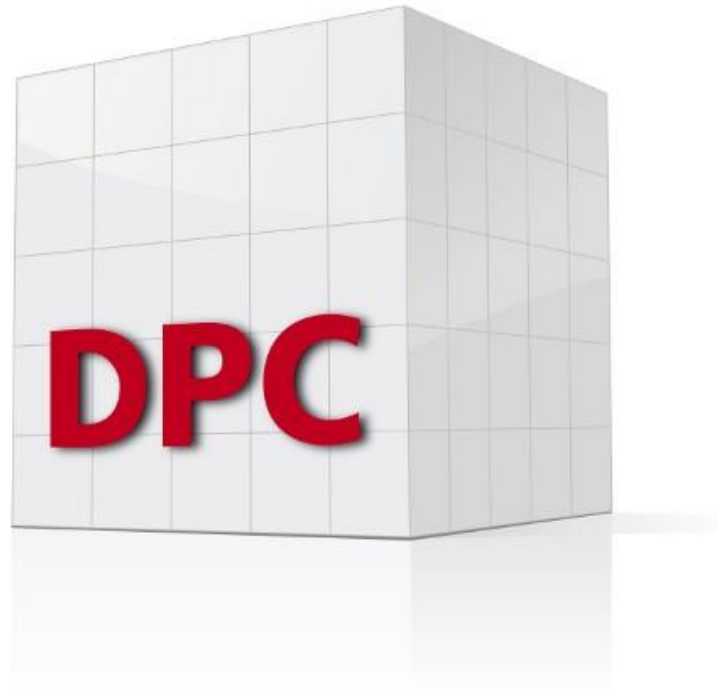
Converting a COM Server to a Jarvis-based Web Service



Finn Flug - DPC

Agenda

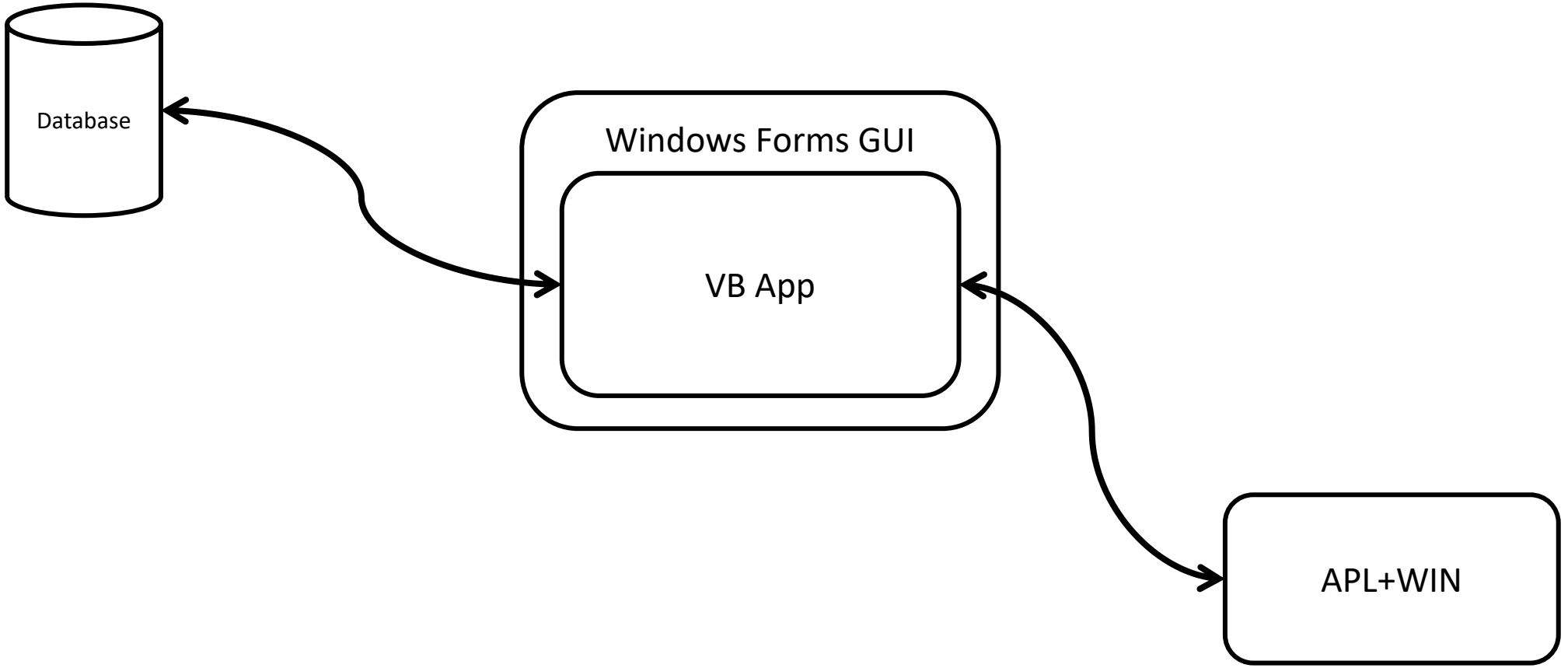
- Overview of the system
- Migrating from APL+WIN to Dyalog-APL
- Converting to a Jarvis-based web service
- Deploying the web service as a Docker container

