DEVOPS
Demo Application

DEVOPS & CLOUD COMPUTING IM JAVA UMFELD

DevOps, Cloud Computing, Iaas, PaaS, Container und Micro-.

CGS – IT Solutions @ 2020/2024
Version 1.0.7

Christian Schaefer
cgs@cgs.at

1 Inhalt

2 3 Basic Cloud Architecture Concepts - Comparison

Using different cloud components, we can implement a cloud based solution in multiple different ways:

1. Classic Approach with Virtual Maschines and 3 Tier Application Layers
2. Serverless Lambda based approach
3. Container based approach with Containers and or full k8s Solution

There endless different possibilities, but those 3 are some basic decisions which have to be taken, and should compared here:

2.1 Traditional 3-Tier VPC/Network Architecture

When deploying a Quarkus-based REST application, there are several cloud architecture patterns to consider. These patterns are tailored to meet different operational requirements, scalability needs, and deployment preferences.

Overview:
This architecture is the classic setup for deploying applications with distinct layers for presentation, logic, and data. The application is hosted on virtual machines (VMs) within a 3-tier Virtual Private Cloud (VPC) network structure.

Components:

- **Tier 1 (Presentation Layer)**:
  - Load balancer (e.g., AWS Elastic Load Balancer) to distribute incoming requests.
- **Tier 2 (Application Layer)**:
  - REST Quarkus app deployed on VMs (e.g., EC2 instances).
  - Auto-scaling groups to manage load.
- **Tier 3 (Data Layer)**:
  - Database hosted on a managed service (e.g., RDS) or self-hosted.

Benefits:

- Provides strong control over networking and instance configurations.
- Well-suited for workloads requiring stable, long-running environments.
- Enhanced security with private subnets and Network Access Control Lists (NACLs).

Drawbacks:

- Requires significant management effort for scaling, patching, and monitoring VMs.
- Not as agile as serverless or containerized architectures.

2.2 Serverless with AWS Lambda

Overview:
Serverless architecture eliminates the need to manage infrastructure. The Quarkus app runs as a lightweight serverless function, triggered by HTTP requests or other events.

Components:

- **Trigger**:
  - API Gateway to expose the REST API endpoints.
- **Application**:
  - The Quarkus app runs as an AWS Lambda function.
- **Data**:
  - DynamoDB for serverless, scalable data storage, or RDS for relational databases.

Benefits:

- No infrastructure management; AWS handles scaling and availability.
- Cost-effective as you pay only for usage (compute time and API Gateway requests).
- Faster deployments due to the serverless model.

Drawbacks:

- Cold starts may cause latency, especially for Quarkus in JVM mode.
- Limited control over the runtime environment compared to VMs or Kubernetes.
- Suitable mainly for stateless applications.

2.3 Containerized Deployment with Kubernetes

2.4 Kubernetes Overview

A modern, containerized approach where the Quarkus app is packaged into containers and deployed to a Kubernetes cluster.

Components:

- **Cluster**:
  - Managed Kubernetes service (e.g., EKS, GKE, or AKS) to orchestrate containers.
- **Application**:
  - Quarkus app runs inside Docker containers.
  - Exposed via Kubernetes Ingress or AWS Application Load Balancer (ALB).
- **Data**:
  - Database hosted either inside the cluster (PostgreSQL) or as a managed service.

Benefits:

- High scalability with Kubernetes' horizontal pod autoscaling.
- Easy integration with CI/CD pipelines for continuous delivery.
- Supports hybrid and multi-cloud deployments.

Drawbacks:

- Higher operational complexity compared to serverless.
- Requires Kubernetes expertise for effective management.
- Overhead in managing container images and configurations.

3 Comparison of the Architectures

Feature	3-Tier VPC/Network	Serverless (Lambda)	Kubernetes
Management Effort	High	Low	Medium
Scalability	Moderate (manual scaling)	High (automatic)	High (pods auto-scale)
Cost Efficiency	Moderate	High (pay-per-use)	Moderate
Complexity	Low to Moderate	Low	High
Ideal Use Case	Stable, long-running apps	Event-driven, lightweight apps	Large-scale, multi-cloud

4 Architecture Conclusion Review

Each architecture suits specific use cases:

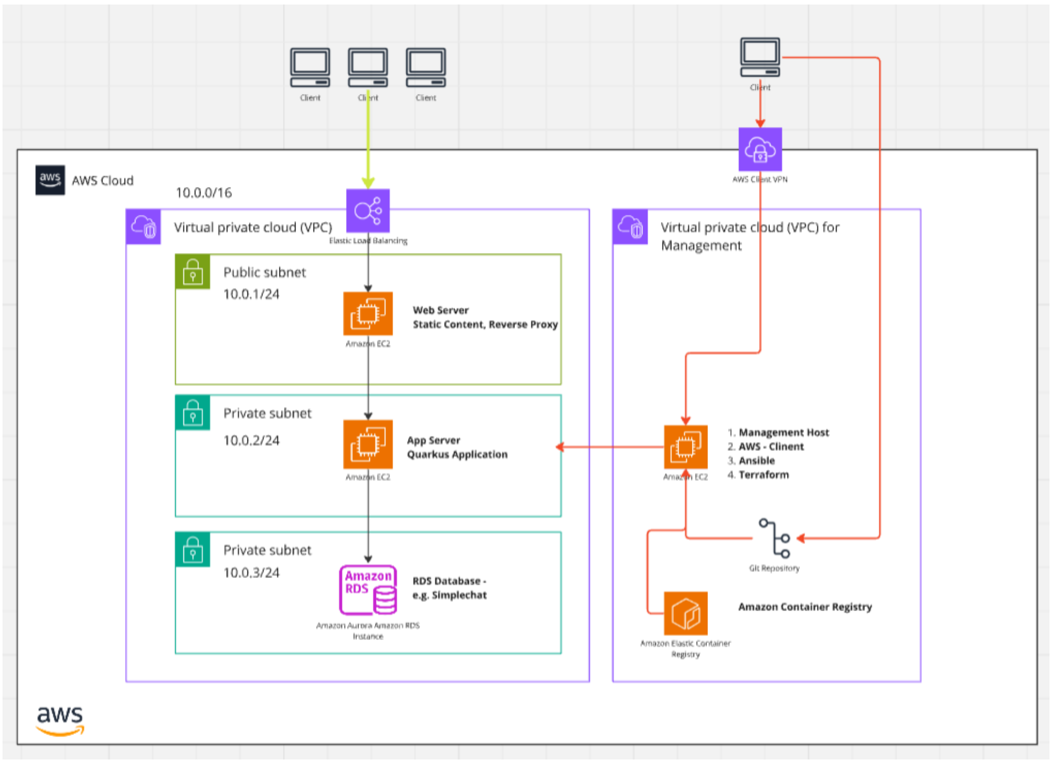
1. **3-Tier VPC/Network**: Ideal for organizations requiring strong control over networking and infrastructure, with stable workloads.
2. **Serverless**: Best for cost-sensitive, event-driven, or lightweight applications where simplicity and agility are key.
3. **Kubernetes**: Perfect for applications requiring container orchestration, hybrid-cloud deployments, or advanced scalability.

Choosing the right architecture depends on the application's scalability requirements, operational complexity, and cost considerations.

5 Training Application Overview

5.1 3 Layer Stacked Application

This architecture diagram illustrates a **3-tier cloud-based application** deployed on **AWS Cloud Infrastructure**.



6 1. Virtual Private Cloud (VPC) - Application Infrastructure

7 Public Subnet (10.0.1/24)

- **Web Server (Amazon EC2):**
  - Hosts static content and acts as a reverse proxy.
  - Connects to the Elastic Load Balancer (ELB), which distributes traffic from clients.

8 Private Subnet (10.0.2/24)

- **Application Server (Amazon EC2):**
  - Runs the main application, specifically a **Quarkus**-based application server.
  - Processes business logic and communicates with the database layer.

9 Private Subnet (10.0.3/24)

- **RDS Database (Amazon Aurora):**
  - Managed relational database service used to persist data.
  - Connected to the application server (port 5432 for PostgreSQL).

