# Архитектура Облачно-Нативных Систем: Углубленный Теоретический Анализ Оркестрации, Сетевого Взаимодействия и Персистентности Данных

## 1. Введение: Смена Парадигмы от Императивного Кода к Декларативной Инфраструктуре

Переход к пятнадцатой неделе обучения в программе подготовки Enterprise-разработчика знаменует собой фундаментальный онтологический сдвиг. Если предыдущие этапы фокусировались на внутренней механике языка C# — управлении памятью в кучах SOH/LOH, работе сборщика мусора и асинхронных конечных автоматах, — то текущий модуль требует переосмысления среды, в которой этот код существует.1 Для разработчика с бэкграундом в Unity, где среда исполнения (Runtime) часто воспринимается как статичная константа, предоставляемая игровым движком и операционной системой конечного пользователя, концепции облачно-нативной разработки могут показаться контринтуитивными.

В мире Unity и десктопной разработки понятие «окружение» жестко связано с физической машиной пользователя. Файловая система персистентна по умолчанию, сеть доступна напрямую через интерфейсы хоста, а localhost всегда обозначает локальную машину. В облачной архитектуре эти аксиомы становятся ложными. Среда исполнения эфемерна, файловая система уничтожается при перезапуске процесса, сеть виртуализирована и изолирована, а понятие localhost становится контекстно-зависимым и часто вводящим в заблуждение.1

Второй день модуля, посвященный Docker Compose, сетям и хранению данных, является точкой бифуркации, где инженер перестает мыслить категориями «исполняемых файлов» и начинает мыслить категориями «распределенных систем». Сервис «Календарь», разрабатываемый в рамках курса, перестает быть изолированным Web API и становится узлом в сложной топологии, включающей базы данных, брокеры сообщений и сетевые шлюзы.

Данный отчет представляет собой исчерпывающее теоретическое исследование технологий, обеспечивающих работу этой топологии. Мы выйдем за рамки синтаксиса YAML-файлов и погрузимся в примитивы ядра Linux, обеспечивающие изоляцию (Namespaces и Cgroups), архитектурные паттерны обнаружения сервисов (Service Discovery), механику ввода-вывода драйверов хранения (Storage Drivers) и иерархию конфигурационных провайдеров ASP.NET Core, необходимую для создания приложений, соответствующих методологии Twelve-Factor App.

## 2. Философия Оркестрации и Инфраструктура как Код (IaC)

### 2.1. От Императивных Скриптов к Декларативной Топологии

Исторически настройка серверного окружения выполнялась императивно. Системный администратор выполнял последовательность команд: «установить PostgreSQL», «создать пользователя», «открыть порт», «запустить приложение». Этот подход, аналогичный написанию игрового скрипта в Update() методе Unity, где состояние изменяется кадр за кадром, страдает от хрупкости. Если один шаг дает сбой, система оказывается в неопределенном состоянии.

Docker Compose внедряет декларативную модель. В файле docker-compose.yml инженер описывает не *последовательность действий*, а *желаемое конечное состояние* системы (Desired State Configuration). Мы определяем, что сервис базы данных *должен* существовать, использовать определенный образ и иметь примонтированный том. Задача оркестратора — привести текущее состояние системы к желаемому, самостоятельно вычислив необходимые действия (создание сетей, запуск контейнеров в правильном порядке).

Docker Compose выступает в роли локального инструмента «Инфраструктура как Код» (Infrastructure as Code, IaC).2 В то время как Kubernetes является стандартом для оркестрации кластеров из множества узлов, Docker Compose является де-факто стандартом для описания мультиконтейнерных приложений на одном хосте. Для проекта «Календарь» файл docker-compose.yml служит трем критическим целям:

1. **Документация Архитектуры:** Файл явно перечисляет все инфраструктурные зависимости. В отличие от документации в Wiki, которая может устареть, docker-compose.yml является исполняемой документацией. Если зависимость не описана, приложение не запустится.
2. **Изоляция Среды:** Compose позволяет создавать изолированные пространства имен для проекта, гарантируя, что сеть проекта «Календарь» не пересечется с сетью проекта «Биллинг», даже если они запущены на одной машине разработчика.4
3. **Воспроизводимость (Reproducibility):** Он гарантирует, что топология сети и конфигурация томов в тестовой среде (CI Pipeline) будут математически идентичны топологии на локальной машине разработчика, устраняя класс ошибок «работает на моей машине».1

### 2.2. Абстракция Сервиса против Контейнера

Важнейшим концептуальным различием в Docker Compose, часто упускаемым новичками, является разница между «Сервисом» (Service) и «Контейнером» (Container). В спецификации Compose мы определяем **Сервисы**.5 Сервис — это абстрактная конфигурация вычислительного ресурса, включающая образ, порты, переменные окружения и тома.

