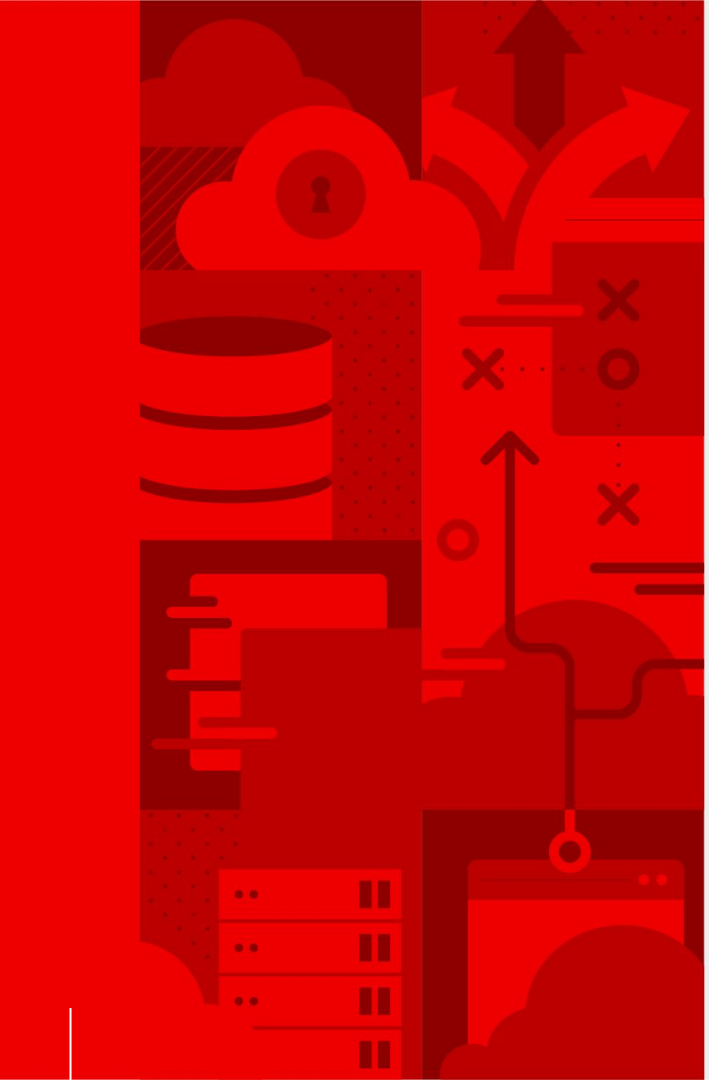


Kubernetes Native Developer

GENERAL DISTRIBUTION

Architecture Workshop

Working with Quarkus

https://github.com/wpernath/quarkus-grumpycat

https://github.com/wpernath/quarkus-demo-service

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat

Self introduction

Name: Wanja Pernath

Email: wpernath@redhat.com

Base: Germany (very close to the Alps)

Role: EMEA Technical Partner Development Manager - OpenShift and MW

Experience: Years of Consulting, Training, PreSales at Red Hat and before

Twitter: https://twitter.com/wpernath

LinkedIn:https://www.linkedin.com/in/wanjapernath/

GitHub: https://github.com/wpernath

First book just published

Getting GitOps

A technical blueprint for developing with Kubernetes and OpenShift based on a REST microservice example written with Quarkus

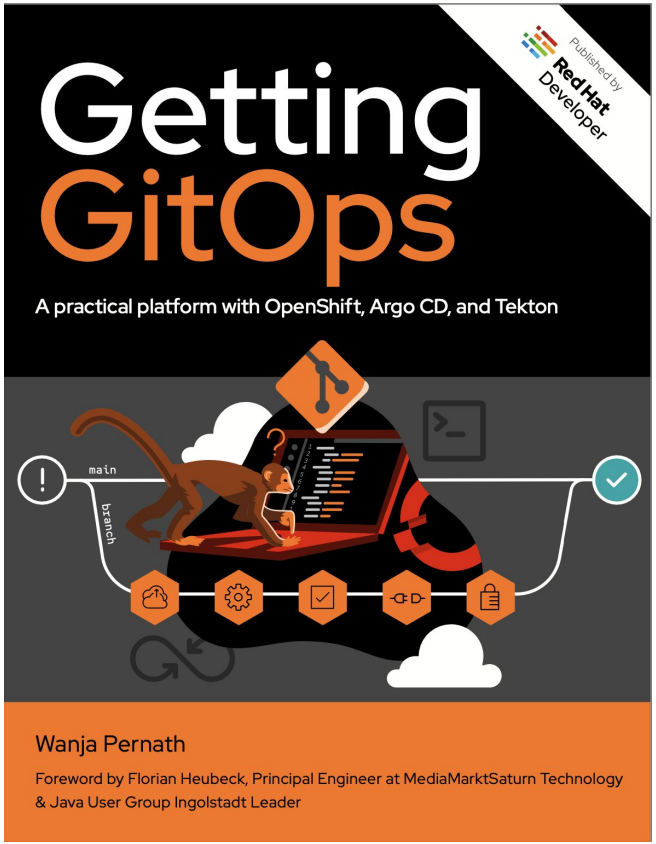
Technologies discussed:
Quarkus, Helm Charts, Kustomize, Tekton Pipelines, Kubernetes Operators, OpenShift Templates, ArgoCD, CI/CD, GitOps….

Download for free at:

https://developers.redhat.com/e-books/getting-gitops-practical-platform-openshift-argo-cd-and-tekton

Interview with full GitOps Demo:

https://www.youtube.com/watch?v=znMfVqAlRzY&ab_channel=OpenShift



Agenda

Agenda

- **Quarkus Concepts**
  - Basics
  - CDI
  - Dev Services
  - Extensions
- **Database / CRUD REST services**
- **Messaging (smallrye.io)**
  - WebSockets
  - Kafka
- **Qute Templating Engine**
- **Container Images**
  - Kubernetes
  - Health
  - Metrics
- **Security**
- **Caching**
- **Using Reactive with Mutiny**

Quarkus Concepts

Concepts

- **Runtime & CDI Framework**
- Runtime based on Eclipse Vert.x & Netty
  - Vert.x is a reactive, resource-efficient, concurrent & asynchronous runtime environment (think about NodeJS for Java)
- Eclipse MicroProfile based Framework
  - With use of SmallRye, Mutiny components for Reactive and Messaging
- Extension based!
  - You don’t need no Servlets? – Don’t use them
  - No JPA required? – Do not include jpa extension
- Can be used for CLI & “web” apps

Context & Dependency Injection (CDI)

```
import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.metrics.annotation.Counted;

@ApplicationScoped (1)
public class Translator {

    @Inject
    Dictionary dictionary; (2)

    @Counted (3)
    String translate(String sentence) {
      // ...
    }
}
```

1. This is a scope annotation. It tells the container which context to associate the bean instance with. In this particular case, a **single bean instance** is created for the application and used by all other beans that inject `Translator`.
2. This is a field injection point. It tells the container that `Translator` depends on the `Dictionary` bean. If there is no matching bean the build fails.
3. This is an interceptor binding annotation. In this case, the annotation comes from the MicroProfile Metrics. The relevant interceptor intercepts the invocation and updates the relevant metrics. We will talk about interceptors later.

Context & Dependency Injection (CDI)

- Quarkus is based on CDI 2.0 programming model
  - With some limitations
  - Read https://quarkus.io/guides/cdi
- Quarkus has its own CDI Reference Guide, which you can find here: https://quarkus.io/guides/cdi-reference

Runtime - Vert.x

- Quarkus is using Eclipse Vert.x underneath, a toolkit for building reactive applications (or NodeJS concepts for Java)
- Vert.x was originally invented by some former JBoss'ians and is now an Eclipse project
- **Important:** Although Quarkus is using Vert.x as it's runtime, you do NOT have to be reactive in your code (but you always could).