System Overview – Current State

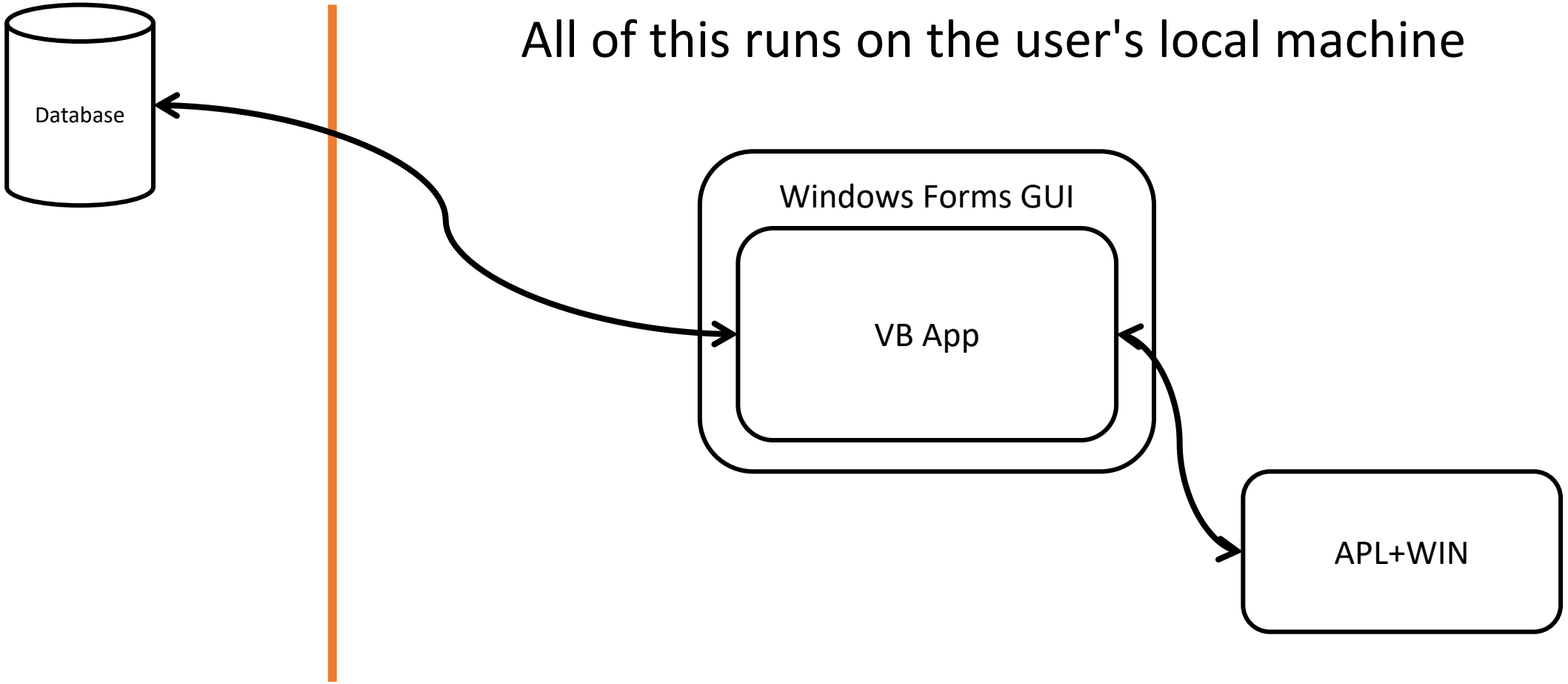
- Application written in Visual Basic 6
- Windows Forms GUI
- Calculating engine written in APL+WIN
- Calculating engine is provided as a COM Server
  - The Visual Basic app calls the APL as follows (translated to Dyalog):

```
WSengine←⎕NEW'OLEClient'(⊂'ClassName' 'APLW.WSEngine')
WSengine.SysCommand⊂'Load /path/to/workspace'
WSengine.Call1 'foo' arg
```

