Pool Guard — Workshop

Table of Contents

> As we embark on the journey of implementing an access control system for a public outdoor pool, we are not merely creating a process for ticket validation and management—we are enhancing the experience of every visitor and ensuring the safety and enjoyment of our community. This assignment offers a unique opportunity to design a system that will transform the way we manage and monitor access to a beloved public facility, turning a simple pool visit into a well-organized and secure event. With our innovative approach, we aim to streamline operations, prevent unauthorized access, and create a welcoming environment where each visitor feels valued and assured of their safety.

As an introductory project for creating proper WebApis we start with leveraging the skills you gained last year in WMC by creating a simple (minimal) REST API. This system will manage the tickets and access to the glorious outdoor pool of HTL Leonding.

1. Project Setup

First, we need to create the project. Hopefully you have the latest version of the templates installed!

```
dotnet new leominiapi -n PoolGuard -o PoolGuard
```

Once the template has been scaffolded, open it in your IDE, build & run. You should now have a simple API to greet people running on `http://localhost:5200`.

1.1. HTTP Request File

A very good way of testing a REST API ist to use a `.http` file. Those request files are supported by all relevant IDEs and allow you to not only define, but also run various REST requests. As an added benefit, those files can be checked into version control and shared with your team.

Our template already created a `PoolGuard.http` file for you. Run the `GET` request and verify, that you get a valid response — your API is ready!

2. Minimal Web API

In ASP.NET Core there are two major, official ways for creating Web APIs:

1. `Controller` based application

- More powerful & flexible feature wise
- Better structure, especially for larger projects
- Less (startup) performance focused and not AOT capable (we are talking about hundreds of thousands of requests per second here, congratulations if your app needs that!)

2. Minimal API based applications
   - Very similar to `Express` and other, similar web frameworks
   - Can get you up and running very quickly
   - Harder to maintain for larger projects
   - Very (startup) performance focused and AOT capable

> An AOT compiled application will start very fast and consume a little less memory, while a 'regular' JIT-based application starts slower, consumes a bit more memory, but also has a higher throughput, because the JIT can dynamically optimize actual *hot paths*, something a static AOT compiler can't do. That makes AOT applications especially useful for serverless functions in the cloud where a small microservice is started and stopped for every function and every request. On the other hand, a JIT-based application is better suited for larger, long-running services, where the startup time is not that important. Always pick the right tool for the job!

During the year — once we add authentication & authorization, model validation, 3bases, many endpoints,... — we will get to 'full', controller based applications, but for now we will stick with the minimal API approach.

The goal is that you are able to quickly create a simple Web API against which we can work when doing frontend development. Most of the complicated things we will ignore for the moment and leave for later.

2.1. The first Endpoint

To begin, let's take a look at the endpoint defined in the template. Don't worry about the initial parts of the code — those are required to set up the application, and we will talk about them soon. But for now, we focus on the thing we are most interested in: how to define a REST endpoint.

Program.cs

```
app.MapGet("/hello/{name}", (string? name, IClock clock) => ①②③④⑤
   {
       if (string.IsNullOrWhiteSpace(name))
       {
           return Results.BadRequest(); ⑥
       }

       var now = clock.GetCurrentInstant();

       return Results.Ok(new Greeting(name, now)); ⑦
   })
```

```
    .Produces<Greeting>(StatusCodes.Status200OK) ⑧
    .Produces(StatusCodes.Status400BadRequest) ⑧
    .WithName("Greetings") ⑧
    .WithOpenApi(); ⑧
```

① Similar to `Express`, we use `app.MapGet` to define a `GET` endpoint

② Next up is the *route* — in this case a *route parameter* (`name`) is also defined

③ Then we need to pass a *lambda* function that will be executed when the endpoint is called. Good, that you are already very comfortable with lambdas!

④ The first parameter of the lambda will be set to the value from the route parameter — *for this to work the name **has** to match* and the type must be compatible

⑤ Finally, we also get an instance of `IClock` — where is that coming from? *Who* is calling this method and the lambda anyway, and *where*? □□ We will talk about that in just a moment.

⑥ Based on the required logic and the values submitted by the client, we can return different results and status codes (HTTP status codes are the same everywhere, so you can make use of your knowledge from WMC) — in this case a required parameter (`name`) is missing, so we return a `BadRequest` result

⑦ If everything is fine, we return a 200 status with a `Greeting` object (the object itself is not important, it is just an arbitrary `class`)

⑧ At the end we can see a bunch of *extension methods* which are used to configure the *metadata* for OpenApi — we'll talk about what that is later

If you run the `GET` request from the `PoolGuard.http` file, you should see a response similar to this:

```
{
  "message": "Hello, Schwammal!",
  "timestamp": "2024-08-11T16:46:43.7213775Z"
}
```

That works, because the application created by the template is configured to serialize the response objects to JSON.

⇒ **You have a working REST API!** □

To add more endpoints, you can simply add more `app.MapGet`, `app.MapPost`, `app.MapPut`, `app.MapDelete`,… calls.

2.2. OpenApi

Before we continue, let's ask a question:

> How would someone else, without access to the code, know which endpoints exist and what to do with them?

