



Academia Tehnică Militară "Ferdinand I"  
Facultatea de Sisteme Informaticе și Securitate Cibernetică

# PARADIGME DE PROGRAMARE (ÎN JAVA)

## CURS 11 - QUARKUS FRAMEWORK

COL(R) TRAIAN NICULA

# TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- Programare funcțională și asincronă (3)
- Programare reactivă (1)
- Servicii web REST (2)
- **Quarkus Framework** (1/4)

# CUPRINS CURS

- Ce este Quarkus
- Cloud Native Computing
- Prima aplicație cu Quarkus
- Context and Dependency Injection (CDI)
- Configuration
- Bibliografie

# CE ESTE QUARKUS

- **Quarkus** este un **open source framework** care facilitează **dezvoltarea de aplicații de tip backend**
- **Quarkus** este **alcătuit din cele mai valoroase biblioteci și standarde Java** cu suport pentru următoarele implementări JVM: **Oracle HotSpot, OpenJDK și GraalVM**
- **Suportă Kubernetes, sisteme reactive** precum și o parte din specificațiile **Jakarta EE și MicroProfile**
- Permite **crearea de aplicații native cloud** folosind **microserviciile** ca stil arhitectural
- **Microserviciile** scurtează semnificativ timpul de adăugare de noi caracteristici aplicațiilor prin **dezvoltarea, testarea și instalarea individuală a fiecărui serviciu fără a le afecta pe celelalte**



# CE ESTE QUARKUS MICROSERVICII

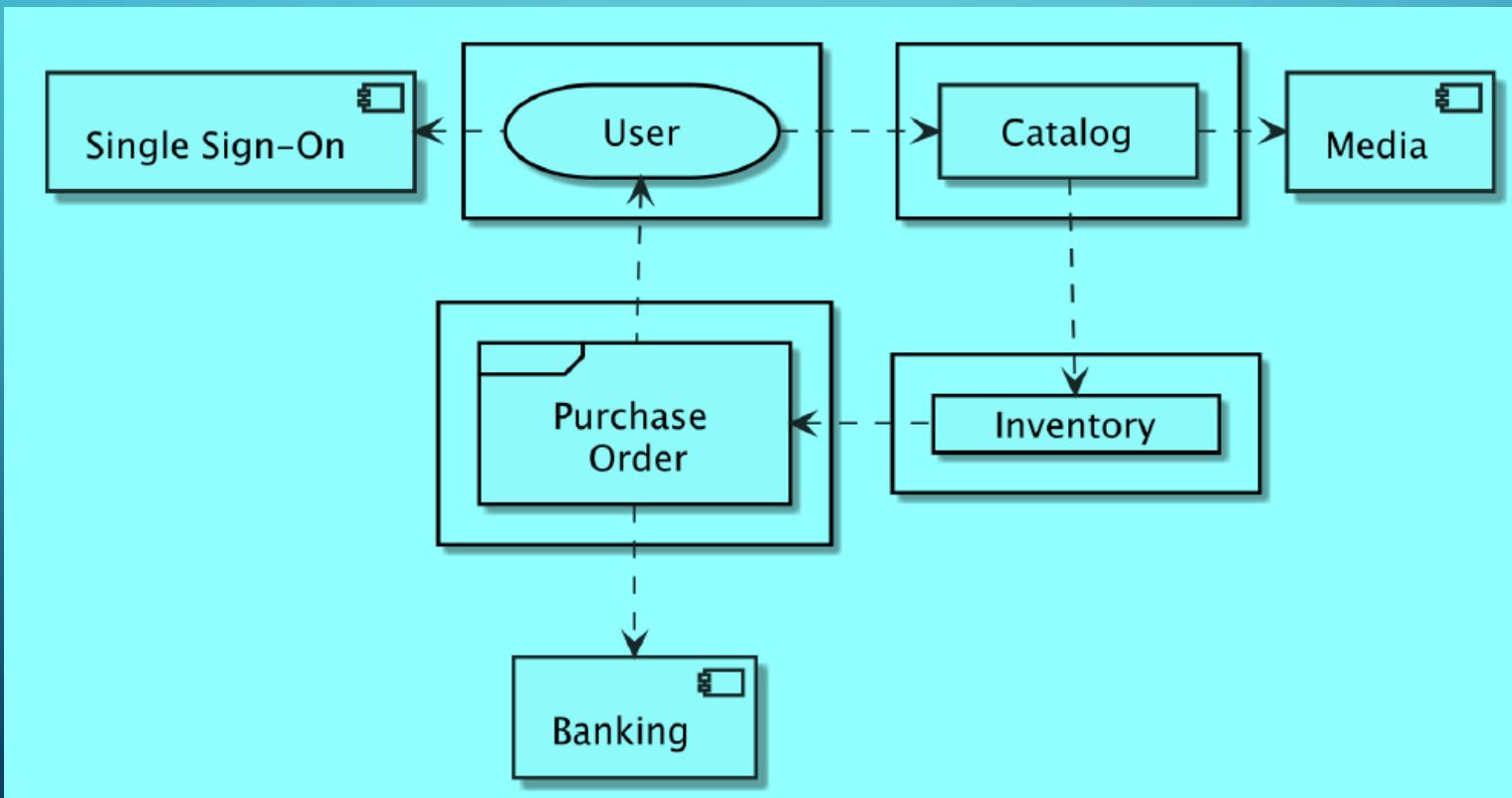
- Alternativa la *microservicii* o constituie *aplicațiile monolitice*
- *Aplicațiile monolitice* au apărut cu **introducerea Java EE** și au permis **dezvoltarea de aplicații** care **conțin toate cerințele de business** și **rulează izolat pe o singură mașină fizică**
- **Orice modificare efectuată presupune reinstalarea întregii aplicații ca o singură unitate logică**
- Astăzi **arhitectura aplicațiilor** trebuie să **suporte integrarea și dezvoltarea continuă (CI/CD)**, care se realizează prin descompunerea sistemului în servicii mai mici, responsabile de anumite domenii de business
- **Descompunerea unei aplicații monolitice** în *microservicii independente*, care să **păstreze aproape neschimbată funcționalitatea aplicației**, are **multe avantaje**



# CE ESTE QUARKUS

## MICROSERVICII

- Arhitectura bazată pe microservicii folosește de multe ori servicii externe
- De exemplu, pentru autentificare folosesc Google, Facebook sau Twitter iar pentru plăți Paypal





# CE ESTE QUARKUS MICROSERVICII

**Beneficiile microserviciilor:**

- **Structură modulară**
- **Independent Deployment** – fiind autonome, microserviciile sunt ușor de instalat și nu afectează celelalte componente dacă ceva nu funcționează corespunzător
- **Diversitate tehnologică** – se poate folosi un amestec de limbi, framework-uri și tehnologii de stocare

**Dezavantaje:**

- **Distribution** – răspunsul poate fi lent
- **Eventual Consistency** – menținerea consistenței datelor distribuite
- **Operational Complexity** – managementul unei multimi de microservicii, care necesită reinstalări frecvente, este dificilă



