# Отчет по теории и архитектуре распределенных систем: Трансформация ментальной модели Unity-разработчика в парадигму Backend-инженерии (День 1, Неделя 7)

## 1. Введение: Архитектурный сдвиг парадигмы

Переход от разработки интерактивных приложений реального времени (GameDev) на движке Unity к проектированию корпоративных бэкенд-систем на платформе.NET Core представляет собой не просто смену инструментария или изучение нового синтаксиса языка C#. Это фундаментальная трансформация инженерного мышления, требующая пересмотра базовых концепций управления временем, состоянием, памятью и взаимодействием компонентов. Учебный план седьмой недели, сфокусированный на создании сервиса календаря (Calendar Service), служит практическим полигоном для этого перехода, однако теоретический фундамент, закладываемый в первый день, является критически важным для успеха всей дальнейшей деятельности.

Данный документ представляет собой исчерпывающее руководство по теоретическим аспектам проектирования REST API, адаптированное специально для разработчиков с бэкграундом в игровой индустрии. Мы детально разберем онтологические различия между игровым циклом и жизненным циклом HTTP-запроса, проанализируем эволюцию сетевого взаимодействия от удаленного вызова процедур (RPC) до архитектурного стиля REST, и погрузимся в тонкости модели зрелости Ричардсона. Цель этого анализа — не просто объяснить «как» писать код, но сформировать глубокое понимание «почему» индустрия пришла к текущим стандартам и как избежать распространенных ментальных ловушек при смене специализации.

### 1.1 Фундаментальное различие: Игровой цикл (Game Loop) против Конвейера запросов (Request Pipeline)

В центре любой игровой системы, будь то Unity или Unreal Engine, лежит **Игровой Цикл (Game Loop)**. Это бесконечный цикл while(true), который выполняется десятки или сотни раз в секунду. В этой парадигме время непрерывно, а состояние (State) является локальным, мутабельным и персистентным в рамках сессии. Когда разработчик Unity пишет скрипт MonoBehaviour, он оперирует объектами, которые живут в оперативной памяти (Heap) от момента загрузки сцены до ее выгрузки.1 Переменная private float health; хранит свое значение между кадрами, и любой метод Update() имеет мгновенный доступ к этому значению. Взаимодействие между компонентами происходит синхронно, часто в рамках одного кадра, а сетевой код (например, через UDP) оптимизирован для минимизации задержек (Latency), где потеря пакета часто предпочтительнее задержки ради повторной отправки.1

В противоположность этому, бэкенд-разработка оперирует в дискретной, событийно-ориентированной среде. Здесь нет Update(), который вызывается каждый кадр. Вместо этого существует **Конвейер обработки запросов (Request Processing Pipeline)**. Жизнь приложения измеряется не кадрами, а транзакциями. Когда клиент (браузер, мобильное приложение или игровой клиент) отправляет HTTP-запрос, сервер создает изолированный контекст для его обработки. Все объекты, созданные в рамках этого запроса (контроллеры, сервисы с областью видимости Scoped), существуют ровно столько, сколько требуется для формирования ответа — обычно от нескольких миллисекунд до секунд.2 После отправки ответа 200 OK, все локальное состояние уничтожается сборщиком мусора.

Это различие диктует необходимость перехода от **Stateful** (сохраняющего состояние) мышления к **Stateless** (не сохраняющему состояние) архитектуре. В Unity вы можете хранить список подключенных игроков в статическом списке public static List<Player> Players. В бэкенде это является грубейшей архитектурной ошибкой. Если ваше приложение масштабируется горизонтально (запущено на 10 серверах), статический список на Сервере А не будет знать о пользователях, подключенных к Серверу Б. Понимание этой эфемерности памяти сервера является первым шагом к проектированию надежных REST API.

## 2. Управление состоянием: Stateful против Stateless

Понимание того, где и как хранится состояние системы, является водоразделом между архитектурой игровых серверов и веб-сервисов. В контексте разработки Calendar Service, который мы проектируем, выбор стратегии управления состоянием определяет масштабируемость и отказоустойчивость системы.