System Overview – Current State



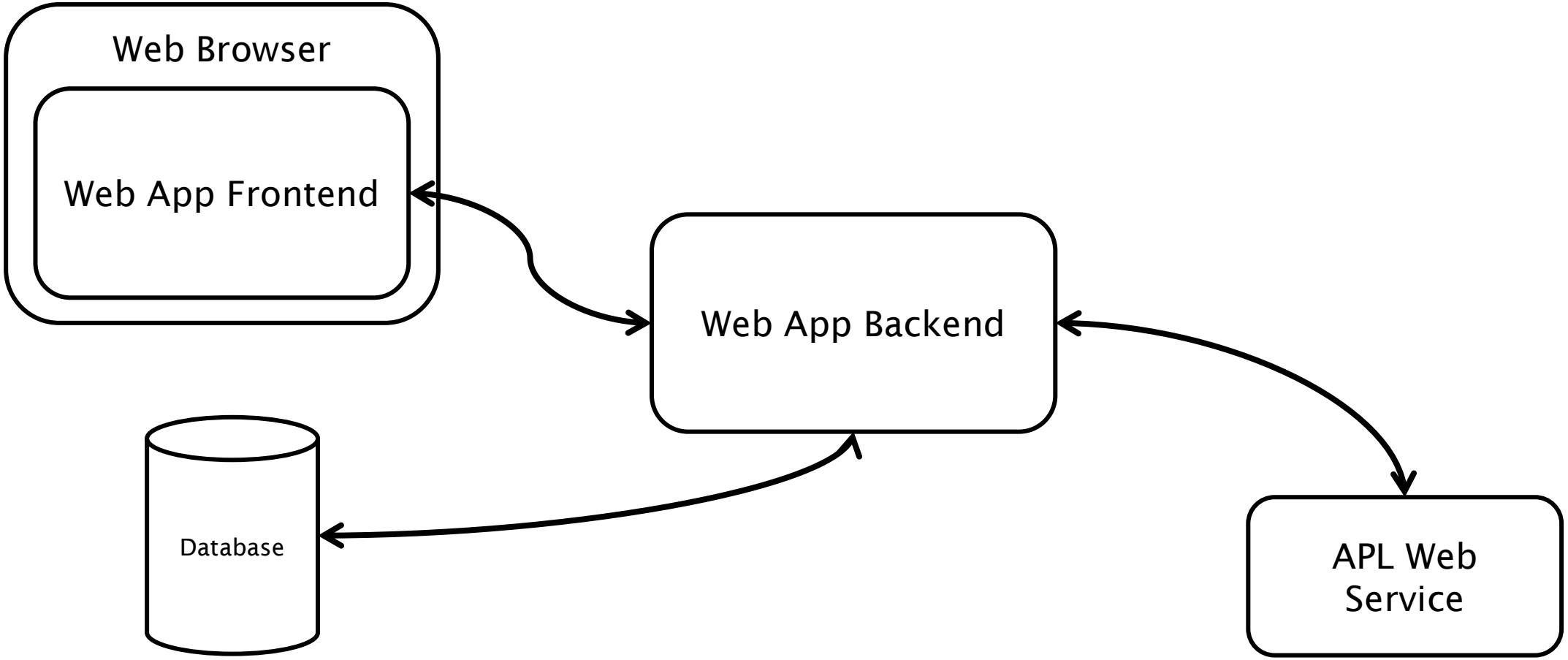
System Overview – Current State



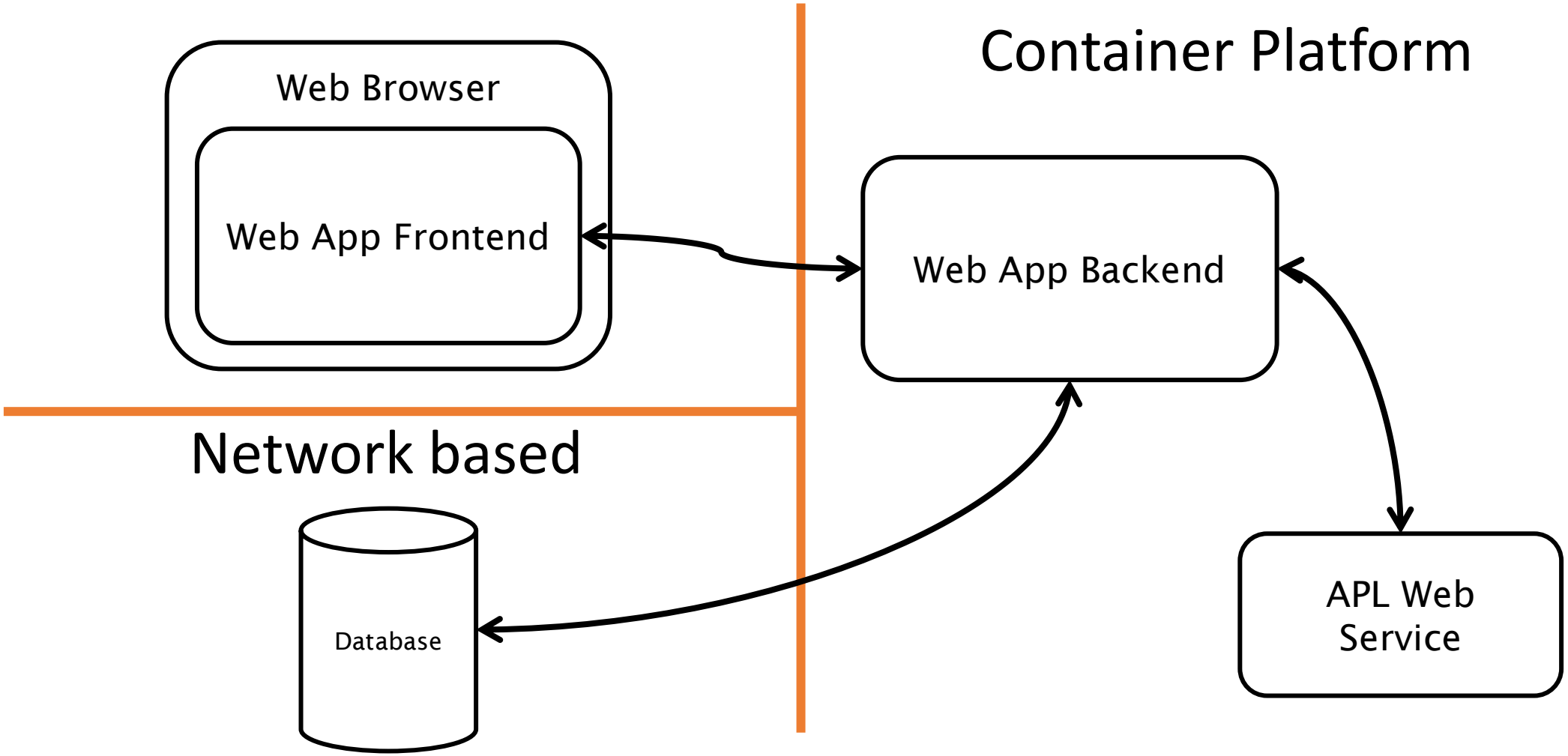
System Overview – Current State

- Why even change the current system?
  - Support for Visual Basic 6 is running out
  - Current architecture of the system does not fit into the infrastructure of the customer
- The goal is:
  - Move to a browser-based solution
  - Replace the outdated components

