployment'. Если новая версия приложения требует миграции базы данных (изменение схемы), как это усложняет Blue/Green развертывание и какие стратегии миграции тут применяются?"

## 8. Итоговые Проекты Недели (Capstone Projects)

Эти проекты призваны консолидировать знания по конфигурации, безопасности, контейнеризации и CI/CD.

### Проект 1: Refactor Legacy "12-Factor" Calendar

**Сценарий:** Вы унаследовали "Legacy" версию сервиса Календаря (написанную вами же на 7-й неделе), где строка подключения захардкожена, JWT-секрет лежит в appsettings.json, и нет никакой валидации настроек.

**Требования:**

1. **Options Pattern:** Полностью переписать работу с конфигурацией на IOptions<CalendarSettings>.
2. **Validation:** Реализовать интерфейс IValidateOptions<CalendarSettings> или использовать Data Annotations, чтобы гарантировать, что JwtSecret имеет длину не менее 32 символов, а MaxEventDuration положительное число.
3. **Secret Injection:** Модифицировать docker-compose.yml, чтобы пароль от БД передавался через file mount (/run/secrets/db\_password).
4. **KeyPerFile:** Настроить ConfigurationBuilder на чтение секретов из директории, маппя файл db\_password в конфигурационный ключ Database:Password.
5. **Результат:** Pull Request с рефакторингом и скриншот логов, где приложение успешно стартует с примонтированными секретами и падает с ошибкой валидации, если секрет убрать.

### Проект 2: "Zero-Click" Notification Pipeline

**Сценарий:** Микросервис уведомлений (Notification Service), который слушает RabbitMQ, сейчас деплоится вручную. Необходимо построить для него полностью автоматический конвейер.

**Требования:**

1. **CI Pipeline:**
   * Триггер на Pull Request.
   * Поднятие RabbitMQ в Service Container.
   * Запуск интеграционных тестов, которые публикуют реальное сообщение в очередь и проверяют (через TestConsumer или сайд-эффекты), что сервис его обработал.
2. **CD Pipeline:**
   * Триггер на Push в main.
   * Сборка оптимизированного Docker-образа (Multi-stage).
   * Пуш в GHCR с тэгами latest и sha-{commit\_hash}.
3. **Documentation:** Добавить в README.md бейдж (Badge) статуса сборки из GitHub Actions.

## 9. Глубокий Анализ и Инсайты (Deep Dive)

### 9.1. Конфликт Состояний: Unity vs. Enterprise.NET

Самый значительный ментальный барьер, идентифицированный в документе 1, — это переход от stateful (Unity) к stateless (.NET) архитектуре. На 18-й неделе это проявляется в Конфигурации.

В Unity объект ScriptableObject, хранящий настройки, существует в памяти и может быть изменен во время выполнения для отладки через Инспектор. В распределенной системе.NET изменение синглтона конфигурации в памяти одного инстанса сервиса приведет к рассинхронизации состояния ("State Drift") — другие инстансы (или Green-версия при деплое) не узнают об изменении.

Инсайт: Именно поэтому использование IOptionsSnapshot предпочтительнее ручного кэширования значений. Snapshot заставляет приложение перечитывать "источник правды" (провайдер конфигурации) для каждого запроса. Если внешний источник (например, Kubernetes ConfigMap, смонтированный как файл) обновится, приложение подхватит это изменение автоматически, сохраняя согласованность поведения всего кластера.

### 9.2. CI/CD как "Игровой Цикл" (Game Loop) Разработки

Подобно тому как Update() управляет кадром игры, CI/CD Пайплайн управляет жизненным циклом продукта.

Наблюдение: Многие Unity-разработчики воспринимают билд-сервер как просто "удаленный компилятор".

Коррекция: В Enterprise архитектуре CI пайплайн — это единственная точка входа кода в продакшен. Если код не проходит CI, его не существует. Навыки Дня 4 (Service Containers) создают так называемые "эфемерные окружения" (ephemeral environments), где каждый Pull Request поднимает полную копию инфраструктуры. Это архитектурная невозможность реализации принципа "работает на моей машине".

### 9.3. Безопасность через файловую систему

Требование использовать envdir и файловые маунты для секретов 1 не случайно.

Глубокое погружение: Хотя.NET имеет провайдер переменных окружения, сами переменные окружения в Linux не являются полностью безопасными — они видны в листингах процессов (/proc/{pid}/environ) и могут утекать в логи ошибок. Файлы, смонтированные в RAM-диск (как делают Docker Secrets и Kubernetes Secrets в /run/secrets), намного безопаснее.

Импликация: Студент учится воспринимать файловую систему контейнера не как место для сохранения данных (SaveGame), которое исчезает при перезапуске, а как механизм безопасной инъекции конфигурации времени выполнения. Это полная инверсия опыта Unity, где Application.persistentDataPath используется для сохранений, а память — для конфигов.