That is a quite important question, because your are most likely not going to work on your project in isolation. There are other teams, customers, suppliers, users etc. who all have an interest in the API you are providing.

A common way to communicate the structure and capabilities of a REST API is to use an OpenApi *specification* (also commonly known as *Swagger*). The template already prepared your Web API to automatically create a specification based on the *metadata* of the endpoint(s).

You can test this by executing the second request in the `http` file, which will return the OpenApi specification of your application. It looks something like this:

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "PoolGuard",
    "version": "1.0" ①
  },
  "paths": {
    "/hello/{name}": { ②
      "get": {
        "tags": [
          "PoolGuard"
        ],
        "operationId": "Greetings",
        "parameters": [ ③
          {
            "name": "name",
            "in": "path",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": { ④
          "200": {
            "description": "OK",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Greeting" ⑤
                }
              }
            }
          },
          "400": {
            "description": "Bad Request"
          }
        }
      }
    }
```

```
  },
  "components": {
    "schemas": {
      "Greeting": { ⑥
        "type": "object",
        "properties": {
          "message": {
            "type": "string",
            "nullable": true,
            "readOnly": true
          },
          "timestamp": {
            "$ref": "#/components/schemas/Instant"
          }
        },
        "additionalProperties": false
      },
      "Instant": {
        "type": "object",
        "additionalProperties": false
      }
    }
  }
}
```

① The version of the API, this is important, because you are going to do *semantic versioning* in the real world. However, for our needs this will stay at 1.0 forever

② All routes are listed

③ For each route the parameters and their types are listed

④ The possible responses of the endpoint (a 500 is always an option, though, because it means the server itself panicked)

⑤ If a *complex* object is returned, a *schema* of that object is referenced

⑥ And here is the specification of the `Greeting` object

With this document we can not only learn a lot about an API without menacing trial & error, but there are also tools to *generate client stubs* based on such a specification — which can speed up client development.

2.3. Configuration

After this great success of having a working endpoint and knowing how to document it, we can take a look at the remaining pieces that make this Web API work.

Starting in the `Program.cs` file, we see the following:

Program.cs

```
var builder = WebApplication.CreateBuilder(args); ①
```

```
builder.Services.AddEndpointsApiExplorer(); ②
builder.Services.AddOpenApi(); ②
builder.Services.RegisterServices(); ③
builder.Services.ConfigureServices(builder.Environment.IsDevelopment()); ④

var app = builder.Build(); ⑤

if(builder.Environment.IsDevelopment()) ⑥
{
    app.MapOpenApi();
}

// endpoint

await app.RunAsync(); ⑦

internal sealed class Greeting(string name, Instant timestamp) ⑧
{
    public string Message => $"Hello, {name}!";
    public Instant Timestamp => timestamp;
}
```

① We want to build a web application, for that we need a builder (and apply the *builder pattern*)

② Needed for OpenApi, we will talk about that soon

③ Registering services — what are services?

④ Configuring services — what are these services??

⑤ Done configuring the builder, now we construct the application

⑥ When running in development mode, we create a OpenApi endpoint

⑦ Finally, we start the application! There is also a synchronous version of this method, but we prefer the async version

⑧ Only the simple class we returned in the endpoint earlier

Lots of talk about services. We will dive deeper and deeper into this topic over the course of the year, but to get started, imagine the following architecture:

```
@startuml
package "Web Application" {
  [Endpoint] --> [Service]
  [Service] --> [Data Storage]
  [Service] --> [Service]
}

package "Client" {
  [ClientApp] -r-> [Endpoint]
}
```

```
note as n1
    Most of those arrows also point in
    the other direction (responses).
    Omitted here to make clear in which
    order components are becoming active
    when a request is made
end note
@enduml
```

- Clients talk to the endpoints (controllers), which are responsible for receiving the requests and calling the appropriate service. After they receive the result from the service, they return a properly formatted response to the client.
  - So they deal with all the REST-Tasks like routes, status codes, most of the validation, object (de)serialization (including DTOs),… but do not contain (a lot of) business logic.
  - Also, they never talk to the data storage directly.
- Services are responsible for the actual business logic — they are the heart of the application. A service might call into the data storage. It might also call other services. It does not know anything about REST or the client, it just handles the business logic.
- Date Storage is a complex topic to which we will get later. For the moment just imagine an few `List<T>` which contain some data we can get and store.

'Service' is a very broad term. There are not only the typical business logic services, but also utilities etc. A good way to remember is that a service is a thing that knows how to do a certain job. To accomplish that job it usually needs help from its *dependencies*.

2.3.1. Setup

If we take a look at the `Setup` class, we can see a few extension methods on `IServiceCollection` — which is the *container* for all services in the application. We add additional services to the container, and we configure some of them.

Setup.cs

```
public static void RegisterServices(this IServiceCollection services)
{
    services.AddSingleton<IClock>(SystemClock.Instance); ①
}

public static void ConfigureServices(this IServiceCollection services, bool
isDevelopment)
{
    services.ConfigureHttpJsonOptions(options =>
    {
        options.SerializerOptions.PropertyNamingPolicy = JsonNamingPolicy.CamelCase;
②
        options.SerializerOptions.WriteIndented = isDevelopment; ③
```