### 2.1 Stateful архитектура: Наследие игровых миров

В классической сетевой игре или MMO (Massively Multiplayer Online game) сервер является "властелином мира". Он загружает всю карту и всех персонажей в оперативную память. Это **Stateful** архитектура.3

* **Механизм:** Когда клиент подключается, устанавливается постоянное соединение (TCP или UDP сокет). Сервер выделяет блок памяти для сессии этого клиента. Все изменения (движение, стрельба) применяются к этому блоку памяти мгновенно.
* **Преимущества:** Экстремально низкая задержка. Данные находятся в "горячей" памяти, нет необходимости тратить время на обращения к базе данных для каждого действия. Это критично для игр, где задержка выше 50-100 мс становится заметной.3
* **Недостатки:** Сложность масштабирования. Если сервер переполнен, вы не можете просто добавить еще один сервер и перекинуть туда половину игроков "на лету", так как состояние их мира находится в памяти первого сервера. Требуются сложные механизмы миграции или шардирования мира. Отказоустойчивость также страдает: если сервер падает, все несохраненные в БД данные сессий теряются.5

Для Unity-разработчика эта модель естественна. NetworkManager в Unity хранит состояние всех NetworkIdentity объектов. Переход на бэкенд требует отказа от этой интуиции.

### 2.2 Stateless архитектура: Основа REST

REST (Representational State Transfer) фундаментально опирается на принцип **Stateless** (отсутствие состояния). Согласно диссертации Роя Филдинга, каждое взаимодействие между клиентом и сервером должно быть независимым.7 Сервер не должен хранить никакой информации о состоянии сессии клиента между двумя запросами.5

* **Механизм:** Клиент отправляет запрос GET /events. В запросе содержится вся необходимая информация для авторизации (например, токен JWT) и параметры выборки. Сервер обрабатывает запрос, обращается к базе данных (внешнему хранилищу состояния), формирует ответ и "забывает" о клиенте.6 Следующий запрос от того же клиента будет обработан как совершенно новый, возможно, даже другим экземпляром сервера.
* **Преимущества:**
  + **Масштабируемость (Scalability):** Поскольку сервер не хранит контекст сессии, любой сервер в кластере может обработать любой запрос. Это позволяет легко добавлять новые мощности (горизонтальное масштабирование).6
  + **Надежность (Reliability):** Если один экземпляр сервиса падает, балансировщик нагрузки просто перенаправляет следующий запрос на другой экземпляр. Пользователь этого даже не заметит, так как состояние хранится в надежной базе данных, а не в памяти упавшего процесса.5
  + **Кэширование:** Отсутствие зависимости от скрытого состояния сессии делает ответы предсказуемыми и кэшируемыми.9
* **Недостатки:** Оверхед (накладные расходы). В каждом запросе приходится передавать служебную информацию (токены, ID), и сервер вынужден часто обращаться к БД, что медленнее, чем чтение из RAM.10

Для нашего сервиса календаря мы выбираем **Stateless** архитектуру. Это стандарт индустрии для веб-сервисов. Календарь не требует миллисекундной реакции, как шутер от первого лица, но требует надежного хранения данных и доступности 24/7, что обеспечивается именно stateless подходом.1

### 2.3 Сравнительный анализ архитектур

В таблице ниже представлено детальное сравнение подходов, помогающее Unity-разработчику соотнести свой опыт с новыми требованиями.

| **Характеристика** | **Stateful (Игровой сервер)** | **Stateless (REST API)** |
| --- | --- | --- |
| **Хранение сессии** | В оперативной памяти сервера (RAM) | На клиенте (Токен) или во внешней БД (Redis) |
| **Зависимость запросов** | Высокая. Контекст сохраняется. | Нулевая. Каждый запрос изолирован. |
| **Масштабирование** | Вертикальное (мощнее CPU) или сложное шардирование | Горизонтальное (больше серверов) |
| **Отказоустойчивость** | Низкая (потеря памяти = потеря прогресса) | Высокая (потеря сервера = ретрай запроса) |
| **Пример в Unity** | NetworkManager, Photon Room | UnityWebRequest, BestHTTP |
| **Пример в.NET** | SignalR (WebSocket), Orleans | ASP.NET Core Web API |

Источники данных для таблицы:.5

## 3. Эволюция коммуникации: От RPC к REST

Понимание истории развития сетевых протоколов помогает осознать, почему REST стал доминирующим стилем. Многие начинающие бэкенд-разработчики интуитивно изобретают RPC (Remote Procedure Call), думая, что пишут REST. Это классическая ошибка, которую мы должны предотвратить на этапе проектирования.

