Developing Cloud Serverless Components in Python

Domain-Driven Design Perspective

Asher Sterkin
asher.sterkin@gmail.com
About Myself

- Software technologist/architect
- VP Engineering @ BST
- Previously CTO @ IRKI, DE @ Cisco, VP Technologies @ NDS
- Cross-discipline approach (connecting the dots):
  - (Strategic) Domain-Driven Design
  - Serverless Architecture
  - Cynefin
  - Wardley Maps
  - Lean Startup
  - Promise Theory
  - ...

- Articles on Medium
- Presentations on Slideshare
- @asterkin on Twitter
- Source code for this talk on GitHub
Connecting the Dots
What is Serverless Architecture?

- No servers to manage
- Highly available
- Flexible scaling
- Two flavors:
  - AWS Lambda - no pay for idle
  - AWS Fargate - log running fully managed containers
- More details
- Serverless computing is a tectonic shift in the software industry
  - S. Wardley “Why the fuss about serverless?”
“New Platform Capabilities Lead to New Architectural Style and New Operational Practices

S. Wardley (liberal quoting of mine)
Serverless-native Architecture is yet to Emerge

Some exploration and “safe-to-fail” experimentation are required

Cloud VM-native architecture
Cloud Container-native architecture
Cloud Serverless-native architecture
Tools have strong influence on thinking
Python is a Superb Tool for Exploration and Experimentation

Outcomes of this research will be easy to port to other languages
AWS Lambda Function in Python

```python
print('Loading function')  # executed at cold start

def lambda_handler(event, context):  # executed at every invocation
    print("value1 = " + event['key1'])  # input data usually comes as a dict
    print("value2 = " + event['key2'])
    print("value3 = " + event['key3'])
    return event['key1']  # Return something if invoked synchronously
    # raise Exception('Something went wrong')
```
AWS Lambda Performance

Source: Yun Zhi Lin, “Comparing AWS Lambda performance of Node.js, Python, Java, C# and Go”
Serverless Architecture Needs Some Organizing Principles

To prevent creation of a distributed monolith
Domain-Driven Design
Domain-Driven Design at a Glance

Language
Model
Boundaries
Nesting
Language

- Common (ubiquitous) language for domain and software experts
- Vocabulary reflected directly in software
- Normally a subset of domain-specific jargon

**DDD is essentially a *linguistic* process**
Boundaries

- “One size fits all” models are impractical
- G-d like models result in Big Ball of Mud
- DDD insists on identifying multiple models:
  - Coherent within their own boundaries
  - Mapped onto others where appropriate

DDD is about maintaining boundaries for- and mappings between- models
Subdomains (my interpretation)

What justifies developing software in the first place

From the same area of activity, but not specific enough

Completely generic, applicable to any domain
Subdomains (my interpretation)

What justifies developing software in the first place

From the same area of activity, but not specific enough

Completely generic, applicable to any domain
Evaluation Criteria

- Is domain language reflected directly in the code?
- Is core logic clearly separated from supporting, and generic subdomains?
- Is there an appropriate model underneath?
DDD Sample Application: Cargo Tracking System
Questions I wanted to address

- What would be an objective reason to prefer Python over JavaScript?
- What is involved in implementing DDD cargo Sample in Serverless Python?
- What is missing in the Serverless Python infrastructure, if any?
I was Very Surprised with the Answers

Let’s see why
You Always Start with a Use Case Model, Don’t You

In fact nobody does, and it’s a big problem
Cargo Tracking Use Cases

**External System**
- Record Cargo Tracking Event

**Operator**
- Book Cargo
- Check Delivery Status
- Change Destination
- Re-route Cargo
You Start with Architecturally Essential Use Case(s)

It’s far from trivial - you need an architect
Cargo Tracking Use Cases

- Interfacing with external systems
- High volume
- High velocity
- Security concerns

External System

- Record Cargo Tracking Event

Operator

- Book Cargo
- Check Delivery Status
- Change Destination
- Re-route Cargo
Diving One Inch Deeper