# CE ESTE QUARKUS REACTIV

- În sens larg, **reactiv** înseamnă să acționezi ca răspuns la un stimул

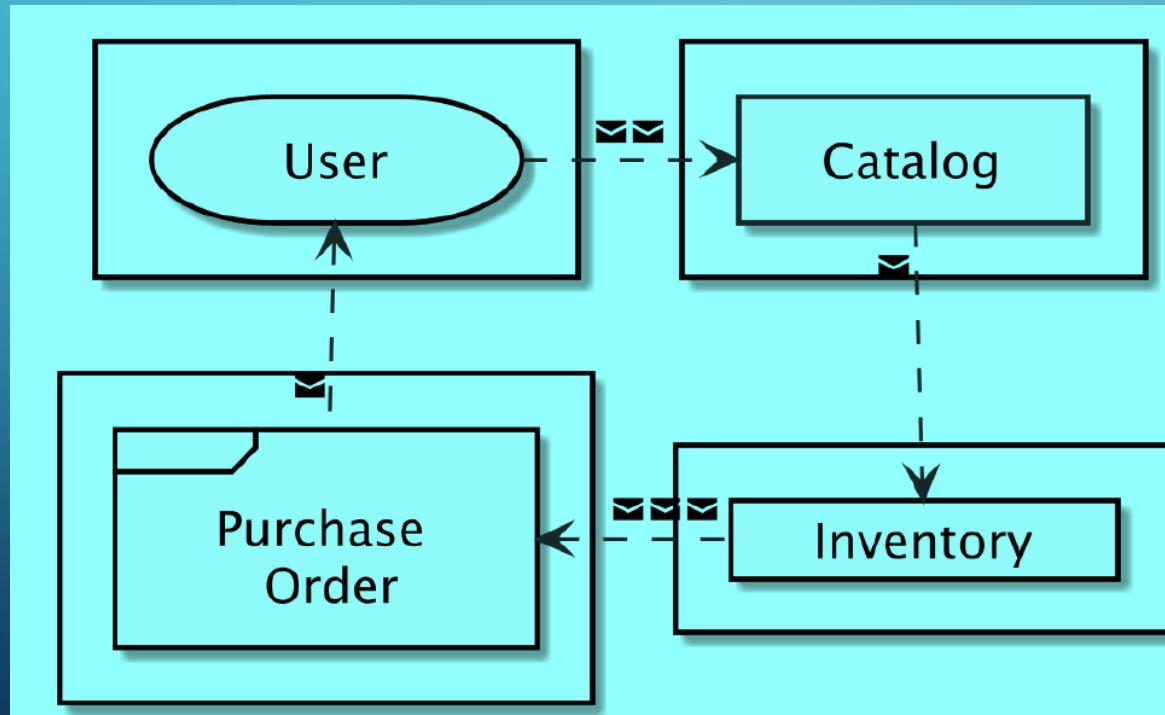
Tipuri de reactivitate:

- **Sisteme reactive** – sisteme modelate să reacționeze la un stimул precum un mesaj, o cerere, etc.
- **Stream-uri reactive** – procesarea stream-urilor folosind backpressure (ex. RxJava)
- **Programare reactivă** – model de programare bazat pe stream-uri implementat în sisteme reactive
- Hardware-ul actual din cloud folosește arhitecturi multinucleu, iar în acest context sistemele reactive sunt utile



# CE ESTE QUARKUS REACTIV

- Sistemele reactive se bazează pe **mesaje asincrone** (evenimente) pentru a comunica între microservicii
- Mesajele pot fi distribuite la mai multe instanțe iar **fluxul de mesaje între producători și consumatori** poate fi **controlat folosind backpressure**





# CE ESTE QUARKUS MICROPROFILE

- Quarkus integrează întregul set de specificații de la *Eclipse MicroProfile* prin *SmallRye*
- *Eclipse MicroProfile* constituie un set de specificații pentru **microservicii Java** de nivel enterprise
- *MicroProfile API* stabilește baza dezvoltării aplicațiilor bazate pe **microservicii**, adoptând un **subset** al standardelor Jakarta EE și extinzându-l pentru microservicii
- *SmallRye* este un proiect open source care implementează specificațiile *Eclipse MicroProfile*

# CE ESTE QUARKUS

## MICROPROFILE

### Specificații:

- ***Context and Dependency Injection (CDI)*** – modelul de programare al CDI transformă aproape **orice componentă** într-un **bean injectabil, interceptabil și administrabil**
- ***Java API for RESTful Web Services (JAX-RS)*** – specificație care furnizează suport pentru **crearea serviciilor REST** conform stilului arhitectural REST
- ***JSON Binding (JSON-B)*** – specificație pentru **conversia obiectelor Java la/de la documente JSON**
- ***JSON Processing (JSON-P)*** – specificație pentru **procesarea JSON în Java**. Include mecanisme de parsare, generare, transformare și interogare JSON



# CE ESTE QUARKUS

## MICROPROFILE

### Specificații:

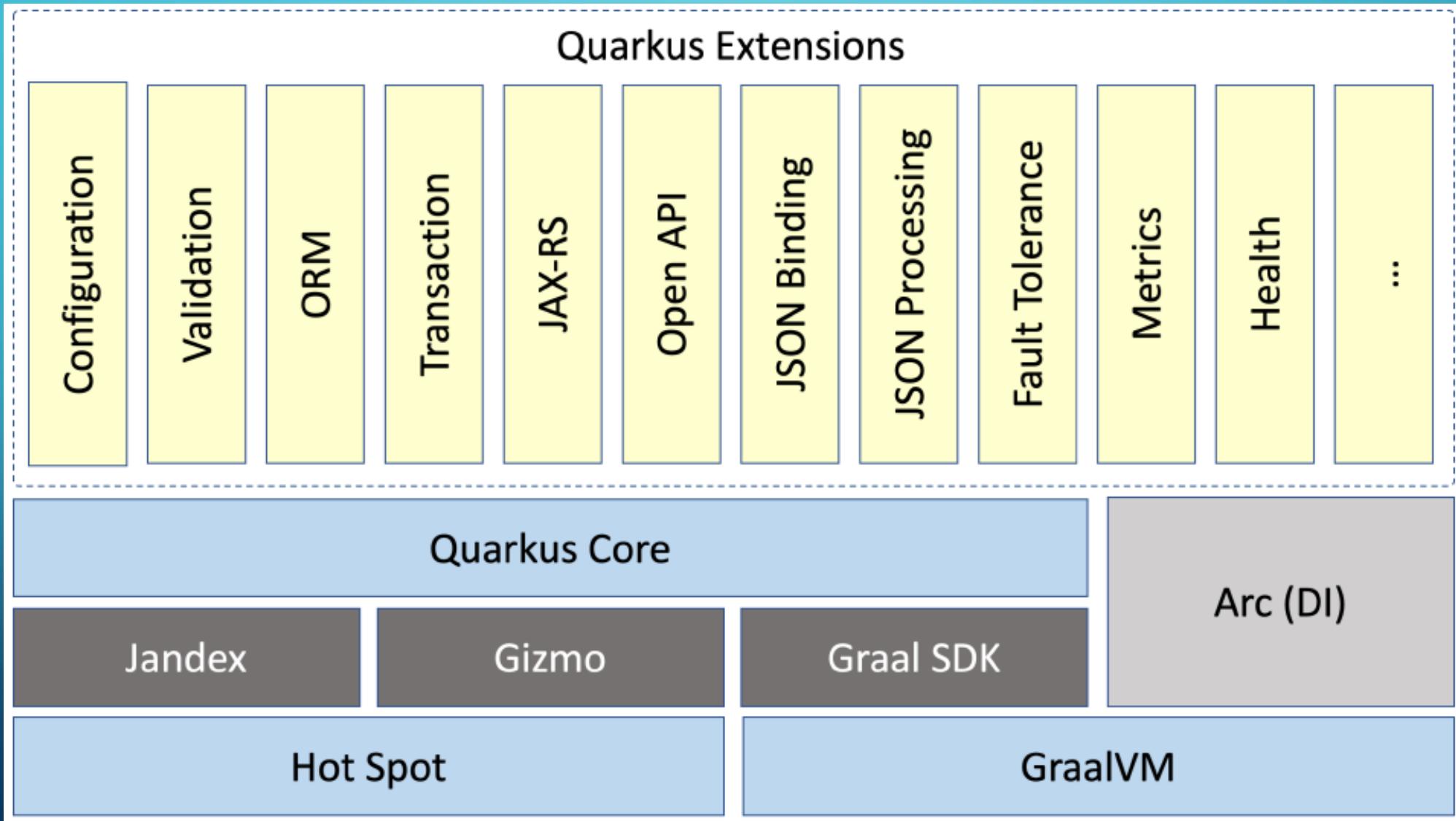
- **Common Annotations** - adnotări comune pentru o varietate de tehnologii (ex. adnotări de securitate)
- **Configuration** – permite **obținerea proprietăților de configurare din mai multe surse** (interne sau externe aplicației) dependency injection sau lookup
- **Fault Tolerance, Health și Metrics**
- **OpenAPI** – mecanism **standard de documentare a serviciilor REST** cu posibilități de testare
- **REST Client** – se creează ușor cu interfețe proxy și adnotări pentru invocarea serviciilor REST
- **JSON Web Tokens (JWT)** – mecanism pentru **autentificare și autorizare distribuită**