### 3.1 Remote Procedure Call (RPC): Императивный подход

RPC — это старейшая парадигма распределенных вычислений, зародившаяся еще в 70-х годах.7 Идея RPC заключается в том, чтобы скрыть (абстрагировать) сеть. Разработчик вызывает функцию CreateEvent(data) в своем коде, а система скрыто отправляет запрос на удаленный сервер, который выполняет эту функцию и возвращает результат.7

**Философия RPC:**

* **Ориентация на Действия (Verbs):** API представляется как набор функций (процедур). Мыслительный процесс разработчика: "Я хочу *сделать* что-то".
* **Тесная связность (Tight Coupling):** Клиент и сервер должны точно знать сигнатуры методов друг друга. Если сервер изменит название функции или порядок аргументов, клиент сломается.12
* **Туннелирование:** Обычно использует один URL (эндпоинт) и один HTTP метод (чаще всего POST) для всех операций. Сама команда передается внутри тела запроса (в JSON или XML).7

Пример RPC-мышления в веб-разработке:

Допустим, мы хотим удалить событие из календаря. В стиле RPC мы бы создали такой эндпоинт:

* **URL:** /api/calendarService
* **Method:** POST
* Body: { "method": "DeleteEvent", "eventId": 123 }  
  Или, в стиле "RPC через URL" (частая ошибка новичков):
* GET /api/deleteEvent?id=123

Это подход Unity-разработчика, отправляющего сетевое сообщение (RPC) TargetRPC или Command в Mirror/UNET. Вы отправляете команду на выполнение действия.

Современный RPC (gRPC):

Стоит отметить, что RPC не умер. Современные реализации, такие как gRPC от Google, используют Protobuf для бинарной сериализации и HTTP/2 для транспорта. Они экстремально эффективны и типизированы, что делает их идеальными для внутреннего общения между микросервисами (Service-to-Service).14 Однако для пулбичного API, которым является наш Calendar Service, RPC часто избыточно сложен для интеграции и менее гибок.

### 3.2 Representational State Transfer (REST): Декларативный подход

REST — это архитектурный стиль, описанный Роем Филдингом в 2000 году, как ответ на сложность и громоздкость существующих RPC/SOAP решений.7 REST меняет фокус с *действий* на *ресурсы*.

**Философия REST:**

* **Ориентация на Ресурсы (Nouns):** Мы не вызываем функции, мы манипулируем объектами (существительными). В системе есть "События" (Events), "Пользователи" (Users). У каждого объекта есть уникальный адрес (URI).7
* **Единообразный интерфейс (Uniform Interface):** Вместо придумывания названий функций (Create, Add, Insert, New), REST использует стандартизированные глаголы протокола HTTP: GET, POST, PUT, DELETE, PATCH.12
* **Самодокументируемость:** Зная, что /events — это коллекция событий, любой разработчик поймет, что DELETE /events/123 удалит событие, а GET /events/123 вернет его данные. Не нужно читать документацию, чтобы узнать название метода удаления (RemoveEvent? DeleteEntry? KillItem?).13

Тот же пример в REST:

Чтобы удалить событие:

* **URL:** /api/events/123
* **Method:** DELETE
* **Body:** (Пусто)

Здесь действие (DELETE) отделено от цели (/api/events/123). Это делает систему более гибкой и понятной.

