March 2023

Applying Domain Driven Design patterns to transform a monolith to a microservices based architecture.

"A simple guy in a complex world trying to make a difference."

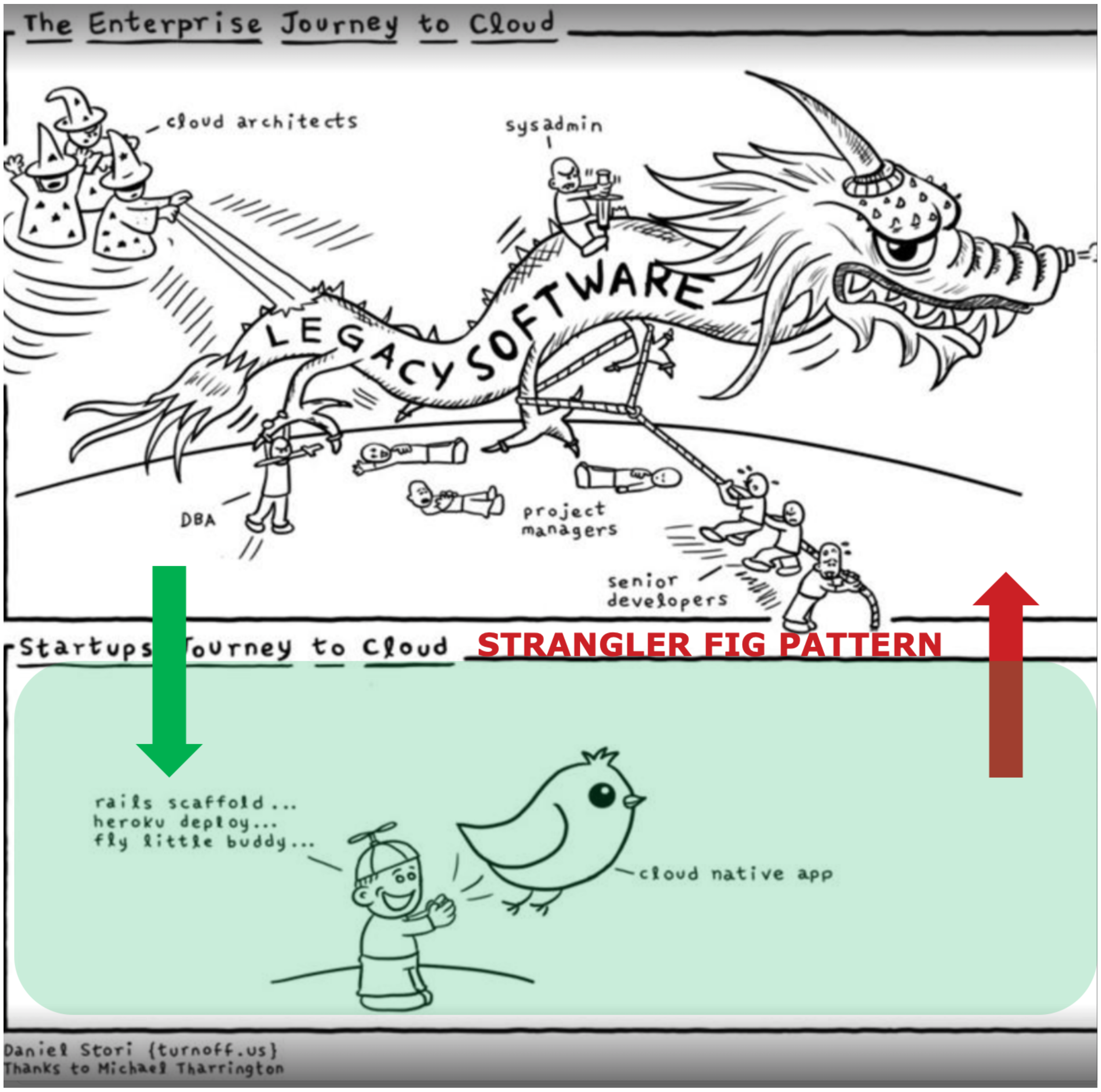
Nicholas Gulrajani

REFERENCES:

https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

https://martinfowler.com/bliki/StranglerFigApplication.html

"The reality"



The intent of this brief is to show, how to apply (with a practical example) industry recognized cloud native compute foundation aka 'cncf' patterns not only when building an 'N' Tier / Layered Application but also when building microservices.

