Library Data ReST APIs: Design to Deploy

Samvera Connect 2018 Workshop

github.com/PenguinParadigm/samvera18apis github.com/PenguinParadigm/samvera18apis

Link to Slides, Datasets, & Other Workshop Materials

Schedule (ish)

9-9:30	Introduction, Logistics, Goals
9:30-10:15	Designing our API (ReST, PCDM, OAS)
10:15-10:30	*Break*
10:30-12	Developing our API (Go, Go-Swagger, Localstack)
12-14:00	*Lunch Break (on your own)*
14:00-14:45	Containerizing our API (Docker)
14:45-15	*Break*
15-16:30	Deploying our API (AWS)
16:30-17	Conclusion & Workshop Retrospective

Your Workshop Authors

Justin Coyne, *@j_coyne, Stanford University Libraries*

Aaron Collier, *@aaronisbrewing, Stanford University Libraries*

Christina Harlow, *@cmh2166, Stanford University Libraries*

Special Thanks to

Erin Fahy, *@eefahy, Protocol Labs*

Introduction

Our Expectations of You

Personal

- Follow the Recurse Center Social Rules (a.k.a. "Hacker School Rules")

Technical

- Have `Go`, `Docker`, `localstack (docker image)`, and `aws-cli` (with free AWS account connection) ready to go on your laptop
- Be ready to participate!

Recurse Center Social Rules
(a.k.a. Hacker School Rules)

- No feigning surprise
- No well-actually's
- No back-seat driving
- No subtle -isms

More info:

- https://www.recurse.com/blog/38-subtle-isms-at-hacker-school
- https://www.recurse.com/manual#sub-sec-social-rules

Technical Prep

We hope you have before this point...

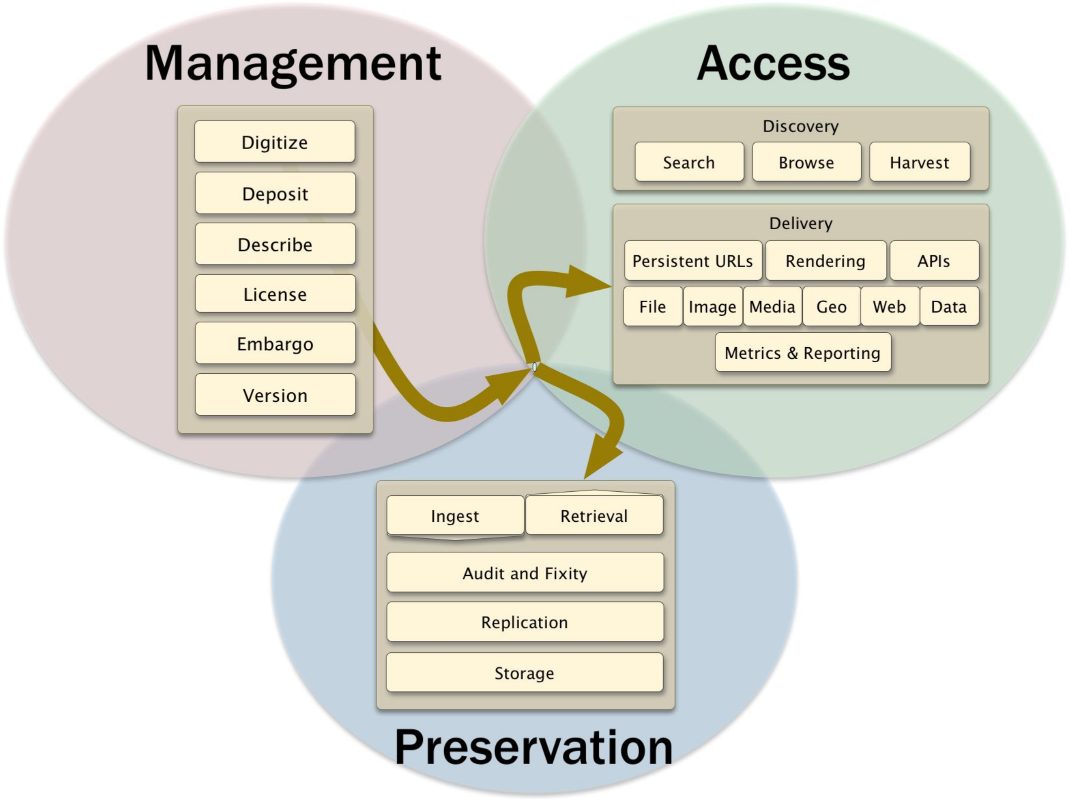
1. Brought a laptop with internet connection & modern web browser.
2. Installed latest stable `Go` on said laptop & set up your workspace.
3. Installed latest stable Docker Community Edition on said laptop.
4. Have our workshop GitHub repository on your computer (with mechanism to update / pull down latest changes on Tuesday morning.
5. Set up a free AWS account & `awscli` on said laptop for said account.
6. Pulled down & can run this localstack docker image in your Go workspace.

Our Goals for this Workshops

- Share Stanford’s recent work on a Fedora 3+ replacement, aka TACO
- Go end to end in this API process
  - We aren’t experts in any single part of this
  - I’m especially not an expert in any of this
  - We don’t want to deep dive today on any particular issue, but share the end to end process
- Learn enough to discuss Pros / Cons of
  - ReSTful API selection for what parts of our system
  - Data models & validation mechanisms
  - Go as middleware language selection
  - Docker as our container / deployment unit selection
  - AWS ECS versus local, serverless, other options
- Get feedback from you on our work so far

Your Goals for this Workshop?