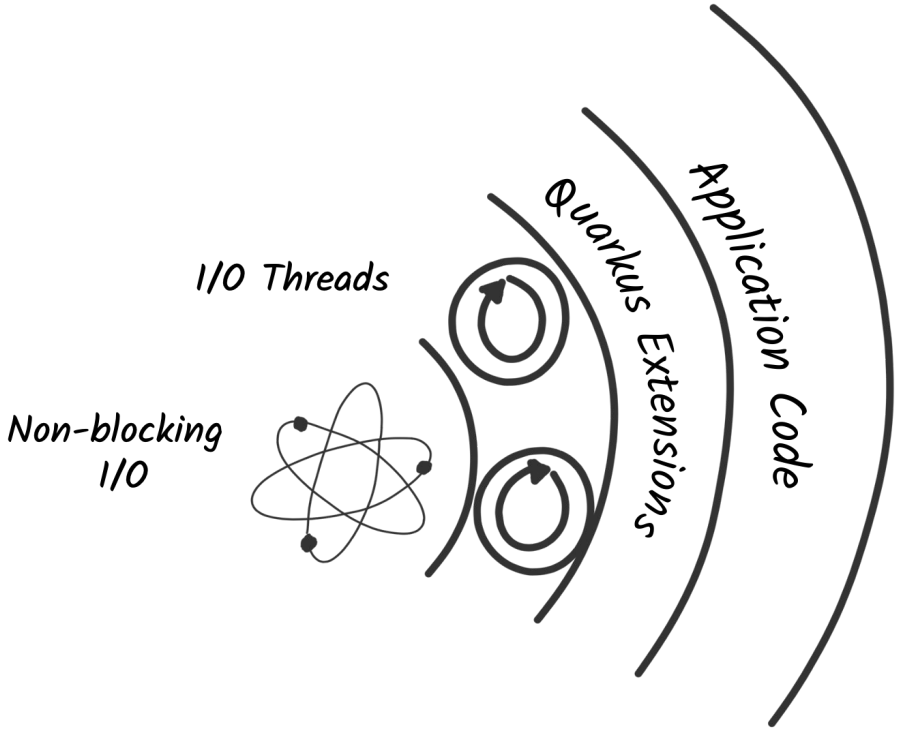
Runtime - Vert.x

Quarkus provides a managed Vert.x instance, which you could always directly use, if you have the need for it. – But you typically don't need to

Quarkus is hiding the complexity for you.

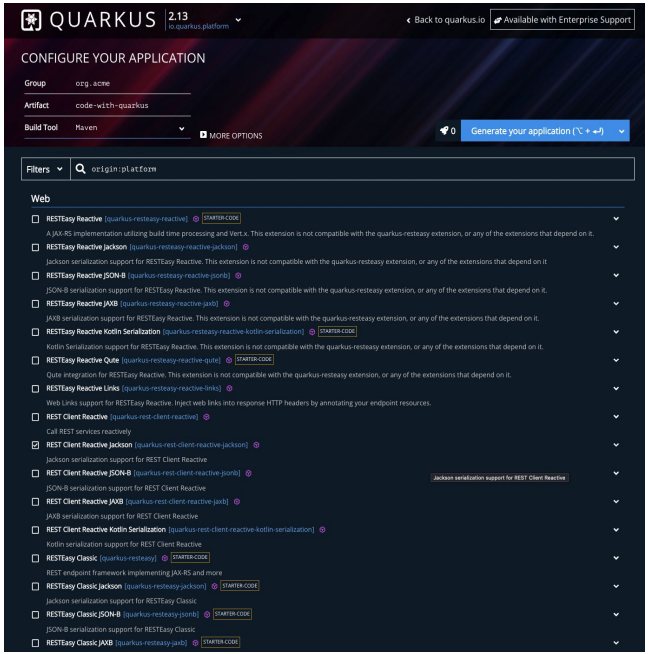
https://quarkus.io/guides/vertx

https://quarkus.io/guides/vertx-reference



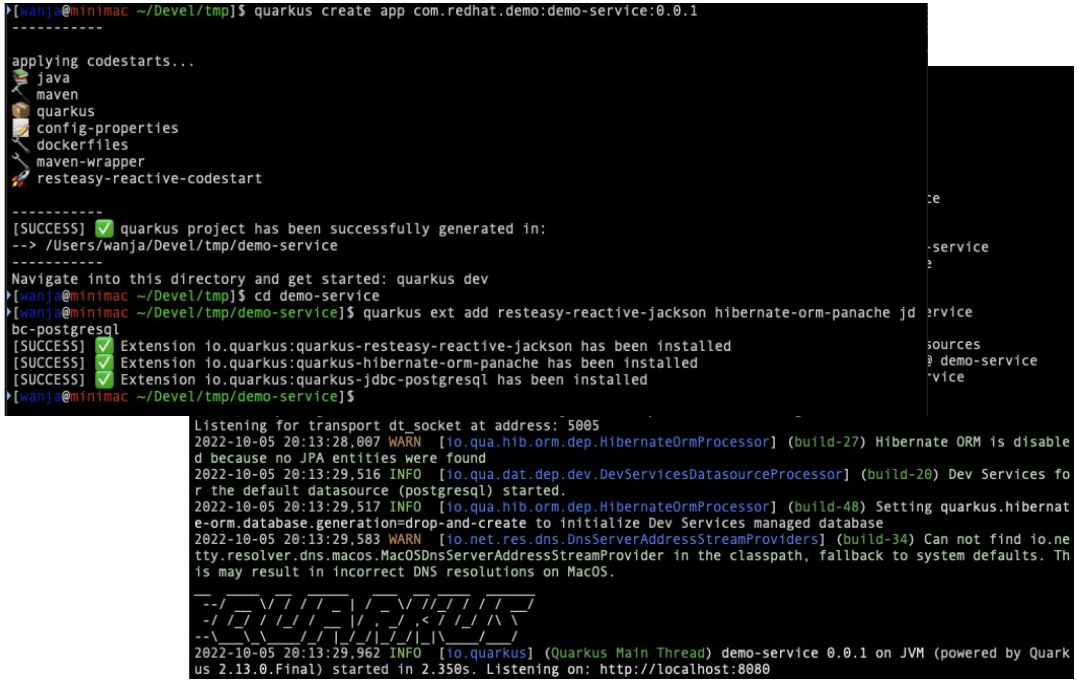
Extensions based

- Quarkus is organized in extensions. There is a core one (which you already have by using the project initialization of Quarkus). And then there are extensions sorted by use case.
  - REST
  - ORM
  - Messaging
  - Security (quickly integrate Keycloak?)
  - Kubernetes, Container-Images, Helm-Charts
  - Etc.
- All public extensions have an own how-to guide



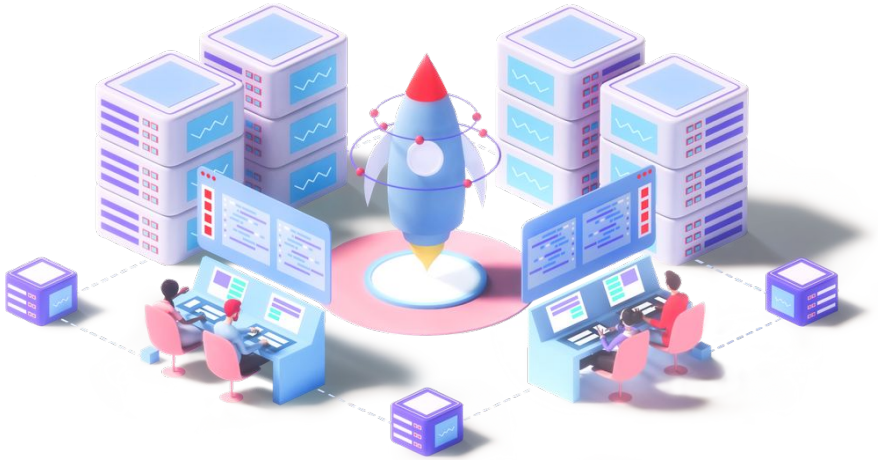
Extensions based

- Want to create a REST CRUD service with PostgreSQL?
  - Create your Quarkus skeleton
  - Add extensions to your code
    - Hibernate ORM
    - PostgreSQL JDBC driver
    - Add jackson / json-b
  - Create the Entity class
  - Create the REST service
  - Configure your service
  - Run quarkus dev
- You're done within 5 minutes!