Когда Docker Compose запускается, он инстанцирует на основе определения Сервиса один или несколько **Контейнеров**.

* **Сервис:** calendar-api (описание того, *как* запускать приложение).
* **Контейнер:** calendar\_app-calendar-api-1 (конкретный запущенный процесс).

Эта абстракция является фундаментом для масштабирования. Команда docker compose up --scale calendar-api=3 заставит оркестратор создать три независимых контейнера на основе одного определения сервиса. При этом Compose автоматически обрабатывает коллизии имен и портов (если порты не привязаны жестко к хосту), включая их в общую виртуальную сеть. Понимание этого различия критично при переходе к Kubernetes, где аналогичная дихотомия существует между понятиями Deployment (Сервис) и Pod (Контейнер).

### 2.3. Изоляция Проектов и Пространства Имен

При выполнении команды docker compose up оркестратор должен гарантировать, что ресурсы, создаваемые для текущего проекта, не повлияют на другие проекты. Механизм изоляции в Docker Compose базируется на понятии «Имя Проекта» (Project Name).4

По умолчанию имя проекта выводится из названия директории, в которой находится файл конфигурации. Если исходный код сервиса «Календарь» находится в папке calendar\_backend, то все создаваемые ресурсы будут иметь префикс calendar\_backend:

* Сеть: calendar\_backend\_default.
* Том: calendar\_backend\_pg\_data.
* Контейнер: calendar\_backend-calendar-db-1.

**Глубокий инсайт (Second-Order Insight):** В средах непрерывной интеграции (CI/CD) использование имени директории по умолчанию является анти-паттерном. Агенты сборки (Build Agents) могут выполнять несколько пайплайнов параллельно в разных рабочих директориях, но если имена директорий совпадут или будут переиспользованы, возникнет состояние гонки (Race Condition), где тесты одной ветки перезапишут базу данных другой.

* **Решение:** В энтерпрайз-пайплайнах необходимо явно задавать имя проекта через флаг -p или переменную окружения COMPOSE\_PROJECT\_NAME, используя уникальный идентификатор сборки (например, BuildID или GitCommitHash). Это обеспечивает абсолютную изоляцию инфраструктуры тестов, позволяя запускать сотни параллельных прогонов на одном сервере без коллизий портов или данных.6

### 2.4. Управление Профилями и Контекстами Запуска

Современные приложения часто требуют различных наборов сервисов для разных сценариев. Например, для unit-тестирования база данных не нужна (используются моки), для интеграционного тестирования нужна база данных, но не нужен UI, а для отладки нужны и база, и UI, и инструменты мониторинга (Prometheus/Grafana).

До версии 2.28 разработчики были вынуждены создавать множество файлов docker-compose.dev.yml, docker-compose.test.yml и объединять их через флаг -f. Современный Docker Compose внедрил концепцию **Профилей** (Profiles).8

Профили позволяют аннотировать сервисы метками (например, profiles: ["debug", "infra"]). Сервисы, помеченные профилем, не запускаются по умолчанию при вызове docker compose up. Они активируются только при явном указании профиля: docker compose --profile debug up.

Это позволяет хранить описание всей экосистемы проекта (приложение, БД, брокеры сообщений, логгеры, инструменты миграции) в одном файле compose.yaml («Монолитная конфигурация»), но запускать лишь необходимое подмножество сервисов в зависимости от задачи. Для разработчика C# это аналогично использованию директив препроцессора #if DEBUG, но на уровне инфраструктуры.10

## 3. Глубокое Погружение: Сетевая Архитектура Контейнеров

Наиболее сложным аспектом при переходе разработчика с монолитной архитектуры на микросервисную является сеть. Интуитивное понимание сети, сформированное годами работы с локальным сервером IIS или Kestrel, где все компоненты доступны по localhost, становится источником критических ошибок в контейнерной среде.

### 3.1. Базис Ядра Linux: Сетевые Пространства Имен (Network Namespaces)

Чтобы понять, как работает Docker, необходимо опуститься на уровень системных вызовов ядра Linux. Основой изоляции является **Network Namespace (netns)**.11

Пространство имен сети — это логическая копия сетевого стека операционной системы. Каждое пространство имен имеет свои собственные:

* Сетевые интерфейсы (lo, eth0).
* Таблицы маршрутизации (Routing Tables).
* Правила брандмауэра (iptables/nftables).
* Сокеты (TCP/UDP порты).

Когда хост-система загружается, она работает в «корневом» (root) пространстве имен. Когда Docker запускает контейнер, он выполняет системный вызов clone() с флагом CLONE\_NEWNET (или использует утилиту unshare).13 Это создает абсолютно пустой сетевой стек для процесса контейнера.