A brief list appears below.

- *DOMAIN DRIVEN DESIGN 'DDD'*
- DEVOPS
    - *GITOPS (PUSH AND PULL MODEL)*
- SERVICE MESH

    - *TRAFFIC MGMT*
    - *mTLS*
    - *RESILIENCY (CHAOS INJECTION) / CIRCUIT BREAKING*
- EVENT DRIVEN ARCHITECTURES
- DATABASE MANAGEMENT
- STREAM-PROCESSING PATTERN
- API MANAGEMENT AND CONSUMPTION
- OBSERVABILITY AND MONITORING
- 12-FACTOR APPLICATION
- IAC + APP (AUTOMATED) CI/CD PLs + Security Scans
- Container Orchestration
- Policy As Code

The first step however, is to start with domain driven development 'DDD.'

Domain-driven development (DDD) is an approach to software development that emphasizes understanding and modeling the business domain of the software.

The focus is on creating a software system that reflects the real-world domain it is intended to serve.

DDD involves a collaborative effort between software developers, business stakeholders, and subject matter experts to create a shared understanding of the domain and its requirements.

PART A - A (REFERENCE) MONOLITH IMPLEMENTATION

A Python-based program that utilizes Domain Driven Design (DDD) principles to implement product and order services with a basic user interface.

Let's start by defining our domain model, which consists of two entities: Product and Order.

The Product entity represents a product that can be ordered, and the Order entity represents an order that contains one or more products.

See code construct of the monolith application with various domains ….

```
├── app
|   ├── main.py
|   ├── order_service.py
|   ├── product_service.py
|   ├── order.html
|   ├── products.html
|   ├── index.html
|   └── templates
|       ├── base.html
|       ├── order.html
|       ├── products.html
|       └── index.html
└── requirements.txt
```

The **app** directory contains all the source code for the monolith program.

Here's a brief overview of each file in the directory:

- **main.py**: The main entry point for the application. This file contains the Flask application object, along with the routes and views for the user interface.
- **product_service.py**: Contains the **ProductService** class, which provides functionality for creating, retrieving, and updating products.
- **order_service.py**: Contains the **OrderService** class, which provides functionality for creating and retrieving orders.
- **order.html**: The template file for displaying a single order.
- **products.html**: The template file for displaying all products.
- **index.html**: The main template file, which extends **base.html** and provides the overall layout for the user interface.
- **templates/base.html**: The base template file, which defines the overall layout and structure of the HTML pages.
- **templates/order.html**: A partial template file for displaying a single order. Used in **orders.html**.
- **templates/products.html**: A partial template file for displaying a single product. Used in **products.html**.
- **templates/index.html**: The main template file for the user interface.
- **requirements.txt**: A list of dependencies required by the application.

Below is a working eCom domain driven design 'DDD' based application which has been refactored (from PART A) to microservices and containers using Dockerfiles and Docker compose.

Microservice 1: Product Catalog Service

- The Product Catalog Service is responsible for managing the product catalog, including adding new products, updating existing products, and retrieving product information.
- It exposes a REST API that allows other services to retrieve product information.
- The service stores product information in a database.

Microservice 2: Order Service

- The Order Service is responsible for managing customer orders, including creating new orders, updating existing orders, and retrieving order information.
- It exposes a REST API that allows other services to place orders and retrieve order information.
- The service stores order information in a database.

