KGD

[KG1] Endpoint Definitions

File Reference:
`verification-service/app/main_fastapi.py:24-42`

Code Fragment:

```
@app.get("/health", response_model=model.HealthCheckResponse)
async def health_check(x_correlation_id: str = fastapi.Header()) ->
fastapi.Response:
    # Currently, nothing can cause a service to report itself as unhealthy
    status: common.model.HealthCheckStatus = "healthy"
    status_code = 200 if status == "healthy" else 503
    # Use methods defined in /common to run health checks
    dependencies: typing.Dict[str, common.model.HealthCheckDependency] = {
        "redis": common.api.redis.check_health(),
    }

    response_content = model.HealthCheckResponse(
        status=status,
        dependencies=dependencies,
    )

    return fastapi.Response(
        content=response_content.model_dump_json(exclude_none=True),
        status_code=status_code,
    )
```

Justification:
This is the definition of the healthcheck endpoint for the verification service.
This code fragment is an example of the standard way of defining API endpoints with FastAPI.
It also shows that I understand status codes.

[KG2] Containerization

File Reference:
`verification-service/Dockerfile:1-11`

Code Fragment:

```
FROM python:3.11-slim

WORKDIR /app

COPY ./requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY ./app .
COPY --from=common . ./common

ENV PYTHONUNBUFFERED=1
```

Justification:
This Dockerfile defines the build process for the `verification-service` microservice.
By setting the base image, installing dependencies, and defining the startup command, it bundles the application code and environment into a single, isolated, and portable container image, which is the definition of Containerization.
This also shows that I know how to modify Dockerfiles to add the dependencies that I want, since I customize the copy flag, and omit the command, since it is split into a worker and API server.

[KG3] Container Orchestration

File Reference: `scripts/generate_compose.py:59-80`

Code Fragment:

```
dependency-worker:
  build:
    additional_contexts:
    - common=./common
    context: ./dependency-service
    dockerfile: Dockerfile
  command:
  - celery
  - --app
  - main_celery
  - worker
  - --loglevel=info
  - -Q
```

```
    - dependency
    - -c
    - '1'
    depends_on:
      rabbitmq:
        condition: service_healthy
    deploy:
      mode: replicated
      replicas: 2
```

Justification:
The `build` item specifies a common library using additional contexts, which allows for code to be shareed across containers and prevents model diffs from causing internal errors.
It also has a dependency on the `rabbitmq` service, which ensures that it is started at the correct time.
Finally, it is replicated twice, which shows that the container is configured with Docker Compose's scaling mechanism in mind.

[KG4 & KG5] API Gateway & Load Balancing

File Reference: `nginx/nginx.conf:1-3`

Code Fragment:

```
upstream dependency-service {
    server dependency-service:8000;
}
```

Justification:
This is the upstream block for the dependency service in my nginx.conf.
This demonstrates load balancing, since it directs to the `dependency-service:8000` origin, which is resolved by Docker's DNS so as to balance incoming requests across various services.
This demonstrates the usage of an API gateway because the upstream block defines the locations within our network that Nginx can forward external requests to.

[KG6 & KG11] Reliability, Scalability, and Maintainability + Asynchronous Messaging

File Reference: `rabbitmq/definitions.json:23:34`

Code Fragment:

```
"policies": [
    {
        "vhost": "/",
        "name": "max-queue-length",
        "pattern": ".*",
        "apply-to": "queues",
        "definition": {
            "max-length": 3,
            "overflow": "reject-publish"
        }
    }
]
```

Justification:
This is part of my RabbitMQ (async messaging) configuration that sets a default maximum queue length for all queues.
The max length ensures that the system can handle a reasonable amount of overload (scalability) while also preventing massive influxes of work from starving the queue for excessive periods of time (reliability).
This configuration ensures that when a moderate overload is encountered, tasks are queued up, whereas during massive ingresses (eg. DOS attack), the service immediately indicates that it is overloaded (reliability).

[KG7] Service Communication

File Reference: `dependency-service/app/main_fastapi:89-104`

Code Fragment:

```
# Request verification service to verify the proofs
try:
    httpx.post(
        url="http://verification-service:8000/run",
        json={"repo_url": request.repo_url, "commit_hash": request.commit},
        timeout=30,
    ).raise_for_status()
except Exception as e:
    logger.error(f"Exception while requesting verification: {e}")
    return fastapi.responses.JSONResponse(
        content={
            "error": "Failed to request verification",
            "message": str(e),
        },
        status_code=500,
    )
```

Justification:
This code fragment is where `dependency-service` requests for the project to be processed by the `verification-service`.
This is an appropriate location for service commuication, because `dependency-service` and `verification-service` have different responsibilies, but `dependency-service` still needs to trigger an event in `verification-service`.