Quarkus Dev Services

- Most services you require during development provide a Dev Service environment via the extension
- Example services are:
  - Databases
  - AMQP
  - Kafka
  - Keycloak
  - Redis, MongoDB
  - RabbitMQ
  - Vault
  - Infinispan
  - Elasticsearch



https://quarkus.io/guides/dev-services

Quarkus Dev Services

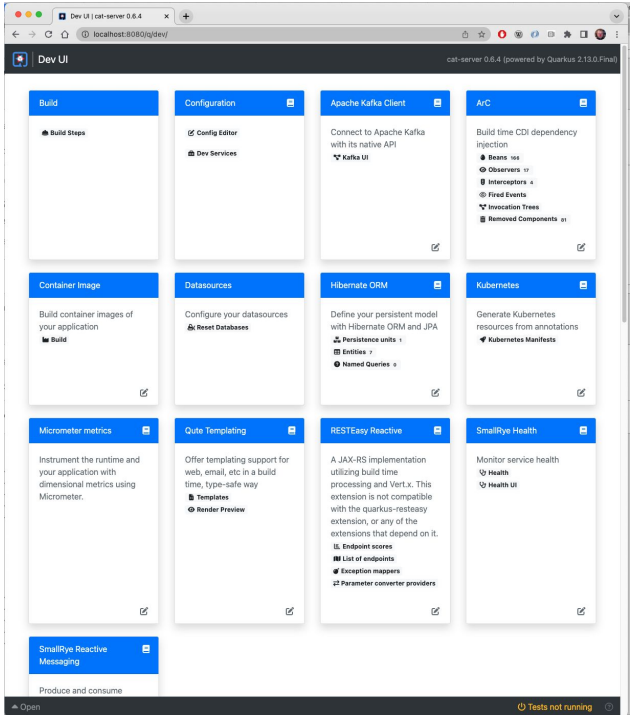
- DevServices require a working Docker Desktop environment (or a running Podman Desktop)
- Benefits?
  - Focus on what you need and implement it
  - Don't worry about local installation of PostgreSQL, Kafka… this comes for free
  - Speed up in Dev
- Configuration via application.properties
  - Use **%prod.**quarkus… to define the location of the service in Quarkus PROD environments
  - Use **${ENV_NAME}** to build your value

```
22  #
23  # Datasource options
24  #
    You, 4 months ago
25  quarkus.datasource.db-kind=postgresql
    wpernath, 2 months ago
26  quarkus.hibernate-orm.log.sql=false
27  quarkus.hibernate-orm.log.format-sql=true
28
29  # the following props only for production
    You, 2 months ago
30  %prod.quarkus.hibernate-orm.log.sql=false
31  %prod.quarkus.hibernate-orm.log.format-sql=false
    You, 4 months ago
32  %prod.quarkus.hibernate-orm.database.generation=update
    You, 4 months ago
33  %prod.quarkus.datasource.username=${DB_user:cat}
34  %prod.quarkus.datasource.password=${DB_password:grumpy}      You, 4 months ago • changed app
    You, 4 months ago
35  %prod.quarkus.datasource.jdbc.url=jdbc:postgresql://${DB_host:catserver}/${DB_dbname:catdb}
```

https://quarkus.io/guides/dev-services

Quarkus Dev UI

- Quarkus provides a Dev UI, available on /q/dev if the app runs in development mode
- This UI is extendable via extensions
- An extension might provide monitoring here (Kafka UI, Health UI etc.) or some configuration stuff (ConfigEditor)
- But NOTE: The UI is only available in Dev mode, i.e. if the app is started via “quarkus dev”



https://quarkus.io/guides/dev-services

Working with a Database

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
```

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>
```

Using Hibernate ORM / JPA

- Quarkus makes it easy to create a CRUD service, all you need to do is to add the **hibernate-orm** or **hibernate-orm-panache** extension to your app
- And you of course need to add the extension with the jdbc driver of your database
- If you have to, you can use hibernate-orm extension with full flexibility and control of the JPA process
- If you don't require full access, use simplified ORM via panache
- Most common databases are part of the Dev Services of Quarkus

```
@Entity
public class Score extends PanacheEntity {
    @Column(name = "player_id")
    public Long playerId;

    @Column(name = "game_id")
    public Long gameId;
    public long score;
    public int level;

    @Column(name = "placed_barriers")
    public int placedBarriers;

    @Column(name = "used_bombs")
    public int usedBombs;

    @Column(name = "bitten_by_spiders")
    public int bittenBySpiders;

    @Column(name = "catched_by_cats")
    public int catchedByCats;

```

https://quarkus.io/guides/hibernate-orm-panache

Using Hibernate ORM / JPA

- Panache makes it possible that you don't require getters / setters for your entities, they're implicitly being created during Quarkus augmentation process
- The REST service looks clean and lean as well
- You have two different ways of using the Entity:
  - **Active Record Pattern**: Here we are using the Active Record Pattern, all access methods are defined as static methods in the entity itself
  - **Repository Pattern:** You can also define an EntityRepository, which encapsulates all accessors

```java
@Path("/highscore")
public class HighScoreResource {

    @GET
    @Path("/{highestX}")
    http://localhost:8080/highscore/{highestX}
    public List<Score> readHighscore(int highestX) {
        List<Score> list = Score.list(query: "order by score desc, time");
        if( highestX > 0 && list.size() > highestX) {
            list = list.subList(fromIndex: 0, highestX);
        }
        return list;
    }

    @POST
    @Transactional
    public List<Score> addScore(Score score) {
        score.id = null;
        if( score.time == null ) score.time = new Date();
        score.persist();
        return readHighscore(highestX: 10);
    }

    @DELETE
    @Transactional
    @Path("/{id}")
    public List<Score> deleteScore(Long id) {
        Score.deleteById(id);
        return readHighscore(highestX: 10);
    }
}
```

https://quarkus.io/guides/hibernate-orm-panache

Using Hibernate ORM / JPA

- The only thing you really have to do is, configuring your datasource by specifying properties in **application.properties**
- If you want to use the database dev service, you need to use a property profile (%prod.) as prefix here
- By using ${ENV_NAME:default} you are able to configure the configuration
    - This is useful if you want to deploy your app to Kubernetes and you're using an environment mapped Secret

```
22  #
23  # Datasource options
24  #
    You, 4 months ago
25  quarkus.datasource.db-kind=postgresql
    wpernath, 2 months ago
26  quarkus.hibernate-orm.log.sql=false
27  quarkus.hibernate-orm.log.format-sql=true
28
29  # the following props only for production
    You, 2 months ago
30  %prod.quarkus.hibernate-orm.log.sql=false
31  %prod.quarkus.hibernate-orm.log.format-sql=false
    You, 4 months ago
32  %prod.quarkus.hibernate-orm.database.generation=update
    You, 4 months ago
33  %prod.quarkus.datasource.username=${DB_user:cat}
34  %prod.quarkus.datasource.password=${DB_password:grumpy}      You, 4 months ago • changed ap
    You, 4 months ago
35  %prod.quarkus.datasource.jdbc.url=jdbc:postgresql://${DB_host:catserver}/${DB_dbname:catdb}
```