System Overview – Goal State



System Overview – Goal State



Roadmap

1. Migrate from APL+WIN to Dyalog
2. Convert the calculating engine to a Jarvis-based web service
3. Run it inside a Docker container

Migrating from APL+WIN to Dyalog

- Consider things like Replicate Each:

```
      1 0/¨(1 2)(3 4) ⍝ Dyalog
1 2
      1 0/¨(1 2)(3 4) ⍝ APL+WIN
1 3
```

- Replace system functions and system variables
- In APL+WIN, all left arguments are optional
- …

Converting to a Jarvis-based Web Service

- Deciding on a paradigm:
  - Jarvis supports two paradigms, JSON and REST
  - We chose the JSON-paradigm because:
    - It is suitable for functional endpoints
    - It is easier to implement
- Important question: Is the application stateless?
  - Luckily, the application at hand is!

Converting to a Jarvis-based Web Service

- Modifying the existing APL-code:
  - Endpoints are result-returning, monadic or dyadic APL-functions
    - Right argument is the request payload
    - (Optional) left argument is the request object itself
  - Jarvis handles the conversion between JSON and APL data structures (using `⎕JSON`)
    - One might have to change the structure of the arguments
    - The APL+WIN application takes strings with parameters separated by semicolons as argument (e.g., `'Finn;1234;5.678'`)
    - No changes were required (although this is probably a suboptimal solution)

Converting to a Jarvis-based Web Service

- Error handling
  - Jarvis (with configuration `Jarvis.Debug←0`) traps all errors and reports them with HTTP response status 500 (internal server error)
  - Use ⎕DMX since it has thread-scope
- We had to expand around the existing error handling

```
⎕TRAP←(500 'C' '→OldErrorhandler')                    ⍝ old error handling
⎕TRAP,←⊂(0 'E' '#.Errorhandling.WriteInfoAndResignal')  ⍝ new error handling
```

Converting to a Jarvis-based Web Service

- Logging
  - We log to stdout.
  - Since we want to run the app inside a container, this allows reporting tools to access the logs

Converting to a Jarvis-based Web Service

- Configuration
  - Configuration parameters can be specified in a JSON-file
  - We use a configuration similar to the following:

```
{
    AppCloseFn: "myAppCloseFn",        // called when Jarvis starts
    AppInitFn: "myAppInitFn",          // called when Jarvis stops
    CodeLocation: "./APLSource",       // by default, all functions in CodeLocation and below are exposed as endpoints
    ExcludeFns: “Errorhandling.*",     // not exposing these functions as endpoints
    Port: 8000,                        // port Jarvis is to list on
}
```

Converting to a Jarvis-based Web Service

At this point, we can run our application as a Jarvis-based web service

…at least on localhost

```
Jarvis.Run '/path/to/config.json'
```

Creating a Custom Docker Image

- Dyalog provides public Docker images for experimentation only.
  - These are not meant to be used in production.
- We used the Dockerfile for the dyalog/jarvis public image as a starting point for our custom image
  - Key differences are:
    - the base image
    - loading of dependencies
    - some added configuration
  - Components of the custom image:
    - Base image
    - Interpreter
    - Jarvis
    - Source code (stored as text files)

Creating a Custom Docker Image

- Simplified version of our custom Dockerfile

```
FROM redhat/ubi8-minimal:8.8

ADD APLSource /app
ADD linux_64_18.2.45405_unicode.x86_64.rpm /dyalog.rpm
RUN git clone https://github.com/dyalog/Jarvis /Jarvis

ENV JarvisConfig="/app/Config.json“
ENV LOAD="/Jarvis/Source“

ENTRYPOINT dyalog
```

