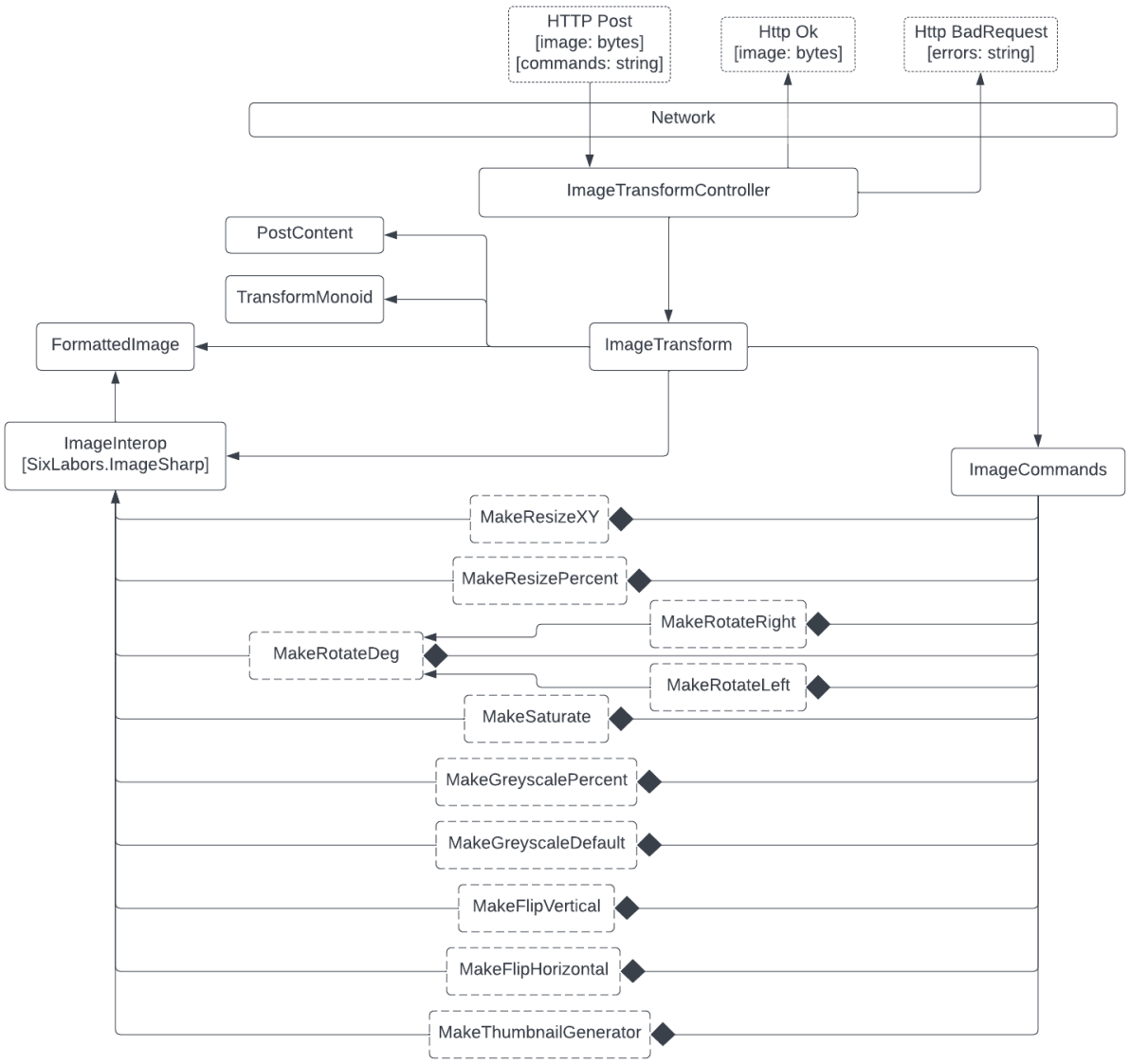
High-Level Summary:

ImageTransformer is a client-server web API built in the ASP.NET Core framework with F#. Clients submit an image file as a byte array and a list of string commands according to a specified language, and the server applies transformations based on those commands in the order supplied and then returns the image to the client in an HTTP 200. If something goes wrong, the server returns error data in an HTTP 400.