# CE ESTE QUARKUS ARHITECTURĂ

- **Quarkus** suportă aplicații pentru **GraalVM** și **HotSpot JVM**
- **Quarkus** este **alcătuit dintr-un nucleu** de dimensiuni reduse care **orchestrează** celelalte părți. **Persistența, tranzacțiile, REST, toleranța la eroare** etc. sunt **adăugate** la aplicație, **ca extensii**, doar dacă sunt folosite
- **Mecanismul de extensie** se bazează pe **ArC** pentru **dependency injection**
- Pentru **optimizarea fazei de compilare** **Quarkus** folosește unelte precum **Jandex** (procesarea adnotărilor) și **Gizmo** (generare bytecode)
- **Quarkus** **unifică** modurile de programare **imperativ** și **reactiv**
- Datorită **nucleului reactiv** bazat pe **Netty** și **Eclipse Vert.x**, **totul în Quarkus este reactiv**
- Fiecare **cerere** este tratată **într-o buclă de evenimente** (event loop), iar în funcție de destinație este trimisă către **cod imperativ (worker thread)** sau către **cod reactiv (IO thread)**

# CE ESTE QUARKUS ARHITECTURĂ





# CLOUD NATIVE COMPUTING

- În **cloud** se plătesc **toate resursele** utilizate
- Dacă **timpul de pornire** al aplicației este **mare, consumă multă memorie sau timp de CPU inutil**, aceste resurse se plătesc
- **Cloud Native Computing (CNC)** - este o abordare care **utilizează cloud-ul privat, public sau hibrid** pentru a **compila și rula aplicații scalabile**
- **Tehnologiile comune** pentru acest stil arhitectural sunt **containerele, microserviciile, funcțiile serverless**
- În CNC fiecare **microserviciu este inclus în propriul container**. Aceste **containere** sunt apoi **orchestrate dinamic** pentru **optimizarea utilizării resurselor**
- **Docker** și **Kubernetes** sunt numele a 2 implementări comune de container și orchestrator

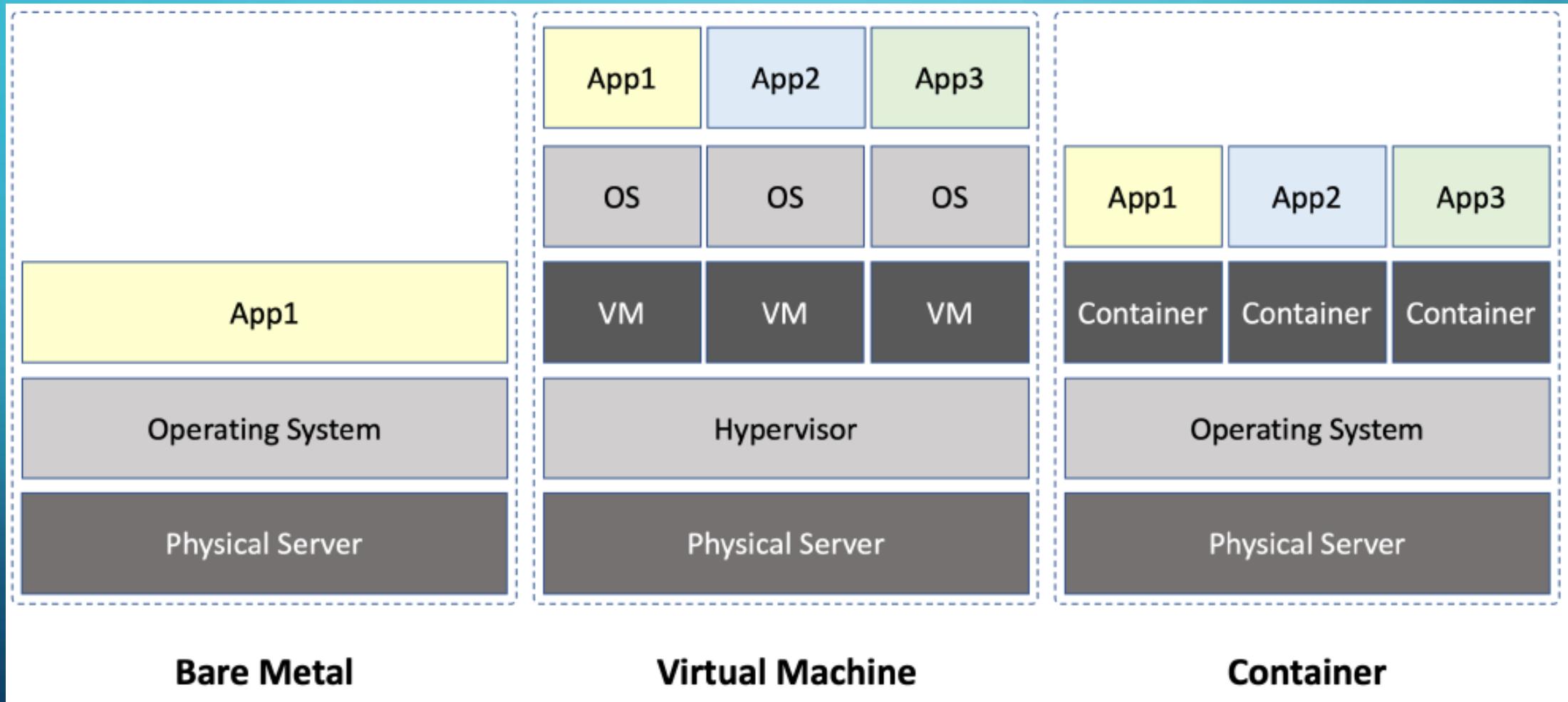
# CLOUD NATIVE COMPUTING

## DOCKER

- Docker este un set de produse **platform-as-a-service (PaaS)** care folosește **virtualizarea la nivel de sistem de operare** pentru a livra software
- Containerele sunt izolate unul de altul și împachetează propriu software, biblioteci și fișiere de configurare. Acestea **comunică** între ele prin **canale bine definite**
- Containerele permit dezvoltatorilor să **împacheteze o aplicație cu toate dependințele** și să o **livreze ca un singur pachet**
- Containerul rulează pe serverului fizic, folosind **sistemul de operare** al acestuia și nu necesită virtualizare
- Docker Image este un **șablon cu instrucțiuni** pentru crearea unui container. Este o **combinăție de sistem de fișiere și parametrii**
- Docker Container este o **instanță executabilă** a imaginii

