FUNCTIONAL VALIDATION

Improving service design with functional concepts

Contents

The problem

API contracts are the base model

- Modeling the data structures is minimal with API contracts often being the only representation
- API contracts are often designed in a way that makes them easier to
  - consume them from the frontend application
  - copy when automation doesn't provide reusable contracts

This leads to various issues with the design of inner layers.

Inconsistent state

- Contracts are based on the exact structure of frontend input model
- Once a contract is used, changing it is a tedious process
- Backend models do not have invariants encoded in them
- Constant possibility of inconsistent state

```C#
// Optional data
public class InsuranceApplicationRequest {
    public bool CustomerHasIssue { get; set; }
    // This must not be NULL when CustomerHasIssue is true
    public int? IssueYear { get; set; }
    // This must not be NULL when CustomerHasIssue is true
    public string IssueReason { get; set; }
}
```

Inconsistent state

- Contracts are based on the exact structure of frontend input model
- Once a contract is used, changing it is a tedious process
- Backend models do not have invariants encoded in them
- Constant possibility of inconsistent state

```C#
// Alternatives that depend on a certain property
public class PurchaseRequest {
    public CustomerCategory CustomerCategory { get; set; }
    // This is only present if CustomerCategory is Category1
    public decimal? CategoryOneAmount { get; set; }
    // This is only present if CustomerCategory is Category2
    public decimal? CategoryTwoEstimate { get; set; }
    // This is only present if CustomerCategory is Category2
    public string? CategoryTwoJustification { get; set; }
}
```

Solution v1: App layer models and FluentValidation

App layer models

- We leave existing API contracts untouched and focus on providing a consistent data structure for application layer
- API service's responsibilities are:
  - validation of API contract instances
  - mapping those instances to application layer models

Optional data: nullable type with required non-nullable members

C#

```
public class InsuranceApplication {
    public CustomerIssue? CustomerIssue { get; set; }
}

public class CustomerIssue {
    // No nulls
    public required int Year { get; init; }
    // No nulls (if we do things right AND maintain the course)
    public required string Reason { get; init; }
}
```

Alternative data: “ADTs” and maintaining requirements in subtypes

```C#
public abstract record Customer {
    public sealed record CategoryOne(decimal Amount): Customer;

    public sealed record CategoryTwo(decimal Estimate, string Justification): Customer;
}
```

App layer models

C#

```
public record Purchase {
    public required Customer Customer { get; init; }
}

// Encoded invariants allow us to be certain what the state is
var amountForCalculations = purchase.Customer switch {
    CategoryOne co => co.Amount,
    CategoryTwo ct => ct.Estimate * 0.9
}

var justificationForNotProvidingExactAmount = purchase.Customer switch {
    CategoryOne co => "",
    CategoryTwo ct => ct.Justification
}
```

Validation part of API service responsibilities covered by FluentValidation package.

```C#
public class RequestValidator: AbstractValidator<InsuranceApplicationRequest> {
  public RequestValidator() {
    When(r => r.CustomerHasIssue), () => {
        RuleFor(iar => iar.IssueYear).NotNull();
        RuleFor(iar => iar.IssueReason).NotEmpty().Length(1, 200);
    };
  }
}
```

C#

```
public class RequestValidator: AbstractValidator<PurchaseRequest> {
  public RequestValidator() {
    When(r => r.CustomerCategory == CustomerCategory.CategoryOne), () => {
        RuleFor(iar => iar.CategoryOneAmount).NotEmpty().Max(200_000);
    };

    When(r => r.CustomerCategory == CustomerCategory.CategoryTwo), () => {
        RuleFor(iar => iar.CategoryTwoEstimation).NotEmpty().Max(100_000);
        RuleFor(iar => iar.CategoryTwoJustification).NotEmpty().Length(1, 200);
    };
  }
}
```

Implicity as a curse

- FluentValidation doesn't affect data in the API contract instances
- After successful validation we need to map the API contract to model
  - Choice 1: we rely on FluentValidation doing its job and map assuming no inconsitencies
  - Choice 2: we check all of the rules again and keep those checks in sync with validator implementations
- Real implementations we have use a somewhat hybrid approach

Solution v2: Validation as transformation

General idea

- API contract instance is validated and transformed at the same time
- Result type contains:
  - a valid transformed app layer model when validation is successful
  - “list” of issues when validation failed
- The described set of alternatives implies all errors are blocking
  - It’s possible to implement a variation with non-critical issues
  - Real use-cases predominantly follow error-only pattern

Sketching the data structure

C#

```
public abstract record Validation<E, A> {
  public sealed record Valid(A Value): Validation<E, A>;
  public sealed record Invalid(List<E> Errors): Validation<E, A>;
}

public static class Validator {
  public static Validation<string, InsuranceApplication> Validate(InsuranceApplicationRequest req) {
    /// implementation here
  }
}
```

Functional requirements

- `Invalid` should contain all of the errors we could find in the validation
- We can explicitly operate on validated pieces of information (no rechecks)
  - Modern C# provides some ad-hoc type guards but it's not enough
- Errors should be linked to data that triggered them

JSON

```json
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "MainApplicant.DateOfBirth": [
      "Applicant must be at least 18 years old and at most 100 years old"
    ]
  },
  "traceId": "00-4fc2b47a18250f1003cf0646b5803f8d-27e604cb6e0c29db-00"
}
```

LanguageExt

- LanguageExt is a library which implements sort of “functional BCL” for C#
- Provides a wide variety of functional data structures and behaviors
- We’re focusing on `Validation<F, A>` type here but a few others are also relevant

Short detour: `Semigroup`

- `Semigroup` defines an "addition" operation

```C#
public interface Semigroup<A> where A : Semigroup<A>
{
  A Combine(A rhs);
}
```