https://quarkus.io/guides/datasource

5-7 Steps to Kubernetes & Container Images

Kubernetes Native Coding

- One of the main goals of Quarkus is to make it easy and user friendly to develop for Kubernetes or Docker / Podman
- So there are several extensions available to reach this goal
  - quarkus-container-image-*
  - quarkus-kubernetes
  - quarkus-kubernetes-config
  - quarkus-openshift
  - quarkus-smallrye-health
  - Smallrye-metrics / micrometer-metrics
  - Centralized log management

https://quarkus.io/guides/deploying-to-kubernetes

Step #1 - Creating container images

- There are several ways to create a container image. The most obvious way is to manually call “docker build…” on one of the predefined Dockerfiles
- But there are better ways via maven by including one of the quarkus-container-image-* extension
  - Jib (my preferred choice)
  - Docker (does NOT work inside Kubernetes)
  - S2i (OpenShift only)
  - Buildpack (standardized s2i)
- Creating and pushing a container is just a matter of calling maven now → This can even be done inside Kubernetes

```
13  #
14  # container image properties
15  #
    You, 4 months ago
16  quarkus.container-image.image=quay.io/wpernath/quarkus-grumpycat:v${quarkus.application.version}
17  quarkus.container-image.builder=jib
18  quarkus.container-image.build=false
19  quarkus.container-image.push=false
```

```
[wanja@minimac ~/Devel/grumpycat/quarkus-grumpycat/quarkus-server]$ mvn clean package -Dquarkus.container-image.push=true
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.wanja.grumpycat:cat-server >------------------
[INFO] Building cat-server 0.6.4
[INFO] --------------------------------[ jar ]---------------------------------
```

```
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Starting (local) container image build for jar using jib.
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] No container image registry was set, so 'docker.io' will be used
[WARNING] [io.quarkus.container.image.jib.deployment.JibProcessor] Base image 'registry.access.redhat.com/ubi8/openjdk-11-runtime:1.11' does not use a specific image digest - build may not be reproducible
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] LogEvent [level=INFO, message=trying docker-credential-desktop for quay.io]
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] LogEvent [level=LIFECYCLE, message=Using credentials from Docker config (/Users/wanja/.docker/config.json) for quay.io/wpernath/quarkus-grumpycat:v0.6.4]
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Using base image with digest: sha256:0aca47bf03430b5ee5033d67ba9b38871470af00fa7d80a38c1cc0ae34270935
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Container entrypoint set to [java, -Djava.util.logging.manager=org.jboss.logmanager.LogManager, -jar, quarkus-run.jar]
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Pushed container image quay.io/wpernath/quarkus-grumpycat:v0.6.4 (sha256:378a5ca73390e9b67e3fdcb53f5623d17bb9e2f2e83e20aaf9bff8cdf6eb37e2)
```

https://quarkus.io/guides/container-image

Step #2 - Creating Kubernetes files

- In order to properly deploy a container image on Kubernetes, you need to define some manifest files
  - Deployment, Service, Route, Ingress, ConfigMap… etc.
- Those files can easily be generated during build by using the **quarkus-kubernetes** extension
- If you're placing some additional files in /src/main/kubernetes, they'll be merged with quarkus own generations and placed in /target/kubernetes/ folder
- You can configure the extension by specifying properties quarkus.kubernetes.* in application.properties file

```
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    app.openshift.io/connects-to: >-
      [{"apiVersion":"apps/v1","kind":"Deployment","name":
  labels:
    app: cat-server
    app.kubernetes.io/component: cat-server
    app.kubernetes.io/instance: cat-server
    app.kubernetes.io/name: cat-server
    app.kubernetes.io/part-of: grumpycat-app
  name: cat-server
spec:
  port:
    targetPort: 8080-tcp
  to:
    kind: Service
    name: cat-server
    weight: 100
  wildcardPolicy: None
---
apiVersion: apps/v1
kind: Deployment
```

```
[INFO] Adding existing Service with name: cat-server.
[INFO] Adding existing Deployment with name: cat-server.
[INFO] Adding existing PostgresCluster with name: cat.
[INFO] Adding existing Kafka with name: grumpy-kafka.
[INFO] Adding existing KafkaTopic with name: player-actions.
[INFO] Adding existing Route with name: cat-server.
```

https://quarkus.io/guides/deploying-to-kubernetes

Step #2 - Creating OpenShift files

- If you really require OpenShift specific things, you can use the **quarkus-openshift** extension instead of the **quarkus-kubernetes** one.
- However, this is not required when you're targeting OpenShift as Kubernetes distribution
- This extension makes it easy to directly deploy to OpenShift and to make use of s2i inside the cluster
- However, by specifying a Route in src/main/kubernetes, you can also easily use the kubernetes extension

https://quarkus.io/guides/deploying-to-openshift

Optional step #2 - Directly use Kubernetes Secrets / ConfigMaps

- By default, you can mount any ConfigMap / Secret to your Pod's environment, either as ENV or as volume.
- If you want to directly use any Secret or ConfigMap as Config Source in your Quarkus Application, you can use the **quarkus-kubernetes-config** extension
- This extension allows you to directly use any key of the Secret / ConfigMap directly in application.properties (and thus of course also in your code, via @ConfigProperty)

One possible way to make Quarkus use these entries to connect the database is to use the following configuration:

```
%prod.quarkus.kubernetes-config.secrets.enabled=true                    1
quarkus.kubernetes-config.secrets=postgresql                            2

%prod.quarkus.datasource.jdbc.url=postgresql://somehost:5432/${database-name}   3
%prod.quarkus.datasource.username=${database-user}                      4
%prod.quarkus.datasource.password=${database-password}                  5
```

1. Enable reading of secrets. Note the use of `%prod` profile as we only want this setting applied when the application is running in production.
2. Configure the name of the secret that will be used. This doesn't need to be prefixed with the `%prod` profile as it won't have any effect if secret reading is disabled.
3. Quarkus will substitute `${database-name}` with the value obtained from the entry with name `database-name` of the `postgres` Secret. `somehost` is the name of the Kubernetes `Service` that was created when PostgreSQL was deployed to Kubernetes.
4. Quarkus will substitute `${database-user}` with the value obtained from the entry with name `database-user` of the `postgres` Secret.
5. Quarkus will substitute `${database-password}` with the value obtained from the entry with name `database-password` of the `postgres` Secret.

https://quarkus.io/guides/kubernetes-config

Step #3 - Adding Health Checks

- In a Kubernetes Native environment, it's essential to have Health checks implemented accordingly to make it easy for the Kubernetes scheduler to see if your app is ready to be used or needs to be killed / restarted.
- For this there is the **smallrye-health** extension available. In combination with the kubernetes extension, the created Deployment will already contain the corresponding checks automatically
- Other extensions (like messaging and databases) will automatically add checks here

Importing the `smallrye-health` extension directly exposes three REST endpoints:

- `/q/health/live` - The application is up and running.
- `/q/health/ready` - The application is ready to serve requests.
- `/q/health/started` - The application is started.
- `/q/health` - Accumulating all health check procedures in the application.