```
        options.SerializerOptions.DefaultIgnoreCondition = JsonIgnoreCondition.Never; ④

        options.SerializerOptions.Converters.Add(new JsonStringEnumConverter()); ⑤
        options.SerializerOptions.ConfigureForNodaTime(DateTimeZoneProviders.Tzdb); ⑥
    });
}
```

① Here we *register a service with DI*, in this case a `SystemClock` instance for an `IClock` interface — we will talk about DI right after this

② Our JSON should use camelCase (instead of PascalCase) for property names, that will help with JS/TS clients and is the default in the web world

③ If running in production, let's not waste bandwidth with pretty printed JSON — but during development it is very helpful

④ We want to serialize all properties, even if they are `null` — this is a common problem with JSON serialization

⑤ Enums serialized as `string` instead of numeric values makes them easier to read and process ⇒ preferable even if it uses more bandwidth

⑥ We also want to (de)serialize `NodaTime` objects in an optimal way, this one line will do the trick

Once the application grows, we will add additional services and configurations here.

3. IoC & DI

This is one of the most important concepts you have to understand this year! It is also not easy to grasp at first, but once you get it, you will see how powerful it is.

Let's start with some definitions:

Inversion of Control (IoC)

In the past, when you wanted to create a new instance of a type, you used the `new` operator. The problem with that is, that it leads to *strong coupling*, meaning that the code that constructs the instance has to know the exact type, its constructor signature **and** *how to create all dependencies **and their dependencies** and so forth*. When we apply IoC we *relinquish* the control over the object creation. Instead, *something else* handles the creation and we simply 'order' the instances we need. That becomes especially powerful in combination with *interfaces* (contracts).

Dependency Injection (DI)

You already know what dependencies are: we defined them as 'all the stuff an object needs to fulfill its job' in the past, and said that they need to be passed in as constructor parameters, because they have to be available right away for the object to work. DI is now an application of this principle in the context of IoC. A *service* defines which *dependencies* it needs by declaring them as constructor parameters. The DI container then goes ahead and builds all those dependencies (and their dependencies etc.) needed and constructs the service instance. So we only need to 'order' and instance of the service, and everything else is done for us.

That sounds great! Someone else is doing all the work for us!

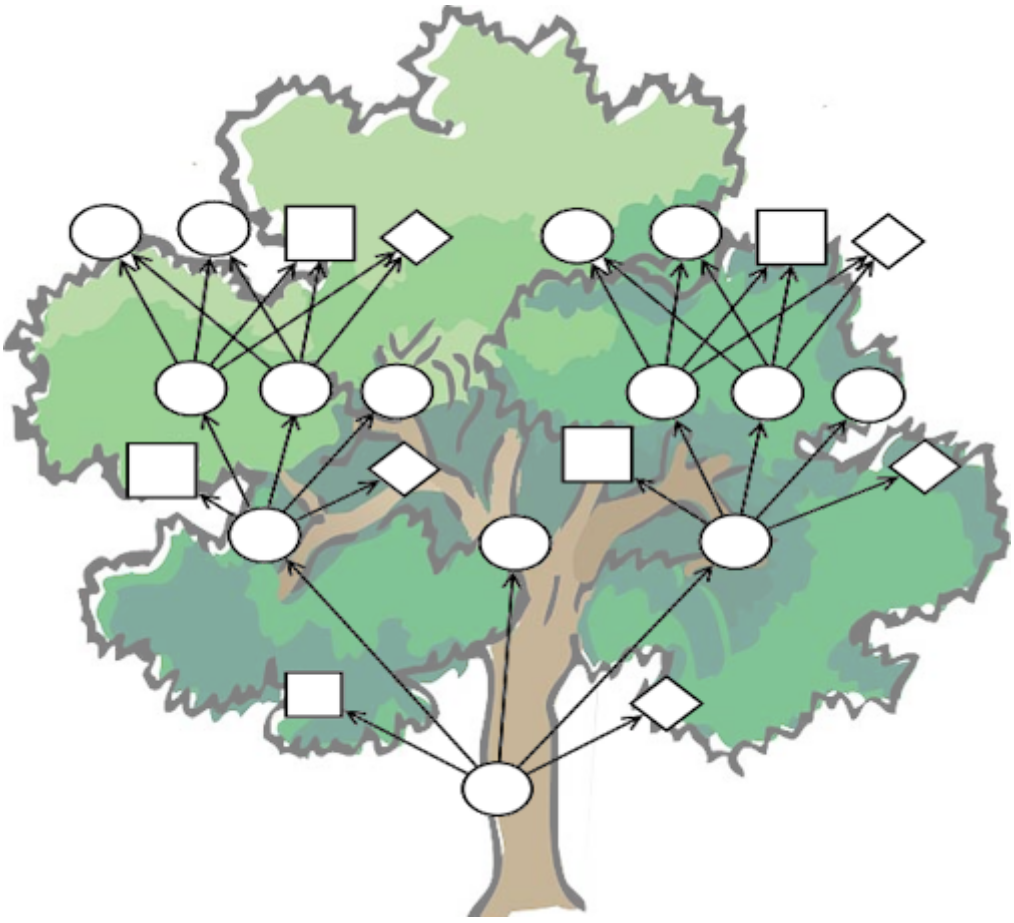
However, there is always a catch:

- Once we start doing DI, we have to go all they way: there is no mix and match, either we get an instance (with *all* of its dependencies) from the container, or we have to create *everything* ourselves.
- The DI container needs to know which types fulfill the requirements, so typically we will *depend on interfaces* and *have to tell the DI container which type it should provide for each interface* ⇒ that's quite a lot of configuration work in larger applications!
- There **must not** be cycles in the dependency graph

But once everything works, the system is really powerful and flexible:

- Imagine getting a mocked data storage for unit tests — nothing in the service's logic has to change to make it work ⇒ contracts at their finest
  - Also, the common issue of getting fixed date/time value for tests — not an issue if a correctly set up `IClock` is provided
- Imagine getting input from touch on mobile and keyboard on desktop without having to change anything, because the DI was configured to provide a touchscreen or keyboard input service depending on the platform *at runtime* ⇒ yes, dependencies can be changed *dynamically* if necessary

Last but not least, it makes code much cleaner to read, because the complex construction logic of the whole **dependency tree** is not put everywhere in the code, but is handled by the DI container.



Even when we use DI for services, we still can manually construct some helper

types. That is common if those objects are an implementation detail (helping to structure complex logic) of a service and are not used outside of it.

3.1. Scopes

With IoC we have given away the control over the *lifetime* of an object: we don't create it, and we also don't know exactly when it will be destroyed. All of that is handled by the DI container. Yet, for a proper implementation, basic knowledge about the lifetime of an object is necessary.

This is controlled already when registering the service with the DI container. Three options exist:

- **Singleton:** you already know what to expect, the DI container will use a single instance throughout the runtime of the whole application
  - That also means that we don't have to use a `static` instance field, because we can rely on the DI container to provide the same instance everywhere
- **Scoped:** a new instance is created for each *scope* — in our case, that will be *one request*
  - Scopes can also be created manually, we'll need that for hosted services and unit testing later
- **Transient:** a new instance is created *every* time the service is requested
  - So even within one scope (request) every object gets its own instance of that service

All of that is only relevant, if the service has **state** — stateless services can be considered transient. However, it might still make sense to use a singleton if some expensive resources are required.

If cleanup is necessary, `IDisposable` & `IAsyncDisposable` are important, because the related `Dispose` & `DisposeAsync` methods will be called by the DI container when the service is destroyed.

3.2. DI for Minimal APIs

In general, dependencies are declared in the constructor and passed there during creation by the DI container. However, in the case of minimal APIs we have `MapGet` etc. *methods* — there *is no constructor*! So how do we get our dependencies?

The solution is quite simple, but can look a little confusing: dependencies are declared as method parameters of the lambda! That means that both route/query/body parameters **and** dependencies are put as parameters in the method signature. **Be very careful which is which!** This problem will go away once we move to controllers later in the year, where we have proper classes. get constructor injection and overall finer control.