Внутри этого нового пространства имен изначально нет ничего, кроме (обычно выключенного) интерфейса loopback. Процесс внутри контейнера не видит сетевых карт хоста, не видит открытых портов хоста и не может взаимодействовать с внешним миром.14

**Virtual Ethernet Pair (veth):** Чтобы соединить этот изолированный мир с реальностью, Docker использует пару виртуальных ethernet-устройств (veth pair). Это, по сути, цифровой патч-корд.

1. Один конец пары (vethXXXX) остается в корневом пространстве имен хоста и подключается к виртуальному мосту.
2. Второй конец перемещается внутрь пространства имен контейнера, переименовывается в eth0 и получает IP-адрес из диапазона подсети моста.14

### 3.2. Драйвер Bridge: Виртуальная Коммутация

Docker Compose по умолчанию создает для проекта сеть типа bridge. Виртуальный мост (bridge) работает как программный Layer 2 свитч (коммутатор).

1. **Default Bridge (docker0):** Исторический режим. Все контейнеры без явного указания сети попадают сюда. Этот режим имеет серьезные недостатки: в нем **не работает** автоматическое разрешение имен (DNS resolution) между контейнерами по имени сервиса. Контейнеры могут общаться только по IP-адресам, которые эфемерны и меняются при перезапуске.16
2. **User-Defined Bridge:** Именно этот тип сети создает Docker Compose (например, calendar\_net). В пользовательских мостах Docker включает встроенный механизм Service Discovery.16

Архитектура прохождения пакета:

Когда calendar-api отправляет HTTP-запрос к calendar-db:

1. Пакет формируется внутри пространства имен API-контейнера.
2. Он покидает контейнер через интерфейс eth0.
3. Через veth-пару он попадает на порт виртуального моста (br-xxxx) на хосте.
4. Ядро Linux, выполняя функции коммутации, анализирует ARP-таблицу моста, находит MAC-адрес контейнера БД и пересылает пакет в соответствующий veth-интерфейс.
5. Пакет попадает в пространство имен БД.

### 3.3. Обнаружение Сервисов и Встроенный DNS (127.0.0.11)

В динамической среде IP-адреса контейнеров назначаются случайным образом из пула подсети (обычно 172.x.x.x). Хардкодить IP-адреса в конфигурации appsettings.json невозможно. Решением является **Service Discovery** через DNS.

Внутри каждого контейнера Docker файл /etc/resolv.conf настроен особым образом. В качестве nameserver там указан адрес 127.0.0.11.18

**Глубокий анализ механизма DNS:**

1. **Адрес-призрак:** 127.0.0.11 не является адресом реального сервиса внутри контейнера. Это адрес из зарезервированного диапазона loopback.
2. **Перехват через Iptables:** Docker не запускает DNS-сервер внутри каждого контейнера. Вместо этого, он использует правила iptables (в таблице NAT, цепочка OUTPUT), чтобы перехватывать весь трафик, идущий на 127.0.0.11:53 (UDP и TCP).
3. **DNAT и Forwarding:** Перехваченные пакеты подвергаются DNAT (Destination Network Address Translation) и перенаправляются на сокет встроенного DNS-сервера Docker Daemon (dockerd), который работает на хосте, но слушает на динамическом порту.18
4. **Логика Разрешения:**
   * Если запрос соответствует имени сервиса (например, calendar-db), DNS-сервер Docker возвращает внутренний IP-адрес соответствующего контейнера в этой сети.
   * Если запрос внешний (например, google.com), Docker пересылает его (Forwarding) на DNS-серверы, настроенные на хосте (например, провайдерские или 8.8.8.8).17

Влияние ndots на производительность:

Важный нюанс конфигурации Linux DNS — параметр options ndots:X в /etc/resolv.conf. В некоторых образах (особенно на базе Alpine или в Kubernetes) это значение может быть больше 1 (часто 5). Это означает, что если приложение запрашивает доменное имя, содержащее меньше точек, чем указано в ndots, резолвер сначала попытается разрешить его как локальное имя, добавляя поисковые суффиксы (search domains).

* *Пример:* Запрос к google.com (1 точка). Если ndots:5, резолвер сначала спросит google.com.calendar\_project\_default, получит NXDOMAIN, и только потом пойдет во внешний мир.
* *Инсайт:* Для высоконагруженных систем это создает паразитный DNS-трафик и латентность. В Docker Compose по умолчанию ndots обычно равен 0, но при использовании специфических базовых образов или Kubernetes об этом нужно помнить.20

### 3.4. Иллюзия Localhost и Loopback-Изоляция

Самая распространенная ошибка разработчиков.NET при переходе в Docker — попытка подключиться к базе данных по адресу localhost или 127.0.0.1.

