Неделя 1. Лекция 1. ASP.NET: как устроен сервис задач

Часть 1 из 2.

После лекции: вы понимаете, как устроены современные веб-сервисы и где в них backend, что такое REST API и как к нему обращаться, как устроен наш проект — сервис задач, как через него проходит запрос и куда добавлять новую функцию. Весь курс мы развиваем именно этот проект. **Инструменты:** Visual Studio 2026, .NET 10 SDK, Postman, браузер.

1. Как устроены современные веб-сервисы

Любое веб-приложение — интернет-магазин, банк, трекер задач — делится на части:

Часть	Что это	Где работает	Из чего сделано
Frontend (клиент)	То, что видит пользователь: страницы, кнопки, формы	В браузере или на телефоне	HTML, CSS, JavaScript (React, Angular, Vue), мобильные приложения
Backend (сервер)	Логика и данные: проверить, сохранить, посчитать, отдать	На сервере или в контейнере	C# (ASP.NET), Java, Go, Python, Node.js
База данных	Надёжное хранение данных	Отдельный сервер или контейнер	PostgreSQL, MySQL, MongoDB

Frontend и backend общаются по **HTTP**: клиент отправляет **запрос**, backend отвечает данными в формате **JSON**. Набор запросов, которые понимает backend, называется **API** (Application Programming Interface) — «меню» сервиса для программ.

```
flowchart LR
  subgraph K["Клиенты"]
    W["Сайт в браузере"]
    M["Мобильное приложение"]
    P["Postman, Swagger<br/>разработчик и тестировщик"]
  end
  K -->|"HTTP-запрос<br/>GET /api/tasks"| B["Backend<br/>сервис задач"]
  B -->|"ответ: 200 + JSON"| K
  B -->|SQL| D[("PostgreSQL")]
```

Один backend обслуживает сразу всех клиентов: сайт, мобильное приложение, другие сервисы. Поэтому backend не рисует страницы, а только отдаёт данные.

Большие системы состоят из многих backend-сервисов: заказы, оплата, уведомления, поиск. У каждого своя задача и своя БД, друг с другом они общаются тоже по HTTP. Каждый сервис собирают, тестируют, выкладывают и мониторят отдельно, часто по нескольку раз в день. Делать это вручную невозможно — этим и занимается **DevOps**: контейнеры, CI/CD, мониторинг. В курсе мы проходим весь этот путь на одном сервисе.

2. Проект курса и как он будет расти

Сервис задач (DemoApi) — backend-сервис с REST API: проекты, в проектах задачи со статусами `Todo → InProgress → Done`. Своего интерфейса у него нет: он принимает HTTP-запросы и отвечает JSON.

За курс проект обрастает всем, что нужно реальному сервису:

Тема	Что появится у сервиса
1	Запускаем в Visual Studio, изучаем код, добавляем свою функцию
2	Dockerfile: сервис работает в контейнере на Docker Desktop
3	PostgreSQL в контейнере, тома, Docker Compose
4	Репозиторий на GitHub, работа через Pull Request
5	CI/CD: тесты и автоматический деплой
6	Разбор сбоев: логи, рестарты, лимиты
7	Мониторинг: метрики, дашборды, алерты
8	Итоговый инцидент: от плохого релиза до постмортема

3. HTTP за 5 минут

Всё общение с сервисом — это HTTP-запросы и ответы.

```
PATCH /api/tasks/5/status HTTP/1.1          ← метод и путь
Host: localhost:8080
Content-Type: application/json               ← заголовки

{ "status": "InProgress" }                   ← тело

HTTP/1.1 200 OK                              ← код ответа
Content-Type: application/json

{ "id": 5, "title": "Написать Dockerfile", "status": "InProgress", ... }
```

Методы выражают действие над ресурсом:

Метод	Смысл	Пример в нашем API
`GET`	Прочитать	`GET /api/tasks?projectId=1` — задачи проекта
`POST`	Создать	`POST /api/projects` — новый проект
`PUT`	Заменить целиком	`PUT /api/tasks/5` — новое название, описание, срок
`PATCH`	Изменить часть	`PATCH /api/tasks/5/status` — только статус
`DELETE`	Удалить	`DELETE /api/tasks/5`

Коды ответа — первое, на что смотрят клиент, тесты и мониторинг:

Код	Значит	Когда у нас
200 OK	Успех	Чтение, изменение
201 Created	Создано, в заголовке `Location` — адрес нового ресурса	`POST` проекта или задачи
204 No Content	Успех без тела	`DELETE`
400 Bad Request	Клиент прислал неверные данные	Пустое название
404 Not Found	Ресурса нет	Задачи с таким id нет
409 Conflict	Запрос противоречит текущему состоянию	Имя проекта занято; переход `Todo → Done`
500 Internal Server Error	Ошибка на стороне сервиса	Исключение в коде

Коды делятся на группы: **2xx** — успех, **4xx** — ошибся клиент, **5xx** — ошибся сервис. Мониторинг в теме 7 следит прежде всего за 5xx.

4. REST API сервиса задач и как к нему обращаться

REST — стиль API, в котором адрес указывает на ресурс (`/api/tasks/5`), а метод — на действие. Поэтому API читается без документации: `DELETE /api/projects/3` понятно без пояснений.

Запрос	Что делает	Коды ответа
`GET /api/projects`	Список проектов	200
`GET /api/projects/{id}`	Один проект	200, 404
`POST /api/projects`	Создать проект	201, 400, 409
`DELETE /api/projects/{id}`	Удалить проект вместе с задачами	204, 404
`GET /api/tasks?projectId=1&status=Todo`	Задачи с фильтрами	200
`GET /api/tasks/{id}`	Одна задача	200, 404
`POST /api/tasks`	Создать задачу	201, 400, 404 (нет проекта)
`PUT /api/tasks/{id}`	Изменить название, описание, срок	200, 400, 404
`PATCH /api/tasks/{id}/status`	Сменить статус	200, 400, 404, 409
`DELETE /api/tasks/{id}`	Удалить задачу	204, 404

Две функции вы допишете сами: статистику проекта (лаба 1) и переименование проекта (лаба 4).

Служебные эндпоинты нужны не пользователям, а эксплуатации (п. 12): `GET /` (версия), `/health/live`, `/health/ready`, `/metrics`, `/api/chaos/*`.

Чем отправить запрос. У сервиса нет своих страниц, поэтому нужен HTTP-клиент. Подойдёт любой — сервису всё равно, кто спрашивает:

Клиент	Где	Удобен для
Swagger — `http://localhost:8080/swagger`	Браузер, открывается сам при F5	Посмотреть все эндпоинты с описаниями и отправить запрос кнопкой *Try it out → Execute*
Postman	Отдельная программа	Сохранять коллекции запросов, работать с разными адресами (локально, в контейнере, на сервере)
`DemoApi.http`	Visual Studio, кнопка *Send Request*	Запросы лежат в git рядом с кодом
`curl.exe`, `Invoke-RestMethod`	Терминал	Скрипты и проверки в CI/CD (тема 5)

Запрос в Postman: метод `POST`, адрес `http://localhost:8080/api/projects`, вкладка *Body → raw → JSON*, тело `{ "name": "Курс DevOps" }`, *Send*. В ответ — код **201** и JSON нового проекта.

5. ASP.NET Core: из чего состоит приложение

ASP.NET Core — фреймворк .NET для веб-приложений и API. Приложение — обычная консольная программа, внутри которой работает веб-сервер **Kestrel**. Он слушает порт (у нас 8080), принимает HTTP-запросы и передаёт их в конвейер обработки.

Всё начинается в `Program.cs` и делится на два этапа:

```
var builder = WebApplication.CreateBuilder(args);   // 1. Настройка: конфигурация, логи, сервисы
builder.Services.AddControllers();
// ... регистрация сервисов

var app = builder.Build();                          // 2. Сборка приложения
app.UseExceptionHandler();                          //    конвейер обработки запроса
app.MapControllers();                               //    конечные точки
app.Run();                                          //    запуск Kestrel
```

1. **`builder`** — что есть в приложении: настройки, логирование, **сервисы** в контейнере зависимостей.
2. **`app`** — как обрабатывается запрос: **middleware** (звенья конвейера) и **endpoints** (конечные точки: контроллеры, health checks, `/metrics`).

6. Структура проекта: что где лежит и зачем

Так проект выглядит в *Обозревателе решений* Visual Studio. Та же карта лежит в проекте — файл `STRUCTURE.md`.

```
src/DemoApi
├── Connected Services          мастер VS для облачных сервисов — не используем
├── Properties/
│   └── launchSettings.json     как запускать из VS: адрес, что открыть в браузере, переменные окружения
├── Зависимости                 .NET и NuGet-пакеты
├── Controllers/                приём HTTP-запросов и ответы клиенту
├── Services/                   действия с данными + запись в лог
├── Domain/                     что такое «проект» и «задача», правила статусов
├── Data/                       хранение: SQL для PostgreSQL
│   └── InMemory/               хранение в памяти процесса (без БД)
├── appsettings.json            настройки
├── DemoApi.http                готовые запросы к API
└── Program.cs                  точка входа: из чего собран сервис
tests/DemoApi.Tests
├── Unit/                       проверка правил без HTTP и БД
└── Integration/                проверка всего сервиса через HTTP
```

Служебные узлы и файлы:

Узел	Зачем	Когда туда идти
Connected Services	Мастер Visual Studio для подключения облачных сервисов. Появляется в любом веб-проекте сам, файла за ним нет	Не используем
Properties/launchSettings.json	Запуск из Visual Studio: адрес `http://localhost:8080`, открыть	Сменить порт,

	`/swagger`, переменные окружения (`ASPNETCORE_ENVIRONMENT=Development`). **В Docker-образ не попадает:** в контейнере переменные задают через `-e` или `compose.yaml`	задать переменную окружения для локального запуска
Зависимости	NuGet-пакеты: `Microsoft.AspNetCore.OpenApi` и `Swashbuckle.AspNetCore.SwaggerUI` — описание API и Swagger; `Npgsql.DependencyInjection` — драйвер PostgreSQL; `AspNetCore.HealthChecks.NpgSql` — проверка БД; `prometheus-net.AspNetCore` — метрики	ПКМ → *Управление пакетами NuGet*
appsettings.json	Настройки: `Storage` (`InMemory` или `Postgres`), уровни логов. `appsettings.Development.json` (раскройте стрелку) перекрывает их при запуске из VS	Сменить хранилище, строку подключения
DemoApi.http	Запросы к API прямо из Visual Studio	Проверить API без браузера
Program.cs	Сборка сервиса: хранилище, Swagger, health checks, метрики, логи. Бизнес-логики здесь нет	Подключить пакет или новую возможность

Папки с кодом:

Папка	Что внутри	Отвечает за	Чего в ней быть не должно
Controllers	`ProjectsController`, `TasksController`, `InfoController`, `ChaosController`. Внизу файла — записи `...Request`	Адрес API, разбор запроса, **что ответить клиенту**: 200, 201, 404, 409	SQL
Services	`ProjectService`, `TaskService`	Действия: создать, переименовать, сменить статус. Запись в лог о том, что сделано	HTTP-кодов, SQL
Domain	`Project`, `TaskItem`, `TaskItemStatus`, `ProjectStats`, `StatusRules`	Что такое проект и задача, какие переходы статуса разрешены. Эти же записи сервис отдаёт клиенту в JSON	Зависимостей от ASP.NET и БД
Data	`IProjectRepository` + `ProjectRepository`, `ITaskRepository` + `TaskRepository`, `Sql.cs`, `DatabaseInitializer.cs`	Сохранить и найти данные. Интерфейс — *что* умеет хранилище, класс — *как* это сделано на SQL. `DatabaseInitializer` создаёт таблицы при старте	Правил, HTTP
Data/InMemory	`InMemoryStore`, `InMemoryProjectRepository`, `InMemoryTaskRepository`	Те же интерфейсы, данные в памяти процесса: сервис работает без БД	—

Записи запросов (`...Request`). Клиент присылает не `TaskItem` целиком, а только то, что ему разрешено задать: в `CreateTaskRequest` есть название, описание и срок, но нет `Id`, `Status` и дат — их ставит сервер. Иначе клиент мог бы создать задачу сразу в `Done`, обойдя правила. Атрибуты `[Required]`, `[StringLength(200)]` — правила проверки: нарушение → **400** автоматически.

Зависимости идут в одну сторону:

```
flowchart LR
  C["Controllers"] --> S["Services"]
  C --> D["Domain"]
  S --> R["Data: интерфейсы"]
  R -.реализуют.- P["PostgreSQL"]
  R -.реализуют.- M["InMemory"]
  S --> D
  P --> D
  M --> D
```

`Domain` ни от кого не зависит. `Services` знают только **интерфейс** хранилища, а какая реализация под ним — решает `Program.cs` по настройке `Storage`. Поэтому в теме 3 мы подключим PostgreSQL, не меняя ни строчки в контроллерах и сервисах.

Почему именно так. Каждая папка — одна забота: меняется формат запроса — правим контроллер, меняется правило — `Domain`, меняется база — `Data`. Меньше делить нельзя: если сложить всё в `Program.cs`, через 15 эндпоинтов файл станет нечитаемым, а правила не проверить тестом отдельно. Больше не нужно: в больших системах слои выносят в отдельные проекты (`.csproj`), но для одного сервиса это лишняя

сложность.

Где что искать:

Задача	Куда идти
Новый эндпоинт, код ответа, поле в запросе	`Controllers/`
Новое поле в ответе	`Domain/`
Новое правило	`Domain/`
Запрос к БД	`Data/` и то же в `Data/InMemory/`
Подключить пакет, выбрать хранилище	`Program.cs`
Строка подключения, уровень логов	`appsettings*.json` или переменные окружения
Порт при запуске из VS	`Properties/launchSettings.json`