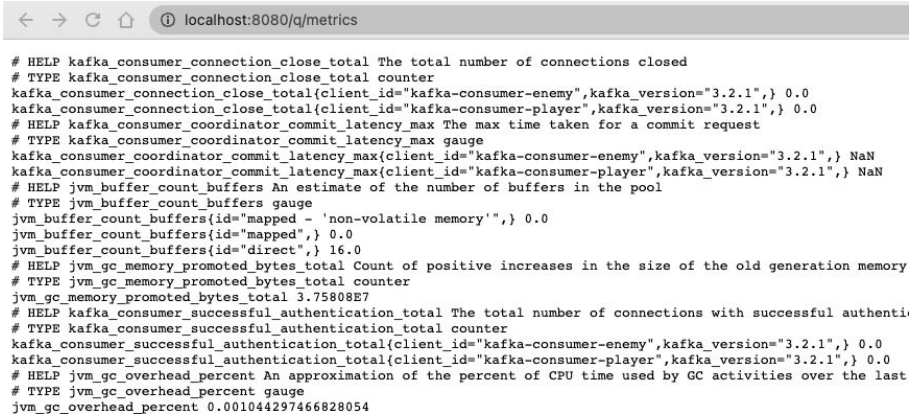
```
        readinessProbe:
          httpGet:
            path: /q/health/ready
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 5
          timeoutSeconds: 1
          periodSeconds: 10
          successThreshold: 1
          failureThreshold: 5
        livenessProbe:
            httpGet:
              path: /q/health/live
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 10
            timeoutSeconds: 1
            periodSeconds: 10
            successThreshold: 1
            failureThreshold: 3
        startupProbe:
            httpGet:
              path: /q/health/started
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 10
            timeoutSeconds: 1
            periodSeconds: 10
            successThreshold: 1
            failureThreshold: 3
```


```
{
    status: "UP",
  - checks: [
      - {
            name: "SmallRye Reactive Messaging - liveness check",
            status: "UP",
          - data: {
                enemy: "[OK]",
                player: "[OK]"
            }
        },
      - {
            name: "Database connections health check",
            status: "UP",
          - data: {
                <default>: "UP"
            }
        },
      - {
            name: "SmallRye Reactive Messaging - readiness check",
            status: "UP",
          - data: {
                enemy: "[OK] - no subscription yet, so no connection to the Kafka broker yet",
                player: "[OK] - no subscription yet, so no connection to the Kafka broker yet"
            }
        },
      - {
            name: "SmallRye Reactive Messaging - startup check",
            status: "UP",
          - data: {
                enemy: "[OK] - no subscription yet, so no connection to the Kafka broker yet",
                player: "[OK] - no subscription yet, so no connection to the Kafka broker yet"
            }
        }
    ]
}
```


https://quarkus.io/guides/smallrye-health

Step #4 - Adding Metrics

- Every good app requires metrics which could be read centrally. Quarkus supports Micrometer by own extensions, for example **micrometer-registry-prometheus**, which could be used for deployments in OpenShift
- All Quarkus extensions will automatically provide desired metrics, like database connections etc.
- Also, all JVM metrics will be available automatically
- /q/metrics



https://quarkus.io/guides/micrometer

Red Hat

Step #5 - Adding Documentation

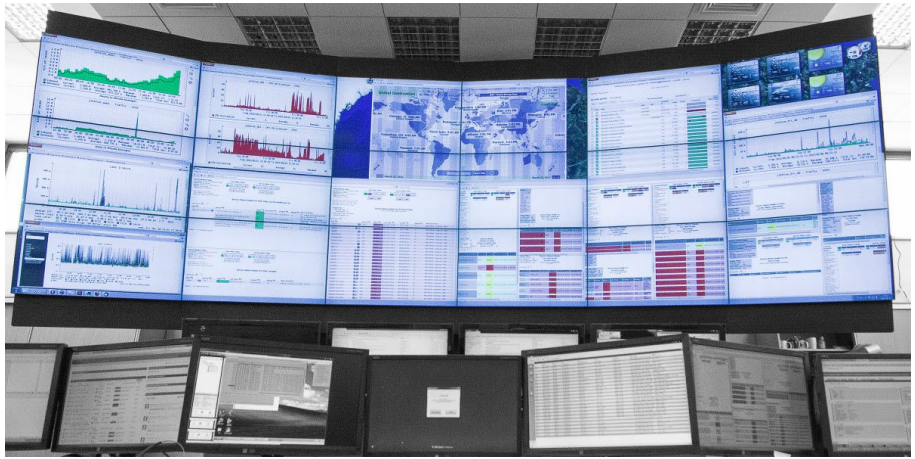
- Every API (regardless if public or private) requires documentation.
- Quarkus provides a **smallrye-openapi** extension, which lets you use MicroProfile OpenAPI annotations in your code
- This also includes swagger-ui to expose the documentation via well-known swagger (available only in dev or test mode by default)
- OpenAPI is available under /q/openapi and
- SwaggerUI is available under /q/swagger-ui

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Operation(summary = "says hello", description = "No Params, just says hello")
public String hello() {
    return "Hello from RESTEasy Reactive";
}
```

https://quarkus.io/guides/openapi-swaggerui

Step #6 - Adding Centralized Logging

- If you want / have to, you can also add your app to a centralized logging consumer like Graylog, Logstash, Fluentd or ELK
- Quarkus supports the GELF log handler by simply adding the extension **quarkus-logging-gelf**



https://quarkus.io/guides/centralized-log-management

Step #7 - Deployment to Kubernetes

```
[INFO] Adding existing Service with name: cat-server.
[INFO] Adding existing Deployment with name: cat-server.
[INFO] Adding existing PostgresCluster with name: cat.
[INFO] Adding existing Kafka with name: grumpy-kafka.
[INFO] Adding existing KafkaTopic with name: player-actions.
[INFO] Adding existing Route with name: cat-server.
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in 2121ms
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.000 s
[INFO] Finished at: 2022-10-06T01:32:09+02:00
[INFO] ------------------------------------------------------------------------
[wanja@minimac ~/Devel/grumpycat/quarkus-grumpycat/quarkus-server]$ oc apply -f target/kubernetes/kubernetes.yml
```

https://quarkus.io/guides/deploying-to-openshift

Messaging

Smallrye Reactive Messaging

- Quarkus makes heavily use of smallrye reactive messaging library, a framework for building event-driven, data streaming and event sourcing applications
- In short, smallrye lets your application interact with various messaging technologies, such as Kafka, AMQP, MQTT etc.
- Even http streams can be seen as messaging (WebSockets)
- Quarkus abstracts away most of the API

https://smallrye.io/smallrye-reactive-messaging/3.20.0/

Creating a WebSocket with Smallrye

- If you quickly need a WebSocket implementation, you can use the **quarkus-reactive-messaging-http** extension and add some configuration in application.properties
- Then you need an ApplicationScoped bean which is processing the incoming events (@Incoming)
- In this example, the processor is just persisting the incoming PlayerAction
- In case of the game, we use this approach for the client sending Player- and Enemy Updates
- Easy, isn't it?

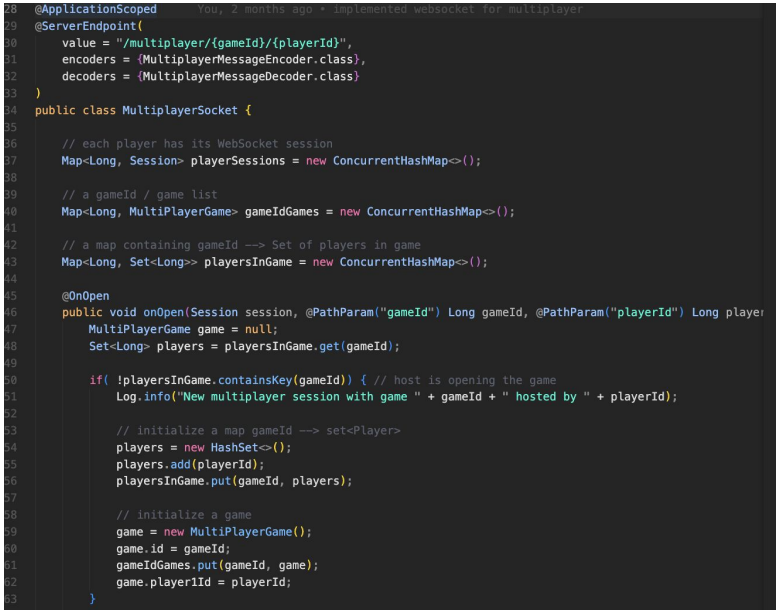
```
55  #
56  # WebSocket for incoming state updates of the client
57  # in single player mode
58  #
    You, 2 months ago
59  mp.messaging.incoming.incoming-states.connector=quarkus-websocket
60  mp.messaging.incoming.incoming-states.path=/player-update
61  mp.messaging.incoming.incoming-states.buffer-size=64
```