```
app.MapGet(pattern:"/hello/{name}", (string? name, IClock clock) =>
                                    ROUTE PARAMETER  DEPENDENCY
```

Be consistent by *always* putting the *parameters first* and *then* the *dependencies* — don't mix them up! Also, the dependencies will typically be

> interfaces, while the route/query/body parameters will be concrete types.

For non-routes, the DI works as usual, so services declare *their* dependencies via constructor parameters and the DI container will provide them.

3.2.1. Explicit attributes

After trying to do everything *convention based* initially, in later .NET versions they added the well known attributes from the controller based approach to minimal APIs as well.

So, optionally, we can now annotate each parameter with:

- `[FromRoute]` for route parameters
- `[FromQuery]` for query parameters
- `[FromBody]` for body parameters
- `[FromServices]` for dependencies

That makes the definition more verbose, but removes any ambiguity or unexpected behavior. We will do this when moving to controllers — for minimal APIs I will leave it up to you.

4. Implementation

With all the theoretical concepts out of the way, let's start implementing the actual endpoints for the Pool Guard system.

The job of the system is to control access to the pool. Each person needs a ticket to enter the pool area. We track who is currently in the pool area. When leaving, the person is checked out again. Only a certain number of people are allowed in the pool area at the same time. People can re-enter the pool area if they have a valid ticket, but not if a person with this ticket number is already inside.

We want to provide the following functionalities:

- Personalized tickets can be created
- A person can try to enter the pool area with a ticket
- A person can leave the pool area
- Status information about the pool area can be requested

> This example does *not* use authentication & authorization to solely focus on the new framework — **never** do that in a real application! For example, *anybody* could create or delete tickets in our system, which is, obviously, not practical in the real world.

Don't worry too much about project structure for now. Simply create a directory `Tickets` within `Core` and put all the classes there. We will talk in detail about how to properly organize such an application later.

4.1. Ticket

Let's start with something you are familiar with: a simple `Ticket` class:

Ticket.cs