- Example: integers and +, strings/lists and concatenation
- `Combine` allows for merging values into one we can return in case of failure

Short detour: `Monoid`

- `Monoid` is a `Semigroup` with a neutral/empty element

```C#
public interface Monoid<A> : Semigroup<A> where A : Monoid<A>
{
  static abstract A Empty { get; }
}
```

- Example: integers and +, strings/lists and concatenation
- Positive integers and + aren't a monoid
- `Empty` allows to start with "zero" and merge other instances into it i.e. aggregate/reduce

LanguageExt's Validation

```C#
public abstract record Validation<F, A>() where F : Monoid<F> {
  ...
}
```

- Technically `Semigroup` is enough: we only need to combine errors we return
- LanguageExt adds a `Monoid` constraint to implement filtering for `A`
  - if filtering fails but there were no errors we return `Monoid<F>.Empty`
- LanguageExt provides `Error` type which implements `Monoid<Error>` for this purpose

Combining successful results

```C#
public class Customer(string FirstName, string LastName);
...
Validation<Error, string> firstName = ValidateFirstName(rawFirstName);
Validation<Error, string> lastName = ValidateLastName(rawLastName);

// How do we extract and use the validated values?
```

Short detour: `Applicative`

- Classic definition is a bit too abstract to dive into here
- Detailed explanations: Scala Cats, Paul Louth's blog
- Practical application: allows to "map" over multiple "containerized" values at once

C#

```
Validation<Error, string> firstName = ValidateFirstName(rawFirstName);
Validation<Error, string> lastName = ValidateLastName(rawLastName);

var customer = (firstName, lastName)
  // fn/ln here are strings
  // new Customer is only called if both tuple items are Valid
  // otherwise a new Invalid is created
  // Errors are merged by Combine provided by Semigroup/Monoid
  .Apply((fn, ln) => new Customer(fn, ln));
```

Dealing with value paths

- If we want to maintain level of information similar to FV, we need property paths

```JSON
"errors": {
  "MainApplicant.DateOfBirth": [
    "Applicant must be at least 18 years old and at most 100 years old"
  ]
}
```

Dealing with value paths

```C#
public record ValuePath {
  private readonly List<ValuePathSegment> _segments;

  public static ValuePath Root => new([]);

  private ValuePath(List<ValuePathSegment> segments) {
    _segments = segments;
  }
  //...
  public ValuePath Combine(string path) {
    return new ValuePath([
      .._segments,
      new ValuePathSegment.Raw(path)
    ]);
  }
```

Dealing with value paths

```C#
public abstract record ValuePathSegment {
  public sealed record Raw(string Path) : ValuePathSegment;
  public sealed record ExpressionBased<T>(Expression<Func<T, object>> PathExpression): ValuePathSegment;
}
```

- `ValidationError` has a property of type `ValuePath`
- Nested validation calls accept a copy of `ValuePath` with added property name
- Mapping `ValidationError` to a problem details item uses `ValuePath` to generate the error key

Demo

Demo repository on Github

- Classic FluentValidation-based implementation
- A variation of the endpoint where validation is not enabled
- Basic `Validation`-based implementation (`LanguageExt.Common.Error` as error monoid, lax domain)
- `Validation`-based implementation with strict domain and `Vogen` library for custom “value object” types
- Same but with `Seq<ValidationError>` as error monoid for a stricter result

Conclusions

Advantages

- Explicit expectations: you deal with `Validation<E, A>`, not `A` with it quietly failing
- Single implementation for validation and mapping
  - Strict modeling is easier now
  - Models are in app or domain layer, so encoded constraints are usable everywhere
- Unit tests are stricter: type system makes assertions direct

Advantages

- The approach is not in any way tied to WebAPIs
  - *Counterargument*: neither is FluentValidation. I'm yet to see it used anywhere else though, might be because of lack of experience
- LanguageExt doesn't need to spread beyond the API layer
  - On the other hand, it certainly helps apply stricter principles on a larger scale

So why don't we do this already?

Object-oriented habits

- Classic OO practices don't make use of these abstractions
- Understanding functional concepts requires cognitive effort
- Modern C# is slowly going somewhere in that direction:
  - Nominal type unions

    `Validation`/`Option` etc. are easier to implement that way
  - "Generic math": IAdditionOperators + IAdditiveIdentity≈ Semigroup + Monoid
  - Built-in implementation is unlikely, see related GH discussion

So why don't we do this already?

Risk/reward balance

- Introducing a core non-MS third-party dependency is hard (case in point: JodaTime/NodaTime)
- Proper use of libs like LanguageExt is as widespread in code base as BCL types (e.g. Tasks, Nullable/NRT)
- Ecosystem around it is not really there
  - Unit testing is, while robust, not eased with ready-to-use helpers
- Licensing has recently become a widely discussed issue
  - FluentAssertion is the most obvious example
  - AutoMapper, MediatR are a bit worse impact-wise but for now not as disruptive

So why don't we do this already?

LanguageExt is a liability for a commercial project

- Technically under MIT license
- Author plans to introduce commercial licensing (see GH discussion), no idea about actual pricing yet
- After a major rework the latest version is a beta version (author treats it as closer to RC with a longer test period)

Going forward

- API contracts will always remain a pain due to interdependencies
- Solution v1 is what we do now in new services
- Solution v2 doesn't have to depend on LanguageExt
  - No similarly convenient alternatives
  - Reusable ad-hoc implementation is not complex provided we see use in it
  - Usage of value objects/Vogen is orthogonal to validation

Going forward

- Understanding the problem is a requirement for considering solutions
- Scope of adoption matters

Thanks!