Неделя 2. Лекция 1. ASP.NET: как устроен сервис задач

Часть 2 из 2. Начало лекции — в материалах предыдущей недели.

7. Контроллеры, маршруты и 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 + задача в JSON
    }
}
```

- **Маршрутизация.** `[Route]` на классе и `[HttpGet("{id:int}")]` на методе вместе дают адрес. `{id:int}` — параметр пути, `:int` — ограничение: `/api/tasks/abc` просто не совпадёт с маршрутом.
- **Привязка модели (model binding).** ASP.NET сам заполняет параметры метода: `id` — из пути, `request` — из JSON-тела, `projectId` в `List` — из строки запроса (`[FromQuery]`).
- **Ответ.** Вернули объект — это **200** и объект в JSON. `NotFound()` — **404**, `NoContent()` — **204**, `CreatedAtAction(...)` — **201**, `Problem("текст", statusCode: 409)` — **409** с объяснением.
- **Валидация.** Атрибуты на записи запроса описывают правила, `[ApiController]` проверяет их **до** вызова метода и сам возвращает **400** со списком ошибок:

```
public sealed record CreateTaskRequest(
    [Range(1, int.MaxValue)] int ProjectId,
    [Required, StringLength(200)] string Title,
    [StringLength(2000)] string? Description,
    DateOnly? DueDate);
```

- **`async`/`await` и `CancellationToken`.** Пока запрос ждёт БД, поток не простаивает и обслуживает другие запросы. Отключился клиент — `ct` отменяет ненужную работу.

8. Внедрение зависимостей (DI)

Контроллер не создаёт `TaskService` сам, а **просит его в конструкторе**. Объекты создаёт **DI-контейнер** (не путать с Docker-контейнером: это объект внутри приложения). Он знает, что чем является, по регистрации в `Program.cs`:

```
builder.Services.AddScoped<TaskService>();
builder.Services.AddScoped<ITaskRepository, InMemoryTaskRepository>();   // интерфейс → реализация
```

На запрос DI собирает цепочку: `TasksController ← TaskService ← ITaskRepository ← хранилище`.

Время жизни	Экземпляр	Пример у нас
`Singleton`	Один на всё приложение	`InMemoryStore` — общее хранилище в памяти; `NpgsqlDataSource` — пул соединений
`Scoped`	Один на HTTP-запрос	Сервисы, репозитории
`Transient`	Новый при каждом запросе зависимости	Лёгкие вспомогательные объекты

Зачем. Классы не знают, откуда берутся их зависимости, поэтому реализацию можно подменить одной строкой. Именно так сервис переключается между хранилищами (п. 9).

9. Хранилище: сейчас в памяти, с темы 3 — PostgreSQL

Сервис умеет хранить данные двумя способами. Выбирает настройка `Storage`:

`Storage`	Где данные	Когда в курсе
`InMemory`	В памяти процесса: списки в `InMemoryStore`	Темы 1–2. Ничего ставить не нужно, но после перезапуска данные пропадают
`Postgres`	В PostgreSQL, SQL через библиотеку Npgsql	С темы 3: БД в контейнере, данные сохраняются

```
        if (usePostgres)
        {
            builder.Services.AddNpgsqlDataSource(connectionString);
            builder.Services.AddScoped<ITaskRepository, TaskRepository>();          // SQL
        }
        else
        {
            builder.Services.AddSingleton<InMemoryStore>();
            builder.Services.AddScoped<ITaskRepository, InMemoryTaskRepository>();  // списки в памяти
        }
```

`TaskService` зависит от интерфейса `ITaskRepository` и не знает, какая реализация под ним. Поэтому в теме 3 мы переключим сервис на PostgreSQL одной настройкой, **не трогая** контроллеры, сервисы и тесты.

При запуске сервис пишет в лог, какое хранилище выбрано: `Хранилище: в памяти` или `Хранилище: PostgreSQL`.

10. Путь запроса

Пример: `PATCH /api/tasks/5/status` с телом `{"status":"Done"}` для задачи в статусе `Todo`.

```
sequenceDiagram
  participant C as Клиент<br/>(Swagger, Postman)
  participant M as Middleware<br/>метрики, ошибки
  participant Ctl as TasksController
  participant S as TaskService
  participant R as Репозиторий<br/>(память или PostgreSQL)
  C->>M: PATCH /api/tasks/5/status
  M->>Ctl: маршрут найден, тело проверено
  Ctl->>S: GetAsync(5)
  S->>R: GetAsync(5)
  R-->>Ctl: TaskItem (Todo)
  Ctl-->>Ctl: StatusRules: Todo → Done запрещён
  Ctl-->>M: Problem(..., 409)
  M-->>C: 409 + JSON с объяснением
```

1. **Маршрутизация** выбирает `TasksController.ChangeStatus`.
2. `[ApiController]` проверяет тело: нет `status` → **400**, наш код даже не вызывается.
3. **Контроллер** просит у сервиса задачу. Нет задачи → **404**.
4. **Контроллер** спрашивает правило в `Domain/StatusRules.cs`. Переход запрещён → **409**.
5. Разрешён → **сервис** меняет статус в хранилище и пишет в лог → **200** и задача в JSON.
6. **Middleware метрик** записывает запрос с его кодом — его увидит Prometheus (тема 7).

Все проверки видны в одном методе контроллера — удобно пройти отладчиком (лаба 1).

11. Конвейер, конфигурация, ошибки

Middleware — звено конвейера обработки запроса. Запрос проходит звенья по порядку, каждое может выполнить работу до и после следующего. Порядок важен:

```
        app.UseHttpMetrics();        // 1. снаружи — видит итоговый код ответа, даже 500
        app.UseExceptionHandler();   // 2. упавший запрос → 500, исключение → в лог
        app.MapControllers();        // 3. конечные точки
```

Конфигурация читается слоями, каждый следующий переопределяет предыдущий:

1. `appsettings.json` — общее для всех окружений (`Storage: InMemory`);
2. `appsettings.Development.json` — при `ASPNETCORE_ENVIRONMENT=Development`, то есть при запуске из Visual Studio;
3. **переменные окружения** — так настраивают контейнер (темы 2–3). `ConnectionStrings:Db` в JSON =

переменная `ConnectionStrings__Db`.

Зачем слои: один и тот же код работает на ноутбуке, в CI и на сервере с разными настройками, а пароли не лежат в git.

Ошибки приходят клиенту в едином JSON-формате **ProblemDetails** (включён строкой `AddProblemDetails()` в `Program.cs`):

Ситуация	Кто отвечает	Ответ
Неверное тело запроса	`[ApiController]` по атрибутам записи запроса	**400**, список полей в `errors`
Нет проекта или задачи	Контроллер: `NotFound()`	**404**
Нарушено правило	Контроллер: `Problem("…", statusCode: 409)`	**409**, объяснение в `detail`
Код упал с исключением	ASP.NET (`UseExceptionHandler`), ошибка — в лог с уровнем Error	**500** без подробностей наружу

```
{ "title": "Conflict", "status": 409, "detail": "Нельзя перевести задачу 5 из Todo в Done" }
```

Почему 500 без подробностей: текст исключения может раскрыть внутренности — пути, SQL, имена таблиц. Клиенту — только код, подробности — в лог, где их найдёт тот, кто чинит (тема 6).

12. Что в коде нужно для эксплуатации

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

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

13. Документация API и тесты

OpenAPI — машиночитаемое описание API (`/openapi/v1.json`). По нему страница **Swagger** (`/swagger`) показывает все эндпоинты и позволяет отправлять запросы прямо из браузера. Это же описание умеет импортировать Postman: *Import* → *Link* → `http://localhost:8080/openapi/v1.json` — и все запросы сервиса появятся коллекцией. Обе страницы включены только в окружении `Development`.

Описания в Swagger — что делает операция, какие коды ответов бывают, что значит каждое поле JSON — берутся из **XML-комментариев** `///` в коде:

```
/// <summary>Сменить статус задачи</summary>
/// <remarks>Разрешены переходы Todo → InProgress → Done и на шаг назад.</remarks>
/// <response code="409">Такой переход статуса запрещён</response>
[HttpPatch("{id:int}/status")]
public async Task<ActionResult<TaskItem>> ChangeStatus(...)
```

Чтобы комментарии попали в описание, в `.csproj` включено `<GenerateDocumentationFile>true</GenerateDocumentationFile>`. Хорошее описание API — часть функции, как и тесты.

Тесты:

- **Юнит** (`Unit/StatusRulesTests.cs`) проверяют правило за миллисекунды, без окружения.
- **Интеграционные** (`Integration/*`): `WebApplicationFactory` поднимает весь сервис в памяти, и запросы идут через настоящий HTTP-конвейер: маршруты, валидация, коды ответов. Сейчас они работают с хранилищем в памяти, с темы 3 — с настоящим PostgreSQL, в теме 5 — на сервере GitHub.

Как добавить функцию: чек-лист

1. **Domain:** новая запись или правило, если нужно.

2. **Data:** метод в интерфейсе репозитория и **в обеих реализациях** — в памяти и PostgreSQL.
3. **Services:** метод-действие, запись в лог.
4. **Controllers:** действие с маршрутом, проверками (404, 409), XML-комментарием; запись `...Request`, если клиент присылает тело.
5. **Tests:** интеграционный тест на успех и на ошибку.
6. **`DemoApi.http`:** пример запроса для коллег.

Типичные ошибки

- **SQL в контроллере:** не подменить хранилище и не проверить без БД.
- **SQL склейкой строк** `"... WHERE id = " + id` — SQL-инъекция. Только параметры.
- **Доменная запись вместо записи запроса** в параметре действия: клиент сможет прислать `id` или `status`, которые должен ставить сервер.
- **Пароли в `appsettings.json`**, закоммиченном в git.
- **`catch (Exception) { return Ok(); }`:** ошибка скрыта от клиента, логов и мониторинга.
- **Синхронные вызовы БД** (`.Result`, `.Wait()`) в веб-коде: потоки заканчиваются под нагрузкой.

Вопросы для самопроверки

1. Чем frontend отличается от backend? Почему у сервиса задач нет своих страниц и кто может к нему обращаться?
2. Чем `PUT` отличается от `PATCH`? Какой код вернёт создание задачи и почему не 200?
3. Почему `POST /api/tasks` принимает `CreateTaskRequest`, а не `TaskItem`?
4. Что вернёт API, если в `PATCH /status` не передать `status`? Кто это делает?
5. Как сервис решает, где хранить данные, и почему смена хранилища не требует менять `TaskService`?
6. Зачем `UseHttpMetrics()` стоит раньше `UseExceptionHandler()`?

Практика: лаба 1

Откроете проект в Visual Studio, запустите и «потыкаете» API через Swagger и Postman. Увидите, что данные в памяти пропадают при перезапуске. Пройдёте запрос отладчиком через все слои и добавите свою функцию — статистику проекта — с тестами.