Structural Overview:



This is a somewhat hybrid diagram of ImageTransformer's structure, with respect to POST messages. Solid boxes are modules, dotted boxes are messages, and dashed boxes are functions.

ImageTransformer is built on top of ASP.NET Core's MVC web api template, using the F# language. F# allows us to focus on the core data transformation functionality of the service, and write highly compact and maintainable code (for functional programmers). A Program module (not shown) is responsible for

spinning up the system with some boilerplate, but ImageTransformController is responsible for all execution in response to incoming messages. The controller functions asynchronously to maintain responsiveness.

On receiving a POST, the controller automatically unpacks the post body into a PostContent object, which it gives to ImageTransform. A PostContent is a simple record type that contains an image as a byte array and commands as a string array.

ImageTransform is responsible for converting the list of commands into a single transform function, and then applying it to the image. For the commands, it relies on pattern matching to select functions from the ImageCommands module, the functional structure of which is shown above to explicitly include the required image transforms (which are all ultimately backed by library functions in the interop module). Some input validation is also performed in the ImageCommands module for commands that take argument(s), for parseability and bounds checking.

ImageTransform uses the TransformMonoid module to simplify error handling and transform composition.
TransformMonoid contains three types – the titular TransformMonoid and two convenience aliases named according to railway-oriented programming patterns.

- ImageSwitch aliases (Image -> Result<Image, Error>)
- ImageTwoTrack aliases (Result<Image, Error> -> Result<Image, Error>)
- TransformMonoid is a type that boxes a Result<ImageTwoTrack, string>

When ImageTransform matches commands against the ImageCommands module, each command becomes a Result<ImageSwitch, string>, which is convertible to a TransformMonoid (Switches are bound into TwoTracks). At that point, the whole list of TransformMonoids can be summed to produce a single Result<ImageTwoTrack, string>, which is either the composite transform or a string of newline-delimited command parsing errors.

To work with the submitted image, ImageTransform loads the image from the submitted byte array using a function from the interop module. This function actually packages the image into a FormattedImage record (which holds the image and a format specifier used later to convert back to bytes), but the FormattedImage module also contains a map method that converts any ImageTwoTrack into a (Result<FormattedImage, a'> -> Result<FormattedImage, a'>), so we don't have to care.

Once it has the image, ImageTransform applies the composite transform and saves the resulting image to bytes, giving a Result<MemoryStream, Error>, which is returned to the controller. The controller unpacks the Result with pattern matching, and either sends the stream contents back to the client as a JSON-encoded byte array in an OK, or sends the errors as a string in a Bad Request.

Not shown, the server will respond to

- GET requests to / with a Swagger UI.
- GET requests to /commands with plaintext instructions on the command syntax.

<u>Messages</u>:

Clients request image transformations by sending an HTTP POST message, whose body is a JSON-encoded object containing a string array of commands and a byte array of an image.

In .NET, clients are advised to make the image byte array with File.ReadAllBytes({path}) and the commands string array with File.ReadAllLines({path}). The message body can be constructed from these with System.Net.Http.Json.JsonContent.Create<T'>({a T'}), where T' is a type containing a field of string[] named commands and a field of byte[] named image, and sent with a POST method in System.Net.Http.HttpClient.

Clients may be implemented in other languages, but the structure of the JSON object is required for the server to correctly decode the request.

ImageTransformer will automatically recognize common image formats[1] and return the result in the same format. For an OK response, the client should deserialize the body as JSON into a byte array, which can be directly saved to a file with the same extension as the original. For a Bad Request response, the client should deserialize the body as a string, may be a single error or a newline-delimited sequence of errors. The sample client prints errors directly to the console.

If there is an error decoding or processing the image, only that error will be returned. If there are any errors with command formatting, they will all be returned.