```
namespace PoolGuard.Core.Tickets;

public sealed class Ticket
{
    private readonly List<AccessEvent> _accessEvents = []; ①
    public Guid Id { get; init; } ②
    public Instant ValidFrom { get; init; } ③
    public Instant ValidTo { get; init; } ③
    public required string PersonName { get; init; } ④

    public bool IsInPoolArea => _accessEvents.Count > 0
                                && _accessEvents[^1].Type is AccessEventType.Entered;
⑤

    public bool AddAccessEvent(AccessEventType accessEvent, Instant timestamp)
    {
        if(timestamp < ValidFrom || timestamp > ValidTo)
        {
            return false; ⑥
        }

        switch (accessEvent) ⑦
        {
            case AccessEventType.Entered when IsInPoolArea:
            case AccessEventType.Exited when !IsInPoolArea:
            {
                return false;
            }
            case AccessEventType.Exited or AccessEventType.Entered:
            {
                _accessEvents.Add(new AccessEvent(timestamp, accessEvent));

                return true;
            }
            default:
            {
                throw new ArgumentOutOfRangeException(nameof(accessEvent),
accessEvent, "Unknown access event type.");
            }
        }
    }
}

public readonly record struct AccessEvent(Instant Timestamp, AccessEventType Type); ⑧
```

```
public enum AccessEventType ⑨
{
    Entered = 10,
    Exited = 20
}
```

① The ticket stores all access events (entering or leaving the pool area) in a collection

② For the ID we will use a `Guid` so that we don't have to worry about generating unique IDs — don't use `Guid` everywhere, though, they are *big & slow*

③ The ticket is valid from `ValidFrom` to `ValidTo`

④ The name of the person who bought the ticket — mind `required` in combination with `init` for this property

⑤ The holder of this ticket is in the pool area if the last access event was an `Entered` event (we assume that all visitors are kicked out at the end of the day, so we will always have an exit event in this case). Why is it important to check if there is *any* element *first* before accessing the last one?

⑥ If the ticket is not valid at the given time, we can't add an access event

⑦ **The patterns in this `switch` are *non trivial*!** Take some time to understand them, especially why the *order* of the cases is super important here!

⑧ An access event is a simple `readonly record struct` with a timestamp and a type. Do you remember why it is important to make a `struct` `readonly` if feasible?

⑨ A simple `enum` to represent the possible events. As usual we assign some explicit values and leave gaps for future extensions.

4.2. Services

We split the required tasks into three services:

- `ITicketService` for handling the access to the pool area and providing tickets
- `ITicketGenerator` for creating tickets
- `IDataStorage` for storing data
    - This is just a simplification, in the future we will access a database and structure this part a bit different

For each service, we need to do three things:

1. Define the interface
2. Implement the interface
3. Register the implementation for the interface with the DI container

We also need some basic settings. Soon, we will use `appsettings` to configure the application, but for now we will hardcode the values.

Const.cs

```
namespace PoolGuard.Core;

public static class Const
{
    public const int MaxCapacity = 9; ①
    public static readonly DateTimeZone TimeZone =
DateTimeZoneProviders.Tzdb["Europe/Vienna"]; ②
    public static readonly LocalTime OpeningTime = new (08, 00, 00); ③
    public static readonly LocalTime ClosingTime = new (17, 15, 00); ④
}
```

① The maximum number of people allowed in the pool area at the same time

② The timezone of the pool area

③ The time the pool area opens

④ The time the pool area closes

4.2.1. Ticket Generator

TicketGenerator.cs

```
namespace PoolGuard.Core.Tickets;

public interface ITicketGenerator ①
{
    public Ticket GenerateTicket(string personName, Instant validFrom);
}

public sealed class TicketGenerator(IClock clock) : ITicketGenerator ②
{
    public Ticket GenerateTicket(string personName, Instant validFrom)
    {
        var validDuration = GetTimeUntilClosing();
        var validTo = validFrom.Plus(validDuration); ⑤

        return new Ticket ⑥
        {
            Id = Guid.NewGuid(), ⑦
            ValidFrom = validFrom,
            ValidTo = validTo,
            PersonName = personName
        };
    }

    private Duration GetTimeUntilClosing()
    {
        var currentTime = clock.GetCurrentInstant().ToLocalDateTime().TimeOfDay;
        var diff = Period.Between(currentTime, Const.ClosingTime) ③
```

```
                        .ToDuration();

        return diff > Duration.Zero ? diff : Duration.Zero; ④
    }
}
```

① The `interface` for the ticket generator

② The implementation of the ticket generator — *depending on* `IClock`

③ Calculate the remaining time until closing

④ If the current time is already after closing time, we fall back to `Duration.Zero`

⑤ Adding the remaining time to the current time to get the valid until time

⑥ Creating a new ticket

⑦ Generating a new GUID for the ticket ID, that way we don't have to check for existing IDs here

You may have noticed, that `ToLocalDateTime` is not found. We need to declare an extension method for that (which we will do, because we need this functionality in multiple places):

Extensions.cs

```
namespace PoolGuard.Core;

public static class Extensions
{
    public static ZonedDateTime ToLocalDateTime(this Instant self) =>
self.InZone(Const.TimeZone);
}
```

We also have to add the registration of the service to the DI container:

Setup.cs

```
public static void RegisterServices(this IServiceCollection services)
{
    // snip
    services.AddTransient<ITicketGenerator, TicketGenerator>(); ①
}
```

① The `TicketGenerator` has no state and is lightweight, so we can use a **transient** lifetime

4.2.2. Data Storage

DataStorage.cs

