Тема 1

ASP.NET: как устроен сервис задач



Разбираем проект, который будем развивать весь курс

План лекции

Сегодня

1. Как устроены веб-сервисы: frontend, backend, БД
2. HTTP и REST API, чем отправлять запросы
3. Как устроено приложение ASP.NET Core
4. Структура проекта, DI и хранилище
5. Путь запроса, ошибки и тесты

После лекции вы сможете

- Понимать, где в системе backend и как с ним общаются
- Знать, что лежит в каждой папке проекта и зачем
- Пройти запрос от HTTP до хранилища
- Знать, куда добавлять новую функцию

Из чего состоит веб-приложение

Интернет-магазин, банк, трекер задач — везде одни и те же три части

Frontend

Что видит пользователь: страницы и кнопки в браузере, мобильное приложение. HTML, CSS, JavaScript

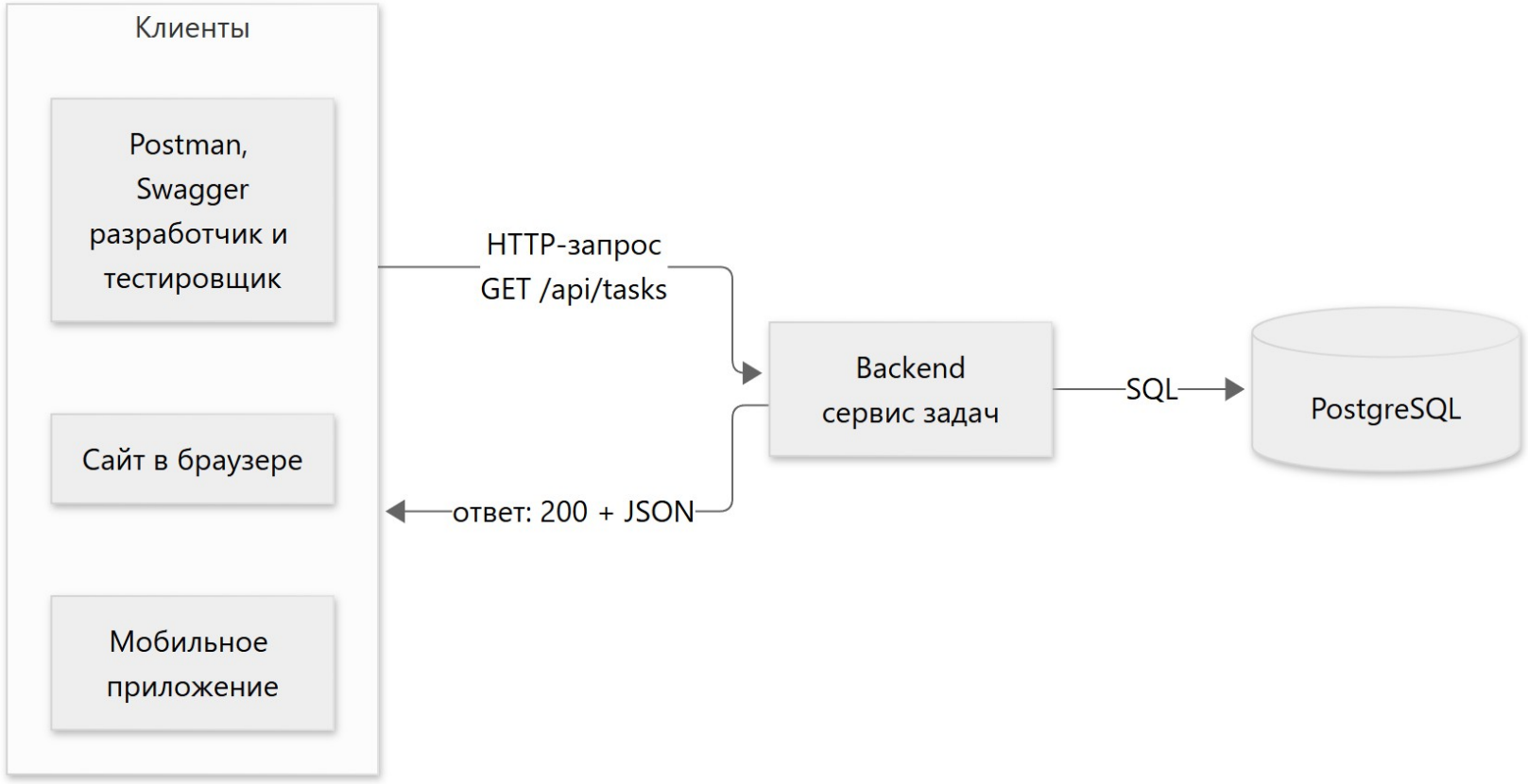
Backend

Логика и данные на сервере: проверить, сохранить, посчитать, отдать. C#, Java, Go, Python

База данных

Надёжное хранение. PostgreSQL, MySQL, MongoDB

Как они общаются: HTTP и JSON



Один backend обслуживает всех клиентов: сайт, телефон, Postman. Набор его запросов — API

Один проект — весь курс

Сервис задач растёт от темы к теме, как настоящий продукт

1 ASP.NET

Запускаем в Visual Studio, пишем свою функцию

2 Docker

Сервис в контейнере на Docker Desktop

3 Данные

PostgreSQL, тома, Docker Compose

4 Git

GitHub, Pull Request, ревью кода

5 CI/CD

Тесты и автодеплой на каждый merge

6 Сбои

Логи, рестарты, нехватка памяти

7 Мониторинг

Prometheus, Grafana, алерты

8 Итог

Инцидент: релиз → алерт → откат

Что за сервис

Backend с REST API: проекты, в них задачи со статусами `Todo → InProgress → Done`

Проекты

Создать, посмотреть, удалить вместе с задачами. Имя проекта уникально