### 3.3 Сравнение RPC и REST для Calendar Service

| **Критерий** | **RPC (Remote Procedure Call)** | **REST (Representational State Transfer)** |
| --- | --- | --- |
| **Центральная концепция** | Функция / Процедура (Глагол) | Ресурс / Сущность (Существительное) |
| **Пример URL** | /api/deleteEvent | /api/events/123 |
| **HTTP Методы** | В основном POST | GET, POST, PUT, DELETE, PATCH |
| **Кэширование** | Сложное (все запросы уникальны) | Простое (GET запросы кэшируются на уровне CDN/браузера) |
| **Связность** | Жесткая (зависимость от сигнатур методов) | Слабая (зависимость от структуры ресурса) |
| **Аналогия с Unity** | Вызов метода component.DoSomething() | Изменение поля в ScriptableObject |

Источники:.7

Для нашего учебного проекта мы выбираем **REST**, так как это основной стиль построения публичных веб-API в экосистеме.NET и Enterprise.1

## 4. Модель Зрелости Ричардсона (Richardson Maturity Model)

Чтобы помочь разработчикам оценить, насколько их API соответствует принципам REST, Леонард Ричардсон предложил модель зрелости, состоящую из четырех уровней (от 0 до 3). Разбор этой модели является обязательной частью теории первого дня, так как она дает четкую дорожную карту проектирования.17

### 4.1 Уровень 0: Болото POX (The Swamp of POX)

Самый примитивный уровень. Здесь HTTP используется просто как транспортный туннель для передачи данных. Аббревиатура POX означает "Plain Old XML" (старый добрый XML), хотя сегодня это чаще JSON.18

* **Характеристики:**
  + Единственный URL для всего API (например, /api/service).
  + Использование единственного метода HTTP (обычно POST) для всех операций.
  + Семантика операции (что именно сделать) зашита внутри тела запроса.
* Пример:  
  Запрос на получение события:  
  HTTP  
  POST /api/endpoint HTTP/1.1  
  Host: calendar.com  
  Content-Type: application/json  
    
  { "action": "getEvent", "id": 123 }  
    
  Запрос на создание события:  
  HTTP  
  POST /api/endpoint HTTP/1.1

...