```
13  @ApplicationScoped
14  public class PlayerMovementProcessor {
15
16      /**
17       * stores the player action into the database
18       * @param action
19       */
20      @Incoming("player-actions")
21      @Blocking
22      @Transactional
23      public void processPlayerAction(PlayerAction action) {
24          if (action.gameId == null || action.playerId == null)
25              throw new IllegalArgumentException(s: "Neither gameId nor playerId must be null");
26
27              Log.debug("Logging player action for " + action.gameId);
28          action.persist();
29      }
```

https://docs.quarkiverse.io/quarkus-reactive-messaging-http/dev/reactive-messaging-http.html?quarkusDocVersion=3.8

Creating a WebSocket with full control

- If you quickly need full control over a WebSocket, you need to use the **quarkus-websockets** extension
- Create an @ApplicationScoped bean and implement all necessary events (@OnOpen, @OnClose, @OnError & @OnMessage)
- You need to implement session handling yourself
- You also need to implement encoders and decoders for non-standard parameters (i.e. JSON)
- In case of the game, we implement this for multi player handling



```
@ApplicationScoped      You, 2 months ago • implemented websocket for multiplayer
@ServerEndpoint(
    value = "/multiplayer/{gameId}/{playerId}",
    encoders = {MultiplayerMessageEncoder.class},
    decoders = {MultiplayerMessageDecoder.class}
)
public class MultiplayerSocket {

    // each player has its WebSocket session
    Map<Long, Session> playerSessions = new ConcurrentHashMap<>();

    // a gameId / game list
    Map<Long, MultiPlayerGame> gameIdGames = new ConcurrentHashMap<>();

    // a map containing gameId --> Set of players in game
    Map<Long, Set<Long>> playersInGame = new ConcurrentHashMap<>();

    @OnOpen
    public void onOpen(Session session, @PathParam("gameId") Long gameId, @PathParam("playerId") Long player
        MultiPlayerGame game = null;
        Set<Long> players = playersInGame.get(gameId);

        if( !playersInGame.containsKey(gameId)) { // host is opening the game
            Log.info("New multiplayer session with game " + gameId + " hosted by " + playerId);

            // initialize a map gameId --> set<Player>
            players = new HashSet<>();
            players.add(playerId);
            playersInGame.put(gameId, players);

            // initialize a game
            game = new MultiPlayerGame();
            game.id = gameId;
            gameIdGames.put(gameId, game);
            game.player1Id = playerId;
        }
```

https://quarkus.io/guides/websockets

Using Apache Kafka

- If you quickly need to integrate into Apache Kafka, for example to store enemy updates, use the **smallrye-reactive-messaging-kafka** extension
- Then configure the extension properly
- The extension will use Dev Services to start up a Kafka server
- So testing is very easy here

```
#
# Strimzi / Kafka
#
wpernath, 3 months ago
kafka.auto.offset.reset=earliest
wpernath, 2 months ago
%prod.kafka.bootstrap.servers=kafka:9092

wpernath, 3 months ago
mp.messaging.incoming.player.topic=player-actions
mp.messaging.incoming.player.connector=smallrye-kafka
wpernath, 3 months ago
mp.messaging.incoming.player.value.deserializer=org.wanja.fatcat.PlayerActionDeserializer
mp.messaging.incoming.player.value.serializer=io.quarkus.kafka.client.serialization.ObjectMapperSerializer

 Unknown property
 'mp.messaging.incoming.enemy.value.deserializer' microprofile(unknown)
 View Problem  Quick Fix... (⌘.)
mp.messaging.incoming.enemy.value.deserializer=org.wanja.fatcat.EnemyActionDeserializer
mp.messaging.incoming.enemy.value.serializer=io.quarkus.kafka.client.serialization.ObjectMapperSerializer
```

```
    @Incoming("incoming-enemy")
    @Outgoing("enemy-actions")
    //@Transactional
    EnemyAction collectEnemy(EnemyAction action) {
        if (action.gameId == null || action.playerId == null) {
            Log.warn("Skipping enemy state action, because gameId || playerId is NULL");
            return null;
        }
        else {
            //enemyEmitter.send(action);
            //action.persist();
            Log.debug("Sending game action to Kafka");
            return action;
        }

    }
```

https://quarkus.io/guides/kafka-reactive-getting-started

Using Apache Kafka with docker-compose

- Docker-compose (or podman-compose) help you to setup and run your multi-service application including a Kafka instance
- Quarkus supports you with the configuration, by setting %prod.kafka.bootstrap.servers which will only be used, if the app does not run in DEV or TEST mode.

```
38  #
39  # Strimzi / Kafka
40  #
    wpernath, 3 months ago
41  kafka.auto.offset.reset=earliest
    wpernath, 2 months ago
42  %prod.kafka.bootstrap.servers=kafka:9092
43
```

```yaml
version: "3.3"
#name: 'grumpycat'

You, 7 days ago | 2 authors (wpernath and others)
services:

  wpernath, 2 months ago | 1 author (wpernath)
  zookeeper:
    image: quay.io/strimzi/kafka:0.23.0-kafka-2.8.0
    command: [
      "sh", "-c",
      "bin/zookeeper-server-start.sh config/zookeeper.properties"
    ]
    ports:
      - "2181:2181"
    wpernath, 2 months ago | 1 author (wpernath)
    environment:
      LOG_DIR: /tmp/logs
    networks:
      - grumpycat-network

  wpernath, 2 months ago | 1 author (wpernath)
  kafka:
    image: quay.io/strimzi/kafka:0.23.0-kafka-2.8.0
    command: [
      "sh", "-c",
      "bin/kafka-server-start.sh config/server.properties --override listeners=${KAFKA_LISTENERS} --overri
    ]
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    wpernath, 2 months ago | 1 author (wpernath)
    environment:
      LOG_DIR: "/tmp/logs"
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    networks:
      - grumpycat-network
```

https://quarkus.io/guides/kafka-reactive-getting-started

Using Apache Kafka with Kubernetes

- In a Kubernetes environment, you can use the Strimzi Operator to declaratively deploy a Kafka broker and a Kafka topic
- Again, Quarkus helps you with the configuration of the service (as in the docker-compose environment).
- Simply install them with your application

```
#
# Strimzi / Kafka
#
wpernath, 3 months ago
kafka.auto.offset.reset=earliest
wpernath, 2 months ago
%prod.kafka.bootstrap.servers=kafka:9092
```

https://quarkus.io/guides/kafka-reactive-getting-started

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: grumpy-kafka
spec:
  kafka:
    version: 3.2.0
    replicas: 1
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    storage:
      type: jbod
      volumes:
        - type: persistent-claim
          id: 0
          size: 5Gi
          deleteClaim: true
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: '3.2'
  zookeeper:
    replicas: 1
    storage:
      type: persistent-claim
      size: 1Gi
      deleteClaim: true
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: player-actions
  labels:
    strimzi.io/cluster: "grumpy-kafka"
spec:
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
```

Red Hat

Qute Templating Engine

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-reactive-qute</artifactId>
</dependency>
```

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-qute</artifactId>
</dependency>
```