Задачи

Название, описание, срок. Фильтры по проекту и статусу

Статусы по правилам

Нельзя перепрыгнуть из `Todo` сразу в `Done` — сервис ответит 409

HTTP: запрос и ответ

```
PATCH /api/tasks/5/status HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{ "status": "InProgress" }


HTTP/1.1 200 OK
Content-Type: application/json

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

1. **Метод и путь**

   Что сделать и с каким ресурсом

2. **Заголовки**

   Служебные данные: формат тела, авторизация

3. **Тело**

   Данные в JSON

4. **Код ответа**

   Первое, на что смотрят клиент, тесты и мониторинг

Методы: 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` — название, описание, срок
PATCH	Изменить часть	`PATCH /api/tasks/5/status` — только статус
DELETE	Удалить	`DELETE /api/tasks/5`

REST: адрес — это ресурс, метод — действие. API читается без документации

Коды ответа

Код	Значит	Когда у нас
200 OK	Успех	Чтение, изменение
201 Created	Создано, адрес в `Location`	`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	Противоречит текущему состоянию	Имя занято; `Todo → Done`
500 Internal Server Error	Ошибка в сервисе	Исключение в коде

Чем отправить запрос к API

Своих страниц у сервиса нет — нужен HTTP-клиент. Сервису всё равно, кто спрашивает

Swagger

`/swagger` в браузере, открывается сам по F5. Все эндпоинты с описаниями

Postman

Коллекции запросов. Import → Link → `/openapi/v1.json`

DemoApi.http

Запросы в Visual Studio, лежат в git рядом с кодом

curl.exe, irm

Терминал, скрипты, проверки в CI/CD

ASP.NET Core: builder и app


```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();     // сервисы
builder.Services.AddScoped<TaskService>();