```
namespace PoolGuard.Core.Tickets;

public interface IDataStorage ①
{
```

```
    public Ticket? GetTicket(Guid ticketId);
    public void SaveTicket(Ticket ticket);
    public IReadOnlyCollection<Ticket> GetAllTickets();
}

public sealed class DataStorage : IDataStorage ②
{
    private readonly Dictionary<Guid, Ticket> _tickets = new(); ③

    public Ticket? GetTicket(Guid ticketId) => _tickets.GetValueOrDefault(ticketId);

    public void SaveTicket(Ticket ticket)
    {
        _tickets[ticket.Id] = ticket;
    }

    public IReadOnlyCollection<Ticket> GetAllTickets() => _tickets.Values;
}
```

① Once we back our application with a database, those operations will be *asynchronous* and return `Task` — for now, this is fine

② This `class` has *no* dependencies

③ A simple in-memory storage for the tickets. I went with a `Dictionary` here, because we will be doing lookups by ID, but a list would also work for few entries.

And the registration:

Setup.cs

```
public static void RegisterServices(this IServiceCollection services)
{
    // snip
    services.AddSingleton<IDataStorage, DataStorage>(); ①
}
```

① We need our data to be available *across multiple requests*, so we have to register it as a **singleton** — as you can see, we don't have to implement the singleton pattern ourselves, the DI container takes care of that

4.2.3. Ticket Service

TicketService.cs

```
namespace PoolGuard.Core.Tickets;

public interface ITicketService
{
    public Ticket? GetById(Guid ticketId); ①
    public Ticket CreateTicket(string personName);
```

```
    public bool TryEnterPool(Guid ticketId);
    public bool TryExitPool(Guid ticketId);
    public Statistics GetStatistics(); ②

    public readonly record struct Statistics(int CurrentVisitors, double FillLevel);
②
}

public sealed class TicketService(
    IClock clock,
    ITicketGenerator ticketGenerator,
    IDataStorage dataStorage) : ITicketService ③
{
    public Ticket? GetById(Guid ticketId) => dataStorage.GetTicket(ticketId); ④

    public Ticket CreateTicket(string personName)
    {
        var now = clock.GetCurrentInstant();
        var currentTime = now.ToLocalDateTime().TimeOfDay;

        if (currentTime < Const.OpeningTime || currentTime > Const.ClosingTime)
        {
            throw new OutsideOpeningHoursException(); ⑤
        }

        var ticket = ticketGenerator.GenerateTicket(personName, now); ⑥
        dataStorage.SaveTicket(ticket); ⑦

        return ticket;
    }

    public bool TryEnterPool(Guid ticketId) => TryAddAccessEvent(ticketId,
AccessEventType.Entered);

    public bool TryExitPool(Guid ticketId) => TryAddAccessEvent(ticketId,
AccessEventType.Exited);

    public ITicketService.Statistics GetStatistics()
    {
        int currentVisitors = GetCurrentVisitors();
        double fillLevel = Math.Round(currentVisitors / (double) Const.MaxCapacity,
2);

        return new ITicketService.Statistics(currentVisitors, fillLevel); ⑧
    }

    private bool TryAddAccessEvent(Guid ticketId, AccessEventType accessEvent)
    {
        if (accessEvent is AccessEventType.Entered)
        {
            if (GetCurrentVisitors() >= Const.MaxCapacity)
```

```
            {
                return false; ⑨
            }
        }

        var now = clock.GetCurrentInstant();
        var ticket = dataStorage.GetTicket(ticketId);

        return ticket?.AddAccessEvent(accessEvent, now) ?? false; ⑩
    }

    // In reality, we would do a count query within the data storage (e.g. database)
and _not_ load
    // all tickets into memory just to count them.
    private int GetCurrentVisitors() => dataStorage.GetAllTickets().Count(ticket =>
ticket.IsInPoolArea); ⑪
}

public sealed class OutsideOpeningHoursException() : Exception("The pool is currently
closed."); ⑫
```

① This method was not in the specification, but we will need it later when we create a ticket — you'll see

② For the statistics we declare and return a custom type — types that are just return types of the `interface` can be declared as nested to make the relationship with the `interface` clear

③ Multiple dependencies here: we need the current time, we need to generate tickets, and we need to access the storage

④ Sometimes simple service methods might just redirect to the data storage — it is still important to keep that extra layer because it makes the code more maintainable, flexible & testable

⑤ Outside the opening hours, the ticket booth is closed, so we throw an exception

⑥ Generate a new ticket

⑦ Save the ticket in the data storage (this would typically be an `async` operation)

⑧ Returning the result type, which we *need to* qualify with the `interface` name here

⑨ If the pool is full, we can't let more people in

⑩ A lot of *nullability* stuff is going on here — make sure you understand it!

⑪ **Never load everything from storage and then count in memory!** We only do this here, because we know that everything is in memory and we want to focus on other aspects.

⑫ A custom exception for the case when the pool is closed — see the nice application of primary constructors here?

If you are working on this at home (during a night shift), be careful with the configured opening hours (in `Const.cs`) ⇒ if you are outside of them, you will get an exception here! In this case, adjust the opening hours to your current time.

Don't forget to register the service:

Setup.cs

