Excerpt from sample chapter
"Domain Driven Design: a Functional Approach using F#"
by Scott Wlaschin

Chapter 4: Building a complete use-case

In this chapter, we'll take one use-case from our order-taking example and design it using the functional principles of composition that we developed earlier.

A business process is often thought of as a series of document transformations, and we'll see we can model the use-case in the same way: as a "pipeline" built from a series of smaller "pipes" using composition.

We'll see that each step in the pipeline will be designed to be stateless, which means that they can be tested independently.

As part of this implementation we'll see how to do error handling in a functional way (no exceptions!) and how common error cases can be included in the domain model.

We'll also look at how to process pipelines in parallel rather than in series, and finally we'll look at how to integrate promises/async into the pipeline.

Determining the components of the use-case

As we saw earlier, our order taking system stores a "customer profile" containing a customer's name, email address, and delivery address.

We will focus on the use-case where we have an existing customer, and they want to update their profile with new information.

The steps in this scenario are:

- Validate the request coming in from the API
- Ensure the data is cleaned up, e.g. by trimming the name and email, etc.
- Update the customer profile in the database
- If the email has changed, send out a message asking them to confirm their new email.

To implement this, we will create a function for each step (such as `validateProfile`, `updateDatabase`) and then compose them into a single function like this:

Here's how that pipeline would be written as code:

```
let updateCustomerProfile profile =
  profile
  |> validateProfile
  |> cleanUpProfile
  |> updateDatabase
  |> sendEmailVerificationMessage
```

That's easy enough. But now the question is: how do we create the smaller functions, and what are their inputs and outputs?

Domain types vs. DTOs

Let's start by looking at the input for the entire pipeline: the customer profile request.

We saw that this customer profile request could be modeled with the following types:

```
type Name = {
  First: FirstName
  Last: LastName }

type DeliveryAddress = {
  Street: StreetAddress
  City: CityName
  Zip: ZipCode
  State: StateCode }

type UpdateCustomerProfile = {
  Id: CustomerId
  Name: Name option
  EmailAddress: EmailAddress option
  DeliveryAddress:  DeliveryAddress option }
```

And recall that `FirstName`, `EmailAddress`, `CustomerId`, etc., are not just simple strings, but constrained values that are guaranteed to be valid.

Now these types model the domain very nicely, but there is a problem – they cannot be modeled in JSON or serialized very easily. If we want to able to make the request via an API we will need *another* type to represent the serializable request.

This type is commonly called a DTO ("data-transfer object") and is designed using the following guidelines:

- Only primitive types such as strings and ints can be used.
- Choice types (such as `ContactEmailAddress`) are represented with flags

Applying those guidelines to this type, the DTO type will look like this:

```
type UpdateCustomerProfileDTO = {
  Id: int
  First: string
  Last: string
  EmailAddress: string
  Street: string
  City: string
  Zip: string
  State: string }
```

So now we have to add another task to our list: how can we convert this ugly DTO type into the elegant domain types? This will have to be the first step in our pipeline.

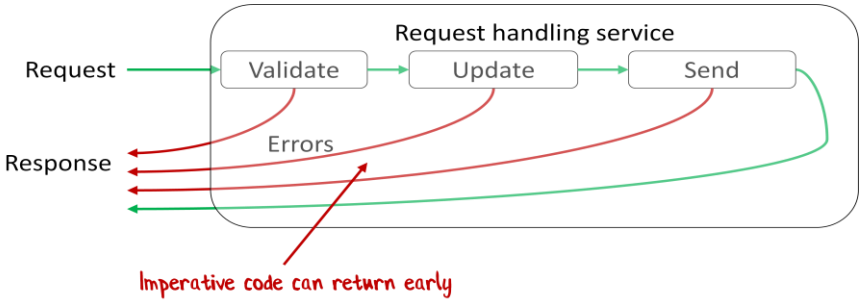
Error handling

Before we address the conversion process, we need to address how to handle errors.

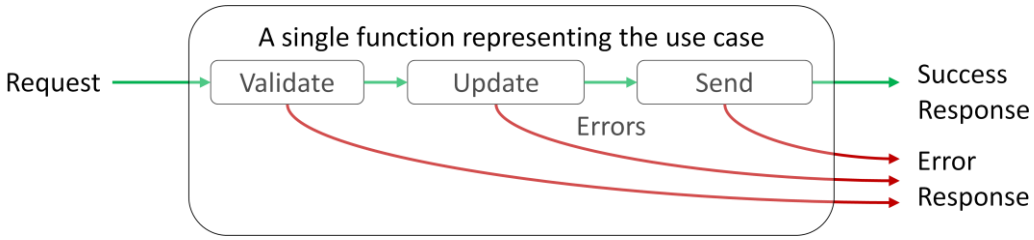
We tend to focus on the happy path, and assume that everything goes well. But what happens if something goes wrong in the pipeline, such as:

- The validation fails and we can't create the domain type.
- The database returns an error.
- The email server cannot be found, or fails to authenticate.

In a non-functional language, we can do early returns or throw exceptions, like this:



But in a functional language, we cannot return early – we have to keep going to the end of the pipeline.



How can this be done? How can a function have more than one output? And how can you bypass downstream functions when an error happens?

