# SOLID Pair Programming

Steps

## 1. Set Up Clean Architecture Structure

- Create solution folders and projects
  - Note: Reference from Api to Infrastructure
  - Check: Nullability is enabled. Consider: Treat Warnings as Errors
- Install Visual Studio Extension:
  - REST Client
  - Create Requests solution folder
    - Create a test request for WeatherForecast
- Remove unnecessary classes etc.
- Remove WeatherForecastController and related classes.

## 2. Implement Create Company + Repository

- Create CreateCompany.http
- Implement CreateCompanyRequest in Contracts/Companies (flat structure with 9 members)
- Implement Company + CompanyAddress + CompanyContact in Domain/Entities
  - `public Guid Id { get; set; } = Guid.NewGuid();`
- Implement CompanyResponse in Contract/Companies (structure mimicking domain Company, but with …Response in Address and Contact)
- Implement (I)CompanyService in Application/Services/Companies
  - Note: Cannot use CreateCompanyRequest in Application layer as it would violate Clean Architecture dependencies.
  - Check in repository if company with cvr exists => Exception
    - **This should probably be some proper custom exception!**
  - Create domain company instance and add to repository.
  - Return newly cerated company as property of result.
- Define ICompanyRepository in Application/Common/Interfaces/Persistence
- Implement InMemoryCompanyRepository in Infrastructure/Persistence/
- Add Infrastructure.DependencyInjection.cs
- Add Application.DependencyInjection.cs
- Add Services to Program.cs
- Run and test.

## 3. Introduce Created Time in Company

- Create property in domain Company
  - `required public DateTime Created { get; init; }`
- Create IDateTimeProvider in Application/Common/Interfaces/Services
- Implement DateTimeProvider in Infrastructure/Services/DateTimeProvide
- Update Infrastructure/DependencyInjection
- Inject IDateTimeProvider into CompanyService
- Timestamp creation of domain Company objects
- Run and test.

## 4. Error Handling using Problems

- Client needs to know problems structure:
  - Problem RFC: https://www.rfc-editor.org/rfc/rfc7807
  - `application/problem+json`
  - See top of Page 4 Example: (Blue = standard, Green = custom)

```
{ "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30,
  "accounts": ["/account/12345",
               "/account/67890"]
}
```

- In Program.cs: UseExceptionHandler("/error") [Ep4, 12:25]
- In Api create ErrorsController with that route
- Set launchBrowser to false in launchSettings.json
- Run and produce error
  - See 500 with correct uri
- Change statusCode to other code, e.g. 422
  - See 422 with correct uri
- Fetch exception from HttpContext and it Problem.Title
  - `HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error`
  - (HttpContext can be injected via IHttpContextAccessor for testability purposes)
- Create a DuplicateCompanyException in Application/Common/Exceptions and use it in service
- Create (statusCode, title) switch from Exception type
  - Not too nice though…
  - Many different solutions exist. But should not have HTTP error codes in Application or Domain!

## 5. ErrorOr and not Exceptions

- A solution is ErrorOr package by Amichai Mantinband [Ep5, 19:14]
  - Add ErrorOr package to Domain
- Create Domain/Common/Errors folder
  - Define DuplicateCvr in Errors.Company.cs
  - Show what the package Error class consists of
- Update the CompanyService with the ErrorOr definition all the way up.
- Use the Match() method in the CompanyController to distinguish between result and error
- Define abstract Application/Controllers/ApiControllerBase
  - Problem(List<Error>)
  - Problem(Error)
  - Move [ApiController] attribute to base

## 6. CQRS

- First add a Get all companies to get the query side
  - CreateCompanyResult -> CompanyResult so that it can be reused
  - Implement GetCompanyByCvrRequest

