Protocol

Who sows overengineering, reaps sorrow. This is a good starting point for discussion of the project. The core part of the app is based on OpenRoute and its services. I had some experience with it and it was not a big deal to understand how API works. So the first thing I started was the creation of **OpenRoute** layer. As it was required we use a layered architecture so it is one of the layers. It is used by the level above - by the **Application Layer** where controllers and the whole web based core resides. Why so? The idea was to push all I/O operations on the controller level to minimize dependencies on I/O on the levels below. Although we do not have in out app a strong business logic and rules it is still a good thing not to spread I/O operations. The **Repository** is also liften to the controller so that we just deliver data we need to required services and they do not care. For example this is a good idea for further **Nodes** in our **Layers** graph. Because actually layers are nothing but nodes in a directed acyclic graph where each layer can not be a parent of itself or in other words all arrows flow in one direction.

So back to the OpenRoute layer. I decided to create an **OpenRouteClient** and added as response types an **ApiResponse** type

```
public record ApiResponse<T>(bool IsSuccess, HttpStatusCode?
ResponseCode = default, T? Response = default, ErrorResponse?
ErrorResponse = null);
```

And then all my responses were polluted with this generic type. And only now I realize that it does not bring any value. It just made things look horrible. I could just create an empty type say **ApiResponse** and then create subtypes based on the semantics. For example one of the **400** responses of **OpenRoute** could signalize that there is no possible route between two points. Instead of returning **ApiResponse<..>** I coudl just inherit from the base **ApiResponse** and name the type **RouteNotAvailable**. Then we could use the **Visitor Pattern** for all possible subtypes to handle them appropriately

```
public interface OpenRouteResponse {
void Accept(OpenRouteResponseVisitor visitor);
}

public class RouteNotAvailable {
void Accept(OpenRouteResponseVisitor visitor) {
visitor.Visit(this);
}
}

public class ReveseGeocodingNotAvailable {...}

public interface OpenRouteResponseVisitor {
void Visit(RouteNotAvailable response);
void Visit(ReveseGeocodingNotAvailable response);
}
```

And then we avoid all this mess with **ifs** and **elses** for checking what to do on certain responses. Good lesson learned!

The CRUD was not that remarkable. It is just CRUD. Thanks to odata it worked fine. I also decided to update and recalculate computed properties on every **nth** update. Because leaving them as real computed properties in EfCore would mean that we will compute them every time we retrieve them. And what if we have 1000000000 tour logs? This is not that efficient. So I just added a counter to know when to recalculate the properties when it reaches **n**.

Full text search was easily configured thanks to your slides. Thank you! The rest was only the import/export and report generation. At the beginning of import export I did not spend much time on designing classes so I just created an interface with two methods **CanHandle(format)** and **Import()** for import and export accordingly. But when I started implementing second importer - this is our unique feature that users can import and export from different file format - I realized that it would be good to extract common code to the base importer/exporter. After that the amount of code in concrete xlsx or csv implementation reduced almost twice. Because the main flow logic was lifted up. For that I used the **Template Method** pattern.

```
public abstract class TourExporter(TourMapper tourMapper,
ILogger<TourExporter> logger)
{
protected string _tempExportFilePath = string.Empty;
```

```
public OperationResult<ExportResult> ExportTours(List<Tour> tours, bool
withTourLogs)
{
try
{
_tempExportFilePath = Path.GetTempFileName();
List<TourExportModel> tourExportModels = tourMapper.MapTours(tours,
withTourLogs);

WriteTourExportModels(tourExportModels);

if (withTourLogs && AnyTourLogs(tourExportModels))
WriteTourLogs(tourMapper.MapTourLogs(tourExportModels));

Save();

return OperationResult<ExportResult>.Ok(new ExportResult(new
FileStream(_tempExportFilePath, FileMode.Open, FileAccess.Read,
FileShare.Read), GetContentType()));
}
catch (Exception e)
{
logger.LogError(e, "Error during tour export");
return OperationResult<ExportResult>.Error();
}
}
public abstract bool CanHandle(string format);

protected bool AnyTourLogs(List<TourExportModel> tourExportModels) =>
tourExportModels.Any(tourExportModel => tourExportModel.TourLogs.Count
!= 0);

protected abstract string GetContentType();

protected abstract void WriteTourExportModels(List<TourExportModel>
tourExportModels);

protected abstract void WriteTourLogs(List<TourLogExportModel>
tourLogs);

protected abstract void Save();
```

```
}
```

Where we just define the main outline and make children implement **Write** method to persist tours and tour logs. Bee smart.