The functional approach to error handling

Let's start by looking at the first question: how can a function have more than one output?

What we're saying is that the output of a function is either a success or a failure, where the failure case is one of the possible errors.

Well, that's easy to model. We just need to use a "choice" type, where the choices are "Success" or "Failure". Here's the definition of a type called `Result` with those two choices:

```
type Result<'SuccessValue,'FailureValue> =
  | Success of 'SuccessValue
  | Failure of 'FailureValue
```

In the success case, we return a valid of type `'SuccessValue` and in the failure case, we return a value of type `'FailureValue`. The tick means that these are generic types: we can reuse this `Result` definition with any success type and any failure type.

How do we use this in code to signal whether the function succeeded or not? First, we might define a list of validation errors such as these:

```
type ValidationError =
    | FirstNameMustNotBeBlank
    | LastNameMustNotBeBlank
```

And then a name validation function might look something like this:

```
let validateFirstName name =
    if String.IsNullOrWhiteSpace(name) then
        Failure FirstNameMustNotBeBlank
    else
        Success (FirstName name)
```

The signature of `validateFirstName` shows that it takes a `string`, and emits a choice of either a valid `FirstName` or a `ValidationError`:

```
val validateFirstName : string -> Result<FirstName,ValidationError>
```

This is a nice approach – the possible errors are explicitly enumerated and documented in the function signature.

Compare this to a more traditional exception throwing approach:

```
let validateFirstName name =
    if String.IsNullOrWhiteSpace(name) then
        failwith "First Name Must Not Be Blank"
    else
        (FirstName name)
```

In this case, the function signature would look like this:

```
val validateFirstName : string -> FirstName
```

We discussed why using exceptions was a bad idea in chapter 2. It's deceptive! This function does not always return a `FirstName` – sometimes it throws an exception instead. The function signature is not being explicit about what it does and is not self-documenting.

So, for domain-driven error handling, do use the "Success/Failure" approach. It signals that (a) the function returns errors and (b) the possible errors are documented in the type signature.

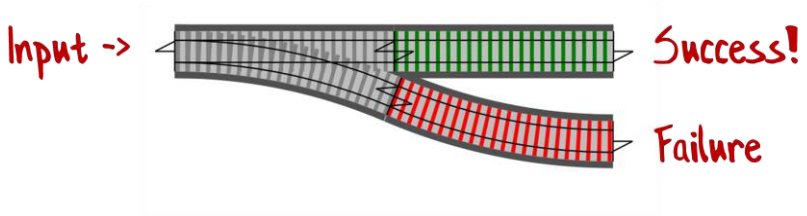
Note that this approach is not suitable for *all* errors, though. Some errors, such as out-of-memory exceptions or null-reference exceptions, are not part of the domain and do not need to be modeled. They should just be thrown and caught at the top level.

Chaining "error generating" functions together

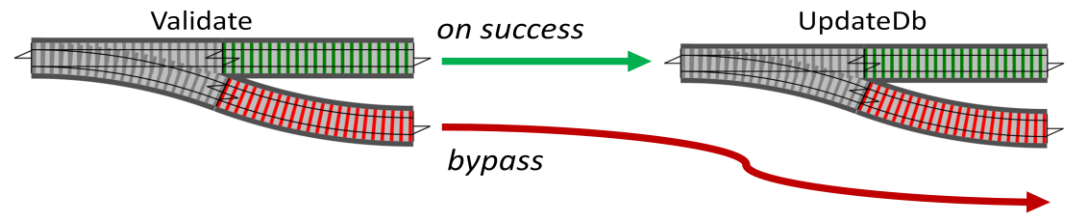
Now we can address the other question about error handling: how can you bypass downstream functions when an error happens?

Let's say that we have a `Validate` function that returns a "Success/Failure" `Result` and an `UpdateDb` function that returns a `Result`. And say that the `UpdateDb` function should be called when `Validate` succeeds, but bypassed when it fails. How can we represent that?

Well, first let's extend our earlier railway track analogy. We can represent an error generating function as a "switch" or "points", like this:



And then the problem of connecting the two functions becomes the problem of connecting two switches, like this:



Anyone who has ever played with a toy train set knows how to do this! We just need to connect the failure track from `Validate` to the failure track of `UpdateDb`, like so:

What we're doing here is converting the `UpdateDb` function from a switch with *one* input into a function with *two* inputs.

This is surprisingly easy to do in a few lines of code. We can convert any one-input switch function into a two-track function using a generic function called "bind" which is written like this:

```
let bind f result =
    match result with
    | Success value -> f value
    | Failure err -> Failure err
```

This `bind` function takes an error-generating function "f" and a `Result` "result" as parameters. If the result parameter is a success, the function "f" is called with the value associated with the success. On the other hand, if the result parameter is a failure, the function "f" is bypassed and the original error is returned.

In this way, you can chain a series of error generating functions together.

If we now go back to our original `updateCustomerProfile` function, and if we assume that all the steps are error-generating functions, we could adjust the implementation to look like this:

```
let updateCustomerProfile profile =
  profile
  |> bind validateProfile
  |> bind cleanUpProfile
  |> bind updateDatabase
  |> bind sendEmailVerificationMessage
```

[Excerpt ends]