* **На хосте:** localhost — это ваш компьютер.
* **В контейнере:** Из-за изоляции Network Namespace, 127.0.0.1 ссылается **исключительно на сам контейнер**.11

**Сценарий провала:**

1. В docker-compose.yml описаны сервисы calendar-api и calendar-db.
2. В appsettings.json сервиса API указана строка подключения: Host=localhost;Port=5432.
3. Контейнер API запускается. Приложение пытается открыть TCP-сокет на 127.0.0.1:5432 *внутри своего пространства имен*.
4. Так как база данных находится в *другом* контейнере (другом пространстве имен), на локальном порту 5432 ничего не слушает.
5. Результат: ConnectionRefused exception.

**Правильное решение:** Использовать имя сервиса как хостнейм: Host=calendar-db;Port=5432. Встроенный DNS (127.0.0.11) разрешит это имя в IP-адрес соседнего контейнера в мостовой сети.23

Подключение к Хосту (host.docker.internal):

Иногда возникает обратная задача: контейнеру нужно обратиться к сервису, запущенному на хост-машине (например, локальному отладчику или базе данных, не упакованной в Docker). Поскольку localhost изолирован, Docker предоставляет специальное DNS-имя host.docker.internal.24 Этот адрес разрешается во внутренний IP-адрес шлюза, через который контейнер может "выглянуть" на хост.

* *Предупреждение:* Использование host.docker.internal допустимо в среде разработки, но является плохой практикой в продакшене, так как нарушает принцип самодостаточности контейнера и привязывает его к конфигурации хоста.

## 4. Персистентность Данных: Стратегии Хранения и Механика I/O

Контейнеры по своей природе эфемерны и не сохраняют состояние (stateless). Это архитектурное требование позволяет легко масштабировать их и заменять. Однако база данных PostgreSQL по определению является stateful-сервисом (хранящим состояние).

### 4.1. Слоевая Файловая Система и Проблема OverlayFS

Файловая система контейнера построена на основе UnionFS (обычно драйвер overlay2). Она состоит из набора неизменяемых (read-only) слоев образа и верхнего тонкого записываемого слоя (writable layer).25

Механизм Copy-on-Write (CoW):

Когда приложение пытается изменить файл, который находится в нижнем (read-only) слое, драйвер overlay2 должен сначала найти этот файл, скопировать его целиком в верхний записываемый слой («copy-up operation»), и только затем применить изменения.

* **Влияние на БД:** Для баз данных, которые выполняют тысячи операций случайной записи в секунду (обновление страниц данных, запись в WAL-логи), накладные расходы на CoW создают катастрофическую латентность. Кроме того, данные в записываемом слое умирают вместе с контейнером.26

### 4.2. Docker Volumes: Золотой Стандарт

Docker Volumes (Тома) — это механизм, позволяющий обойти слоевую файловую систему.27

* **Архитектура:** Том — это директория на файловой системе хоста (обычно в /var/lib/docker/volumes/), которая монтируется в контейнер.
* **Bypassing OverlayFS:** Когда процесс внутри контейнера пишет в директорию тома, он пишет **напрямую** на диск хоста, минуя драйвер хранения Docker. Это обеспечивает нативную производительность ввода-вывода (Native I/O Performance).28
* **Управление:** Тома управляются Docker API. Они не зависят от жизненного цикла контейнера. Даже если удалить контейнер docker compose down, том останется. Для его удаления требуется явная команда docker volume rm или флаг -v.
* **Изоляция прав:** Тома решают проблемы с правами доступа (UID/GID mapping), так как Docker инициализирует права доступа к тому на основе данных внутри образа при первом запуске.29

### 4.3. Bind Mounts: Инструмент Разработчика

Bind Mounts (Привязки) позволяют отобразить произвольную папку хоста в контейнер.30

* **Синтаксис:** ./src:/app.
* **Сценарий:** Идеально подходят для проброса конфигурационных файлов (config.json) или исходного кода для "горячей перезагрузки" (Hot Reload) во время разработки.
* **Проблемы производительности:** На Linux bind mounts работают быстро. Однако на Windows и macOS (где Docker работает внутри микро-VM), каждое обращение к bind mount требует прохождения через трансляцию файловой системы (через gRPC, FUSE или VirtioFS). Это может быть в 10-50 раз медленнее нативных томов при операциях с множеством мелких файлов (например, node\_modules или obj папки.NET).31

Стратегическое решение для Недели 15:

Для базы данных calendar-db использование Named Volume является обязательным требованием. Это гарантирует сохранность данных и производительность. Использование Bind Mount для БД в среде Windows приведет к существенному замедлению тестов.

## 5. Продвинутое Управление Конфигурацией в ASP.NET Core

Перенос сервиса «Календарь» в Docker требует фундаментального изменения подхода к конфигурации. Мы не можем полагаться на ручное редактирование appsettings.json внутри контейнера. Мы должны следовать методологии **Twelve-Factor App**, которая предписывает хранить конфигурацию в переменных окружения (Environment Variables).