# CLOUD NATIVE COMPUTING

## DOCKER





# CLOUD NATIVE COMPUTING

## KUBERNETES

- **Kubernetes este un orchestrator pentru aplicații containerizate**
- Poate **programa, scala, repară, modifica, opri și porni containerele** de pe mașini locale sau la distanță
- **Odată aplicația împachetată** într-un container, se declară starea aplicației într-un fișier manifest, iar mai departe **se ocupă Kubernetes**
- **Kubernetes decide pe ce nod rulează containerul** (funcție de cerințele de resurse) și **câte instanțe** ale containerului **creează**
- **Dacă încărcarea** pe containerele existente **crește**, **Kubernetes creează mai multe instanțe** ale acestora, **dacă scade** încărcarea oprește o parte din container
- Dacă un **container eșuează** va **crea automat** unul nou



# CLOUD NATIVE COMPUTING

## GRAALVM

- **GraalVM** este o extensie a Java Virtual Machine (JVM) pentru a suporta mai multe limbi și moduri de execuție
- **GraalVM** are propriul compilator numit **Graal** care este un compilator de tip **Just-in-Time (JIT)** de înaltă performanță
- **GraalVM** suportă evident Java, limbi bazate pe JVM precum **Scala**, **Groovy** și **Kotlin**, dar și **JavaScript**, **Ruby**, **Python**, **R** și **C/C++**
- **GraalVM** permite **crearea de imagini native** din aplicații JVM prin analizarea codului și **compilarea** de tip **Ahead-Of-Time (AOT)**. **Codul binar** rezultat conține **programul în cod mașină**
- Compilarea de tip **AOT** îmbunătățește **timpul de pornire** al programului



# PRIMA APLICAȚIE CU QUARKUS

- Modul de lucru cu VSC pentru crearea unui **project Quarkus** va fi prezentat la laborator
- **Proiectele Quarkus** pot fi create cu interfața web <https://code.quarkus.io> și apoi deschise cu VSC

Proiectul creat se numește **course11/config-quickstart** are ca dependențe:

- **quarkus-resteasy**: REST framework implementing JAX-RS
- **quarkus-resteasy-jsonb**: JSON-B serialisation support for RESTEasy

Pentru testarea aplicației folosim dependențele:

- **quarkus-junit5**: JUnit 5 support in Quarkus
- **rest-assured**: Framework to easily test REST endpoints



# PRIMA APLICAȚIE CU QUARKUS

- Clasa **ArtistResource** este un serviciu REST simplu având ca URI `/artists` și definește 2 metode:
- `getAllArtists()` – returnează lista de artiști în reprezentare JSON
- `countArtists()` – returnează numărul de artiști în format text
- Datele artiștilor sunt ținute în clasa **Artist** care este un POJO cu attribute, constructori și metode getters și setters
- Clasa utilizează adnotarea JSON-B **@JsonbProperty** pentru a schimba denumirea atributelor `firstName` la `first_name` și `lastName` la `last_name`
- Adnotarea **@JsonbTransient** previne ca identificatorul `id` să apară în JSON

# PRIMA APLICAȚIE CU QUARKUS DEZVOLTARE

```
package atm.paradigms.model;
import java.util.UUID;
import javax.json.bind.annotation.JsonbProperty;
import javax.json.bind.annotation.JsonbTransient;

public class Artist {
    @JsonbTransient
    private UUID id;
    @JsonbProperty("first_name")
    private String firstName;
    @JsonbProperty("last_name")
    private String lastName;

    public Artist() {
    }

    public Artist(UUID id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    // getters and setters
}
```

Clasa **Artist** este un POJO decorat cu adnotări **JSON-B** care influențează modul de serializare/deserializare a obiectelor Java la JSON.

**@JsonbProperty** schimbă denumirea atributelor, iar **@JsonbTransient** previne apariția atributului **id** în reprezentarea JSON.

# PRIMA APLICAȚIE CU QUARKUS DEZVOLTARE

```
@Path("/artists")
public class ArtistResource {
    private static List<Artist> artists = List.of(
        new Artist(UUID.randomUUID(), "John", "Lennon"),
        new Artist(UUID.randomUUID(), "Paul", "McCartney"),
        new Artist(UUID.randomUUID(), "george", "Harrison"),
        new Artist(UUID.randomUUID(), "Ringo", "Starr")
    );

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAllArtists(){
        return Response.ok().entity(artists).build();
    }

    @GET
    @Path("/count")
    @Produces(MediaType.TEXT_PLAIN)
    public Integer countArtists(){
        return artists.size();
    }
}
```

Cod REST similar cu cel prezentat în cursurile anterioare.  
Chiar dacă Quarkus folosește RestEasy ca implementare pentru specificația JAX-RS, toate cunoștințele anterioare despre REST rămân valabile.



# PRIMA APLICAȚIE CU QUARKUS RULARE

Listening for transport dt\_socket at address: 5005

2022-10-06 11:09:23,158 WARN [io.net.res.dns.DefaultDnsServerAddressStreamProvider] (build-4) Default DNS servers: [/8.8.8.8:53, /8.8.4.4:53] (Google Public DNS as a fallback)

2022-10-06 11:09:24,172 INFO [io.quarkus] (Quarkus Main Thread) config-quickstart 1.0.0-SNAPSHOT on JVM (powered by Quarkus 2.13.1.Final) started in 2.427s. Listening on: <http://localhost:8080>

1

# PRIMA APLICAȚIE CU QUARKUS

## TESTARE

```
TestRest.rest X
src > test > java > atm > paradigms > TestRest.rest > GET /artists/count
Send Request
1 GET http://localhost:8080/artists/count HTTP/1.1

Response(156ms) X

1 HTTP/1.1 200 OK
2 Content-Type: text/plain;charset=UTF-8
3 connection: close
4 content-length: 1
5
6 4
```

```
Send Request
1 GET http://localhost:8080/artists/ HTTP/1.1

Response(102ms) X

1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 connection: close
4 content-length: 180
5
6 [
7 {
8     "first_name": "John",
9     "last_name": "Lennon"
10 },
11 {
12     "first_name": "Paul",
13     "last_name": "McCartney"
14 },
15 {
16     "first_name": "george",
17     "last_name": "Harrison"
18 },
19 {
20     "first_name": "Ringo",
21     "last_name": "Starr"
22 }
23 ]
```



# PRIMA APLICAȚIE CU QUARKUS

## TESTARE

- În **modul dezvoltare**, Quarkus permite *reîncărcarea la cald*. La modificarea fișierelor Java și/sau resurse, **invocarea serviciului REST va reflecta modificările făcute**

```
private static List<Artist> artists = List.of(
    // new Artist(UUID.randomUUID(), "John", "Lennon"),
    // new Artist(UUID.randomUUID(), "Paul", "McCartney"),
    new Artist(UUID.randomUUID(), "george", "Harrison"),
    new Artist(UUID.randomUUID(), "Ringo", "Starr")
);
```

Modificarea listei în cod se reflectă imediat la accesarea endpoint-ului, fără alte acțiuni din partea programatorului.