var app = builder.Build();
app.UseExceptionHandler();             // конвейер
app.MapControllers();                  // конечные точки
app.Run();                             // Kestrel на :8080
```

1. **Kestrel**

   Встроенный веб-сервер: слушает порт, принимает HTTP

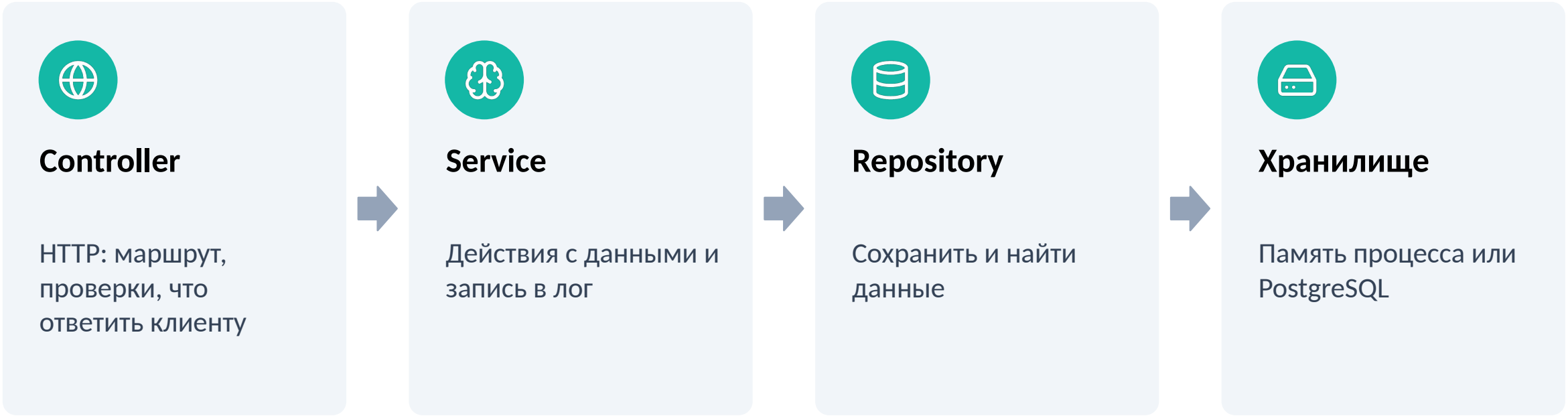
2. **builder**

   Что есть в приложении: настройки, логи, сервисы

3. **app**

   Как обрабатывается запрос: middleware и endpoints

Слои сервиса



Рядом: `Domain` — проект, задача, правила статусов; эти же записи уходят клиенту в JSON

Что где лежит в проекте

Узел в Visual Studio	Зачем
`Properties/launchSettings.json`	Запуск из VS: порт 8080, открыть `/swagger`, переменные окружения. В образ не попадает
Зависимости	NuGet-пакеты: Swagger, драйвер PostgreSQL, health checks, метрики
`Controllers/`	Адреса API и коды ответа. Внизу файла — `...Request`: что присылает клиент
`Services/`	Действия: создать, переименовать, сменить статус. Запись в лог
`Domain/`	`Project`, `TaskItem`, правила статусов `StatusRules`
`Data/`, `Data/InMemory/`	Интерфейс хранилища и две реализации: SQL и память
`appsettings*.json`	Настройки: хранилище, строка подключения, логи
`Program.cs`	Сборка сервиса: хранилище, Swagger, health, метрики
`DemoApi.http`, `tests/`	Готовые запросы к API; юнит- и интеграционные тесты

Та же карта — в файле `STRUCTURE.md` в корне проекта. Connected Services — мастер VS, не используем

Почему такая структура

Вариант	Как выглядит	Итог
Всё в `Program.cs`	Эндпоинты, SQL и правила в одном файле	Прототип. Через 15 эндпоинтов — нечитаемо
Один проект, папки-слои (наш)	`Controllers`, `Services`, `Domain`, `Data`	Сервис одной команды: просто и понятно
Несколько проектов	Отдельный `.csproj` на каждый слой	Большие системы. Для одного сервиса — лишнее

Зачем делить на слои

Меняется по своей причине

Формат запроса — контроллер. Правило — `Domain`. Хранилище — `Data`

Тестируется отдельно

Правило статусов проверяется юнит-тестом за миллисекунды, без HTTP и БД

Подменяется реализация

Сменили хранилище — контроллеры и сервисы не трогаем. Сделаем это в теме 3

Контроллер: маршрут, данные, валидация