### 5.1. Иерархия Провайдеров Конфигурации

ASP.NET Core строит объект IConfiguration из множества источников. Понимание приоритетов (precedence) критически важно, чтобы избежать ситуаций, когда изменение настройки в Docker не влияет на приложение.33

Стандартный CreateBuilder загружает конфигурацию в следующем порядке (от низшего приоритета к высшему):

1. appsettings.json
2. appsettings.{Environment}.json (например, appsettings.Production.json)
3. **Environment Variables** (Ключевая точка интеграции с Docker)
4. Аргументы командной строки (Command Line Args)

**Инсайт:** Поскольку переменные окружения загружаются *после* JSON-файлов, они переопределяют (override) любые значения, заданные в файлах.34 Это позволяет образу контейнера быть неизменяемым (immutable) — он содержит дефолтные настройки в JSON, а специфические для среды настройки (пароли, хосты БД) впрыскиваются извне при запуске.

### 5.2. Конвенция Двойного Подчеркивания (\_\_)

Специфическая проблема возникает при работе с вложенными секциями конфигурации. В JSON структура иерархична:

JSON

{  
 "ConnectionStrings": {  
 "DefaultConnection": "Host=localhost..."  
 }  
}

Чтобы переопределить DefaultConnection через переменную окружения, интуитивно хочется использовать двоеточие: ConnectionStrings:DefaultConnection. Однако двоеточие (:) не является валидным символом для имен переменных окружения во многих оболочках (например, Bash) и запрещено в некоторых оркестраторах (Kubernetes до определенных версий).36

**Решение:** Провайдер конфигурации ASP.NET Core автоматически транслирует двойное подчеркивание (\_\_) в разделитель иерархии конфигурации.

* **Имя переменной:** ConnectionStrings\_\_DefaultConnection
* **Интерпретация:** Секция ConnectionStrings -> Ключ DefaultConnection.

Эта конвенция поддерживается на всех платформах и является стандартом для докеризации.NET приложений.38

### 5.3. Работа с Массивами в Конфигурации

Переопределение массивов через переменные окружения — задача нетривиальная. ASP.NET Core использует синтаксис с индексом (zero-based index).

Чтобы изменить первый элемент массива, нужно задать переменную:

* Logging\_\_LogLevel\_\_Default (для словаря)
* MyArray\_\_0\_\_Value (для списка)

**Второй порядок сложности:** Переменные окружения не могут *заменить* массив целиком или удалить элементы. Они могут только переопределить значения по конкретным индексам. Если в JSON массиве 3 элемента, а через ENV мы переопределили только нулевой, то элементы 1 и 2 останутся. Если требуется полная замена списка (например, список URL для CORS), часто эффективнее передавать их одной строкой с разделителем (например, через запятую) и парсить внутри приложения, либо использовать специализированные Configuration Providers.39

### 5.4. Управление Секретами: Docker Secrets

Передача паролей через блок environment в docker-compose.yml является плохой практикой безопасности, так как значения видны в docker inspect. Docker Compose поддерживает механизм **Secrets**.40

1. **Определение:** Секрет описывается в секции secrets файла compose.
2. **Монтирование:** Docker монтирует секрет как *файл* в директорию /run/secrets/{secret\_name} внутри контейнера.
3. **Потребление:** Приложение должно уметь читать этот файл.
   * **ASP.NET Core Интеграция:** По умолчанию конфигурация не читает файлы из /run/secrets. Для нативной поддержки необходимо добавить провайдер KeyPerFile:

C#  
config.AddKeyPerFile(directoryPath: "/run/secrets", optional: true);  
Это позволяет бесшовно интегрировать Docker Secrets в IConfiguration. Имя файла секрета становится ключом конфигурации (например, файл с именем ConnectionStrings\_\_DefaultConnection будет прочитан как строка подключения).40

## 6. Управление Порядком Запуска и Надежность (Reliability)

В распределенной системе сервисы не стартуют мгновенно. Процесс базы данных может запуститься, но ему потребуется время на инициализацию файловой системы, проигрывание WAL-логов и открытие сетевого сокета.

### 6.1. Состояние Гонки и Заблуждение depends\_on

Распространенная ошибка новичков — полагать, что директива depends\_on в Docker Compose гарантирует готовность сервиса.

* **Реальность:** Стандартный depends\_on ждет только запуска *контейнера* (состояния "Running"). Он не знает, готово ли приложение внутри принимать соединения.42
* **Следствие:** API запускается через 100мс после БД, пытается открыть соединение, получает отказ (сокет еще закрыт) и падает с исключением. Если не настроена политика перезапуска, стек выходит из строя.

### 6.2. Решение: Healthcheck и Condition service\_healthy

Для создания надежной последовательности запуска необходимо комбинировать healthcheck (проверку здоровья) с расширенным синтаксисом depends\_on.

Шаг 1: Определение Healthcheck для PostgreSQL

Мы должны научить Docker понимать, что значит "Postgres готов". Для этого используется утилита pg\_isready, входящая в образ Postgres.