Send Request

1 GET <http://localhost:8080/artists> HTTP/1.1

Response(12ms) X

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 connection: close
4 content-length: 91
5
6 [
7   {
8     "first_name": "george",
9     "last_name": "Harrison"
10   },
11 [
12   {
13     "first_name": "Ringo",
14     "last_name": "Starr"
15   }
]
```

# PRIMA APLICAȚIE CU QUARKUS

## TESTARE

- Proiectul Quarkus conține o **zonă de testare**
- Utilizăm **QuarkusTest runner** pentru a transmite **JUnit** să pornească aplicația înainte de a rula testele
- Clasa de test **ArtistResourceTest** conține 2 metode:
- **shouldGetAllArtists()** – verifică dacă **codul de răspuns este 200 și dimensiunea vectorului returnat este 4**
- **shouldCountArtist()** - verifică dacă **codul de răspuns este 200 și răspunsul conține 4**
- Testele au fost dezvoltate folosind **REST Assured**

# PRIMA APLICAȚIE CU QUARKUS TESTARE

```
@QuarkusTest
public class ArtistResourceTest {
    @Test
    public void shouldGetAllArtists() {
        given()
            .when().get("/artists")
            .then()
            .assertThat()
            .statusCode(is(200))
            .and()
            .body("size()", Matchers.equalTo(4));
    }
    ...
}
```

Metode statice puse la dispoziție de **Rest Assured** pentru testarea endpoint-urilor REST. Modul de înlățuire a metodelor este destul de intuitiv. Apar metode specifice testării (ex. `assertThat`).

```
...
@Test
public void shouldCountArtist() {
    given().when()
        .get("/artists/count")
        .then()
        .assertThat()
        .statusCode(is(200))
        .and()
        .body(is("4"));
}
```



# PRIMA APLICAȚIE CU QUARKUS

## TESTARE

- Aplicația se testează cu comanda  
**./mvnw test**
- Quarkus pornește și rulează aplicația pe portul 8081 înainte de rularea testelor

[INFO]

[INFO] Results:

[INFO]

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO]

[INFO] -----

# PRIMA APLICAȚIE CU QUARKUS

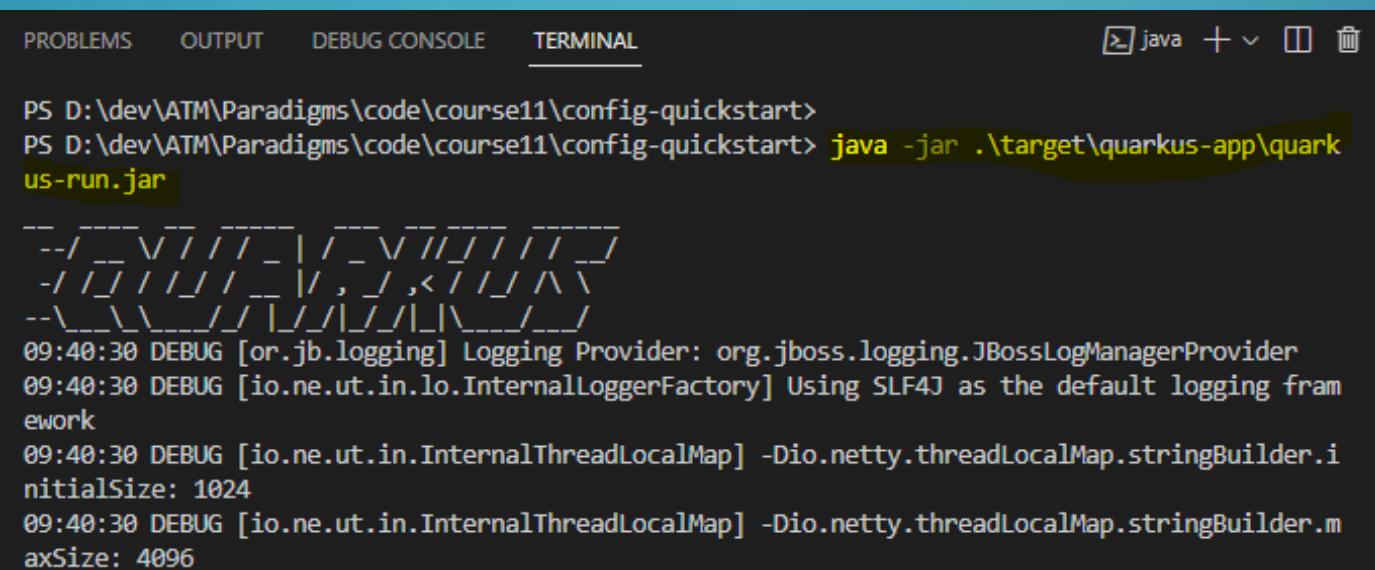
## RULAREA APLICAȚIEI

- Am rulat aplicația în mod dezvoltare cu comanda **quarkus dev** și am beneficiat de reîncărcarea la cald atunci când s-au făcut modificări în cod
- Pentru **mediul de producție** este necesară **împachetarea aplicație** cu comanda  
**./mvnw package**

• **Rezultatul procesului** este folder-ul **quarkus-app** aflat în folder-ul **target**

- Aplicația se rulează cu comanda

**java -jar .\target\quarkus-app\quarkus-run.jar**



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS D:\dev\ATM\Paradigms\code\course11\config-quickstart>
PS D:\dev\ATM\Paradigms\code\course11\config-quickstart> java -jar .\target\quarkus-app\quarkus-run.jar
--/ _ \ \ \ / / / | / _ \ \ / / / / / /
-/ / / / / / / | / , _ / < / / / \ \
--\_\ \_\ / / | / / | / / | / \ / / /
09:40:30 DEBUG [or.jb.logging] Logging Provider: org.jboss.logging.JBossLogManagerProvider
09:40:30 DEBUG [io.ne.ut.in.lo.InternalLoggerFactory] Using SLF4J as the default logging framework
09:40:30 DEBUG [io.ne.ut.in.InternalThreadLocalMap] -Dio.netty.threadLocalMap.stringBuilder.initialSize: 1024
09:40:30 DEBUG [io.ne.ut.in.InternalThreadLocalMap] -Dio.netty.threadLocalMap.stringBuilder.maxSize: 4096
```



# PRIMA APlicațIE CU QUARKUS

## CONTAINERIZAREA APlicațIEI

- Pe Windows este necesară instalarea aplicației **Docker Desktop** care permite **construirea și diseminarea aplicațiilor containerizate**
- Detalii privind comenziile pentru construirea și rularea containerului se găsesc în fișierul **Dockerfile.jvm** din folder-ul **docker** din cadrul proiectului
- **Comanda pentru crearea unui container bazat pe JVM este:**

***docker build -f src/main/docker/Dockerfile.jvm -t quarkus/config-quickstart-jvm .***

- Pentru **rulare** se folosește comanda:

***docker run -i --rm -p 8080:8080 quarkus/config-quickstart-jvm***

The screenshot shows the Docker Desktop application window. On the left, there's a sidebar with icons for 'Images', 'Volumes', and 'Dev Environments' (which has a 'PREVIEW' badge). The main area is titled 'LOCAL' under 'REMOTE REPOSITORIES'. It features a search bar and an 'In Use only' checkbox. A table lists the following information:

NAME	TAG	IMAGE ID	CREATED	SIZE
quarkus/config-quickstar...	IN USE latest	7b1649f257ba	10 minutes ago	420.54 MB



# CONTEXT AND DEPENDENCY INJECTION (CDI)

- Injecția este **componenta centrală a Quarkus** și este implementată prin ArC.  
**CDI este parte a nucleului Quarkus**
- Se pot injecta: **un bean în altul, o configurație într-un component, o resursă într-un component**
- **CDI este tehnologia centrală** în Jakarta EE și **MicroProfile**
- **CDI** vine cu un set de adnotări:

Adnotare	Descriere
@Inject	Marchează constructori, metode și câmpuri injectabile
@Qualifier	Specifică calificatori pentru adnotări
@ApplicationScoped, @SessionScoped, @RequestScoped, @Singleton, @Dependent	Set de adnotări care definesc ciclul de viață al unui bean
@Observes	Marchează un parametru eveniment al unei metode Observer

# CONTEXT AND DEPENDENCY INJECTION (CDI)

- În **Java SE** există **Java Beans** care sunt de fapt **POJO** (Plain Old Java Object) ce se execută în JVM
- În **Quarkus** există **Managed Beans** care sunt **administrate de container și suportă injectia de resurse, managementul ciclului de viață și interceptarea**
- **Managed Beans** există și în implementările JAX-RS
- **Managed Beans** sunt obiecte care păstrează starea, sunt legate de un context, și sunt sigure din punct de vedere al tipului
- **Injectarea** dependințelor se face **prin adnotări**
- Prezentăm un exemplu de injectare a dependințelor prin adnotări

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms.model;

// POJO
public class Book {
    private String title;
    private Float price;
    private String description;
    private String isbn;

    public Book() {
    }

    public Book(String title, Float price,
               String description) {
        this.title = title;
        this.price = price;
        this.description = description;
    }
    // getters and setters
}
```

Vezi proiectul [course11/cdi-injection](#)

```
package atm.paradigms;

public interface NumberGenerator {
    String generateNumber();
}

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class IsbnGenerator implements NumberGenerator{

    @Override
    public String generateNumber() {
        return "13-84356-" +
            Math.abs(new Random().nextInt());
    }
}
```

Clasa **IsbnGenerator** care implementează interfața **NumberGenerator** este decorată cu anotarea **@ApplicationScoped** și poate fi astfel injectată într-un alt bean.

# CONTEXT AND DEPENDENCY INJECTION (CDI)

- Adnotarea `@Inject` pe câmp va informa **containerul** că trebuie să insereze o referință de tip **NumberGenerator** în câmpul **numberGenerator**
- **Locul** în care se află adnotarea se numește **punct de injecție**
- Pentru a funcționa cu **GraalVM**, nu se folosește **private** la punctul de injecție

```
import javax.inject.Inject;
import atm.paradigms.model.Book;

@ApplicationScoped
public class BookService {
    @Inject
    NumberGenerator numberGenerator;

    public Book createBook(String title, Float price, String description){
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}
```

Clasa **BookService** este decorată cu adnotarea `@ApplicationScoped` și poate fi astfel injectată într-un alt **bean**. Conține un **punct de injecție** de tip **NumberGenerator** marcat cu adnotarea `@Inject`.

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import atm.paradigms.model.Book;

@Path("book")
public class BookResource {
    @Inject
    BookService bookService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getBook(){
        Book book = bookService.createBook("Morometii", 56.88F, "About contrymen");
        return Response.ok(book).build();
    }
}
```

# CONTEXT AND DEPENDENCY INJECTION (CDI)

- Interfața **NumberGenerator** are numai implementarea **IsbnGenerator**
- CDI va putea să determine implementarea și să o injecteze folosind **@Inject**
- Acest tip de injectare se numește **injectare implicită**
- Poate fi marcat cu adnotarea **@Default**, dar nu este obligatoriu aici

```
@ApplicationScoped  
@Default  
public class IsbnGenerator implements NumberGenerator{  
    @Override  
    public String generateNumber() {  
        return "13-84356-" + Math.abs(new Random().nextInt());  
    }  
}
```

```
@ApplicationScoped  
public class BookService {  
    @Inject @Default  
    NumberGenerator numberGenerator;  
    ...
```

# CONTEXT AND DEPENDENCY INJECTION (CDI)

- Ce se întâmplă dacă avem 2 implementări pentru **NumberGenerator**?
- Presupunem că avem și cărți cu codul ISSN (ISBN are 13 cifre, ISSN are 8 cifre)
- Vom avea 2 implementări **IsbnGenerator** și **IssnGenerator**. Cum va ști CDI pe care să o injecteze?
- Fălăsim adnotarea **@Qualifier** pentru a crea 2 adnotări corespunzătoare generatoarelor de numere cu 13, respectiv 8 cifre
- Odată definiți calificatorii se aplică implementările corespunzătoare: **@ThirteenDigits** la **IsbnGenerator**, **@EightDigits** la **IssnGenerator**
- Calificatorii creați se aplică la punctele de injecție

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms;
import java.lang.annotation.*;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD})
public @interface ThirteenDigits { }
```

Se definesc adnotările de tip **@Qualifier**.

```
package atm.paradigms;
import java.lang.annotation.*;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD})
public @interface EightDigits { }
```

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms;
import java.util.Random;
import javax.enterprise.context.ApplicationScoped;

@ThirteenDigits
@ApplicationScoped
public class IsbnGenerator implements NumberGenerator{
    @Override
    public String generateNumber() {
        return "13-84356-" + Math.abs(new Random().nextInt());
    }
}
```

Se aplică adnotarea corespunzătoare pe fiecare implementare a interfeței **NumberGenerator**.

```
package atm.paradigms;
import java.util.Random;
import javax.enterprise.context.ApplicationScoped;

@EightDigits
@ApplicationScoped
public class IssnGenerator implements NumberGenerator {
    @Override
    public String generateNumber() {
        return "8-" + Math.abs(new Random().nextInt());
    }
}
```

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import atm.paradigms.model.Book;

@ApplicationScoped
public class BookService {
    @Inject
    @ThirteenDigits
    NumberGenerator numberGenerator;

    public Book createBook(String title, Float price, String description){
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}
```

La punctul de injecție se specifică implementarea dorită a interfeței **NumberGenerator**.

# CONTEXT AND DEPENDENCY INJECTION (CDI)

```
package atm.paradigms;
import javax.inject.Inject;
import atm.paradigms.model.Book;

public class LegacyBookService {
    @Inject
    @EightDigits
    NumberGenerator numberGenerator;

    public Book createBook(String title, Float price, String description) {
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        return book;
    }
}
```

# CONTEXT AND DEPENDENCY INJECTION (CDI)

## SCOP

- CDI înseamnă **injecția** dependențelor dar și **context**
- Fiecare **bean** are **scop** și un **ciclu de viață** bine definit legate de un **context**
- În Java scopul POJO este simplu: se creează o instanță a clasei cu operatorul **new** și ne bazăm pe GC să o șteargă
- În **CDI**, **bean-ul** este **legat de un context** și **rămâne în acel context** până ce **acesta este distrus de container**
- Nu există modalități manuale de ștergere a unui bean dintr-un context

# CONTEXT AND DEPENDENCY INJECTION (CDI)

## SCOP

### Scopuri normale:

- ***Application scope (@ApplicationScoped)*** – se întinde pe **întreaga durată de viață a aplicației**. Bean-ul se creează o singură dată pentru toată durata aplicației și se șterge la oprirea aplicației. Este **util pentru clase utilizare** sau obiecte care stochează date utilizate de întreaga aplicație (problema cu thread-urile rămâne)
- ***Session scope (@SessionScoped)*** – se întinde peste mai multe cereri HTTP sau invocare de metode **în cadrul unei singure sesiuni utilizator**. Bean-ul este **creat pe toată durata sesiunii HTTP** și este **șters la terminarea acesteia**. Acest scop se utilizează pentru obiecte care sunt necesare pe timpul sesiunii precum date de logare sau preferințe utilizator
- ***Request scope (@RequestScoped)*** – se întinde pe **durata unei singure cereri HTTP** sau invocare de metodă. Este utilizat de clase serviciu care sunt necesare doar pe durata unei cereri HTTP



# CONTEXT AND DEPENDENCY INJECTION (CDI)

## SCOP

### Pseudo scopuri:

- **Dependent scope (@Dependent)** – un bean dependent se creează de fiecare dată când este injectat și se șterge când ținta injecției se șterge. Acesta este scopul implicit pentru CDI
- **Singleton scope (@Singleton)** – identifică un bean pe care CDI îl instațiază numai o dată
- Exemplu de scop dependent

