MDA DevOps for Data Science

MDA

2024-06-11

Table of contents

Preface

The MDA team at CORI will do a book club and review the book DevOps for Data Science written by **Alex K Gold** here.

- Book study repo
- Book study site
- Book study google drive

Planning:

1. Introduction + Environments as Code: Olivier
2. Data Project Architecture: John
3. Databases and Data APIs: Brittany
4. Logging and Monitoring: John
5. Deployments and Code Promotion: Camden
6. Demystifying Docker: John
7. ...

Workflow:

1. Clone this repo

2. Create a specific branch:

```
git checkout -b ch-?/???
# (replace `?` with the chapter number and `???` with a name/title)
```

3. Install Quarto CLI

4. Install local dependencies:

```
npm install
```

6. Preview Quarto site:

```
npm run preview
```

7. Make changes and commit as needed
8. Push commits and create a pull request

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Introduction

Learning objectives:

- introducing some definitions and a bit of history

Definitions

production :

> affecting decision in your orgs / world
>
> putting your work in front of someone else's eyes

We want:

- our works to be **reliable**
- in a safe **environment**
- our work to be **available**

DevOps

> DevOps is a set of **cultural norms**, **practices**, and **tooling** to help make developing and deploying software **smoother** and **lower risk**.

.. but is a squishy concept and a "vendor" associated name

It came in opposition to the *waterfall dev process* were you had a team doing Dev. and then one other doing Ops (Ops "make it works on everyone computer").

Process and people

- Are data scientist software developper?
- Are we in the red flags number 3?
- Do we need a workbench (ie using ec2) ?
- should we do the exercice with `penguins` or one of our dataset?

Part I

DevOps Lessons for Data Science

> You are a software developer.

But:

- Writing code for data science is different than writing code:

  > You're pointed at some data and asked to derive value from it without even knowing if that's possible.

- difference between architect and archaeologist

1 Environments as Code

1.1 Environments:

- stack of software and hardware below our code
- should be treated as "cattle not pet" / should be *stateless*
- Risk of it "only works on my machine"

Building a completly reproducible environement is a "fool's errand" but first step should be easy.

(any trouble with renv and sf anyone?)

1.2 Environements have layers

Layer	Contents	Example
Packages	R packages	cori.db
System	R versions / GDAL / MacOS	14.4.1
hardware	Physical / Virtual hardware	Apple M3

Hardware and System should be in the hand of IT (see later chapter 7 and 14), packages layer should be the data scientist.

1.3 The package layer

Package can in 3 places:

- repository: CRAN / GH / "Supermarket"
- library: a folder on a drive / "pantry"
- loaded : "ready to cook"

Each **project** should have it's own "pantry"

Project was higlighted in text but I think it is important: if you do not have a project workflow it is way harder to do it.

A package environement shouldbe :

- isolated and cannot be disrupted (example updating a packge in an other project)
- can be "captured" and "transported"

In R: `{Renv}` ("light"/"not exactly the same" option also exist, Box, capsule)

Author does not like Conda (good to not being alone!)

1.4 Workflow

- Create a standalone directory with a virtual environment

(spend time exploring `renv/` and `.gitignore`)

- Document environment state (see `lockfile`)
- Collaborate / deploy: you can't share package because their binay can be OS or system specific, hence specific package need to be installed (could be a pain point).
- Use virtual env

1.5 Under the hood

- test `.libpaths()` in a specific project and in a "random" R session
- order of Paths matter

1.6 Key points

- being in production is what make a DS a software developper
- kill and create new environment fast is important

2 Data Project Architecture

2.1 Key Takeaways

This chapter gives an opinionated overview of good design and conceptual layout practices in regards to a data project. The areas of responsibility within the project are broken out into

1) *Presentation*,
2) *Processing*, and
3) *Data* layers.

The categories that a given data project may fall into our further divided into

1) *jobs*,
2) *apps*,
3) *reports* and
4) *API's*.

The rest of the chapter discusses how to break a project down into the previously mentioned layers, as well as considerations for optimizing the Processing and Data layers.

2.2 Lab / Project

2.2.1 Initial Setup

The last chapter, Environments as Code, introduced the example project that we will use throughout the book. You can either ~~clone a starter template for~~ fork the project repo from do4ds_project or create the project from scratch yourself using the following Quarto CLI commands (taken from the Quarto documentation):

```
quarto create project website do4ds_project
# Choose (don't open) when prompted

quarto preview do4ds_project
```

… if the `quarto preview` command loads a new website in your web browser, go back to the terminal and use `Ctrl+C` to terminate the preview server. Change to the project directory and setup a local python virtual environment (you can grab the `requirements.txt` file from here, if needed):

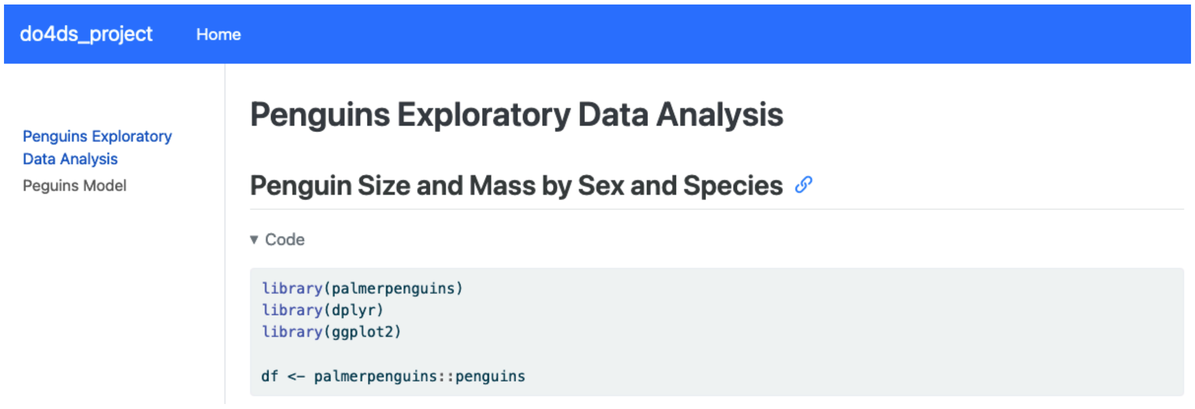
```
cd do4ds_project
# If using python, create and activate a local virtual environment
python -m venv ./venv
source venv/bin/activate
venv/bin/python -m pip install -r requirements.txt
```

Now that you are in the local project directory you can use the `quarto preview` command without arguments to continue seeing updates to the local project in your browser:

```
quarto preview

# Alternately, if you forked the project sample from Github, you can use npm...
npm run preview
```

If you did not fork the project sample, make sure to create the `eda.qmd` and `model.qmd` files from chapter 1 and add them to the sidebar section of `_quarto.yml`:

```
project:
  type: website

website:
  title: "do4ds_project"
  navbar:
    left:
      - href: index.qmd
        text: Home
  sidebar:
    style: "docked"
    search: true
    contents:
      - eda.qmd
      - model.qmd
```



Penguins Exploratory Data Analysis

Penguin Size and Mass by Sex and Species

```
library(palmerpenguins)
library(dplyr)
library(ggplot2)

df <- palmerpenguins::penguins
```

2.2.2 Updates

To complete part 1 of the lab, I had to modify the example code. First, I added a line that would generate a `vetiver` model and assign it to `v` and then I changed the path to the local folder where the model could be stored:

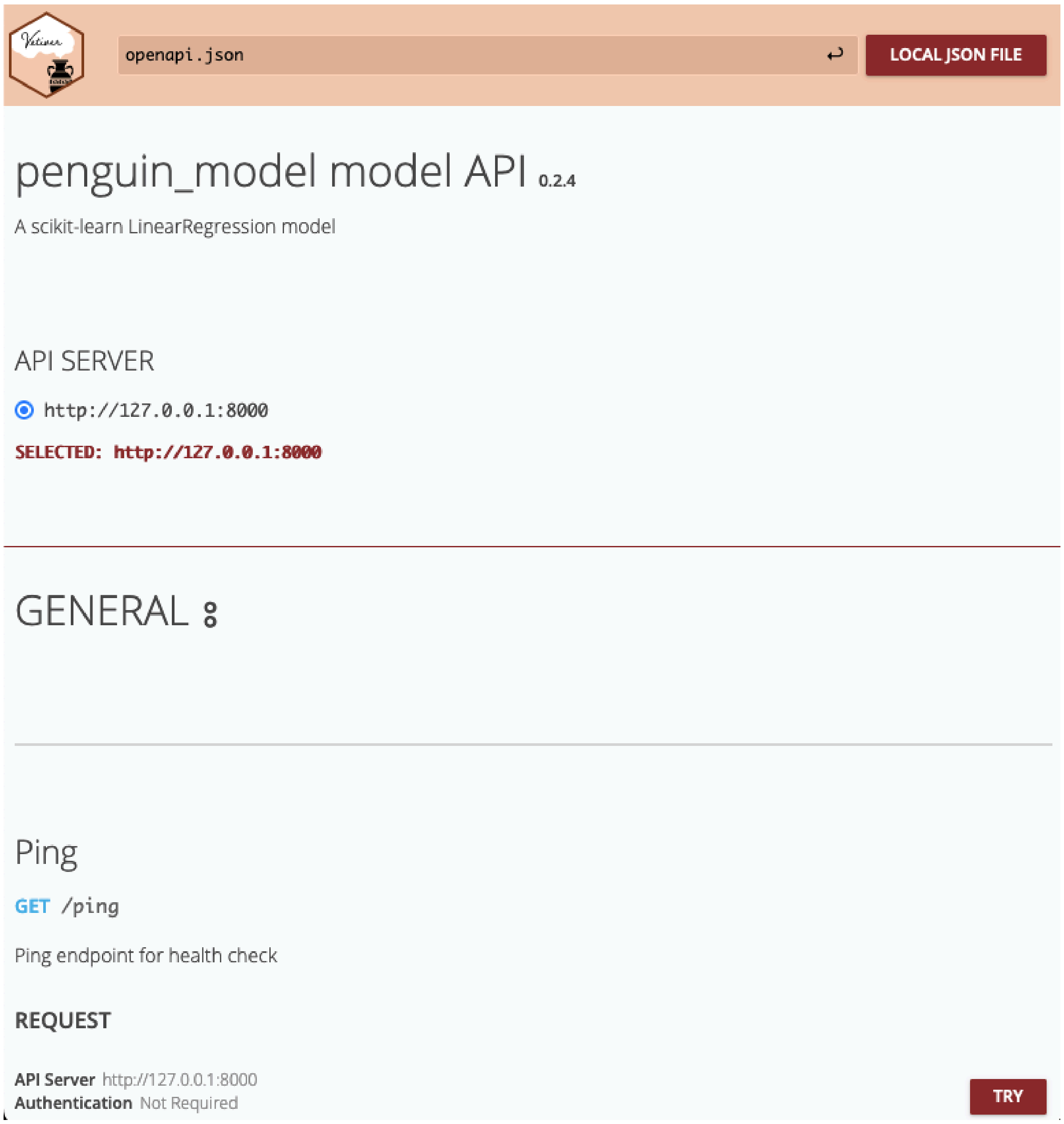
```
from pins import board_folder
from vetiver import vetiver_pin_write
from vetiver import VetiverModel

v = VetiverModel(model, model_name = "penguin_model")

model_board = board_folder(
  "data/model",
  allow_pickle_read = True
)
vetiver_pin_write(model_board, v)
```

In addition to these changes, I created a separate Python file with the code to run the `vetiver` API, called `api.py`, which also required updates to the `VetiverApi` call to ensure that the API server had the correct input params in order to process the prediction:

```
from palmerpenguins import penguins
from pandas import get_dummies
from sklearn.linear_model import LinearRegression
from pins import board_folder
from vetiver import VetiverModel
from vetiver import VetiverAPI
```

```
# This is how you would reload the model from disk...
b = board_folder('data/model', allow_pickle_read = True)
v = VetiverModel.from_pin(b, 'penguin_model')

# ... however VertiverAPI also uses the model inputs to define params from the prototype
df = penguins.load_penguins().dropna()
df.head(3)
X = get_dummies(df[['bill_length_mm', 'species', 'sex']], drop_first = True)
y = df['body_mass_g']

model = LinearRegression().fit(X, y)

v = VetiverModel(model, model_name = "penguin_model", prototype_data = X)

app = VetiverAPI(v, check_prototype = True)
app.run(port = 8000)
```

… and then used `python api.py` to run the API. Once running, you can navigate to http://127.0.0.1:8000/**docs** in a web browser to see the autogenerated API documentation



openapi.json

LOCAL JSON FILE

penguin_model model API 0.2.4

A scikit-learn LinearRegression model

API SERVER

http://127.0.0.1:8000

SELECTED: http://127.0.0.1:8000

GENERAL

Ping

GET /ping

Ping endpoint for health check

REQUEST

API Server http://127.0.0.1:8000
Authentication Not Required

TRY

3 Databases and Data APIs

3.1 Key Takeaways

Although it's not explicitly described in this chapter, a database is, at minimum, two things:

1) a datastore (i.e., a way to store structured data) and
2) a compute engine (i.e., a way to perform operations on structured data).

In most programming languages, the way you begin an interaction with a database is to create a connection object. The methods and params used to create that object will vary from database type to type and from language to language, including how credentials are provided or set, but the successful creation of this object implies that you have an open connection to the database with a specific set of permissions (read/write/remove, etc.).

Another source of data may be an API, or Application Programming Interface. The term "API" is actually used in this chapter in two different contexts; the first is to describe the method (or interface) used in particular programming languages to connect to a database, but in the second usage the context is *REST* or REST-ful API's. These are collections of URL's that allow a server to listen for requests for data from clients. Once a client makes a request to a given URL with specified parameters, the server *evaluates* the request, computes a response and returns the results. Although this is very similar to how a database operates, a REST-ful API uses a specific protocol called the HyperText Transfer Protocol (HTTP). This protocol uses the *parts* of the URL (i.e., the path and the query string) and the *verb* of the client request (i.e., GET, POST, PUT, DELETE, etc.) as a way to specify how the server should evaluate the request. This protocol is widely available in many applications, including those that do no support any database drivers (like web browsers). Each programming languages has specific packages or libraries for making HTTP requests, as well as packages that provide "wrapper" functions; functions that hide the details of each API request from the developer and simply take arguments and return results in a synchronous manner. In addition to parameters, a REST-ful API endpoint may accept a *body* (block of code or text) as part of the request, usually in the form of a JSON object, which is often also the preferred format of the response body (again, block of code or text) chosen by API developers. There are various ways to include credentials with HTTP requests.

While I believe the topic should have been introduced earlier in this book, this chapter also discusses environment variables and how they can be used to provide configuration settings to the process that the code is running within, including database or API credentials.

At CORI/RISI we have our own "data connector" package called `cori.db`, which allows each of us to connect directly to the database and interact with it in a prescribed way. At the moment, we do not have an equivalent set of function to interact with the CORI Data API, although we have created a placeholder package called `cori.data.api` which may one day provide such functions in `R`.

3.2 Lab / Project

3.2.1 Updates

The lab work introduces the `duckdb` package, so I created a new markdown file called "00_data.qmd" in the project sample that demonstrates the most basic syntax for creating a new database (which is stored in a flat file on local disk) and loading the penguins data into a new table in that database. To make sure that this file is processed before the others, I prefixed the names of all the quarto files with sequences numbers ("00_...", "01_...", etc.). The model and api code has been updated to retrieve the penguin table from the local database store with `duckdb`. Olivier added an "02_model_R.qmd" file to demonstrate the `R` code that is equivalent to the modeling example used in the book. The same could be done for the `vetiver` code used to deploy the resulting model as an API endpoint.

4 Logging and Monitoring

4.1 Key Takeaways

In any data processing and/or analysis, something is going to go wrong.

Data science (and by extension programmatic data analysis) is hard, because the output is "novel", so we should "use process metrics to reveal a problem before it surfaces in your results".

> Process metrics - quantitative and qualitative measures related to a process, its performance and its evolution.

The easiest level to observe or monitor process metrics is at the Data and Processing layers (why?).

The author believes that all job-type projects should be encapsulated in a *literate programming* format (i.e. Quarto markdown), so that the output (log) of the process is combined with notes about its context. Is this an evolution over comments in the code?

Things to check when transforming data:

- Quality of each join (row count)
- Cross-tabulation before and after recoding (`dplyr::mutate`)
- Fitness of model (i.e. bias, variance, drift, etc.)

For the purposes of logging, `print` is a good start, but consistently using a logging package is better. The author recommends`log4r`. With this package, the output (like most conventional log outputs) has 3 components:

- *log metadata*
- *log level*
- *log data*

Log level commonly uses the following ranking:

- DEBUG
- INFO
- WARN (or WARNING)
- ERROR

- CRITICAL

Log messages are often formatted as either plain text of JSON. Log output defaults to the current terminal, but can be directed to a file on local disk, a remote store like S3, or to a logging API like AWS CloudWatch.

4.2 Lab / Project

4.2.1 Updates

We started logging messages and errors in the `api.py` and `app.R` scripts, with the respective `logging` and `log4r` packages.

5 Chapter 5 Notes

5.1 Key Takeaways

Benefits of CI/CD: - Ability to build your code from scratch - Source code is always available in a central repository - Automated testing and deployment simplifies and standardizes the testing/deployment process

Tactics: - Separate the dev and prod environments - Changes only happen through a promotion process from dev (through test) and into prod - Code promotion happens via Version Control (e.g., git)

Author's suggestion for managing data science projects: 1. Maintain two long-running branches – main is the prod version of your project and test is a long-running pre-prod version. 2. Code can only be promoted to main via a merge from test. Direct pushes to main are not allowed. 3. New functionality is developed in short-lived feature branches that are merged into test when you think they're ready to go. Once sufficient approvals are granted, the feature branch changes in test are merged into main.

Other: - Use YAML for conditional environments

...

5.2 Lab / Project

5.2.1 Updates

...

6 Key Takeaways

...

6.1 Lab / Project

6.1.1 Updates

...

Part II

IT/Admin for Data Science

7

8

9

10

11

12

13

14

Part III

Enterprise-grade Data Science

15

16

17

18

A

B

C

D