YAML

healthcheck:  
 test:  
 interval: 5s  
 timeout: 5s  
 retries: 5

* **Механика:** Docker периодически выполняет эту команду внутри контейнера. Если она возвращает exit code 0, статус контейнера меняется с starting на healthy.44
* **Важный нюанс (TCP vs Socket):** По умолчанию pg\_isready может проверять локальный Unix-сокет. Однако приложение будет подключаться по TCP. Чтобы проверка была честной, рекомендуется указывать флаг хоста -h 127.0.0.1 или -h localhost, чтобы утилита проверила именно доступность TCP порта, на котором слушает БД. В противном случае возможна ситуация, когда сокет жив, а сетевой интерфейс еще не поднялся.45

Шаг 2: Блокировка запуска API

В сервисе API мы указываем условие:

YAML

depends\_on:  
 calendar-db:  
 condition: service\_healthy

Теперь Docker Compose физически не запустит контейнер API до тех пор, пока pg\_isready в контейнере БД не вернет успех. Это полностью устраняет гонку при старте.42

## 7. Практическая Реализация: Экосистема Календаря

На основе теоретического фундамента мы конструируем итоговый docker-compose.yml. Этот артефакт объединяет знания о сетях, томах, конфигурации и проверках здоровья.

### 7.1. Спецификация Архитектуры

* **Сервис calendar-db:**
  + Использует легковесный образ postgres:15-alpine (на базе musl libc).
  + Хранит данные в именованном томе pg-data (Bypassing OverlayFS).
  + Имеет настроенный healthcheck через pg\_isready.
* **Сервис calendar-api:**
  + Собирается из Dockerfile (Multi-stage build).
  + Подключается к БД через внутренний DNS (имя сервиса calendar-db).
  + Получает строку подключения через переменную окружения с двойным подчеркиванием.
  + Ждет статуса service\_healthy от БД.

### 7.2. Артефакт Docker Compose

YAML

version: '3.9' # Используем современную версию спецификации  
  
services:  
 # Сервис Базы Данных  
 calendar-db:  
 image: postgres:15-alpine  
 container\_name: calendar-db-1  
 # Переменные для инициализации Postgres (только при первом запуске)  
 environment:  
 POSTGRES\_USER: ${DB\_USER:-postgres} # Значение из.env или дефолт  
 POSTGRES\_PASSWORD: ${DB\_PASSWORD:-securepass}   
 POSTGRES\_DB: CalendarDb  
 # Персистентность: Именованный том для обхода OverlayFS  
 volumes:  
 - pg-data:/var/lib/postgresql/data  
 # Сетевая изоляция  
 networks:  
 - calendar-net  
 # Проверка готовности к приему TCP соединений  
 healthcheck:  
 test:  
 interval: 5s  
 timeout: 5s  
 retries: 5  
 start\_period: 10s # Даем время на холодный старт без штрафов  
 restart: unless-stopped  
  
 # Сервис API  
 calendar-api:  
 build:  
 context:.  
 dockerfile: Dockerfile  
 container\_name: calendar-api-1  
 # Инъекция конфигурации (переопределение appsettings.json)  
 environment:  
 # Использование \_\_ для вложенности. Имя хоста = имя сервиса (calendar-db)  
 ConnectionStrings\_\_DefaultConnection: "Host=calendar-db;Port=5432;Database=CalendarDb;Username=${DB\_USER:-postgres};Password=${DB\_PASSWORD:-securepass}"  
 ASPNETCORE\_ENVIRONMENT: Development  
 ASPNETCORE\_URLS: "http://+:8080"  
 # Проброс портов: Хост 5000 -> Контейнер 8080  
 ports:  
 - "5000:8080"  
 networks:  
 - calendar-net  
 # Гарантия порядка запуска  
 depends\_on:  
 calendar-db:  
 condition: service\_healthy  
 restart: on-failure  
  
# Глобальные определения ресурсов  
volumes:  
 pg-data: # Создается в /var/lib/docker/volumes/  
  
networks:  
 calendar-net:  
 driver: bridge # Создает изолированный мост с DNS-резолвером

### 7.3. Анализ Ключевых Решений

1. **Сеть calendar-net:** Мы явно создали пользовательский мост. Это активирует встроенный DNS Docker, позволяя calendar-api резолвить calendar-db в IP-адрес. Пинг по имени будет работать.16
2. **Том pg-data:** Использование именованного тома вместо bind mount (./data:/...) обеспечивает производительность на Windows/Mac и защищает от проблем с правами доступа к файлам на хосте.27
3. **Конфигурация:** Строка подключения формируется динамически. Код C# остается агностиком среды — он просто читает конфиг, не зная, что значения подставлены оркестратором.48
4. **Безопасность:** Использование .env файла (подразумевается синтаксисом ${VAR}) позволяет не хранить пароли в самом YAML файле. Файл .env должен быть добавлен в .gitignore.