```
[ApiController]
[Route("api/tasks")]
public sealed class TasksController(TaskService tasks)
    : ControllerBase
{
    [HttpGet("{id:int}")]                  // GET api/tasks/5
    public async Task<ActionResult<TaskItem>> Get(
        int id, CancellationToken ct)
    {
        var task = await tasks.GetAsync(id, ct);
        if (task is null) return NotFound();   // 404
        return task;                           // 200 + JSON
    }
}

public sealed record CreateTaskRequest(
    [Range(1, int.MaxValue)] int ProjectId,
    [Required, StringLength(200)] string Title);
```

1 Маршрут

`[Route]` + `[HttpGet]` дают адрес; `{id:int}` — параметр

2 Ответ

Объект → 200. `NotFound()` → 404, `Problem(…, 409)` → 409

3 Запрос

`...Request` — только то, что можно прислать клиенту

4 Валидация

Атрибуты на записи; при ошибке — 400 без нашего кода

DI: объекты собирает контейнер

Цепочка на каждый запрос: `TasksController ← TaskService ← ITaskRepository ← хранилище`

Singleton

Один на приложение. `InMemoryStore`, пул соединений с БД

Scoped

Один на HTTP-запрос. Сервисы и репозитории

Transient

Новый при каждом запросе зависимости. Лёгкие объекты

Хранилище: сейчас память, потом PostgreSQL

Storage = InMemory

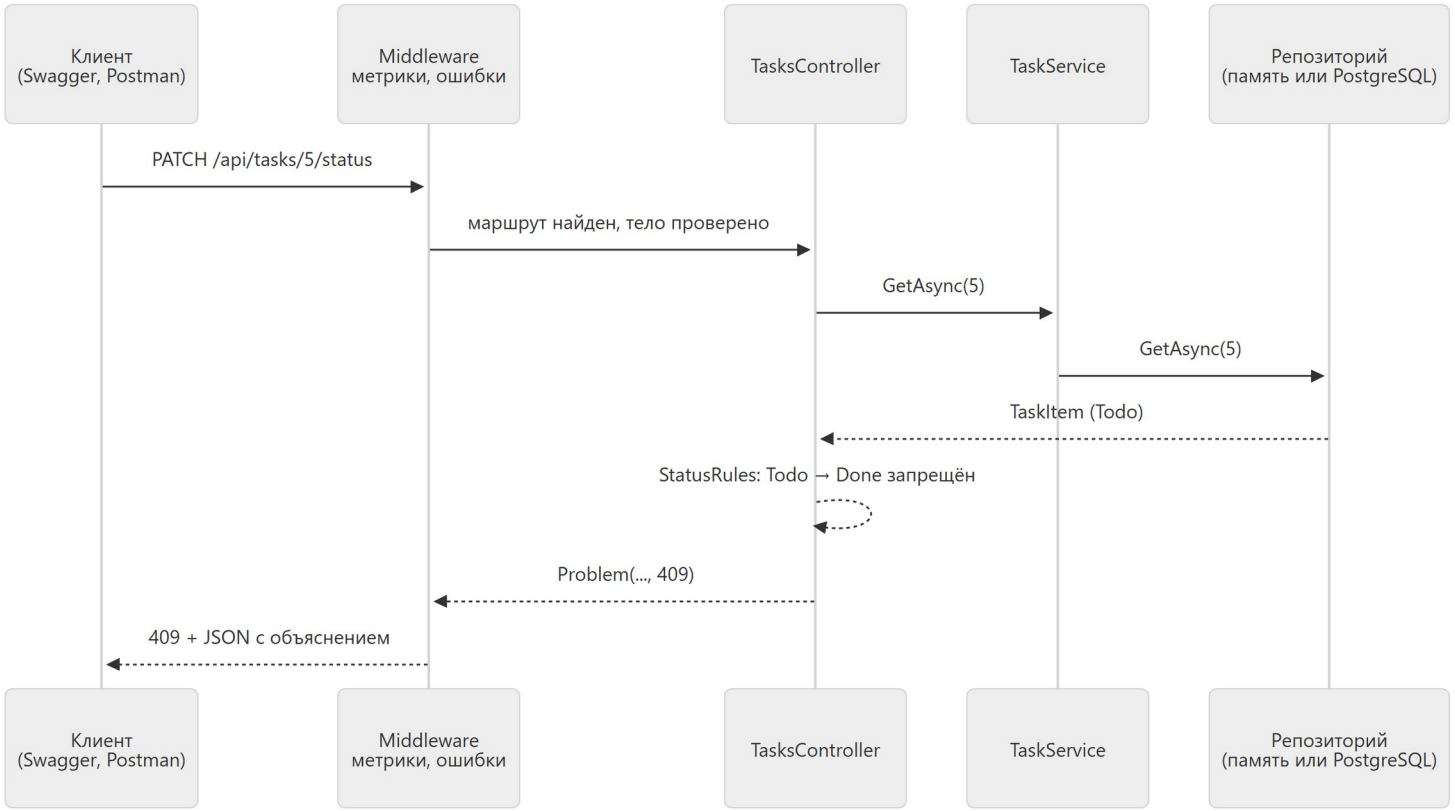
- Темы 1–2
- Списки в памяти процесса
- Ничего ставить не нужно
- После перезапуска данные пропадают

Storage = Postgres

- С темы 3
- PostgreSQL в контейнере
- SQL через библиотеку Npgsql
- Данные живут на томе

Переключение — одна настройка. Контроллеры, сервисы и тесты не меняются

Путь запроса: PATCH из Todo в Done



Маршрут → проверка тела → задача найдена → StatusRules: переход запрещён → 409 → метрика

Конвейер, конфигурация, ошибки

Middleware по порядку

`UseHttpMetrics` снаружи — видит итоговый код, даже 500. Затем обработка ошибок, затем endpoints

Конфигурация слоями

`appsettings.json` → `appsettings.Development.json` → переменные окружения. Последний слой побеждает

Ошибки в едином JSON

Валидация → 400, `NotFound()` → 404, `Problem(…, 409)` → 409, исключение → 500 и запись в лог

Что в коде нужно для эксплуатации