{ "action": "createEvent", "title": "Meeting" }

```

* **Анализ:** Это чистый RPC. С точки зрения сетевой инфраструктуры (прокси, кэши) эти два запроса идентичны. Это делает невозможным автоматическое кэширование.20

### 4.2 Уровень 1: Ресурсы (Resources)

Первый шаг к REST — введение понятия **Ресурсов**. Мы начинаем использовать разные URL для разных объектов, применяя принцип "Разделяй и Властвуй" к сложности системы.17

* **Характеристики:**
  + Множество URL (например, /api/events, /api/users).
  + Каждый объект имеет свой уникальный идентификатор в адресе (/api/events/123).
  + Все еще используется один HTTP метод (чаще всего POST) для всех действий.
* Пример:  
  Получение события: POST /api/events/123 (тело может быть пустым или содержать параметры).  
  Удаление события: POST /api/events/123/delete.
* **Прогресс:** Мы структурировали адресное пространство. Теперь мы знаем, *с чем* мы взаимодействуем, но не *как* стандартным образом.17

### 4.3 Уровень 2: HTTP Глаголы (Цель Недели 7)

Это уровень, который принято называть "RESTful" в современной индустрии. Здесь мы начинаем использовать протокол HTTP так, как он был задуман, применяя стандартные глаголы для семантического описания действий.18

* **Характеристики:**
  + **GET:** Для получения данных. Этот метод должен быть **безопасным** (Safe) — его вызов не меняет состояние сервера, и **идемпотентным**.
  + **POST:** Для создания новых ресурсов. Обычно не идемпотентен (два вызова создадут два объекта).
  + **PUT:** Для полной замены ресурса. Идемпотентен (сколько бы раз ни вызвали, результат один — объект обновлен).
  + **DELETE:** Для удаления. Идемпотентен.
  + **Статус-коды:** Использование кодов ответа для коммуникации результата (200 OK, 201 Created, 204 No Content, 404 Not Found, 400 Bad Request) вместо возврата 200 OK с текстом ошибки внутри JSON.18
* Пример:  
  Получение: GET /api/events/123 -> Возвращает 200 OK и JSON.  
  Создание: POST /api/events -> Возвращает 201 Created и заголовок Location.
* **Преимущества:** Инфраструктура интернета (CDN, браузеры) теперь может помогать нам. Браузер знает, что GET можно кэшировать, а POST нельзя повторять без предупреждения пользователя.17 Именно на этот уровень мы ориентируемся в нашем учебном плане.1

### 4.4 Уровень 3: HATEOAS (Hypermedia As The Engine Of Application State)

Высший пилотаж REST, который добавляет возможность **обнаружения (Discoverability)**. Ответ API содержит не только данные, но и ссылки (Hypermedia) на возможные следующие действия.17

* **Концепция:** Клиент не должен знать структуру URL заранее. Он должен знать только корневой URL, а дальше перемещаться по ссылкам, как человек по веб-сайту.
* Пример:  
  Ответ на GET /api/events/123:  
  JSON  
  {  
   "id": 123,  
   "title": "Meeting",  
   "links":  
  }
* **Аналогия с Unity:** Это похоже на динамический UI, где кнопка "Удалить" рендерится только если сервер прислал соответствующую ссылку в списке доступных действий.
* **Реальность:** Хотя это идеал REST, на практике HATEOAS внедряют редко из-за сложности реализации на клиенте. Для нашего курса мы остановимся на **Уровне 2**, так как он покрывает 99% потребностей рынка.1

## 5. Проектирование Контрактов: API-First против Code-First

Прежде чем писать код контроллеров, мы должны определить "Контракт" — соглашение о том, как API будет выглядеть снаружи. Существует два основных подхода к этому процессу.

### 5.1 Code-First (Сначала код)

В этом подходе разработчик сначала пишет реализацию на C# (Model, Controller), а затем использует инструменты типа Swashbuckle/NSwag для автоматической генерации документации (Swagger/OpenAPI Specification).21

* **Плюсы:**
  + **Скорость:** Для одного разработчика это самый быстрый путь. Вы пишете код, запускаете, и документация готова.23
  + **Синхронизация:** Документация никогда не устаревает, так как она генерируется из кода.
* **Минусы:**
  + **Утечка абстракций:** Часто внутренняя структура БД (Entity) просачивается в API, если разработчик ленится создавать отдельные DTO.
  + **Блокировка фронтенда:** Фронтенд-разработчик не может начать работу, пока бэкенд не напишет код.24

### 5.2 API-First / Design-First (Сначала контракт)

Здесь сначала пишется спецификация API (в формате YAML/JSON OpenAPI), обсуждается и утверждается. Только после этого начинается написание кода. Спецификация служит "чертежом".25

* **Плюсы:**