3rd Party Operator

Operator

api.cts.com

HTTP

HTTP/2

MQTT

CargoTrackingEvents

CargoTrackingEventsArchive

CargoTrackingErrors

CargoTrackingEvents

Cargo Tracking System
Cargo Handling Event

CargoTrackingEvent
- completionTime
- eventType

Location
- loadLocation
- unloadLocation

Voyage

CarrierMovement
- loadTime
- unloadTime

Cargo
Record Cargo Tracking Event: First Cut

REST or IoT End Point
invoke
get_item(cargo)
get_item(location)
get_item(voyage)
put_item(event)
Record Cargo Handling Event: Architecture

Can we program this in half-an-hour?
import boto3
import os

dynamodb = boto3.resource('dynamodb')

def checkItem(table, key, event, value):
    if value in event:
        if 'Item' in table.get_item(Key={key:event[value]},ProjectionExpression=key):
            return []
        else:
            return ['%s not found' % value]
    else:
        return ['%s is missing' % value]

def flatten(status):
    return [e for es in status for e in es]
cargoTable = dynamodb.Table(os.getenv('CARGO_TABLE'))

def checkCargo(event):
    return checkItem(cargoTable, 'trackingId', event, 'cargo')

locationTable = dynamodb.Table(os.getenv('LOCATION_TABLE'))

def checkLocation(event):
    return checkItem(locationTable, 'unLocationCode', event, 'location')

VOYAGE_REQUIRED = {'LOAD', 'UNLOAD'}

voyageTable = dynamodb.Table(os.getenv('VOYAGE_TABLE'))

def checkVoyage(event):
    if not 'eventType' in event: return ['eventType is missing']
    if event['eventType'] in VOYAGE_REQUIRED:
        return checkItem(voyageTable, 'id', event, 'voyage')
    else:
        return ['unexpected voyageId'] if 'voyage' in event else []
from botocore.exceptions import ClientError
import time

eventTable = dynamodb.Table(os.getenv('EVENT_TABLE'))

def recordEvent(event):
    try:
        eventTable.put_item(
            Item= {
                'cargoTrackingId': event['cargo'],
                'completionTime': event['completionTime'],
                'registrationTime': int(time.time()*1000),
                'voyageId' : event['voyage'],
                'eventType' : event['eventType'],
                'location': event['location']
            },
            ConditionExpression = 'attribute_not_exists(cargoTrackingId) AND attribute_not_exists(completionTime)'  
        )
        return 'OK'
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            return 'DUPLICATE'
        raise
import json

def reportError(event, error):
    print(json.dumps({'event': event, 'error': error}))

def lambda_handler(event, context):
    inputStatus = flatten([checkCargo(event), checkLocation(event), checkVoyage(event)])

    if len(inputStatus) > 0:
        reportError(event, inputStatus)
    else:
        recordingStatus = recordEvent(event)
        if recordingStatus != 'OK': reportError(event, recordingStatus)
Evaluation Criteria

- Is domain language reflected directly in the code?
- Is core logic clearly separated from supporting, and generic subdomains?
- Is there an appropriate model underneath?
What would be an alternative?

Without alternatives it’s an ideology, not engineering
What if we used inappropriate tool for modelling the system logic?

Always a good question to ask
Input validations could be performed in parallel.
How could we implement such logic?

- Use Python asyncio?
- Switch to JavaScript?
- Any other option?
Look, Ma!

There is (almost) no code here!
"""Encapsulates Cargo Repository and all relevant computations."""
import matte

CARGO = 'cargo'
CARGO_KEY = {CARGO : str}

Cargoes = matte.repository('Cargoes', hashKey = CARGO_KEY)
Specifications = matte.repository('Specifications', hashKey = CARGO_KEY)
Itineraries = matte.repository('Itineraries', hashKey = CARGO_KEY)

""" Check whether cargo reference is correct."""
from Cargo import Cargoes
cargoes = Cargoes.get_client()

def check_cargo(cargo):
    return cargoes.has_item(cargo)
What About Other Use Cases?