Qute Templating Engine

- Qute is an easy to use templating engine for Quarkus. It's designed especially for Quarkus:
  - Minimizes reflection (for native compilation)
  - Contains imperative and non-blocking APIs
  - Understands Dev Services (templates are being watched for changes during development)
  - Syntax & runtime checks during build time
- Qute has two parts
  - The template file itself, located under src/main/resources/templates
  - The Java part

```java
@Path("/reports")
public class ReportResource {

    Open `src/main/resources/templates/openGames.txt`
    @Inject
    Template openGames;

    @Inject
    MultiPlayerResource multiPlayerResource;

    @Path("/open-games")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Blocking
    http://localhost:8080/reports/open-games
    public TemplateInstance listOpenGames() {
        return openGames.data(
            key: "games",
            multiPlayerResource.listOpenGames()
        );
    }
```

```
ReportResource#openGames | games : List<MultiPlayerGame>
This report lists all currently open games

ID,Time,player1name,player2name,player3name,player4name
{#for game:MultiPlayerGame in games}
    {game.id},{game.timeStarted},{game.player1.name},{game.player2 ? game.player2.name : "Unused"}
{/for}
```

https://quarkus.io/guides/qute

Qute Templating Engine - The Java part

- You could use it through a REST service (other use cases are possible, for example via Scheduler)
- All you need to do is:
  - @Inject a Template object
  - The object's name is the name of the template file (so openGames needs to have a corresponding openGames.txt file)
  - The GET method of the service is returning a TemplateInstance, which you can prefill with parameters.
  - Parameters are key/value pairs

```
13  @Path("/reports")
14  public class ReportResource {
15
        Open `src/main/resources/templates/openGames.txt`
16      @Inject
17      Template openGames;
18
19      @Inject
20      MultiPlayerResource multiPlayerResource;
21
22      @Path("/open-games")
23      @GET
24      @Produces(MediaType.TEXT_PLAIN)
25      @Blocking
        http://localhost:8080/reports/open-games
26      public TemplateInstance listOpenGames() {
27          return openGames.data(
28              key: "games",
29              multiPlayerResource.listOpenGames()
30          );
31      }
```

https://quarkus.io/guides/qute

Qute Templating Engine - The template file

- By default, all templates are in /src/main/resources/templates (could be overwritten via application.properties).
- All templates need to have a suffix like ".txt", ".html" (could be overwritten via application.properties).
- A template has access to all parameters provided via key/value
- A template has also access to all CDI beans available globally in the quarkus application
- Comprehensive syntax to render the output

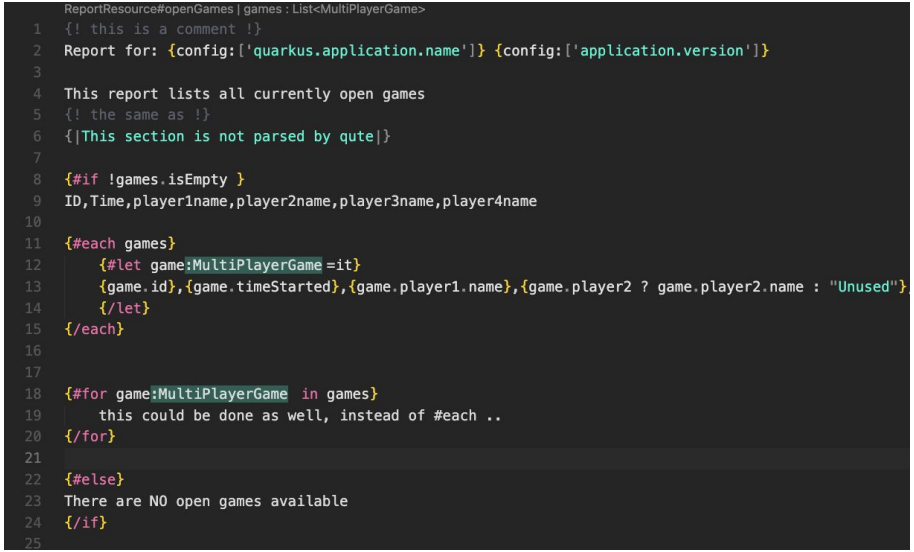
```
ReportResource#openGames | games : List<MultiPlayerGame>
This report lists all currently open games

ID,Time,player1name,player2name,player3name,player4name
{#for game:MultiPlayerGame  in games}
    {game.id},{game.timeStarted},{game.player1.name},{game.player2 ? game.player2.name : "Unused"}
{/for}
```

https://quarkus.io/guides/qute-reference

Qute Templating Engine - The template file

- Qute language is easy and contains the following parts
  - **Comments**: {! this is comment !}
  - **Expressions**: {foo.name} outputs the evaluated value
  - **Sections**: A section may contain static text, expressions and more sections. {#if foo.isActive } {foo.name} {/if}
  - **Unparsed data**: Marks content that should be rendered but not parsed. {| alert("hi"); |}



https://quarkus.io/guides/qute-reference

Red Hat

Qute Templating Engine - Advanced features

- **Template Extension Methods**: A data class can be extended with new functionality, w/o changing the original Java class
- **Template Global** variables: A class annotated with @TemplateGlobal contains variables which ALL TemplateInstances can use out of the box
- **Modularize your templates** by using includes and user defined tags

```java
package org.acme;

class Item {

    public final BigDecimal price;

    public Item(BigDecimal price) {
        this.price = price;
    }
}

@TemplateExtension
class MyExtensions {

    static BigDecimal discountedPrice(Item item) { 1
        return item.getPrice().multiply(new BigDecimal("0.9"));
    }
}
```

https://quarkus.io/guides/qute-reference

Security

Secure your REST end-points easily

- By adding the **quarkus-oidc** extension to your pom.xml, you're able to quickly secure your REST end-points with an external provider, like Keycloak.
- Quarkus Dev Services make it possible to just use Keycloak - no dev configuration etc.
- Use @Authenticated or @RolesAllowed annotations to make sure certain parts of your services are properly secured
- If you need to use such a secured app in production, you should install the Keycloak operator into your target Kubernetes cluster

```
#
# OIDC
#
You, 4 hours ago
%prod.quarkus.oidc.auth-server-url=https://localhost:8543/realms/quarkus
You, 4 hours ago
quarkus.oidc.client-id=cat-server      You, 4 hours ago • Uncommitted changes
You, 4 hours ago
quarkus.oidc.credentials.secret=secret
You, 4 hours ago
quarkus.oidc.tls.verification=none
# Enable Policy Enforcement
You, 4 hours ago
quarkus.keycloak.policy-enforcer.enable=true
# Tell Dev Services for Keycloak to import the realm file
# This property is not effective when running the application in JVM or Native
You, 4 hours ago
quarkus.keycloak.devservices.realm-path=quarkus-realm.json
```

https://quarkus.io/guides/security-keycloak-authorization

https://quarkus.io/guides/security-openid-connect

Caching

Caching with different providers

- By adding the extension **quarkus-cache**, you're already able to use an internal in-memory cache provided by **Caffeine**. This allows you to easily use method result caching by just annotating the method – by additionally using the micrometer extension, you get HEALTH APIs automatically
- However, if you need to synchronize data between services, you should use **Infinispan** or **Redis** extensions

```
@CacheResult(cacheName = "weather-cache") 1
public String getDailyForecast(LocalDate date, String city) {
    try {
        Thread.sleep(2000L);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return date.getDayOfWeek() + " will be " + getDailyResult(date.getDayOfMonth() % 4) + " in " + city;
}
```

```
package org.acme.cache;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import io.quarkus.cache.Cache;
import io.quarkus.cache.CacheName;
import io.smallrye.mutiny.Uni;

@ApplicationScoped
public class CachedExpensiveService {

    @Inject 1
    @CacheName("my-cache")
    Cache cache;

    public Uni<String> getNonBlockingExpensiveValue(Object key) { 2
        return cache.get(key, k -> { 3
            /*
             * Put an expensive call here.
             * It will be executed only if the key is not already associated with
             */
        });
    }

    public String getBlockingExpensiveValue(Object key) {
        return cache.get(key, k -> {
            // Put an expensive call here.
        }).await().indefinitely(); 4
    }
}
```

```
# HELP cache_size The number of entries in this cache. This may be
# TYPE cache_size gauge
cache_size{cache="foo",} 8.0
# HELP cache_puts_total The number of entries added to the cache
# TYPE cache_puts_total counter
cache_puts_total{cache="foo",} 12.0
# HELP cache_gets_total The number of times cache lookup methods ha
# TYPE cache_gets_total counter
cache_gets_total{cache="foo",result="hit",} 53.0
cache_gets_total{cache="foo",result="miss",} 12.0
# HELP cache_evictions_total cache evictions
# TYPE cache_evictions_total counter
cache_evictions_total{cache="foo",} 4.0
# HELP cache_eviction_weight_total The sum of weights of evicted er
# TYPE cache_eviction_weight_total counter
cache_eviction_weight_total{cache="foo",} 540.0
```