## 8. Заключение и Перспективы

Второй день 15-й недели является критической точкой зрелости инженера. Освоение Docker Compose, понимание работы сетевых пространств имен и стратегий хранения данных трансформирует подход к разработке. Мы перешли от мышления в масштабах «одной машины» к мышлению в масштабах «распределенной топологии».

* **Сеть:** Знания сместились от простого открытия портов к пониманию механизмов изоляции, виртуальных мостов и DNS-маршрутизации трафика внутри кластера.
* **Хранение:** Понимание разницы между CoW-слоями и томами позволяет проектировать высокопроизводительные базы данных в контейнерах, избегая распространенных ловушек производительности.
* **Оркестрация:** Мы научились описывать самовосстанавливающиеся (self-healing) системы, где порядок запуска строго детерминирован проверками здоровья, а не случайными задержками (Thread.Sleep).

Этот фундамент является необходимым пререквизитом для следующих модулей. Понимание работы сети bridge и внутреннего DNS критично для настройки **RabbitMQ** (Неделя 16), где микросервисы должны находить брокер сообщений. Принципы healthchecks и изоляции ресурсов, изученные здесь, имеют прямое отображение в **Kubernetes** (Liveness/Readiness probes, Pods, PV/PVC), к которому готовит данный курс. Предоставленный docker-compose.yml — это не просто скрипт запуска, а переносимая, исполняемая спецификация архитектуры всей системы.

### Таблица 1: Сравнительный Анализ Стратегий Хранения в Docker

27

| **Характеристика** | **Docker Volumes (Тома)** | **Bind Mounts (Привязки)** | **tmpfs (В памяти)** |
| --- | --- | --- | --- |
| **Расположение** | Управляется Docker (/var/lib/docker/volumes) | Произвольный путь на хосте | ОЗУ хоста |
| **Производительность (Linux)** | Высокая (Нативная) | Высокая (Нативная) | Максимальная |
| **Производительность (Win/Mac)** | Высокая (Оптимизирована в VM) | Низкая (Накладные расходы на синхронизацию) | Максимальная |
| **Управление** | CLI (docker volume...), API | Команды ОС, права доступа хоста | Эфемерно |
| **Сценарий использования** | Базы данных, персистентные данные приложения | Конфиги, Исходный код (Dev), Горячая перезагрузка | Секреты, Временные файлы, Кэш |
| **Изоляция** | Высокая (Абстрагировано от структуры хоста) | Низкая (Зависит от структуры папок хоста) | Высокая |

### Таблица 2: Приоритет Конфигурации в ASP.NET Core (DefaultBuilder)

33

| **Приоритет** | **Источник** | **Описание** | **Стратегия Docker** |
| --- | --- | --- | --- |
| **1 (Высший)** | Аргументы командной строки | --Key=Value | Используется редко, для переопределения в Entrypoint |
| **2** | Переменные Окружения | Key=Value или Key\_\_SubKey=Value | **Основной метод для Docker Compose** |
| **3** | User Secrets | Локальное хранилище секретов | Монтируется как том только в Dev-среде |
| **4** | appsettings.Env.json | Специфичный для среды JSON | Включен в образ (Baked-in) |
| **5 (Низший)** | appsettings.json | Базовая конфигурация | Включен в образ (Baked-in) |