Some Context: Stanford Digital Repository



https://github.com/PenguinParadigm/samvera18apis/

High-Level Overview of Stanford Digital Repository ecosystem June 2017

Ingest

- Hydrus
- ETDs
- Pre-Assembly (Digitization, Google books, Other)
- Goobi
- Assembly Utils
- Dir Validator
- Assembly ObjectFile
- Assembly Image
- WAS

DOR Services

- Dor Services App
- Dor Services Gem
- SURI
- Dor Workflow Service
- Workflow Service
- Fedora 3

PURL+

- Stacks
- Wowza
- PURL
- PURL Fetcher

Robots

- Robots Master
- Lyber Core
- Robot controller
- Assembly
- Common Accessioning
- Item Release
- ETD Robots
- Gis robot suite
- Goobi Robot
- Dor gsb robots
- Sdr Pres Core Robots
- WAS Robot Suite
- WAS Metadata Extractor

Argo+

- Dor Indexing App
- Argo
- Modsulator & Modsulator App Rails
- SUL MQ
- Dor Camel Routes

Preservation

- SDR preservation core
- moab versioning
- Archive catalog
- File system & tapes?
- SDR Services App

- Spotlight Exhibits Portfolios

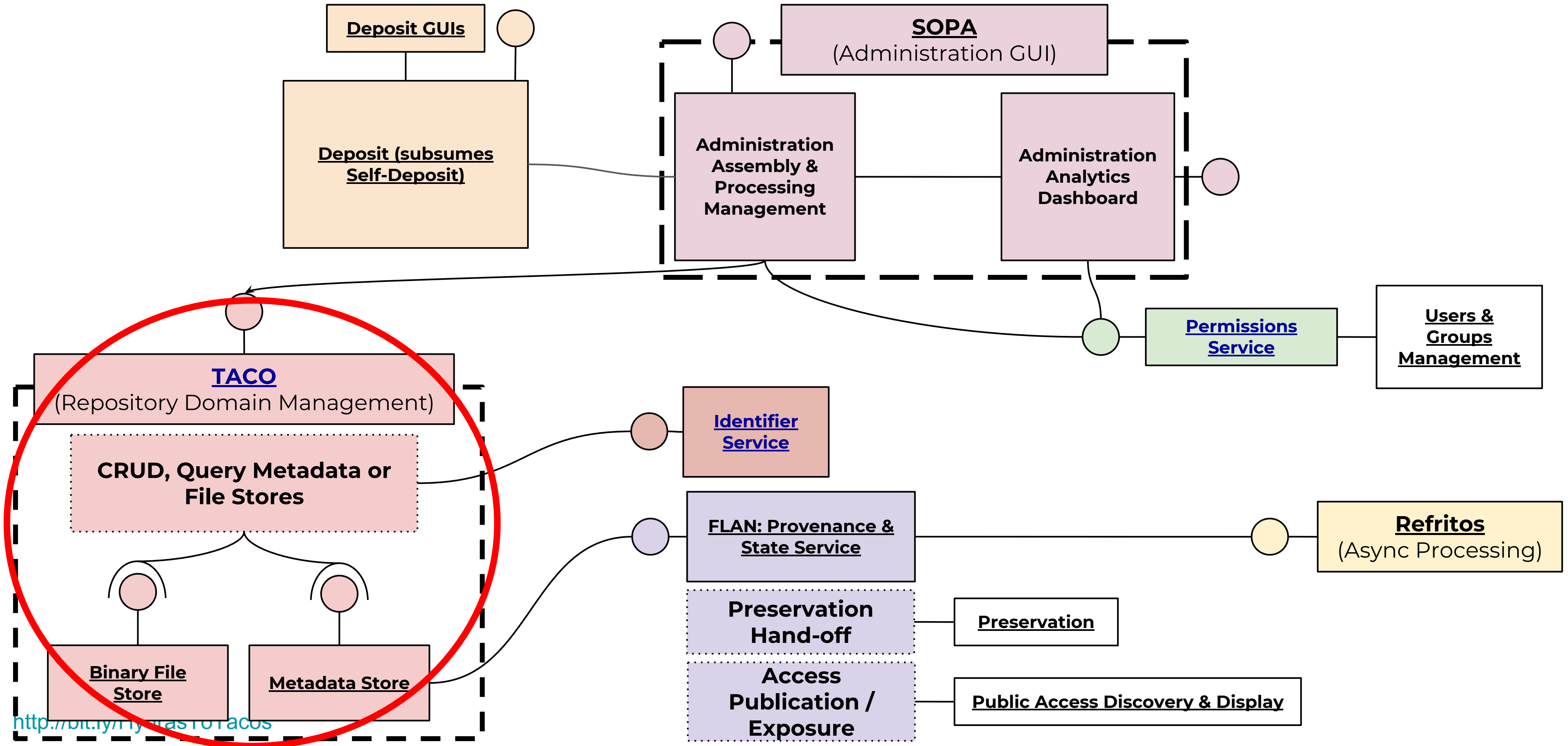
Stacks / Shelves

- Stacks
- various file systems
- A/V?
- Geo?
- …?

Indexing, Access, & Discovery

- Discovery Dispatcher
- sw-indexer
- sul-embed
- SearchWorks
- SWAP
- Mods profiling indexer

SDR3 High Level Conceptual Design (so far)


Deposit GUIs
Deposit (subsumes Self-Deposit)
SOPA
(Administration GUI)
Administration Assembly & Processing Management
Administration Analytics Dashboard
Permissions Service
Users & Groups Management
TACO
(Repository Domain Management)
CRUD, Query Metadata or File Stores
Binary File Store
Metadata Store
Identifier Service
FLAN: Provenance & State Service
Refritos
(Async Processing)
Preservation Hand-off
Preservation
Access Publication / Exposure
Public Access Discovery & Display

http://bit.ly/HydrasToTacos

Today’s Example: TAQUITO (little TACO)

TAQUITO, a Simple Digital Repository Management Layer API

Based on TACO (a prototype & WIP): https://github.com/sul-dlss-labs/taco/

TACO is meant to make database selections independent from more involved repository business logic. See more here: https://sul-dlss.github.io/taco-truck/

For TAQUITO, we will work through...

ReST API for the new service interface

JSON[-LD] for the service’s data representation

Swagger for the API specification

Go for the service’s programming language

Docker for deployment

AWS for infrastructure (ECS primarily)

For TAQUITO, we will work through...