Could we apply the same approach?
Now, We Speak Domain Language in the Code

As we are supposed to
from lib.Itinerary import Itinerary
import CargoHandlingEvent

def expected_at_interim_leg(last_event, itinerary):
    leg = itinerary.find(last_event)
    return (True, leg.is_on_schedule(last_event)) if leg else [False, '']

def expected_at_edge(last_event, leg):
    expected = leg.is_expected(last_event)
    return (expected, leg.is_on_schedule(last_event) if expected else '')

def expected_at_first_leg(last_event, itinerary):
    return expected_at_edge(last_event, itinerary.first())

def expected_at_final_leg(last_event, itinerary):
    return expected_at_edge(last_event, itinerary.last())

def default(last_event, itinerary):
    return (False, '')

def expect(last_event, itinerary):
    return {
        CargoHandlingEvent.RECEIVE: expected_at_first_leg,
        CargoHandlingEvent.LOAD: expected_at_interim_leg,
        CargoHandlingEvent.UNLOAD: expected_at_interim_leg,
        CargoHandlingEvent.CUSTOMS: expected_at_final_leg,
        CargoHandlingEvent.CLAIM: expected_at_final_leg
    }.get(last_event.event_type, default)(last_event, itinerary)

def calculate_delivery_status(last_event, itinerary):
    if not itinerary: return 'UNKNOWN'
    (expected, on_schedule) = expect(last_event, Itinerary(itinerary))
    return on_schedule if expected else 'MISDIRECTED'
Cargo Tracking System (Micro?) Services

- CargoTrackingSystem
  - Cargo
  - CargoHandlingEvent
  - Voyage
Missing Infrastructure Elements

- Conversion to/from JavaScript-style objects
- `find()` with predicate function
- Dropping None values from dict
- High-level encapsulation of managed services
- Automatic generation of SAM templates
- Automatic generation of Step Functions
Introducing Matte (Staff in Hebrew)

Staff of Moses that, upon command, was transformed to a snake, forth and back
Matte

Experimental Python3 infrastructure toolkit to remove as much boilerplate from Serverless DDD code as possible

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>js0</td>
<td>lazy dict -&gt; object conversion</td>
</tr>
<tr>
<td>find</td>
<td>list lookup with predicate</td>
</tr>
<tr>
<td>drop_nulls</td>
<td>remove None values from dict</td>
</tr>
<tr>
<td>make_handler</td>
<td>Automatic code generator of lambda_handler</td>
</tr>
<tr>
<td>make_sam_template</td>
<td>SAM template builder</td>
</tr>
<tr>
<td>Repository</td>
<td>DynamoDB resource descriptor</td>
</tr>
<tr>
<td>RepositoryClient</td>
<td>DynamoDB client with permissions</td>
</tr>
<tr>
<td>DecimalEncoder</td>
<td>Proper handling of Numeric fields</td>
</tr>
<tr>
<td>...</td>
<td>More to follow</td>
</tr>
</tbody>
</table>
Cargo Tracking System Subdomains

Core

Cargo, Specification, Itinerary, CargoHandlingEvent

Supporting Services

Voyage, Location

Generic Services

Matte
Software Architecture is a Context Mapping Process

From Use Cases to Core, to Supportive, to Generic sub-domains
Beware of Coupling

And always make it explicit
But isn’t this approach terribly inefficient?

Too many forth and backs between Step Function and Lambda
You never know until you measure

It might actually scale better than we think
Why it might scale better?

- Tiny Footprint → Easy to find resources for new instance
- Single-purpose Function → Few policies to check
- Built-in basic monitoring → Data-driven ML-based optimization is possible
What Comes After Serverless?

Some Wild Ideas, Inspired by this Research
I Feel Monadish

Pure Functions

Domain Logic

Activity Flow

Use Case Controllers

Interfacing Managed Services

Amazon S3

Amazon DynamoDB

Amazon SNS

Collect Data and Optimize

Collect Data

AWS IoT

Amazon API Gateway

AWS Appsync

Collect Data

Collect Data

Collect Data

Collect Data

Collect Data
Maybe We Should Call it Threadless Architecture?

Who knows? Time will Tell.
Please, Disagree with Me

“Consensus is poisonous for innovation”

D. Snowden