```
public static void RegisterServices(this IServiceCollection services)
{
    // snip
    services.AddScoped<ITicketService, TicketService>(); ①
}
```

① We could also declare it as *transient*, but services are typically (request) scoped, so we use *scoped* here

I recommend grouping your service registrations together in the `RegisterServices` method, so you can more easily find a specific one later, once there are *dozens* of registration calls:

Setup.cs

```
public static void RegisterServices(this IServiceCollection services)
{
    services.AddSingleton<IClock>(SystemClock.Instance);
    services.AddSingleton<IDataStorage, DataStorage>();

    services.AddScoped<ITicketService, TicketService>();

    services.AddTransient<ITicketGenerator, TicketGenerator>();
}
```

4.3. Endpoints

So far, we did not do a lot of Web API stuff, but only created some business logic and registered services. Now we will create the actual endpoints. But they need some *DTOs* to send and receive, which we will define first. If there are only a few, it is fine to put them into the file with the endpoint, but once there are many extra files are better.

Regarding naming conventions for DTOs there are many opinions. As so often with naming, the most important aspect is *consistency*. However, we will use the following conventions to keep our code comparable:

- DTOs representing a *projection* of an entity are named `<EntityName>Dto`
- Objects which are sent as a request body and are not related to an entity are named `<Action>Request`
- Objects which are sent as a response body and are not related to an entity are named `<Action>Response`

TicketEndpoint.cs

```
namespace PoolGuard.Core.Tickets;

public sealed record TicketCreationRequest(string Name);
```

```
public sealed record TicketDto(Guid Id, Instant ValidFrom, Instant ValidTo, string
PersonName) ①
{
    public static TicketDto FromTicket(Ticket ticket) => ②
        new(ticket.Id, ticket.ValidFrom, ticket.ValidTo, ticket.PersonName);
}

public sealed record StatisticsDto(int CurrentVisitors, double FillLevel) ③
{
    public static StatisticsDto FromStatistics(ITicketService.Statistics statistics)
=>
        new(statistics.CurrentVisitors, statistics.FillLevel);
}
```

① As you can see, the `TicketDto` does **not** have all the properties of the `Ticket` class — we only send what is necessary, and we might even sometimes *transform* the data. That is often called a *projection*.

② A factory method to create a `TicketDto` from a `Ticket` — this is a common pattern for DTOs, so we have a *single point of truth* for the transformation

③ Some DTOs have all the same properties as the entity they represent, but we *still* use a DTO because we want to be consistent and things may change in the future, and then we have the transformation logic in place already

TicketEndpoint.cs

```
namespace PoolGuard.Core.Tickets;

public static class TicketEndpoint ①
{
    private const string ApiBasePath = "api/tickets"; ②
    private const string GetByIdEndpointName = "GetTicketById"; ③

    public static void MapTicketEndpoints(this IEndpointRouteBuilder app) ①
    {
        var group = app.MapGroup(ApiBasePath); ④

        group.MapGet("{id:Guid}", (Guid id, ITicketService service) => ⑤
            {
                var ticket = service.GetById(id);

                return ticket is not null ⑥
                   ? Results.Ok(TicketDto.FromTicket(ticket))
                   : Results.NotFound();
            })
            .Produces<TicketDto>(StatusCodes.Status200OK) ⑦
            .Produces(StatusCodes.Status404NotFound)
            .WithName(GetByIdEndpointName); ③

        group.MapPost("", (TicketCreationRequest request, ITicketService service) =>
```

```
⑧
            {
                if (string.IsNullOrWhiteSpace(request.Name)) ⑨
                {
                    return Results.BadRequest();
                }

                try
                {
                    var ticket = service.CreateTicket(request.Name); ⑩

                    return Results.CreatedAtRoute(GetByIdEndpointName, new { id =
ticket.Id },
                                                  TicketDto.FromTicket(ticket)); ⑪
                }
                catch (OutsideOpeningHoursException)
                {
                    return Results.StatusCode(StatusCodes.Status403Forbidden); ⑫
                }
            })
            .Produces<TicketDto>(StatusCodes.Status201Created)
            .Produces(StatusCodes.Status403Forbidden)
            .Produces(StatusCodes.Status400BadRequest);

        group.MapPost("{id:Guid}/entries", (Guid id, ITicketService service) =>
            {
                bool success = service.TryEnterPool(id); ⑬

                return success ? Results.NoContent() :
Results.StatusCode(StatusCodes.Status403Forbidden);
            })
            .Produces(StatusCodes.Status204NoContent)
            .Produces(StatusCodes.Status403Forbidden);

        group.MapPost("{id:Guid}/exits", (Guid id, ITicketService service) =>
            {
                bool success = service.TryExitPool(id); ⑭

                return success ? Results.NoContent() : Results.BadRequest();
            })
            .Produces(StatusCodes.Status204NoContent)
            .Produces(StatusCodes.Status400BadRequest);

        group.MapGet("statistics", (ITicketService service) =>
            {
                var statistics = service.GetStatistics();

                return Results.Ok(StatisticsDto.FromStatistics(statistics)); ⑮
            })
            .Produces<StatisticsDto>(StatusCodes.Status200OK); ⑯
    }
```