В коде	Кто использует	Тема
`/health/live`, `/health/ready`	Docker, Compose, проверка после деплоя	2, 3, 5, 6
`/metrics`	Prometheus и Grafana	7
Логи в stdout, JSON	`docker logs`, Dozzle	2, 6
`APP_VERSION` **в** `GET /`	Какая версия задеплоена	5
Настройки из окружения	Один образ для всех окружений	2, 3
`/api/chaos/*`	Учебные сбои	6–8

Сервис пишут не только для пользователей, но и для тех, кто его запускает и чинит

Как проверять сервис

Swagger: `/swagger`

Страница со всеми эндпоинтами, запросы прямо из браузера. Только в Development

Юнит-тесты

Правила без окружения: `StatusRulesTests`. Миллисекунды

Интеграционные

Весь сервис через HTTP. Сейчас на памяти, с темы 3 — на PostgreSQL, в теме 5 — в CI

Как добавить функцию

1. **Domain**
   Запись или правило
2. **Data**
   Метод в интерфейсе и в обоих хранилищах
3. **Service**
   Действие и запись в лог
4. **Controller**
   Маршрут, проверки 404/409, XML-описание
5. **Tests + .http**
   Тест на успех и ошибку, пример запроса

В лабе 1 так добавите `GET /api/projects/{id}/stats`

Типичные ошибки

- SQL в контроллере: не подменить хранилище и не проверить без БД
- SQL склейкой строк — SQL-инъекция. Только параметры
- `TaskItem` вместо `CreateTaskRequest` в параметре — клиент подменит `id` и `status`
- Пароли в `appsettings.json`, закоммиченном в git
- `catch (Exception) { return Ok(); }` — ошибка скрыта от всех

Проверьте себя

1. Чем frontend отличается от backend? Кто может обращаться к сервису задач?
2. Чем `PUT` отличается от `PATCH`? Почему создание возвращает 201, а не 200?
3. Почему `POST /api/tasks` принимает `CreateTaskRequest`, а не `TaskItem`?
4. Кто вернёт 400, если в `PATCH /status` нет поля `status`?
5. Почему смена хранилища не требует менять `TaskService`?



Практика

Лаба 1. Запускаем сервис задач и добавляем свою функцию

Открываем проект в Visual Studio и работаем с ним руками

- Запуск по F5, запросы через Swagger и Postman
- Проверяем сами: данные в памяти пропадают при перезапуске
- Проходим запрос отладчиком через все слои
- Пишем `GET /api/projects/{id}/stats` с тестами