Creating a Custom Docker Image

- Simplified version of our custom Dockerfile

Base Image →

```
FROM redhat/ubi8-minimal:8.8
```

Add all components (specify a version!) →

```
ADD APLSource /app
ADD linux_64_18.2.45405_unicode.x86_64.rpm /dyalog.rpm
RUN git clone https://github.com/dyalog/Jarvis /Jarvis
```

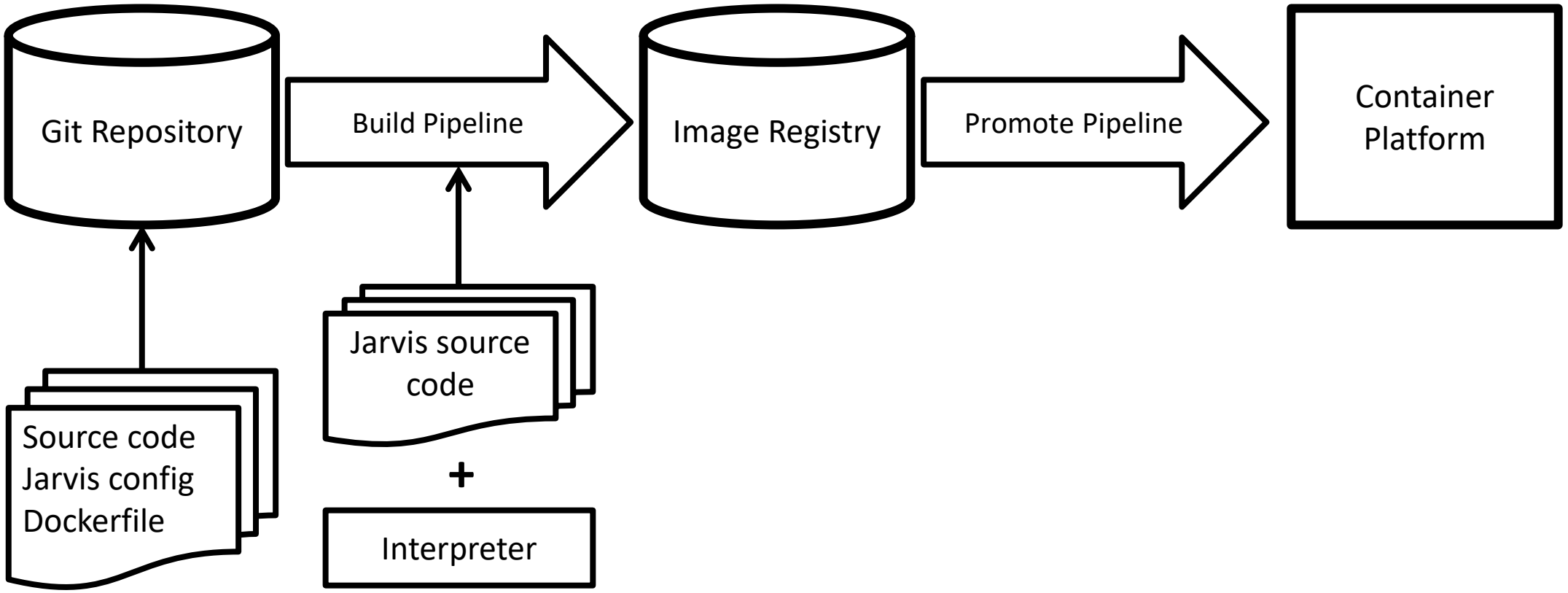
Specify environment variables →

```
ENV JarvisConfig="/app/Config.json“
ENV LOAD="/Jarvis/Source“
```

Executable which runs at startup →

```
ENTRYPOINT dyalog
```

Build & Deploy



What Else?

- Security
- Testing
- Updating the components of the image
- …

Summary

- Until now, everything runs without issues
- App is yet to go into production, but test results are promising
- Conversion process went smoothly