    - Implement new CvrNotFound in Domain/Common/Errors class
    - Add method to CompanyController
    - Add method to CompanyService
    - Create GetCompanyByCvr.http
    - Run and test
        - Run GetCompanyById.http query first and see 404.
        - Run CreateCompany.http and see 200
        - Run GetCompanyById.http query again and see 200.
- Note: This way the services will slowly become God Class! ☹️
    - And everything is wired up explicitly in an almost suffocating way.
    - Need to creates smaller "services" => **"Service pr. Use Case" => CQRS!**
- First Step: Split CompanyService into Commands and Queries folders and services
    - Note: Contracts are Api <-> Application, so they do not change!
    - Inject both services into controller. Update DependencyInjection etc.
    - Run and test
        - Run GetCompanyById.http query first and see 404.
        - Run CreateCompany.http and see 200
    - Run GetCompanyById.http query again and see 200.
- Note: Communication is still very explicit and ugly around controller.
- Second step: Introduce CQRS and MediatR to **"Split Logic by Feature"**
- Add Application/Companies/{Commands, Queries, Common} folders
- Install MediatR package
- Create GetCompanyQuery : IRequest<ErrorOr<CompanyResult>>
- Create GetCompanyQueryHandler
  : IRequestHandler<GetCompanyQuery, ErrorOr<CompanyResult>>
- Modify CompanyController to send via MediatR
    - Watch out with Task.FromResult and if-statements might be needed
- Note: No handlers need to be explicitly injected or registered
    - MediatR.Extensions.Microsoft.DependencyInjection package for AddMediatR(Assembly)
- Create CreateCompanyCommand : IRequest<ErrorOr<CompanyResult>>
- Create CreateCompanyCommandHandler
  : IRequestHandler<CreateCompanyCommand, ErrorOr<CompanyResult>>
- Modify CompanyController to send via MediatR
    - Watch out with Task.FromResult and if-statements might be needed
- **Remove *all* occurrences of services!!**
- Finally: IMediator -> ISender

## 7. Mapping Discussion

- To do things very cleanly, we could/should map between layers
    - Either use AutoMapper, Mapster or similar
    - Or write explicit mapping code, e.g.
        - Constructors
        - Extension Methods
        - Operators
- Be careful about exposing Domain objects directly:
    - We should have stopped the Domain object Company to be sent directly, so…

- Create Contract/CompanyResponse mapping from Domain/Company for the purpose of returning an “independent” structure to clients.

## 8. Fluent Validation via Pipeline Behaviors

- In Application
  - Create ValidateCreateCompanyCommandBehavior as IPipelineBehavior<CreateCompanyCommand,ErrorOr<CompanyResult>>
- Register behavior in Application/DependencyInjection
- Run and check that it is being hit when requests arrive
- Add the FluentValidation package
- Create CreateCompanyCommandValidator
- Register validator in Application/DependencyInjection
  - Can be done for all validators using the package FluentValidation.AspNetCore:
  - **services.AddValidatorsFromAssembly(typeof(DependencyInjection.Assembly)?**
- Inject CreateCompanyCommandValidator into ValidateCreateCompanyCommandBehavior
  - ConvertAll errors from FluentValidation errors into ErrorOr errors:

```
Error.Validation(
    code: failure.PropertyName,
    description: failure.ErrorMessage
)
```

- Run with missing parameters and check that we get we get errors.

- Refactor to a generic ValidationBehavior<TRequest, TResponse>
  - IErrorOr constraint and dynamic conversion
- Update Application/DependencyInjection with

```
.AddScoped(
   typeof(IPipelineBehavior<,>),
   typeof(ValidateBehavior<,>)
)
```

- Make sure that the validator might not be present, so make it nullable (and default to null!)
- Run with missing parameters and check that we get we get errors.

### Model Validation Errors Integration [Ep8, 17:08]

- Can create reusable helper methods for getting model validation errors integration
- If all errors are validation errors, use the ControllerBase.ValidationProblem() method!
- Refactor to three methods inside ApiControllerBase
  - Problem(List<Error>)
    - Calls ValidationProblem(List<Error>) is all errors are Validation
    - Calls Problem(Error) with the first error otherwise
    - Call Problem() if errors is empty
  - Problem(Error)
    - As before
  - ValidationProblem(List<Error>)
    - Convert ErrorOf errors to ModelStateDictionary and calls built-in ValidationProblem(ModelStateDictionary)
- Run and check that model state validation errors are returned.

### Easy Validation of GetCompanyQuery

- Create GetCompanyQueryValidator

- Check that it automatically is activated and works as soon as it is added!
- Run and check that model state validation errors are returned.

Y. Implement API Key Autorization

Y. CompanyLogo get/set with caching (abstract file system)

Y. Unit + Integration Tests 😊

Y. Towards DDD -> Structured Types / Value Objects, e.g. Phone, Email, …

Y. “Real” repository saves demand Unit of Work pattern