After our application runs, we now want to operate this application. This tutorial cares about monitoring.

Task 1: Install Monitoring in Kubernetes

Our cluster is running but we have no idea what resources are used an how.
Within Kubernetes, Prometheus is the de facto standard tool for monitoring. Working with loosely coupled components, Prometheus is extensible and stable….and yet easy to install in vanilla Kubernetes via helm.

Since we will adapt Prometheus, we will consume it via helm and deploy it via helm. The referenced helm chart containing dashboards, configuration and so on is available under https://prometheus-community.github.io/helm-charts

1. Create a new new space called "monitoring"

2. Install the chart via Helm:
   ```
   helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
   helm repo update
   helm install prometheus prometheus-community/kube-prometheus-stack -n monitoring
   ```

3. The stack is complete: Prometheus, Grafana, Alertmanager…all is included.
   However, the possibility to gain metrics from grafana, argocd and the applications is missing. We have to adjust the setting and use an own values.yml next week.

Task 2: Adapt Eliza/ConnectingWorlds to expose metrics

As for now, Eliza and Connecting Worlds do now expose any metrics regarding their runtime or their business metrics.

1. Enable Quarkus to expose prometheus-readable metrics is as easy as to insert a new maven-dependency:
   ```
   <dependency>
     <groupId>io.quarkus</groupId>
     <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
   </dependency>
   ```
   Afterwards, metrics should be visible from the started application under http://localhost:8080/q/metrics

2. So far, only JVM-/ and Runtime-Metrics are exposed. However, we would also like to create business metrics. You can find documentation under https://quarkus.io/guides/micrometer and https://quarkus.io/guides/telemetry-micrometer-tutorial .
   Annotate the get-function in quarkus to generate information about the time consumed within each invocation of the get-request.

3. Business metrics are important and easy to integrate. Think of a custom metric and add at least one:

1. Add the `MeterRegistry` into the class according to the documentation above

2. You can add afterwards in any method counters and gauges like:
   `registry.counter("example.message", "label1","value1").increment();`

4. Test your implementation locally: just invoke the application. The metrics are visible under http://localhost:8080/q/metrics

5. Do the same steps 1-4 for connection worlds.

   Please note that you need to invoke the application before any custom metrics for methods appear.

Task 3: Adapt Roberta to expose metrics

Now, we focus on Roberta / Flask to expose metrics as well as liveness/readiness.

1. I recommend using prometheus-flask-exporter as dependency. Just add to to the requirements.txt and install it

2. Standard-Metrics can be exposed via:
   `metrics = PrometheusMetrics(app, group_by='endpoint')`

   Check http://127.0.0.1:7070/metrics for the metrics exposed

3. Think of one (or more) useful custom metric(s) and add them to the exporter according to the documentation:
   https://pypi.org/project/prometheus-flask-exporter/ . There is a direct applicable example under https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend/app.py

   One way is the instantiation of e.g. a Counter before all methods:
   `my_counter = Counter('metric_name', 'Metric Description', ['label1', 'label2'])`

   In a method, you can increment it via:
   `my_counter.labels(labelValue1, labelValue2).inc()`