## 10. Ресурсы и Литература

В ходе составления плана использовались следующие материалы, рекомендуемые к детальному изучению:

* **ASP.NET Core Configuration & Options:**.2
* **Docker Optimization & Security:**.9
* **Secret Management (Envdir/UserSecrets):**.5
* **GitHub Actions & CI/CD:**.7
* **Deployment Strategies:**.31

*Конец отчета по Плану Обучения Недели 18.*

#### Источники

1. Разработка бэкенда: план обучения C#
2. How to Use the Options Pattern in ASP.NET Core for Strongly Typed Configuration, дата последнего обращения: декабря 6, 2025, <https://www.ottorinobruni.com/how-to-use-the-options-pattern-in-asp-net-core-for-strongly-typed-configuration/>
3. Configuration in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-10.0>
4. Options pattern in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-10.0>
5. .NET Core : Using .env Files for Deployment Containerization | by Sayedur Rahaman | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@sayed.cse01/net-core-using-env-files-for-containerization-9065d9be4c93>
6. Secrets in Compose - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/use-secrets/>
7. Dependency caching reference - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching>
8. Building and testing .NET - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/guides/building-and-testing-net>
9. Publishing image to GitHub Container Registry using GitHub Actions - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/reniciuspagotto/publishing-docker-image-to-github-container-registry-using-github-actions-20dg>
10. Automating .NET Deployment with GitHub Actions and Docker | by Roko Kovač - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@kova98/automating-net-deployment-with-github-actions-and-docker-d43109d34f88>
11. ASP.NET Core runtime environments - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-10.0>
12. The confusion of ASP.NET Configuration with environment variables | by Georgi Parlakov, дата последнего обращения: декабря 6, 2025, <https://gparlakov.medium.com/the-confusion-of-asp-net-configuration-with-environment-variables-c06c545ef732>
13. Safe storage of app secrets in development in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-10.0>
14. Options pattern - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/options>
15. Options Pattern in ASP.NET Core - NikolaTech, дата последнего обращения: декабря 6, 2025, <https://www.nikolatech.net/blogs/options-pattern-asp-dotnet-core>
16. Options Pattern Validation in ASP.NET Core With FluentValidation, дата последнего обращения: декабря 6, 2025, <https://www.milanjovanovic.tech/blog/options-pattern-validation-in-aspnetcore-with-fluentvalidation>
17. Validation of ASP.NET Core options during startup - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/51692665/validation-of-asp-net-core-options-during-startup>
18. Why should I use User secrets only in the Development environment? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/56626305/why-should-i-use-user-secrets-only-in-the-development-environment>
19. Manage sensitive data with Docker secrets, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/swarm/secrets/>
20. Using Docker Secrets to replace environment vars without modifying images - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/docker/comments/9k4tdi/using_docker_secrets_to_replace_environment_vars/>
21. Tutorial: Create a GitHub Action with .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/devops/create-dotnet-github-action>
22. Docker Samples - Nuget Caching - Why is this a separate build step in all the examples? #6123 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/dotnet/dotnet-docker/discussions/6123>
23. Run Integration tests with Real DB on Github Workflow | I am Raghuveer, дата последнего обращения: декабря 6, 2025, <https://www.iamraghuveer.com/posts/run-db-integration-tests-sqlserver-postgres/>
24. Creating PostgreSQL service containers - GitHub Docs, дата последнего обращения: декабря 6, 2025, <https://docs.github.com/actions/guides/creating-postgresql-service-containers>
25. Running Testcontainers Tests Using GitHub Actions and Testcontainers Cloud - Docker, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/running-testcontainers-tests-using-github-actions/>
26. How to create and manage unit test resources (best practices) #438 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/testcontainers/testcontainers-dotnet/discussions/438>
27. Setup .NET Core SDK · Actions · GitHub Marketplace, дата последнего обращения: декабря 6, 2025, <https://github.com/marketplace/actions/setup-net-core-sdk>
28. Integration Test Postgres using GitHub Actions - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/kashifsoofi/integration-test-postgres-using-github-actions-3lln>
29. Build and push Docker images · Actions · GitHub Marketplace, дата последнего обращения: декабря 6, 2025, <https://github.com/marketplace/actions/build-and-push-docker-images>
30. Publish .NET Docker images using .NET SDK and GitHub Actions - Laurent Kempé, дата последнего обращения: декабря 6, 2025, <https://laurentkempe.com/2023/10/30/publish-dotnet-docker-images-using-dotnet-sdk-and-github-actions/>
31. blue-green-deployment · GitHub Topics, дата последнего обращения: декабря 6, 2025, <https://github.com/topics/blue-green-deployment>
32. How I created a simple blue/green deployment using only Docker Compose and GitHub Actions - Primfeed Blog, дата последнего обращения: декабря 6, 2025, <https://blog.primfeed.com/2025/07/how-i-created-a-simple-blue-green-deployment-using-only-docker-compose-and-github-actions/>
33. elafarge/docker-bluegreen: A proof of concept for zero downtime blue/green deployments with HAProxy, Docker and docker-compose. - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/elafarge/docker-bluegreen>
34. Deploy Docker Compose applications with zero downtime using GitHub Actions - jmh, дата последнего обращения: декабря 6, 2025, <https://jmh.me/blog/zero-downtime-docker-compose-deploy>
35. GitHub Actions cache - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/cache/backends/gha/>
36. Docker Best Practices: Using ARG and ENV in Your Dockerfiles, дата последнего обращения: декабря 6, 2025, <https://www.docker.com/blog/docker-best-practices-using-arg-and-env-in-your-dockerfiles/>
37. Load .env Files in C# .NET - rmauro.dev {blog}, дата последнего обращения: декабря 6, 2025, <https://rmauro.dev/read-env-file-in-csharp/>