  + **Параллельная разработка:** Фронтенд может генерировать мок-серверы и писать UI, пока бэкенд пишет логику.24
  + **Качество дизайна:** Фокус на интерфейсе, а не на реализации. Это заставляет думать о потребителе API.25
* **Минусы:**
  + **Трудоемкость:** Поддержка YAML файлов вручную может быть утомительной. Требуется дисциплина для синхронизации кода и спецификации.21

### 5.3 Стратегия для Недели 7

В учебном плане мы используем гибридный подход: Design-Thinking with Code-First Implementation.

В День 1 мы проектируем API на бумаге или в Swagger Editor (как в подходе API-First), определяя ресурсы, URL и типы данных. Но реализовывать мы будем через атрибуты C# (Code-First), чтобы использовать мощь типизации.NET. Это развивает навык проектирования без чрезмерного усложнения инструментария на старте.1

## 6. Анатомия Ресурса и Лучшие Практики Проектирования

При проектировании ресурса Event для календаря необходимо следовать строгим правилам, выработанным индустрией.

### 6.1 Именование: Существительные и Множественное число

* **Правило 1: Только существительные.** URL должен называть ресурс, а не действие.
  + *Плохо:* /getEvents, /createEvent, /updateEvent.
  + *Хорошо:* /events.
* **Правило 2: Множественное число для коллекций.** Используйте /events (события), а не /event. Это логично, так как по адресу /events мы ожидаем получить *список*, а конкретный элемент доступен по ID внутри коллекции: /events/123.16
* **Правило 3: Kebab-case в URL.** Для составных слов в URL принято использовать дефис (kebab-case), так как он лучше читается и индексируется поисковиками, чем camelCase или underscores.
  + *Пример:* /api/v1/calendar-events вместо /api/v1/calendarEvents.29

### 6.2 Версионирование API

В отличие от игры, где мы можем заставить игрока обновить клиент, веб-API могут использоваться старыми клиентами годами. Изменения не должны ломать их работу.

* **URI Versioning:** Включение версии в путь URL (например, /api/v1/events).
  + *Плюсы:* Наглядно, просто, легко кэшируется.
  + *Минусы:* "Загрязняет" URL.30
* Мы выбираем **URI Versioning** как самый понятный и распространенный стандарт для начала.1

### 6.3 Работа со Временем: 5 Законов API Dates

Для Unity-разработчика время часто ограничивается Time.deltaTime или локальным временем компьютера. В распределенной системе время — это минное поле.

**Законы времени в API:**

1. **Всегда используйте UTC.** Никогда не храните и не передавайте локальное время сервера или клиента. Сервер может быть в Ирландии, БД в США, а пользователь в Японии. UTC — единственная константа.24
2. **Используйте ISO 8601.** Формат строки для передачи времени должен быть стандартизирован: 2023-10-05T14:30:00Z. Буква Z в конце явно указывает на UTC (Zulu time).33
3. **Типы данных.NET:** Используйте DateTimeOffset вместо DateTime. DateTime в.NET амбивалентен (непонятно, это UTC или Local), тогда как DateTimeOffset хранит смещение явно, предотвращая ошибки конвертации.1
4. **Длительность (Duration):** Не передавайте длительность в секундах (int). Используйте формат ISO 8601 Duration: PT1H30M (Period Time 1 Hour 30 Minutes). Это защищает от нюансов, таких как секунды координации или смена длины дня.33

### 6.4 Идентификаторы: UUID против Int

В локальных БД часто используют автоинкрементные int (1, 2, 3...). В распределенных системах это опасно:

* **Безопасность:** Зная ID=100, хакер может попробовать ID=99 (Insecure Direct Object Reference).
* **Конфликты:** При слиянии баз данных или генерации ID на клиенте (офлайн-режим) int вызовет коллизии.

Мы будем использовать **UUID (Universally Unique Identifier)**, в C# это тип Guid.

* *Пример:* 550e8400-e29b-41d4-a716-446655440000.
* Это позволяет генерировать ID где угодно (даже на клиенте перед отправкой), гарантируя уникальность во вселенной.37

## 7. Практическое задание: Дизайн API для Calendar Service

Опираясь на теорию, мы формулируем спецификацию для нашего сервиса. Это и есть результат работы первого дня.

Ресурс: Event (Событие календаря)

Базовый URL: /api/v1/events

### Модель данных (DTO)

При проектировании DTO мы избегаем Mass Assignment, не включая служебные поля БД в запрос на создание.

**CreateEventRequest (Запрос на создание):**

JSON

{  
 "title": "Daily Standup",  
 "description": "Sync with team",  
 "startTime": "2023-10-25T09:00:00Z", // ISO 8601 UTC  
 "duration": "PT15M", // 15 минут  
 "location": "Zoom"  
}

**EventResponse (Ответ сервера):**

JSON

{  