ReST API for the new service interface

JSON[-LD] for the service's data representation

Swagger for the API specification

Go for the service's programming language

Docker for deployment

AWS for infrastructure (ECS primarily)

… aka A LOT OF STUFF. That we are learning as we go.

API Design

Overview

★ **ReST API** for the new service interface

JSON[-LD] for the service’s data representation

Swagger for the API specification

Go for the service’s programming language

Docker for deployment

AWS for infrastructure (ECS primarily)

TAQUITO’s contract within our system

What does this TAQUITO API promise to do?

- Really simple / ‘stupid’ CRUD for our core digital repository object models.
- Keep the database selection separate from the rest of the system.
- Manage our canonical metadata & metadata store.

Representational State Transfer (ReST)

ReST is an architectural style that gives some constraints. These constraints include but are not limited to...

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System

We selected ReST here for the ability to work across machines as well as keeping components boundaries clear.

TAQUITO (TACO-inspired) Routes

- `POST /resource` : Deposit New TAQUITO Resource.
  operationId: depositResource, consumes JSON or JSON-LD in body
- `PATCH /resource/{ID}` : Update TAQUITO Resource.
  operationId: updateResource, consumes JSON or JSON-LD in body
- `GET /resource/{ID}?version=#` : Retrieve TAQUITO Resource Metadata.
  operationId: retrieveResource, produces: JSON
- `DELETE /resource/{ID}` : Delete a TAQUITO Resource.
  operationId: deleteResource
- `GET /healthcheck` : Health Check.
  operationId: healthCheck

Overview

ReST API for the new service interface

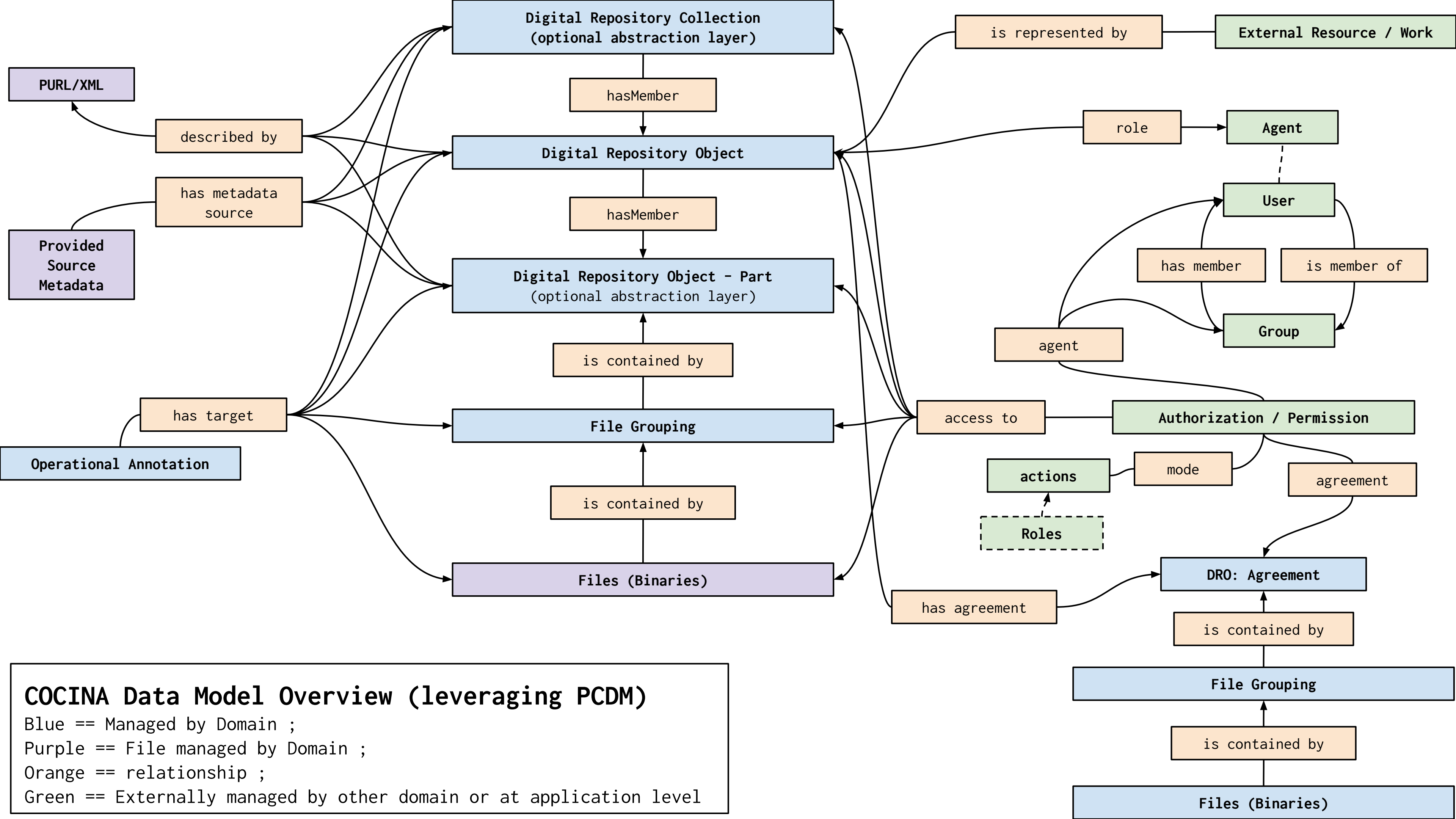
★ **JSON[-LD]** for the service’s data representation

Swagger for the API specification

Go for the service’s programming language

Docker for deployment

AWS for infrastructure (ECS primarily)



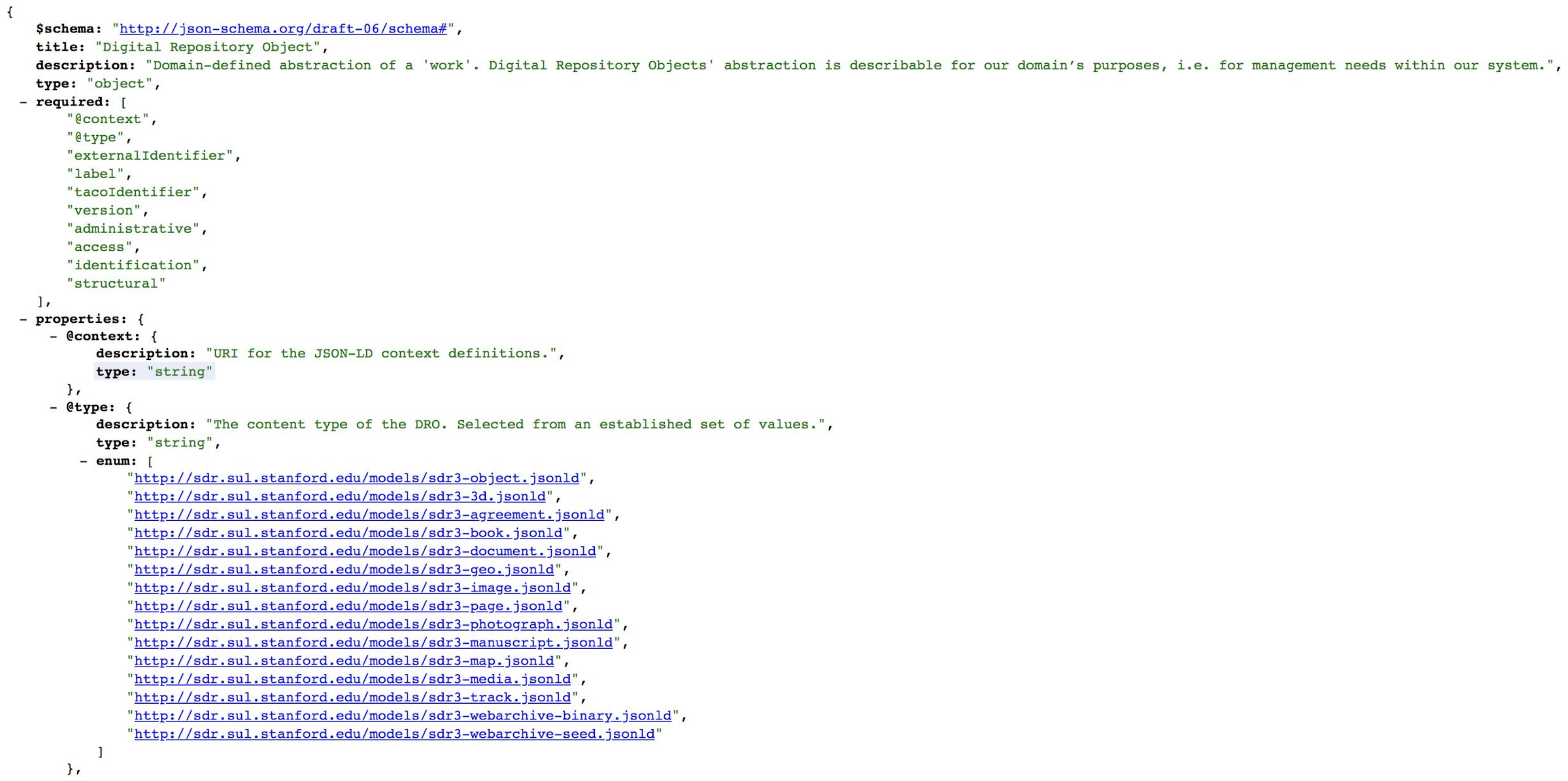
COCINA Data Model Overview (leveraging PCDM)

Blue == Managed by Domain ;
Purple == File managed by Domain ;
Orange == relationship ;
Green == Externally managed by other domain or at application level

TAQUITO API Data Models

- Resource (ResourceResponse | DepositResource)
  - Relation to our Data Models & MAPs
  - Relation to JSON, JSON Schema
- Agent
  - For Permissions more than Authorization
  - Logging information like Depositor
- Sequence
  - Handing multiple orders of resources
- HealthCheckResponse
- ErrorResponse
- Error

COCINA Metadata Application Profiles



```
{
    $schema: "http://json-schema.org/draft-06/schema#",
    title: "Digital Repository Object",
    description: "Domain-defined abstraction of a 'work'. Digital Repository Objects' abstraction is describable for our domain’s purposes, i.e. for management needs within our system.",
    type: "object",
  - required: [
        "@context",
        "@type",
        "externalIdentifier",
        "label",
        "tacoIdentifier",
        "version",
        "administrative",
        "access",
        "identification",
        "structural"
    ],
  - properties: {
      - @context: {
            description: "URI for the JSON-LD context definitions.",
            type: "string"
        },
      - @type: {
            description: "The content type of the DRO. Selected from an established set of values.",
            type: "string",
          - enum: [
                "http://sdr.sul.stanford.edu/models/sdr3-object.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-3d.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-agreement.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-book.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-document.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-geo.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-image.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-page.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-photograph.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-manuscript.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-map.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-media.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-track.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-webarchive-binary.jsonld",
                "http://sdr.sul.stanford.edu/models/sdr3-webarchive-seed.jsonld"
            ]
        },
```

https://github.com/sul-dlss-labs/sdr3-models/

Overview

ReST API for the new service interface

JSON[-LD] for the service's data representation

★ **Swagger** for the API specification

Go for the service's programming language

Docker for deployment

AWS for infrastructure (ECS primarily)

Break (15 minutes)

Quick Introduction to Swagger / OpenAPI

OpenAPI Specification (formerly, Swagger) is API description format or API definition language. Basically, OpenAPI Specifications let you describe:

- General information about the API
- Available paths (`/resources`)
- Available operations on each path (get `/resources`)
- Input/Output for each operation

OpenAPI / Swagger is a subset (though not entirely faithful) of JSON Schema.

2 Go Projects in Our Repo in Your Workspace

In our repository, we have 2 go projects:

- **`simpleDev`** just lets us get a handle on Swagger & Go-Swagger
- **`taquito`** is a stripped down version of TACO

Start with **`simpleDev`** first
(cd into that directory please)

SimpleDev Swagger Walk Through

Take a few minutes to find, review & add “update” & “delete” routes to the simpleDev Swagger Spec.

Development

Overview

ReST API for the new service interface

JSON[-LD] for the service's data representation

Swagger for the API specification

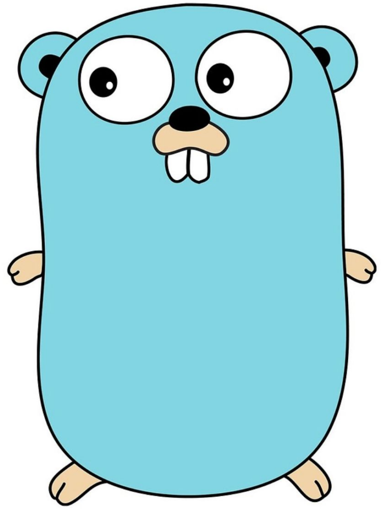
★ **Go** for the service's programming language

Docker for deployment

AWS for infrastructure (ECS primarily)

Go quick intro

- Go is a concurrent programming language introduced by Google in 2009
- Originally developed as a language for servers, but has grown to have a more general purpose
- “Go is a statically typed compiled language in the tradition of C” (thanks wikipedia)
- It is fast(er than Ruby) hence our usage



Check Your Go Development Environment

1. Install Go (I hope already done!).
2. Setup your Go workspace (where your Go code, binaries, etc. are kept together. See some helpful docs here):

```
$ mkdir -p ~/go
$ export GOPATH=~/go
$ export PATH=~/go/bin:$PATH
$ cd ~/go
```

3. Go code repositories will reside within `~/go/src/...` in `$GOPATH`. Name these paths to avoid library clash, for example Bootcamp Go code could be in `~/go/src/github.com/PenguinParadigm/samvera18apis`

Populate SimpleDev's Dependencies

Handle Go project dependencies with the Go `dep` package:

1. Install Go Dep via `brew install dep` then `brew upgrade dep` (if Mac OSX)
   a. If not using brew:
      ```
      $ curl
      https://raw.githubusercontent.com/golang/dep/master/install.sh | sh
      ```
2. If your project's `Gopkg.toml` & `Gopkg.lock` aren't populated, add an inferred list of dependencies via `dep init`.
3. If your project has those files populated, sync dependencies via `dep ensure`.
4. To add new dependencies, run `dep ensure -add github.com/pkg/errors`.
5. This adds dependency & put new dependency in your `Gopkg.*` files.

Dep will probably be deprecated in near future, but it is currently used.

Populate SimpleDev’s Dependencies

```
## Install Go Dep via (if Mac OSX)

$ brew install dep
$ brew upgrade dep

## Add an inferred list of dependencies via

$ dep init
$ dep ensure

## To add new dependencies, *you can* run

$ dep ensure -add github.com/pkg/errors
```

Take a few minutes to check your simpleDev workspace & install dependencies.

SimpleDev's Swagger Spec & Go-Swagger

The API code is generated from `swagger.json` using `go-swagger` library. You'll need to install `go-swagger` (for OSX, otherwise: https://goswagger.io/install.html):

```
$ brew tap go-swagger/go-swagger
$ brew install go-swagger
$ brew upgrade go-swagger
```

This should give you the `swagger` binary command in your `$GOPATH` and allow you to manage versions better. Try running swagger validation then docs generation:

```
$ swagger validate swagger.json
$ swagger serve swagger.json
```

SimpleDev’s Swagger Spec & Go-Swagger

Now generate the start of our API code from our Swagger spec by running:

```
$ mkdir generated
$ swagger generate server -t generated --exclude-main --principal \
    models.Agent
```

(there appears to be no best way to handle specification-based re-generation of the `generated/` API code)

Generated Code Deep Dive

Add info / pointers to the generated code to explain:

- `Models` (leveraging JSON Schema to become Go structs with validation functions and marshal/unmarshal interfaces)
- Configurations for the server
- `Operations` (based off the routes, operation per route & action)
  - OperationName
  - URL Builder
  - Responses
  - Parameters
- Operation for Primary API

`https://github.com/PenguinParadigm/samvera18apis/`

Generate Swagger Code

```
## Install Go-Swagger (if Mac OSX)

$ brew tap go-swagger/go-swagger
$ brew install go-swagger
$ brew upgrade go-swagger

## Validate Your Swagger Spec

$ swagger validate swagger.json

## Generate Swagger Server

$ mkdir generated
$ swagger generate server -t generated --exclude-main --principal
    \ models.Agent | dep ensure
```

https://github.com/PenguinParadigm/samvera18apis/

Take a few minutes to validate your Swagger.json & generate your Go code from Swagger.

Write main.go

`func main()` actually runs the server

`func createServer(port int) *restapi.Server` takes that Server instance, and add ours handlers (which then are called by the handlers generated for each route with the Swagger-generated portion)

Our SimpleDev Deposit Handler

Note: Deposit doesn't actually persist that metadata yet, it just prints it out to stdout.

`Handler` files are what run for each Handler / route.

You have generated code to help with data models, HTTP call Params, & responses.

Running the simpleDev Go Code

Running the Go Code locally without a build / binary:

```
$ go run main.go
```

Build Go binary for the local OS & Running that binary:

```
$ go build -o simpleDev main.go
$ ./simpleDev
```

Test Your Endpoint in New Shell

```
## Check the Health check (should say not implemented)

$ curl http://localhost:8080/v1/healthcheck

## Try Loading an Invalid Object

$ curl -H "On-Behalf-Of: lmcrae" -H "Content-Type: \
    application/json" -d@examples/create-bs646cd8717.json \
   http://localhost:8080/v1/resource

## Try Loading a Valid Object (Will Just Return JSON)

$ curl -H "On-Behalf-Of: lmcrae" -H "Content-Type: \
   application/json" -d@examples/deposit_object.json \
   http://localhost:8080/v1/resource
```

Let’s stub out the RetrieveResourceHandler (with static data) and test that it works

Lunch Break (120 minutes)
Restart at 2 PM

Move to TAQUITO now (go ahead & cd into that directory)

TAQUITO has infrastructure (database) ideas

- AWS DynamoDB for our JSON metadata
- To then do local development, we need localstack (hoping you have this installed!)
- awscli (aws) can point to localstack for querying it directly (or you can use awslocal)
- Go AWS SDK for our Handler code to connect to DynamoDB

Side Note: Why DynamoDB?