#### Источники

1. План обучения: Docker и Тестирование неделя 11
2. Building best practices - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/build/building/best-practices/>
3. Infrastructure as a Code : r/selfhosted - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/selfhosted/comments/18leeto/infrastructure_as_a_code/>
4. Specify a project name - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/project-name/>
5. Services | Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/reference/compose-file/services/>
6. How Compose works - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/intro/compose-application-model/>
7. Configure pre-defined environment variables in Docker Compose, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/environment-variables/envvars/>
8. docker compose - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/reference/cli/docker/compose/>
9. Use service profiles - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/profiles/>
10. Leveraging Docker Compose profiles | by Szabolcs Toth - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@kicsipixel/leveraging-docker-compose-profiles-01bb3b02c2fc>
11. Docker network and network namespaces in practice, дата последнего обращения: декабря 6, 2025, <https://learn-docker.it-sziget.hu/en/latest/pages/advanced/kernel-namespaces-network.html>
12. Building containers by hand using namespaces: The net namespace - Red Hat, дата последнего обращения: декабря 6, 2025, <https://www.redhat.com/en/blog/net-namespaces>
13. Network Namespaces Basics Explained in 15 Minutes - YouTube, дата последнего обращения: декабря 6, 2025, <https://www.youtube.com/watch?v=j_UUnlVC2Ss>
14. Basic understanding of network namespace - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/frfahim/network-namespace>
15. Container Networking Demystified: A Deep Dive into Linux Network Namespaces, дата последнего обращения: декабря 6, 2025, <https://dev.to/ajinkya_singh_2c02bd40423/container-networking-demystified-a-deep-dive-into-linux-network-namespaces-3gll>
16. Bridge network driver - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/network/drivers/bridge/>
17. Networking | Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/network/>
18. learn-docker/docs/docker-networking-deep-dive.md at master - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/KamranAzeem/learn-docker/blob/master/docs/docker-networking-deep-dive.md>
19. how does Docker Embedded DNS resolver work? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/41707573/how-does-docker-embedded-dns-resolver-work>
20. ndots option in hosts /etc/resolv.conf breaks container name resolution #41819 - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/moby/moby/issues/41819>
21. ndots:5 in /etc/resolv.conf in kubernetes pod | by jun sUTTHIPHONG SUWANICH | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@sutthiphong2005/ndots-5-in-etc-resolv-conf-in-kubernetes-pod-7a80a8ef48cb>
22. How do i conncet a service running on localhost inside the container? : r/docker - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/docker/comments/17w7q9e/how_do_i_conncet_a_service_running_on_localhost/>
23. Access docker container by host name instead of localhost:port - DevOps Stack Exchange, дата последнего обращения: декабря 6, 2025, <https://devops.stackexchange.com/questions/14764/access-docker-container-by-host-name-instead-of-localhostport>
24. From inside of a Docker container, how do I connect to the localhost of the machine? [closed] - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/24319662/from-inside-of-a-docker-container-how-do-i-connect-to-the-localhost-of-the-mach>
25. Role of Docker Storage Drivers In Container Performance - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@deepeshtripathi/docker-storage-drivers-in-and-out-11f188f9bfc6>
26. Storage drivers - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/drivers/>
27. Docker Volumes vs Bind Mounts: The Definitive Guide for Scalable Containers - Mihir Popat, дата последнего обращения: декабря 6, 2025, <https://mihirpopat.medium.com/docker-volumes-vs-bind-mounts-the-definitive-guide-for-scalable-containers-76c90eb4f248>
28. OverlayFS storage driver - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/drivers/overlayfs-driver/>
29. docker - volumes vs mount binds. what are the use cases? - Server Fault, дата последнего обращения: декабря 6, 2025, <https://serverfault.com/questions/996785/docker-volumes-vs-mount-binds-what-are-the-use-cases>
30. Bind mounts - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/bind-mounts/>
31. Just Discovered the Difference Between Volumes and Bind Mounts. Do I need to convert to volumes? : r/docker - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/docker/comments/mfh5aj/just_discovered_the_difference_between_volumes/>
32. Docker bind mount directory vs named volume performance comparison - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/64629569/docker-bind-mount-directory-vs-named-volume-performance-comparison>
33. ASP.NET Core Configuration, дата последнего обращения: декабря 6, 2025, <https://aspnetzero.com/blog/asp.net-core-configuration>
34. 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>
35. Configuration - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/configuration>
36. Overwriting Configuration using Environmental Variables in ASP.NET Core - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/thirddev/overriding-configuration-using-environmental-variables-in-asp-net-core-d38079475654>
37. Environment Variables: Naming Conventions and Runtime Access in C# .NET, дата последнего обращения: декабря 6, 2025, <https://dev.to/phougatv/environment-variables-naming-conventions-and-runtime-access-in-c-net-5adc>
38. Environment Variables in ASP .NET Core | by Egor Tarasov - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@vosarat1995/environment-variables-in-asp-net-core-d6b4ea6cff9f>
39. Override array settings in appsettings.json with those in appsettings.Production.json - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/52755027/override-array-settings-in-appsettings-json-with-those-in-appsettings-production>
40. Docker, AspNetCore, DB connection string best practices - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/49437336/docker-aspnetcore-db-connection-string-best-practices>
41. Secrets in Compose - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/use-secrets/>
42. Control startup and shutdown order with Compose - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/compose/how-tos/startup-order/>
43. Docker wait for postgresql to be running - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/35069027/docker-wait-for-postgresql-to-be-running>
44. How to wait for container X before starting Y using docker-compose healthcheck - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/peter-evans/docker-compose-healthcheck>
45. Make it easy to wait for connection · Issue #880 · docker-library/postgres - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/docker-library/postgres/issues/880>
46. Safe ways to specify postgres parameters for healthchecks in docker compose, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/65115627/safe-ways-to-specify-postgres-parameters-for-healthchecks-in-docker-compose>
47. Docker Compose Health Checks: An Easy-to-follow Guide - Last9, дата последнего обращения: декабря 6, 2025, <https://last9.io/blog/docker-compose-health-checks/>
48. 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>
49. Volumes - Docker Docs, дата последнего обращения: декабря 6, 2025, <https://docs.docker.com/engine/storage/volumes/>