 "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6", // GUID  
 "title": "Daily Standup",  
 "description": "Sync with team",  
 "startTime": "2023-10-25T09:00:00Z",  
 "endTime": "2023-10-25T09:15:00Z", // Вычисляемое поле для удобства клиента  
 "duration": "PT15M",  
 "location": "Zoom"  
}

### Спецификация Эндпоинтов (Контракт)

| **Метод** | **URI** | **Описание** | **Статус успеха** | **Идемпотентность** |
| --- | --- | --- | --- | --- |
| **POST** | /api/v1/events | Создать новое событие. Сервер генерирует ID. | 201 Created + Заголовок Location | Нет |
| **GET** | /api/v1/events | Получить список событий. Поддержка фильтров ?from=...&to=... | 200 OK | Да |
| **GET** | /api/v1/events/{id} | Получить детальную информацию об одном событии. | 200 OK или 404 Not Found | Да |
| **PUT** | /api/v1/events/{id} | Полностью перезаписать событие (замена всех полей). | 200 OK или 204 No Content | Да |
| **DELETE** | /api/v1/events/{id} | Удалить событие. | 204 No Content | Да |

*Примечание об идемпотентности:* DELETE идемпотентен, потому что результат (события больше нет) одинаков после 1 или 100 вызовов. Даже если второй вызов вернет 404, состояние системы не изменится.1

## 8. Заключение

Первый день седьмой недели закладывает теоретический фундамент для всей дальнейшей работы. Мы деконструировали привычный Unity-разработчику мир игрового цикла и stateful-памяти, заменив его на дискретный мир HTTP-запросов и stateless-архитектуры. Понимание различий между RPC и REST, а также следование модели зрелости Ричардсона, позволяет проектировать системы, которые не просто "работают", но являются масштабируемыми, поддерживаемыми и интегрируемыми в современную экосистему веб-технологий. Сформированный контракт API служит картой, по которой в следующие дни будет написан код на C#.

Документ подготовлен на основе материалов учебного плана Недели 7 1 и анализа современных практик разработки распределенных систем.

#### Источники

1. План обучения REST API для бэкенда неделя 7
2. ASP.NET Core Application Lifecycle - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/39800616/asp-net-core-application-lifecycle>
3. Embracing unconventionality: Stateful vs Stateless architectures | by Gameberrylabs Blogs, дата последнего обращения: декабря 5, 2025, <https://medium.com/@gamberrylabs/embracing-unconventionality-stateful-vs-stateless-architectures-a226901c940a>
4. Stateful v. Stateless Design: Key Considerations for Game Developers - Metaplay, дата последнего обращения: декабря 5, 2025, <https://metaplay.io/blog/stateful-vs-stateless-design-key-considerations-for-game-developers>
5. Stateful vs stateless applications - Red Hat, дата последнего обращения: декабря 5, 2025, <https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless>
6. Stateful vs. Stateless Architecture - GeeksforGeeks, дата последнего обращения: декабря 5, 2025, <https://www.geeksforgeeks.org/system-design/stateful-vs-stateless-architecture/>
7. RPC vs REST - Difference Between API Architectures - AWS, дата последнего обращения: декабря 5, 2025, <https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/>
8. Stateful vs Stateless: Choosing the Right Backend Architecture - F22 Labs, дата последнего обращения: декабря 5, 2025, <https://www.f22labs.com/blogs/stateful-vs-stateless-choosing-the-right-backend-architecture/>
9. Understanding RPC Vs REST For HTTP APIs - Smashing Magazine, дата последнего обращения: декабря 5, 2025, <https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/>
10. Stateful vs. Stateless Web App Design - DreamFactory Blog, дата последнего обращения: декабря 5, 2025, <https://blog.dreamfactory.com/stateful-vs-stateless-web-app-design>
11. RPC vs. REST: A Comprehensive Comparison | by Utkarsh Shukla | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@utkarshshukla.author/rpc-vs-rest-a-comprehensive-comparison-88d0c7e13687>
12. RPC vs REST: Main Differences and which one to choose! - Ankr, дата последнего обращения: декабря 5, 2025, <https://www.ankr.com/blog/rpc-vs-rest/>
13. Understanding the difference between RPC and REST for web APIs - Merge.dev, дата последнего обращения: декабря 5, 2025, <https://www.merge.dev/blog/understanding-the-difference-between-rpc-and-rest-for-web-apis>
14. RPC vs REST - DEV Community, дата последнего обращения: декабря 5, 2025, <https://dev.to/dhanush___b/rpc-vs-rest-a-comparative-analysis-for-advanced-systems-architecture-2d8d>