ImageTransformer listens for POST messages on https://{host}:7261/imagetransform
Responses are returned over the same connection.

Commands:

The expected syntax for image commands follows, by requirement, quotes not included. Commands are case-insensitive, and each line is trimmed server-side. Commands are interpreted such that there is no possibility of injection of any type.

- Flip horizontal and vertical
  - "flipH"
  - "flipV"
- Rotate +/- n degrees
  - "rotate {degrees}"
    - degrees: unbounded float (clockwise, processed modulo 360)
- Convert to fixed grayscale
  - "greyscale"
  - "grayscale"
- Convert to a user-specified grayscale
  - "greyscale {percent}"
  - "grayscale {percent}"
    - percent: float in [0, 100]
- Saturate / desaturate
  - "saturate {percent, nonnegative float}"
    - < 100 will desaturate, >100 will oversaturate
- Resize (x, y)
  - "resize {width} {height}"
    - width and height: nonegative integer (of pixels). At most one may be zero.

[1] ImageSharp automatically recognizes Bmp, Gif, Jpeg, Pbm, Png, Tga, Tiff, and Webp

        - If with or height is zero, it is calculated relative to the other to maintain the original aspect ratio.
- Resize percentage
    - "resize {percent}"
        - percent: positive float
- Generate a thumbnail
    - "thumbnail"
        - Performs a resize 0 177, but with a thumbnail-specific resampling algorithm.
- Rotate left
    - "rotateLeft"
        - Alias for rotate -90
- Rotate right
    - "rotateRight"
        - Alias for rotate 90

Commands must be received as an array of these command string forms. The sample client reads commands by line from a text file.

This syntax is also available through the service with a GET request to /commands

ImageSharp:

All direct image-processing functions in ImageTransformer are backed by functions from the ImageSharp[2] library. The client does not need to use or even be aware of this library, since all processing takes place on the server.

Some implementation notes:
- Greyscaling uses the ITU-R Recommendation BT.709 filter.
- Resizing, except for thumbnails, uses a general-purpose Bicubic resampling algorithm.
- Thumbnail resizing uses a Lanczos3 resampling algorithm, which produces sharper thumbnails.
- Rotation is measured in degrees clockwise.

Explicit design styles and patterns:
- Railway-oriented programming[3]
    - A functional programming paradigm that replaces OO exception throwing by propagating an elevated type (here, Result) through the full program, using binds and maps to elevate intermediate functions to operate on the elevated type.
- Higher-order functions
    - Rather than use Command-pattern objects, in a functional language we can return functions directly. We see this most prominently in ImageTransform.parseCommand, which we use to map a sequence of strings into a sequence of ImageSwitch.
- Function composition
    - In a functional language, functions can be composed into composite functions that hide the details of their components. ImageTransform uses composition to assemble a

[2] https://docs.sixlabors.com/articles/imagesharp/index.html
[3] Term borrowed from https://fsharpforfunandprofit.com/rop/

number of utility functions, and TransformMonoid uses composition in its addition overload.

- Monoid
    - A monoid consists of a type, an associative combining operation that takes in two elements of that type and outputs an element of that type, and an identity element under that operation.
    - Monoid types are used in functional languages to enable simple list comprehension of complex objects.
        - In ImageTransformer, the subtlely-named TransformMonoid is a monoid. Structurally, it's a box for a Result<ImageTwoTrack, string>, and its additive rule either composes two ImageTwoTracks (for Ok + Ok), concatenates two strings (for Error + Error), or just keeps the string (for one of each). The zero member is the identity function, elevated into a two-track with map.
        - During command parsing, the list of ImageSwitch is mapped to TransformMonoids and summed to produce the overall transform. The top-level code is deliriously compact to eyes used to OOP.
- Library-in-a-box
    - Aside from a couple of type references in FormattedImage, all code that directly references the image library resides in the ImageInterop module. Should we need to change the image processing library later, this pattern greatly simplifies finding all affected code.
- Interpreter (sort of)
    - The command syntax above can be considered a little language, even if it has very little syntax and is 'interpreted' by a match statement.