https://quarkus.io/guides/cache

https://quarkus.io/guides/redis

https://quarkus.io/guides/infinispan-client

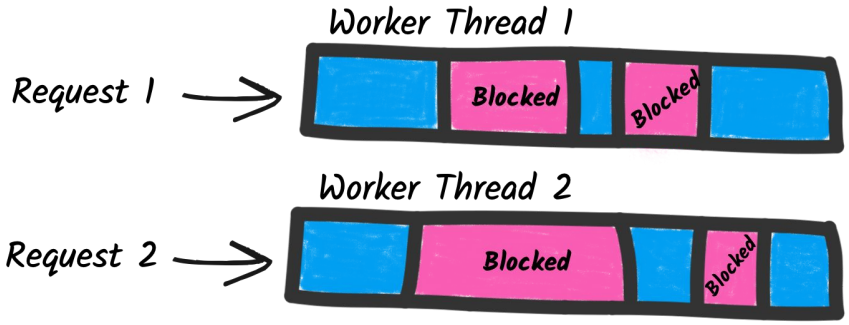
Red Hat

Using Reactive with Mutiny

Blocking vs. Non blocking Model

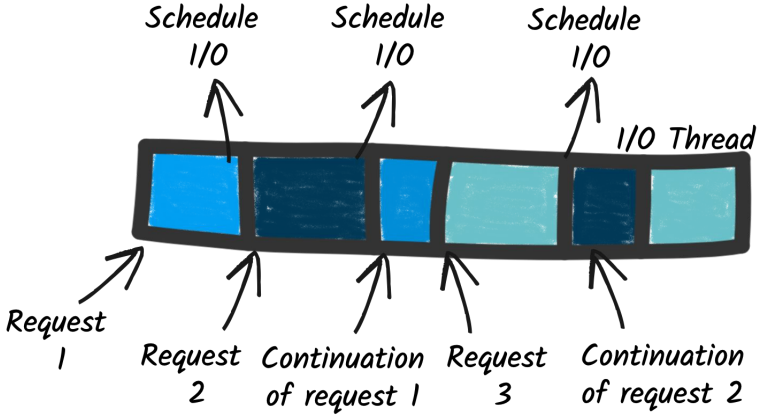
Blocking:

- Every request requires a thread
- This thread does not really do anything (in fact it sleeps most of the time)
- Waste of resources



Non Blocking:

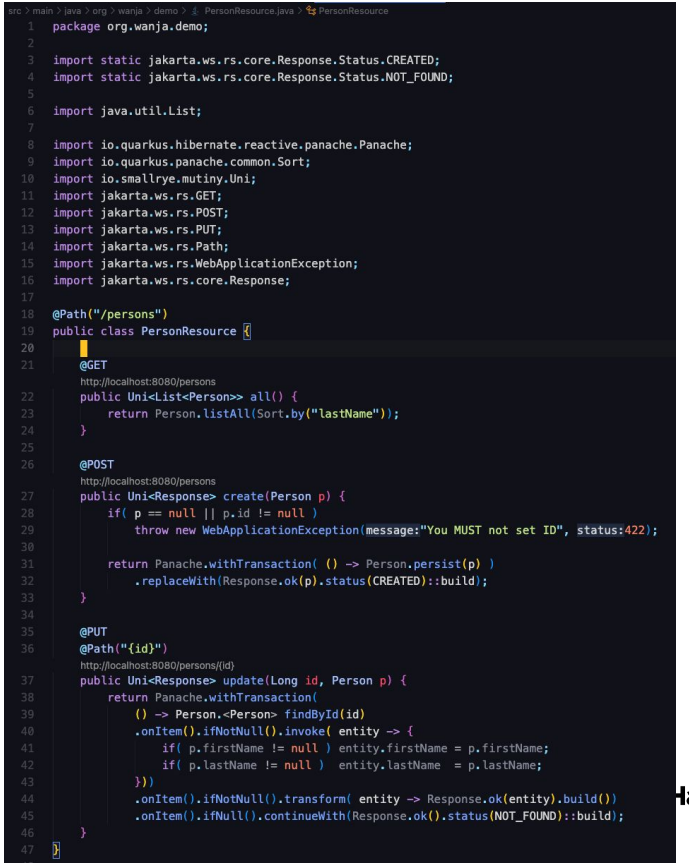
- A thread is shared by most requests
- Every resource intensive action needs to be scheduled on a worker thread and gets executed when possible
- Resources are being used best possible



https://quarkus.io/guides/quarkus-reactive-architecture

A reactive CRUD REST service

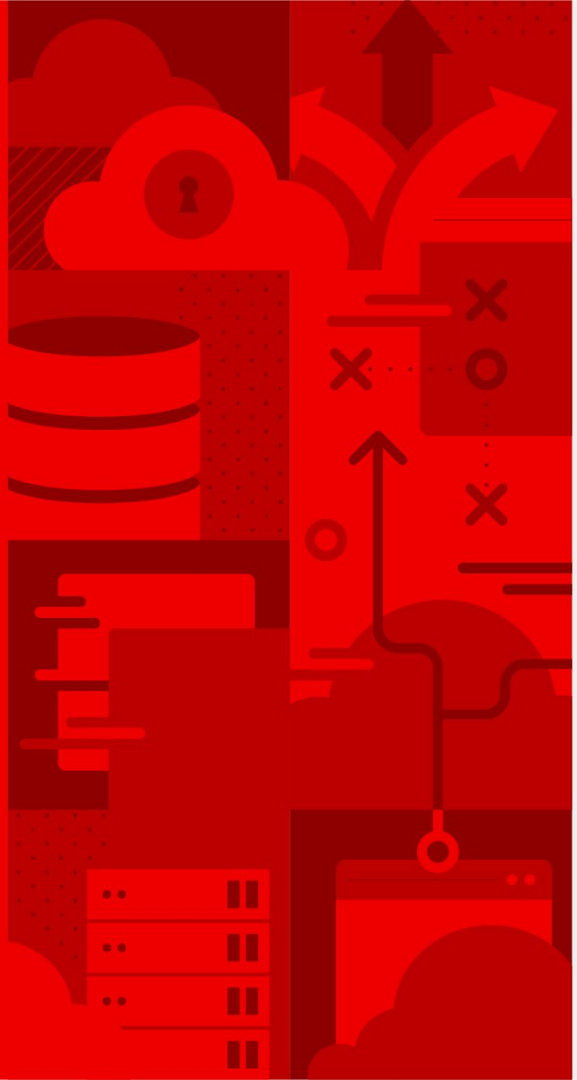
- Use reactive extensions for Hibernate and JDBC drivers **quarkus-hibernate-reactive-panache** and **quarkus-reactive-pg-client**
- Have a look at the Mutiny project to understand the reactive programming model with Quarkus
- Implement the Entity as you'd implement it using blocking code
- The REST service implementation is a bit different:
  - You're using **Uni**'s as return types
  - You're starting a transaction with **Panache.withTransaction()**
- Remember: You're on an Event Thread! Don't do blocking code



https://quarkus.io/guides/hibernate-reactive-panache

https://quarkus.io/guides/mutiny-primer

https://smallrye.io/smallrye-mutiny/2.4.0/



Thank you

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat