First Web App in Rust

Adrian Kosmaczewski

2021-05-21

My exploration of Rust continues; this week, I rewrote a Python Flask application I use for demos at work.

- Python Application
- Let's Get Rusty
- The HTML Template
- Container Image and Dockerfile

Python Application

The original app is a simple Flask application, running with Python 3.7, which just outputs a random number and a funny message when you use it. The funny message comes from the fortune program, which must be installed in the host.

Here is the code:

```
import os
from flask import Flask, request, Response, jsonify
from flask.templating import render_template
from subprocess import run, PIPE
from random import randrange

app = Flask(__name__)

version = '1.0'

def get_fortune():
    number = randrange(1000)
    fortune = run('fortune', stdout=PIPE, text=True).stdout
    return number, fortune

@app.route("/")
def fortune():
    number, fortune = get_fortune()
    mimetype = request.mimetype
    if mimetype == 'application/json':
```

```
        resp = jsonify({ 'number': number,
                         'fortune': fortune,
                         'version': version })
        return resp

    if mimetype == 'text/plain':
        result = 'Fortune %s cookie of the day #%s:\n\n%s' % (version, str(number), fortune)
        resp = Response(result, mimetype='text/plain')
        return resp

    html = render_template('fortune.html', number=number, fortune=fortune, version=version)
    resp = Response(html, mimetype='text/html')
    return resp

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=os.environ.get('listenport', 9090))
```

There's just one endpoint, and the whole thing listens on port 9090. Depending on the `Content-Type` header of your request, the app spits out JSON, plain text, or just good old HTML, generated using Jinja.

Let's Get Rusty

After learning about Rust web frameworks, here's my translation of the application above using the Actix crate, together with the Askama template library.

```
use actix_web::{get, App, HttpRequest, HttpResponse, HttpServer, Responder};
use askama::Template;
use rand::Rng;
use serde::{Deserialize, Serialize};
use std::process::Command;

#[derive(Template)]
#[template(path = "fortune.html")]
#[derive(Serialize, Deserialize)]
struct FortuneData {
    number: u32,
    fortune: String,
    version: String,
}

impl FortuneData {
    fn new() -> FortuneData {
        let mut rng = rand::thread_rng();
        let command = Command::new("fortune")
            .output()
            .expect("failed to execute process");
```

```
        FortuneData {
            fortune: format!("{}", String::from_utf8_lossy(&command.stdout)),
            number: rng.gen_range(0..1000),
            version: String::from("1.0-rust"),
        }
    }

    fn to_plain(&self) -> String {
        format!(
            "Fortune {} cookie of the day #{}:\n\n{}",
            self.version, self.number, self.fortune
        )
    }
}

fn get_content_type<'a>(req: &'a HttpRequest) -> Option<&'a str> {
    req.headers().get("Content-Type")?.to_str().ok()
}

#[get("/")]
async fn fortune(req: HttpRequest) -> impl Responder {
    let response = FortuneData::new();
    match get_content_type(&req) {
        Some(content_type) => {
            if content_type == "text/plain" {
                return HttpResponse::Ok().body(response.to_plain());
            }
            if content_type == "application/json" {
                return HttpResponse::Ok().json(response);
            }
        }
        None => {}
    }
    HttpResponse::Ok().body(response.render().unwrap())
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().service(fortune))
        .bind("0.0.0.0:9090")?
        .run()
        .await
}
```

This Rust application does exactly the same as the previous one in Python, and both can be stored in a nice container, to be executed in your nearest Kubernetes

cluster.

The HTML Template

The nice thing of this rewriting was that I could reuse the same `struct FortuneData` for creating JSON, and also to feed the Askama template. Very handy and very elegant.

This is, by the way, the HTML template used by both applications, reused without changes from one app to the other, and bearing the filename `templates/fortune.html` in both cases.

```
<html>

<head>
    <title>FORTUNE COOKIE</title>

    <style>
        * {
            color: green;
            font-family: Cambria, Cochin, Georgia, Times, 'Times New Roman', serif;
            font-size: x-large;
        }

        #main {
            width: 80%;
        }
    </style>
</head>

<body>
    <div id="main">
        <h1>Fortune cookie of the day #{{ number }}</h1>

        <p>{{ fortune }}</p>

        <hr>

        <p>Version {{ version }}</p>
    </div>
</body>

</html>
```

Container Image and Dockerfile

The main difference between both apps, beyond the obvious syntactic differences, had to do with the creation of container images and the required `Dockerfile`. While in the case of the Python app, the build process was short and the resulting image was huge, in the case of Rust the build process was awfully long, and the resulting image was subatomically small. Talk about tradeoffs.

To reduce the build times of the container image, I used the trick shown in this blog post which consists of having two separate `cargo build` commands; the first one installs the dependencies, which seldom change, while the second one actually compiles the executable from the main Rust file, which of course changes more often.

Let's compare the Dockerfiles, both using Alpine as a basis. First, the Python version:

```
FROM python:3.7-alpine
RUN apk add fortune
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py /usr/src/app
COPY templates /usr/src/app/templates/
USER 1001
EXPOSE 9090
CMD [ "python", "app.py" ]
```

Simple and sweet. Of course, the Python runtime increases the size of the final image quite dramatically.

And now the Rust one, itself using a multi-step approach:

```
# Step 1: builder image
# We're using musl to generate smaller images based on Alpine Linux
# Base image: https://github.com/emk/rust-musl-builder
FROM ekidd/rust-musl-builder:stable as builder

# This Dockerfile contains two `cargo build` commands;
# The first compiles the dependencies of the application
# while the second one compiles and links the app itself, which
# changes less often than the dependencies.
# Inspired from
# https://blog.logrocket.com/packaging-a-rust-web-service-using-docker/

# Compile dependencies
RUN USER=root cargo new --bin rust-fortune
WORKDIR ./rust-fortune
```

```
COPY ./Cargo.lock ./Cargo.lock
COPY ./Cargo.toml ./Cargo.toml
RUN cargo build --release

# Compile the app itself
RUN rm src/*.rs
RUN rm ./target/x86_64-unknown-linux-musl/release/deps/rust_fortune*
ADD src ./src
ADD templates ./templates
RUN cargo build --release

# Step 2: runtime image
FROM alpine:latest
EXPOSE 9090

# Add dependencies
ENV TZ=Etc/UTC
RUN apk update \
    && apk add --no-cache fortune ca-certificates tzdata \
    && rm -rf /var/cache/apk/*

# Copy self-contained executable
COPY --from=builder /home/rust/src/rust-fortune/target/x86_64-unknown-linux-musl/release/rus

# Never run as root
USER 1001:0
CMD ["/usr/local/bin/rust-fortune"]
```

The final result is a super small image, holding a single executable that contains everything it needs to run; even the HTML template is embedded inside of the application, and you don't need to copy it separately. Of course, containers based on this image start and stop super fast, and consume less memory than the other.

So, as I said; your tradeoff is a bit longer build time for less memory consumption and faster execution. I'd say it's totally worth it, particularly if your final image is created in a CI/CD system such as a GitLab pipeline, where you could not care less about building times.