Finally in the **Reporting** is also tried to abstract away from pure pdf coding - yeah pdf look horrible, i am not a design person and probably did not want to spend much time with designing pdf documents. But still from the design perspective I decided to break the flow into general **Coordination, Data Provisioning,File Handling** and **Report Generation**. **Coordinator** does not bother itself how and where the data is provisioned to each type of reports. It just controls the flow. Firstly it maps the incoming type which is in such format

```
{
"reportType" : "SingleTourReport",
"payload" : "{\"tourId\" : \"28538d7f-520b-425f-a1ee-41dc9458a131\"}"
}
```

It tries to find the appropriate report type in the assembly and then populates needed for this report type files in the newly created report. For example single report needs an id so that we can find the tour in the database. After we determined the report type we try to populate report with data so that we can display it later. And here comes the visitor pattern

```
public abstract class AbstractReport
{
public string CreatedAt => DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");

public abstract ValueTask<ProvisionResult> ProvisionData(DataProvisioner
dataProvisioner);

public abstract void GenerateReport(ReportGenerator reportGenerator);
}

public interface DataProvisioner
{
ValueTask<ProvisionResult> Provision(SingleTourReport report);

ValueTask<ProvisionResult> Provision(TourSummaryReport report);
}
```

For provisioning as well as for generation we define visitors. And then for each concrete report type appropriate actions are taken. This allows us to make the code more modular and type safe. So after we provisioned our data we move back to creating a temp file + save it on our file handler so that we can retrieve it later and stream to the client and then we generate the report. Here the design is not that ideal because I assume that all reports need a file to be generated. But maybe we do not need it. So could be improved. Finally we return the result

```
public async ValueTask<ReportGenerationResult> GenerateReport()
{
try
{
ProvisionResult provisionResult = await
_report.ProvisionData(dataProvisioner);

if (provisionResult is not ProvisionedOk) return _reportGenerationResult
= new GenerationFailed();

fileHandler.GenerateFile();
reportGenerator.SetFilePath(fileHandler.FilePath);
reportGenerator.Init();
_report.GenerateReport(reportGenerator);
_reportGenerationResult = new GeneratedOk { Stream =
fileHandler.GetFileStream() };
}
catch (Exception e)
{
logger.LogError(e, "Error generating report");
_reportGenerationResult = new GenerationFailed();
}
return _reportGenerationResult;
}
```

Here we could use not only pdf reports but any type of files as reports. So quite flexible. Not that robust but for a student quite okay.

And finally unit tests. I added unit tests for OpenRoute layer and Reports, because I though that there can be more problems than in other areas. And the OpenRoute is actually our critical point so it is essential for testing. Reporting was new to me so it goes also to critical part. And import export were quite straightforward. Ideally would be to cover all projects with tests but you know... All in all I have on the backend 18 tests and we also have 6 on the frontend. I used mock library to mock dependencies for example when generating a report so that we do not generate a real report or in case of open route so that we do not make real http calls. XUnit was used as testing library.

Okay not finally - **UML**. In the uml I did not draw all the classes and did not model all the layers. This does not make sense to model **DataAccess** layer where we have only repository interfaces and their implementations. So I modelled only interesting parts - **Reports + Import/Export**. Modeling the application level also does not make much sense, because we use there classes from all layers and this will not give us any useful information.

All in all I spent 30-40 hours on the backend part and this was an interesting experience. I also touched MVVM on the frontend side and it turned out that Blazor and JS with all their event based approaches + challenges with rendering can be quite interesting. But for now I stay on the backend. However I see a lot of space for improvements in my design skills. I think this is really essential for designign flexible and robust systems.