```
@Dependent @ThirteenDigits  
public class IsbnGenerator implements NumberGenerator{ . . . }
```

- **@Dependent** poate fi omis

# CONTEXT AND DEPENDENCY INJECTION (CDI)

## EVENIMENTE

- Evenimentele CDI permit bean-urilor să interacționeze fără dependințe la compilare
- O dată ce un bean definește un eveniment (interfața `javax.enterprise.event.Event`), un alt bean poate genera evenimentul (cu metoda `fire()`) iar un altul poate trata evenimentul cu adnotarea `@Observes` (observer/observable design pattern)
- Bean-urile pot fi în pachete separate sau chiar în biblioteci separate (JAR) ale aplicației
- Modificăm exemplul anterior astfel încât `BookService` generează un eveniment ori de câte ori un obiect `Book` este creat
- Evenimentul este generat cu metoda `bookAddedEvent.fire(book)` și ajunge la un consumator

# CONTEXT AND DEPENDENCY INJECTION (CDI)

## EVENIMENTE

- Evenimentele sunt generate de clasa **BookService** prin metoda **createBook()**
- Se **injectează** o **instanță** a **clasei Event**

Vezi proiectul **course11/cdi-events**

```
@ApplicationScoped
public class BookService {
    @Inject
    NumberGenerator numberGenerator;

    @Inject
    Event<Book> bookAddedEvt;

    public Book createBook(String title, Float price, String description){
        Book book = new Book(title, price, description);
        book.setIsbn(numberGenerator.generateNumber());
        bookAddedEvt.fire(book);
        return book;
    }
}
```



# CONTEXT AND DEPENDENCY INJECTION (CDI)

## EVENIMENTE

- Evenimentele sunt **consumate** de un bean de tip **Observer**
- Metodele **Observer** primesc ca parametru **evenimentele** de un anumit tip **dacă sunt adnotate cu @Observes**

```
@Singleton
public class InventoryService {
    @Inject
    Logger logger;

    List<Book> inventory = new ArrayList<>();

    public void addBook(@Observes Book book){
        logger.info("Adding book " + book.getTitle() + " to inventory");
        inventory.add(book);
    }
}
```

Metoda **addBook()** a clasei **InventoryService** primește evenimentele de tip **Book** generate de clasa **BookService**. **InventoryService** are scop **Singleton**.

- Rezultat la accesarea URI <http://localhost:8080/book>  
2022-10-10 13:19:40,677 INFO [atm.par.InventoryService] (executor-thread-0) Adding book Morometii to inventory

# CONTEXT AND DEPENDENCY INJECTION (CDI)

## EVENIMENTE

- Dacă sunt **mai multe evenimente**, se folosesc **calificatori pentru a distinge între ele**, atât cu adnotarea **@Inject** cât și **@Observes**

```
@ApplicationScoped  
public class BookService {  
    . . .  
  
    @Inject @Added  
    Event<Book> bookAddedEvt;  
  
    @Inject @Removed  
    Event<Book> bookRemovedEvt;  
    . . .
```

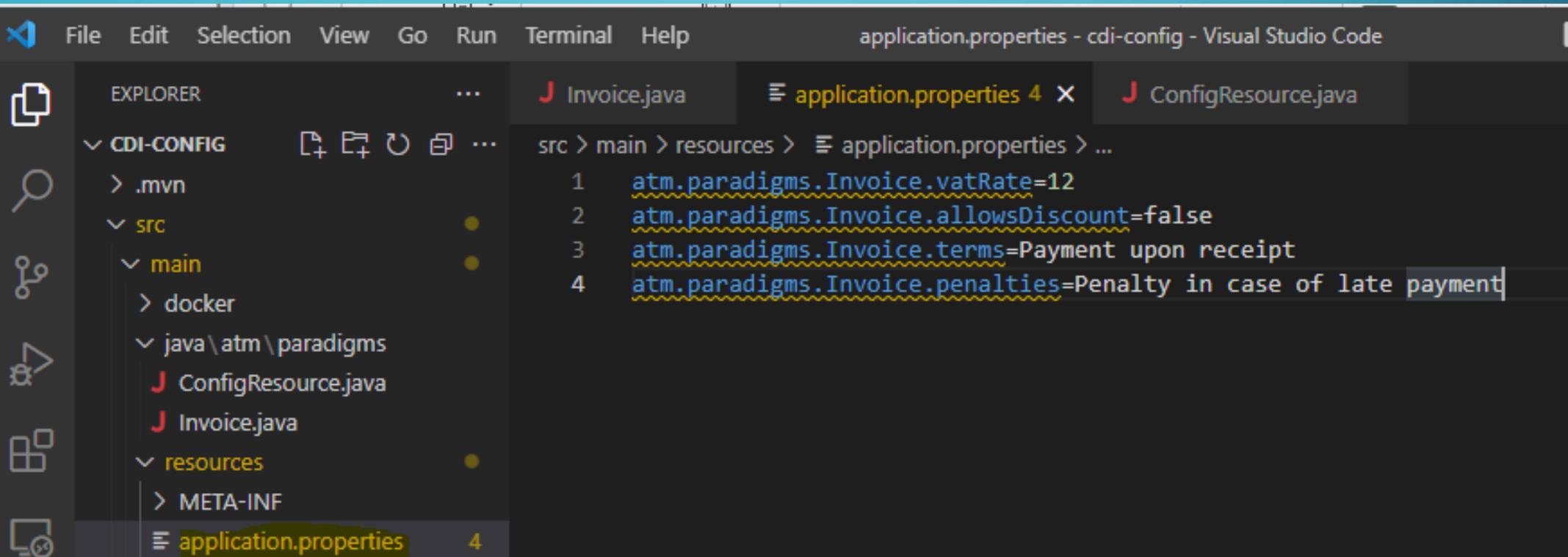
```
@Singleton  
public class InventoryService {  
    . . .  
  
    public void addBook(@Observes @Added Book book) {  
        logger.info("Adding book " + book.getTitle()  
                   + " to inventory");  
        inventory.add(book);  
    }  
  
    public void removeBook(@Observes @Removed Book book) {  
        logger.info("Removing book " + book.getTitle()  
                   + " from inventory");  
        inventory.remove(book);  
    }  
}
```

# CONFIGURATION

- **Eclipse MicroProfile Configuration** permite aplicațiilor și microserviciilor să obțină proprietățile de configurare din mai multe surse interne și externe, injecția dependențelor sau căutare
- API de configurare vine cu adnotarea `@ConfigProperty` care leagă punctul de injecție cu valoarea din configurație
- De multe ori este necesară schimbarea datelor de configurare din afara aplicației (ex. conexiunea la baza de date, date de logare pentru un serviciu extern) fără compilarea și reîmpachetarea aplicației
- Datele de configurare pot veni în **diferite formate** (.properties, .xml, .yaml) și din **diferite surse** (proprietăți și variabile de sistem, fișiere externe, baze de date)

# CONFIGURATION

- Implicit proprietățile sunt stocate în fișierul **application.properties** din folderul **resources** folosind **convenția de nume cu punct**  
**<fully qualified package name>.<bean name>.<attribute name>**



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** application.properties - cdi-config - Visual Studio Code.
- Explorer:** Shows the project structure:
  - CDI-CONFIG
  - .mvn
  - src
    - main
      - docker
      - java\atm\paradigms
        - ConfigResource.java
        - Invoice.java
      - resources
        - META-INF
        - application.properties
- Editor:** The application.properties file is open, showing the following content:

```
atm.paradigms.Invoice.vatRate=12
atm.paradigms.Invoice.allowsDiscount=false
atm.paradigms.Invoice.terms=Payment upon receipt
atm.paradigms.Invoice.penalties=Penalty in case of late payment
```

# CONFIGURATION

```
package atm.paradigms;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@ApplicationScoped
public class Invoice {
    Float subtotal;
    @Inject
    @ConfigProperty(defaultValue = "10")
    Float vatRate;
    Float vatAmount;
    Float total;
    @Inject
    @ConfigProperty(defaultValue = "true")
    Boolean allowsDiscount;
    @Inject
    @ConfigProperty(defaultValue = "2.5")
    Float discountRate;
    ...
}
```

Vezi proiectul [course11/cdi-config](#)

# CONFIGURATION

- Folosim o clasă resursă simplă pentru testare

```
@Path("config")
public class ConfigResource {
    @Inject
    Logger logger;

    @Inject
    Invoice invoice;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getInvoice(){
        logger.info(invoice);
        return invoice.toString();
    }
}
```

- Rezultate: în browser și la consolă
- Invoice [subtotal=null, vatRate=12.0, vatAmount=null, total=null, allowsDiscount=false, discountRate=2.5, terms=Payment upon receipt, penalties=Penalty in case of late payment]*



# CONFIGURATION

- Adnotarea **@ConfigProperty** are 2 parametrii:
  - *Numele cheii proprietății de configurare* pentru care să caute valoarea (ex. **invoice.vatRate** sau **invoice.terms**)
  - *Valoarea implicită* în caz că valoarea proprietății nu există

```
@Singleton
public class OptionalInvoice {
    Float subtotal = 0.0F;
    @ConfigProperty(name = "invoice.vatRate", defaultValue = "10")
    Float vatRate;
    Float vatAmount = 0.0F;
    Float total = 0.0F;
    @ConfigProperty(name = "invoice.allowsDiscount", defaultValue = "true")
    Boolean allowsDiscount;
    @ConfigProperty(name = "invoice.discountRate", defaultValue = "2.5")
    Float discountRate;
    @ConfigProperty(name = "invoice.terms")
    Optional<String> terms;
    @ConfigProperty(name = "invoice.penalties")
    Optional<String> penalties;
}
```

# CONFIGURATION

- Datele din fișierul de configurare *application.properties*

```
invoice.vatRate=15  
invoice.allowsDiscount=false  
invoice.terms=Payment upon receipt  
invoice.penalties=Penalty in case of late payment
```

- Rezultat obținut cu clasa resursă

```
2022-10-11 10:15:16,626 INFO [atm.par.OptionalInvoiceResource] (executor-thread-0) OptionalInvoice [subtotal=0.0, vatRate=15.0, vatAmount=0.0, total=0.0, allowsDiscount=false, discountRate=2.5, terms=Optional[Payment upon receipt], penalties=Optional[Penalty in case of late payment]]
```



# CONFIGURATION

- Proprietățile pot fi puse și într-un fișier YAML situat în folder-ul **resources** din proiect
- Quarkus va căuta mai întâi în fișierul YAML

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure:
  - CDI-CONFIG
  - .mvn
  - .vscode
  - src
    - main
      - docker
      - java
    - resources
      - META-INF
      - application.properties
      - application.yaml
    - test
- Terminal (Top Right):** Shows the current working directory: src > main > resources > application.yaml
- Code Editor (Right):** Displays the contents of application.yaml:

```
1 app:
2   invoice:
3     vatRate: 50
4     allowsDiscount: false
5     terms: Payment upon receipt
6     penalties: Penalty in case of late payment
```

# CONFIGURATION

```
@Singleton
public class InvoiceYaml {
    Float subtotal;
    @ConfigProperty(name = "app.invoice.vatRate", defaultValue = "10")
    Float vatRate;
    Float vatAmount;
    Float total;
    @ConfigProperty(name = "app.invoice.allowsDiscount", defaultValue = "true")
    Boolean allowsDiscount;
    @ConfigProperty(name = "app.invoice.discountRate", defaultValue = "2.5")
    Float discountRate;
    @ConfigProperty(name = "app.invoice.terms")
    String terms;
    @ConfigProperty(name = "app.invoice.penalties")
    String penalties;
    @Override
}
```

2022-10-11 11:13:08,020 INFO [atm.par.InvoiceYamlResource] (executor-thread-0) InvoiceYaml  
[subtotal=null, vatRate=50.0, vatAmount=null, total=null, allowsDiscount=false, discountRate=2.5,  
terms=Payment upon receipt, penalties=Penalty in case of late payment]

# CONFIGURATION

- Implicit Quarkus citește configurația aplicației din fișierul **application.properties** aflat în directorul `src/main/resources`
- **Eclipse MicroProfile Configuration** permite mai multe surse pentru configurarea aplicației, fiecare sursă având un număr de ordine
- Sursele cu număr de ordine mai mare suprascriu valorile celor cu prioritate mai scăzută

## Ordinea descrescătoare a priorităților surselor:

- Proprietăți de sistem : `-Dinvoice.vatRate=50`
- Variabile de mediu: `INVOICE_VATRATE=50`
- Fișierul cu numele `.env` situat în directorul de lucru: `INVOICE_VATRATE=50`
- Directorul cu numele `config` situat în directorul de lucru: `config/application.properties`
- Resursa `src/main/resources/application.properties`



# CONFIGURATION

- Quarkus însuși se configurează în același fel ca aplicația, dar rezervă spațiul de nume **quarkus**, pentru **configurarea proprie**
- Quarkus are în jur de 100 de chei de configurare
- Câteva exemple:

Proprietate	Valoare implicită
quarkus.http.root-path Calea HTTP rădăcină	/
quarkus.http.port Port HTTP	8080
quarkus.http.test-port Port HTTP folosit de teste	8081
quarkus.http.ssl.protocols Lista de protocoale SSL activate	TLSv1.3,TLSv1.2
quarkus.http.auth.form.login-page Pagina de logare	/login.html

# CONFIGURATION

- **Quarkus suportă noțiunea de profil de configurare**
- Pot fi **stocate mai multe profile** în același fișier și pot fi selectate folosind numele
- **Quarkus are implicit 3 profile:**
  - **dev**: se activează în modul dezvoltare
  - **test**: se activează la rularea testelor
  - **prod**: se activează în modul producție
- Sintaxa este **%{profile}.config.key=value**
- Exemplu de proprietate pentru profilul dezvoltare **%dev.isbn.prefix**

```
java -Dquarkus.profile=staging -jar profiles-1.0-runner.jar
```

# BIBLIOGRAFIE

- **Understanding Quarkus**, Antonio Goncalves, Independently Published, 2020
- **Quarkus Cookbook**, Alex Soto Bueno and Jason Porter, O'Reilly Media, 2020
- **Beginning Quarkus Framework**, Tayo Koleoso, Apress, 2020
- [Quarkus Guides -Latest](#)