- Something simple & fast (simple being relative)
- Anecdotally, best up-time
- AWS SDK for Go already existed
- RDS (AWS) / PostgreSQL (local) is our fall back plan

Amazon DynamoDB

Using LocalStack with TAQUITO

```
## Start localstack & leave this running in terminal

$ docker run -p 4567-4583:4567-4583 -e SERVICES=dynamodb
localstack/localstack

## In new terminal, Make Localstack resources

$ make resources

## You can now interact with Localstack DynamoDB

$ aws --endpoint-url=http://localhost:4569 dynamodb list-tables

$ awslocal dynamodb describe-table --table-name 'resources'
```

https://github.com/PenguinParadigm/samvera18apis/

Take a few minutes to run LocalStack with DynamoDB.

Take some time to generate dependencies, start TAQUITO & call some routes using cURL.

Calling TAQUITO with cURL

```
## Create a Resource

$ curl -H "On-Behalf-Of: lmcrae@stanford.edu" -X POST -H \
  "Content-Type: application/json" \
  -d@examples/deposit_object.json http://localhost:8080/v1/resource

## Retrieve the Resource

$ curl -H "On-Behalf-Of: lmcrae@stanford.edu" \
  http://localhost:8080/v1/resource/{UUID from Above} | json_pp
```

https://github.com/PenguinParadigm/samvera18apis/

TAQUITO Deep Dive: Main.go / Server.go

TAQUITO Deep Dive: “Full” Handlers

TAQUITO Deep Dive: Internal Services (Identifier, Permissions)

TAQUITO Deep Dive: Data is Hard (aka what validation where & validators)

Schedule (ish)

9-9:30	Introduction, Logistics, Goals
9:30-10:15	Designing our API (ReST, PCDM, OAS)
10:15-10:30	*Break*
10:30-12	Developing our API (Go, Go-Swagger, *Localstack*)
12-14:00	*Lunch Break (on your own)*
15-15:30	Containerizing our API (Docker)
15:30-15:45	*Break*
15:45-16:30	Deploying our API (AWS)
16:30-17	Conclusion & Bootcamp Retrospective

Infrastructure & Deployment: Docker

Overview

ReST API for the new service interface

JSON[-LD] for the service's data representation

Swagger for the API specification

Go for the service's programming language

★ **Docker** for deployment

AWS for infrastructure (ECS primarily)

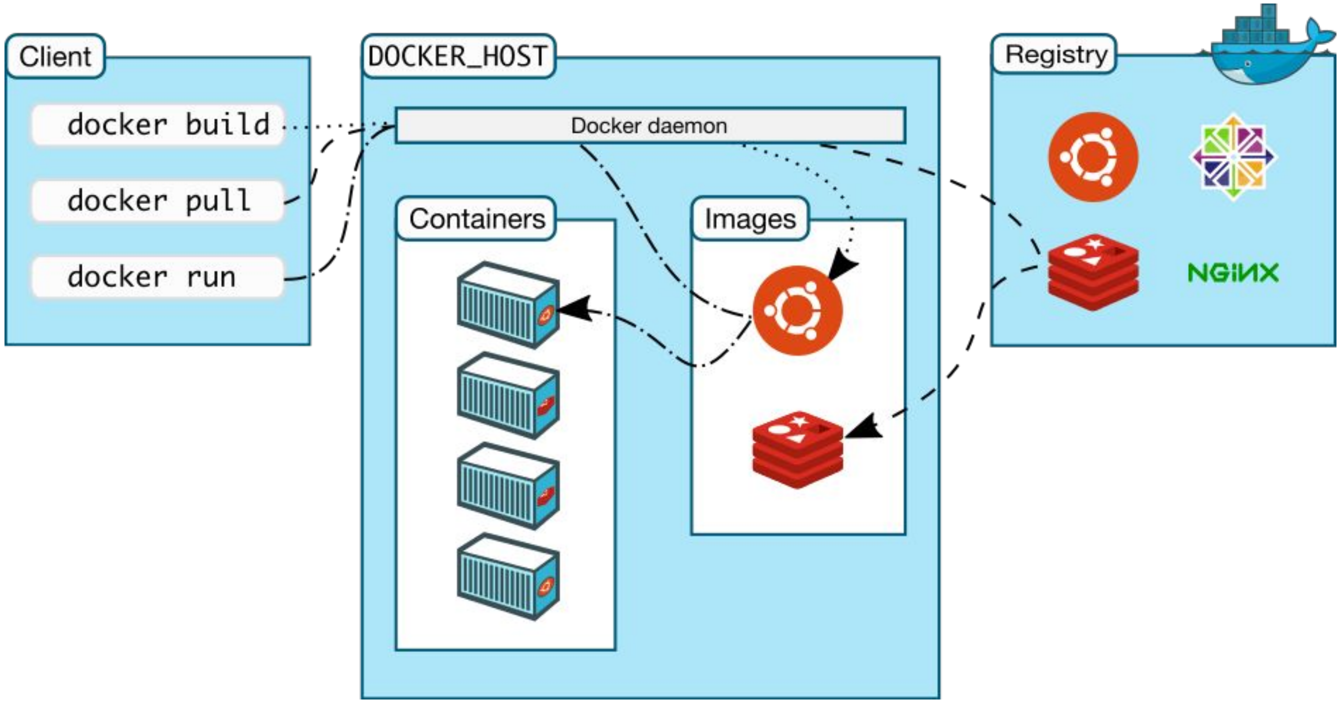
Docker quick intro

“Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.”

“Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel.”

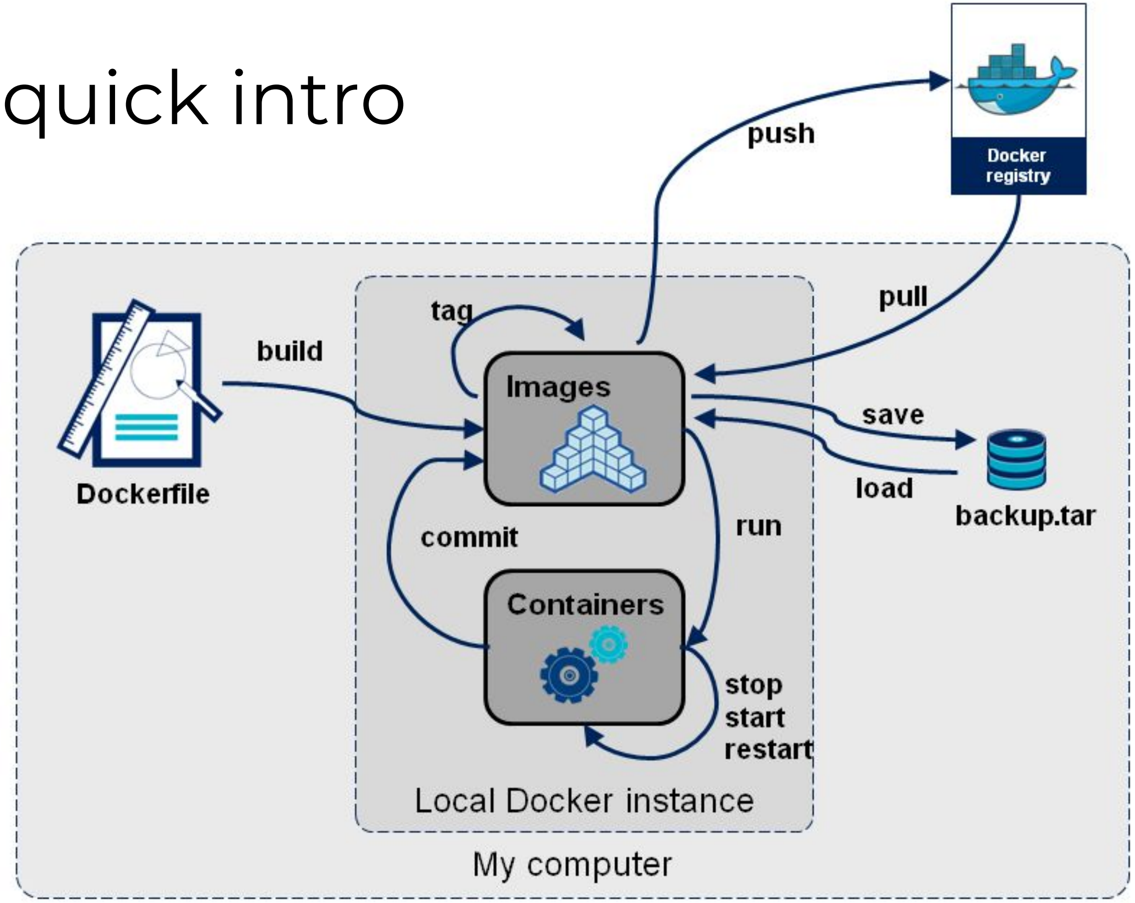
https://docs.docker.com/engine/docker-overview

Docker quick intro



https://docs.docker.com/engine/docker-overview

Docker quick intro



https://stackoverflow.com/questions/23735149/what-is-the-difference-between-a-docker-image-and-a-container

Docker quick intro

```
## List Docker CLI commands
docker
docker --help

docker container --help


## Display Docker version and info
docker --version
docker version
docker info
```

Docker quick intro

```
## Execute Docker image
docker run hello-world


## List Docker images
docker image ls


## List Docker containers (running, all, all in quiet mode)
docker container ls
docker container ls --all
```

Build our Docker container

```
FROM golang:alpine as BASE
WORKDIR /go/src/github.com/PenguinParadigm/samvera18apis/taquito
COPY . .
RUN apk update && apk add --no-cache ca-certificates && \
   apk add --no-cache --virtual .build-deps git && \
   go get -u github.com/golang/dep/cmd/dep && \
   dep ensure && \
   apk del .build-deps
RUN CGO_ENABLED=0 GOOS=linux go build -ldflags "-s" [...] main.go
```

https://github.com/PenguinParadigm/samvera18apis/

Build our Docker container

```
FROM scratch
EXPOSE 8080
COPY --from=BASE /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=BASE
/go/src/github.com/PenguinParadigm/samvera18apis/taquito .
CMD ["./api"]
```

Build our Docker container

```
## Build Taquito Docker image
docker build -t taquito .


## With a configured AWS cli, run the image
docker run -p 8080:8080 taquito


## cURL the running container
curl http://localhost:8080/v1/healthcheck
```

Coffee Break (15 minutes)

Take a few minutes to build & run your docker container, then do simple test calls with cURL.

Some Notes on TACO Deployment

- Circle-CI usage
- Docker-compose used for local testing
- Docker set-ups able to:
    - Run with localstack in its own container
    - Run with access to localstack’s endpoints on local machine
- “Continuous Deployment” (if you’re a member of Ops) via Docker Registry & GitHub repository branches management & tagging

Infrastructure & Deployment: AWS

Overview

ReST API for the new service interface

JSON[-LD] for the service’s data representation

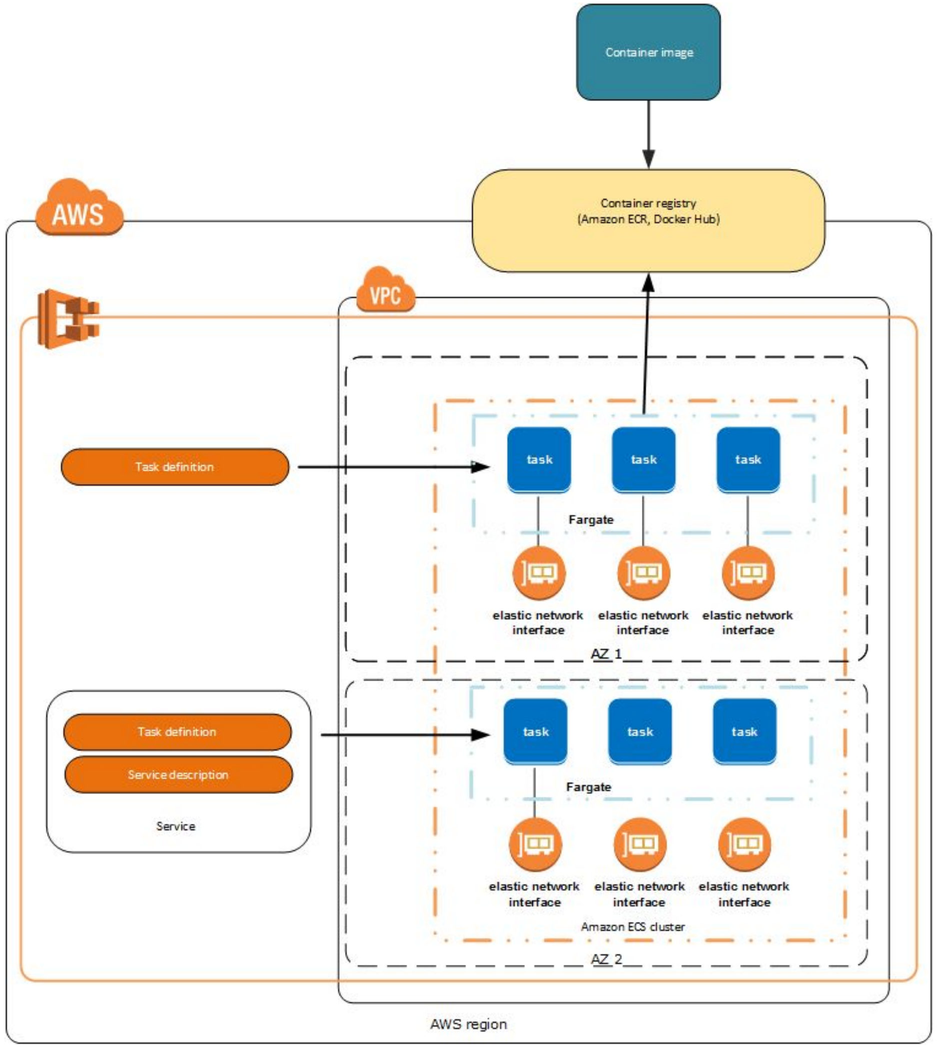
Swagger for the API specification

Go for the service’s programming language

Docker for deployment

★ **AWS** for infrastructure (ECS primarily)

AWS ECS Intro



https://github.com/PenguinParadigm/samvera18apis/

AWS Infrastructure Needs

So we know we need in AWS:

- ECS with appropriate task loaded
- DynamoDB with appropriate indices
- IAM Policies / Roles for the two to talk

Then after building that, we need a way to deploy our Docker container to that ECS cluster and test it.

Overview of the following steps

- Check your AWS Environment
- Check you are an admin (you should have done this already for AWS setup)
- Review our ECS Task
- Create an appropriate security group
- Create a cluster
- Launch a container instance
    - Select an AMI, t2.micro (*free tier eligible)
    - Check configurations, make sure its in your subnet, security group, etc.

Check Your AWS Environment Setup

```
$ aws --version

## See your profiles via ~/.aws/config

$ aws [--profile profile-name] configure list

$ aws [--profile profile-name] ecs list-clusters
```

Go through the steps in /aws directory in GitHub repository

Notes on TACO's Infrastructure

- Terraform
- CircleCI (again)
- Continuous Deployment via ECS setup

Tear down your AWS stuff

```
## Scale Down Your Service
$ aws ecs update-service --cluster taquito --service taquito \
     --desired-count 0 --region us-east-1


## Delete Service
$ aws ecs delete-service --cluster taquito --service taquito \
     --region us-east-1
```

Tear down your AWS stuff

```
## Deregister Container Instance

$ aws ecs list-container-instances --cluster taquito

$ aws ecs deregister-container-instance --cluster taquito
--container-instance {instance-id} --region us-east-1 --force


## Deregister Cluster

$ aws ecs delete-cluster --cluster taquito --region us-east-1
```

Tear down your AWS stuff

```
## Delete EC2 Instance

$ aws ec2 terminate-instances --instance-ids {ec2 instance id} \
     --region us-east-1


## Delete Security Group

$ aws ec2 delete-security-group --group-id {security group id}
```

Tear down your AWS stuff

```
## Delete Keypair

$ aws ec2 delete-key-pair --key-name ec2-key


## Delete Security Group

$ aws ec2 delete-security-group --group-id {security group id}
```

Tear down your AWS stuff

```
## Remove Roles

$ aws iam remove-role-from-instance-profile \
    --instance-profile-name ec2-profile --role-name ec2-role


## Remove Instance Profile

$ aws iam delete-instance-profile --instance-profile-name \
    ec2-profile
```

Tear down your AWS stuff

```
## Detach Policies from Role
$ aws iam detach-role-policy --role-name ec2-role --policy-arn
arn:aws:iam::aws:policy/service-role/AmazonEC2ContainerServiceRole

$ aws iam detach-role-policy --role-name ec2-role --policy-arn
arn:aws:iam::aws:policy/service-role/AmazonEC2ContainerServiceforEC2
Role

## Remove Role

$ aws iam delete-role --role-name ec2-role
```

Conclusion & Retro

Mini-Retro or Plus / Delta on this Workshop

Plus (what you appreciated)

Deltas (what you would change)

Share your takeaways, questions, or feedback on our work so far?

Thanks!