```
}
```

① Structuring minimal APIs can be challenging. I suggest you create one `static class` for each and within that define an extension method for the `IEndpointRouteBuilder` — then put all the related endpoints in there

② The base path for all ticket related operations. We will need that `string` for some manual path construction (sadly), so let's keep it in a `const`.

③ The name of the `GetById` endpoint, we will need that later for generating a location route for created tickets

④ Defining a group applied the same base route to all endpoints within that group

⑤ Mapping a `GET` request — do you see, which of the parameters is a route parameter and which we will get via dependency injection?

⑥ Convert the `Ticket` to a `TicketDto` and return it — if the ticket is not found, we return a `404`

⑦ If we properly specify all possible responses, the OpenApi specification will be generated correctly. Mind how we can declare the type as a type parameter if a non-empty response body is expected.

⑧ Here the route is empty (translating to 'api/tickets'), but we get one parameter from the body and the other via DI

⑨ A very basic validation — we will learn how to do proper and real validation later!

⑩ Using the service to create a ticket

⑪ Here we (correctly) return a `201 Created`, with the created resource (converted to DTO). But all created responses **have** to contain an url where this resource can be retrieved as well. Sadly, with minimal API we cannot use `nameof`, so instead we rely on the `const` name of the method. And this is also the reason, why we had to implement the `GetById` method in the first place — to *correctly* implement the `201 Created` response.

⑫ We handle the expected exception `OutsideOpeningHoursException` here and return an appropriate status code. Be careful *not* to use `Results.Forbid` here, because that would require authentication (for which the `403` status is most commonly used, while we represent the pool being closed here).

⑬ Trying to enter the pool — if successful, we return `204 No Content`, otherwise `403 Forbidden`. We don't make a difference between not found and not allowed here, which is a good approach for password-like data, so it's not possible to guess existing ticket IDs.

⑭ Trying to exit the pool — if successful, we return `204 No Content`, otherwise `400 Bad Request` (because the given ticket is not in the pool area)

⑮ Returning the statistics after mapping them to a `StatisticsDto`

⑯ For this endpoint we only expect a `200 OK` response, because even if no tickets exist a valid statistics object can be created and returned

4.3.1. Generating a Route

Sometimes you need to create route, e.g. for the created result. Minimal APIs are not very good at this, because the individual endpoints don't have names (they are anonymous lambdas), so we can't use `nameof` etc. to get the route. Instead, we have to define string constants and use an extension

method to assign that name to an endpoint. That name can then be used with `CreatedAtRoute`.

This will get easier once we move to controllers, but the principle stays the same.

4.4. Testing the Endpoints

Don't forget to set up the new endpoint:

Program.cs

```
app.MapTicketEndpoints(); ①

await app.RunAsync();
```

① Calling the extension method we just created to add the group of ticket related endpoints

You can use the following `http` file requests to test the endpoints:

PoolGuard.http

```
@BaseUrl = http://localhost:5200
@Tickets = {{BaseUrl}}/api/tickets ①

### Create Ticket
POST {{Tickets}}
Content-Type: application/json ②

{
  "name": "S. Schwammal"
}

> {% client.global.set("ticketId", response.body.id); %} ③

### Get Ticket
GET {{Tickets}}/{{ticketId}} ④

### Enter Pool Area
POST {{Tickets}}/{{ticketId}}/entries

### Statistics
GET {{Tickets}}/statistics

### Exit Pool Area
POST {{Tickets}}/{{ticketId}}/exits

### OpenAPI
GET {{BaseUrl}}/openapi/v1.json ⑤
```

① We define the base route for all ticket related operations

② Don't forget to set the `Content-Type` header to `application/json` when sending JSON as `POST` body

③ This handy feature of the `http` file allows us to store the `id` of the created ticket (in the response body) in a global variable, so we can use it in later requests — **execution order matters!**

④ We use the stored `id` to retrieve the ticket we just created — **if the `ticketId` variable has not been set before this will fail!**

⑤ Your automatically generated OpenAPI specification — only available in development mode in the current setup

You now have a working WebAPI — Congratulations! 🎉🎉

5. Assignment

And now it's your turn to show what you've learned! Adjust the application as follows:

1. If someone wants a ticket, they should pay!
    - Define a price for a ticket, maybe per hour since tickets may be issued for only a part of a day
    - It is sufficient to return the amount to pay when a ticket is created — you may assume, that people will then dutifully pay that amount
2. Allow more flexibility in the ticket creation
    - Customers want tickets that last for multiple days
        - Yet they are still not allowed to enter outside of the opening hours!
    - Customers want to buy tickets in advance

5.1. Expert Challenge

- Add a unit test project to the solution
- Write some tests for the services — **figure out how to deal with the dependencies**!
    - Hint: remember that we used `NSubstitute` in 2nd grade

5.1.1. Creating a Solution

To add a second project, we need to move from a single project to a solution. You can do that manually, with your IDE, or you can try this script which makes use of the `dotnet` CLI.

<script src="https://gist.github.com/haslingerm/f788296c1dbc2240653ea57ed223b49b.js"></script>