Containerization

- Both microservices are containerized using Docker.
- Each microservice is deployed as a separate container.
- The containers are managed by a container orchestration system such as Kubernetes.

Domain Model

- The domain model for the e-commerce platform includes two main entities: Product and Order.
- The Product entity has attributes such as name, description, price, and vendor information.
- The Order entity has attributes such as customer information, product information, and order status.

Communication between Microservices

- The Product Catalog Service and Order Service communicate with each other using REST APIs.
- When a customer places an order, the Order Service calls the Product Catalog Service to retrieve product information.
- The Order Service then stores the order information in its database.

In summary, this is a simple domain driven design based on two microservices containerized.

The Product Catalog Service manages the product catalog, and the Order Service manages customer orders.

Both microservices are containerized using Docker and communicate with each other using REST APIs.

The domain model includes two entities: Product and Order.

Let’s start with the code directory construct of these microservices.

Both services are written in Python for this example.

```
├── product_catalog_service/
|   ├── Dockerfile
|   ├── product_catalog_service.py
|   ├── requirements.txt
|   └── ...
├── order_service/
|   ├── Dockerfile
|   ├── order_service.py
|   ├── requirements.txt
├── docker-compose.yml
```

Explanation of the directory structure:

- **product_catalog_service/**: This directory contains all the files related to the Product Catalog Service microservice.
    - **Dockerfile**: This file contains the Docker instructions for building the container image for the Product Catalog Service.
    - **product_catalog_service.py**: This file contains the Python code for the Product Catalog Service.
    - **requirements.txt**: This file lists all the Python dependencies required for running the Product Catalog Service.
- **order_service/**: This directory contains all the files related to the Order Service microservice.
    - **Dockerfile**: This file contains the Docker instructions for building the container image for the Order Service.
    - **order_service.py**: This file contains the Python code for the Order Service.
    - **requirements.txt**: This file lists all the Python dependencies required for running the Order Service.
- **docker-compose.yml**: This file defines the Docker Compose configuration for running both microservices together. It specifies the container images, ports, and other configuration options.

Note: This is just an example directory structure, and it can vary depending on the project's requirements and team preferences.

Docker Compose YAML file for running the Product Catalog Service and Order Service microservices together:

```
version: '3'

services:
  product_catalog_service:
    build: ./product_catalog_service
    ports:
      - "5000:5000"
  order_service:
    build: ./order_service
    ports:
      - "5001:5001"
```

Explanation of the Docker Compose YAML file:

- **version**: This specifies the version of the Docker Compose YAML file format.
- **services**: This section lists all the microservices that need to be started.
    - **product_catalog_service**: This section specifies the configuration options for the Product Catalog Service microservice.
        - **build**: This specifies the path to the directory containing the Dockerfile and other files required to build the container image for the Product Catalog Service.
        - **ports**: This maps the container port 5000 to the host port 5000, so that the Product Catalog Service API can be accessed from the host machine.
    - **order_service**: This section specifies the configuration options for the Order Service microservice.
        - **build**: This specifies the path to the directory containing the Dockerfile and other files required to build the container image for the Order Service.
        - **ports**: This maps the container port 5001 to the host port 5001, so that the Order Service API can be accessed from the host machine.

To run the microservices, navigate to the directory containing the Docker Compose YAML file and run the following command:

Below is an example requirements.txt file for the Order Service microservice:

Explanation of the requirements:

- **Flask**: This is a micro web framework for building web applications in Python. It is used to create the RESTful API endpoints for the Order Service.
- **requests**: This is a Python library for making HTTP requests. It is used to call the Product Catalog Service API to retrieve product information.

Note: This is just an example requirements file, and it can vary depending on the project's requirements and team preferences. You may need to add additional dependencies based on the functionality and features of your Order Service microservice.

Below an example requirements.txt file for the Product Catalog Service microservice:

```
Flask==2.1.0
```

Explanation of the requirements:

- **Flask**: This is a micro web framework for building web applications in Python. It is used to create the RESTful API endpoints for the Product Catalog Service.

Note: This is just an example requirements file, and it can vary depending on the project's requirements and team preferences. You may need to add additional dependencies based on the functionality and features of your Product Catalog Service microservice, such as a database driver or an Object Relation Model aka 'ORM'.

Below is an example requirements.txt file for the Order Service microservice:

```
Flask==2.1.0
requests==2.26.0
```

Explanation of the requirements:

- **Flask**: This is a micro web framework for building web applications in Python. It is used to create the RESTful API endpoints for the Order Service.
- **requests**: This is a Python library for making HTTP requests. It is used to call the Product Catalog Service API to retrieve product information.

Note: This is just an example requirements file, and it can vary depending on the project's requirements and team preferences. You may need to add additional dependencies based on the functionality and features of your Order Service microservice.

Sample code for the Product Catalog Service:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/products')
def get_products():
    # Retrieve product information from the database
    products = [
        {'id': 1, 'name': 'Product 1', 'description': 'Description 1', 'price': 10.0, 'vendor': 'Vendor 1'},
        {'id': 2, 'name': 'Product 2', 'description': 'Description 2', 'price': 20.0, 'vendor': 'Vendor 2'},
        {'id': 3, 'name': 'Product 3', 'description': 'Description 3', 'price': 30.0, 'vendor': 'Vendor 3'}
    ]
    return jsonify(products)

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Here's an example Dockerfile for the Product Catalog Service:

```
FROM python:3.9

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

CMD ["python", "product_catalog_service.py"]
```

Sample code for the Order Service:

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/orders', methods=['POST'])
def create_order():
    # Retrieve product information from the Product Catalog Service
    product_id = request.json['product_id']
    # Call Product Catalog Service API to retrieve product information
    product_info = {'id': product_id, 'name': 'Product 1', 'description': 'Description 1',
'price': 10.0, 'vendor': 'Vendor 1'}

    # Create a new order
    order = {'customer_name': request.json['customer_name'], 'product_info':
product_info, 'status': 'New'}
    # Save order information to the database

    return jsonify(order)

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5001)
```

And here's an example Dockerfile for the Order Service:

```
FROM python:3.9

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt.

COPY . .

CMD ["python", "order_service.py"]
```

Make sure to replace **product_catalog_service.py** and **order_service.py** with the actual filenames of your Python scripts, and **requirements.txt** with the actual filename of your requirements file.

BUILD AND RUN THE MICROSERVICES

Once you have your microservices containerized and your Docker Compose YAML file ready, you can test the microservices by running the following command in the directory containing the Docker Compose YAML file:

```
docker-compose up
```

This will build the container images for both microservices and start them.

You can then use a tool such as **curl** or a web browser to test the APIs exposed by the microservices.

For example, assuming the Product Catalog Service is running on port 5000 and the Order Service is running on port 5001, you can test the APIs by making HTTP requests to **http://localhost:5000/products** and **http://localhost:5001/orders** respectively.

Below is an example '**curl'** command to test the Product Catalog Service API:

```
curl http://localhost:5000/products
```

And here's an example '**curl'** command to test the Order Service API:

```
curl -d '{"customer_name": "John Doe", "product_id": 1}' -H "Content-Type: application/json" -X POST
http://localhost:5001/orders
```

Make sure to replace the customer's name and product ID in the above **curl** command with valid values for your test.

Note:

This is one example of how to test microservices running in Docker containers.

Depending on the complexity of your microservices and the tools used, you may need to write additional tests and perform more thorough testing to ensure that your microservices are working as expected.

Conclusions

This brief illustrates with a practical example a ‘DDD’ pattern applied to a microservices and container-based design.

Subsequent briefs as follow on’s will encompass various other ‘CNCF’ patterns as listed above.