15. REST vs. RPC: Choosing the Right Architecture for Your APIs | by Kumud Sharma | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@kumud.sharma.0206/rest-vs-rpc-choosing-the-right-architecture-for-your-apis-560019633137>
16. Best practices for RESTful web API design - Azure - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>
17. Richardson Maturity Model - Martin Fowler, дата последнего обращения: декабря 5, 2025, <https://martinfowler.com/articles/richardsonMaturityModel.html>
18. Richardson Maturity Model and HATEOAS: Understanding the Evolution of RESTful APIs | by Anita Liberatore | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@liberatoreanita/richardson-maturity-model-and-hateoas-understanding-the-evolution-of-restful-apis-53702173a138>
19. Richardson Maturity Model - Wikipedia, дата последнего обращения: декабря 5, 2025, <https://en.wikipedia.org/wiki/Richardson_Maturity_Model>
20. Richardson Maturity Model - REST API Tutorial, дата последнего обращения: декабря 5, 2025, <https://restfulapi.net/richardson-maturity-model/>
21. Current Best Practices for Go HTTP API Design: Code-First, Schema-First, or Protobuf? Looking for the sweet spot : r/golang - Reddit, дата последнего обращения: декабря 5, 2025, <https://www.reddit.com/r/golang/comments/1pd6g9k/current_best_practices_for_go_http_api_design/>
22. Code-First vs. Design-First: Eliminate Friction with API Exploration - Swagger, дата последнего обращения: декабря 5, 2025, <https://swagger.io/blog/code-first-vs-design-first-api/>
23. Api-First vs Code-First: Which One Should You Choose? - Manuel Sánchez, дата последнего обращения: декабря 5, 2025, <https://www.manuelsanchezdev.com/blog/api-first-vs-code-first>
24. API-First, API Design-First, or Code-First: Which Should You Choose | Stoplight, дата последнего обращения: декабря 5, 2025, <https://blog.stoplight.io/api-first-api-design-first-or-code-first-which-should-you-choose>
25. API-first vs Code-first: Choosing the Right Approach for Your Project | by Tioka Chiu, дата последнего обращения: декабря 5, 2025, <https://medium.com/@tiokachiu/api-first-vs-code-first-choosing-the-right-approach-for-your-project-868443e73052>
26. A Developer's Guide to API Design-First, дата последнего обращения: декабря 5, 2025, <https://apisyouwonthate.com/blog/a-developers-guide-to-api-design-first/>
27. API Design-First vs Code First - APIs You Won't Hate, дата последнего обращения: декабря 5, 2025, <https://apisyouwonthate.com/blog/api-design-first-vs-code-first/>
28. How to Choose the Right REST API Naming Conventions | Zuplo Learning Center, дата последнего обращения: декабря 5, 2025, <https://zuplo.com/learning-center/how-to-choose-the-right-rest-api-naming-conventions>
29. REST API URI Naming Conventions and Best Practices, дата последнего обращения: декабря 5, 2025, <https://restfulapi.net/resource-naming/>
30. API versioning: URL vs Header vs Media Type versioning - Lonti, дата последнего обращения: декабря 5, 2025, <https://www.lonti.com/blog/api-versioning-url-vs-header-vs-media-type-versioning>
31. API Versioning Strategies: Best Practices Guide - Daily.dev, дата последнего обращения: декабря 5, 2025, <https://daily.dev/blog/api-versioning-strategies-best-practices-guide>
32. REST API Versioning: How to Version a REST API?, дата последнего обращения: декабря 5, 2025, <https://restfulapi.net/versioning/>
33. OpenAPI String Type Format - APIMatic, дата последнего обращения: декабря 5, 2025, <https://www.apimatic.io/openapi/string-type-format>
34. Dates in OpenAPI Files | Baeldung, дата последнего обращения: декабря 5, 2025, <https://www.baeldung.com/openapi-dates>
35. DateTimeOffset.Add(TimeSpan) Method (System) - Microsoft Learn, дата последнего обращения: декабря 5, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.datetimeoffset.add?view=net-10.0>
36. What is a good format for Duration in JSON? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/50471779/what-is-a-good-format-for-duration-in-json>
37. Is there any difference between a GUID and a UUID? - Stack Overflow, дата последнего обращения: декабря 5, 2025, <https://stackoverflow.com/questions/246930/is-there-any-difference-between-a-guid-and-a-uuid>
38. UUID and GUID. GUID (Globally Unique Identifier) and… | by Shubham Padhi | Medium, дата последнего обращения: декабря 5, 2025, <https://medium.com/@shubhampadhi404/uuid-and-a-guid-b484f5b05006>