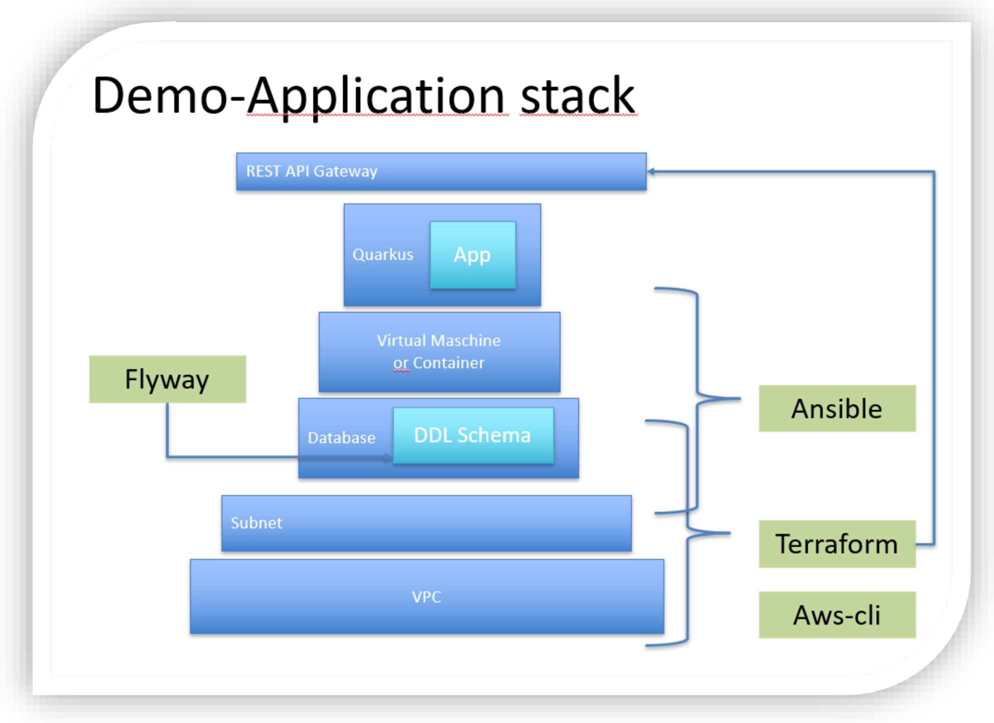
9.1 Management VPC for CI/CD and Infrastructure Management

- **Management Host (Amazon EC2):**
    - Hosts tools for DevOps processes, including:
        1. AWS CLI (Command Line Interface)
        2. Ansible (for configuration management)
        3. Terraform (for infrastructure as code)
    - Pulls code from the Git repository and manages deployment pipelines.
- **Amazon Elastic Container Registry (ECR):**
    - Stores containerized application images for deployment.
- **AWS Client VPN:**
    - Enables secure access to the management VPC and AWS infrastructure for authorized clients.

9.2 3-Tier Connections and Traffic Flow

1. **Client Access:**
   - Clients access the application via the Elastic Load Balancer, which forwards requests to the web server in the public subnet.
2. **Internal Communication:**
   - The Web Server forwards application requests to the Application Server in the private subnet.
   - The Application Server queries the RDS database for data.
3. **Management Traffic:**
   - The Management Host (in the Management VPC) pulls code from the **Git Repository** and deploys application updates to Amazon EC2 instances.
   - Infrastructure changes are managed using Terraform and Ansible.
4. **Secure Data Storage:**
   - All sensitive data is stored in the **RDS database**, which resides in a private subnet.

10 Infrastructure Stack for the Application - Pyramid



10.1 Technology Stack

1. **Infrastructure**:
   - **Amazon VPC**: A virtual private cloud for isolating and securing resources.
   - **Three-tier Subnet Architecture**: Separation of public, private, and restricted layers for better scalability and security.
   - **Virtual Machine Instance**: A compute resource to host the Quarkus microservice.
2. **Application**:
   - **Quarkus**: A modern Java framework optimized for Kubernetes and designed for building lightweight, cloud-native microservices.
   - **Flyway**: Database migration tool for managing schema changes.
   - **REST API**: Provides CRUD (Create, Read, Update, Delete) operations.

11 Microservice Description

12 The microservice is designed to manage TestEntity objects via a REST API. Here’s a breakdown of the features provided by the updateTestEntity method:

12.1 REST Method: PUT /{id}/{name}/{version}

- **Purpose**: Update an existing TestEntity with a new name and version for optimistic locking.
- **Inputs**:
  - id: The unique identifier of the entity (must be a positive number).
  - name: The new name of the entity (cannot be empty).
  - version: Version number for optimistic locking to prevent concurrent updates.
- **Validation**:
  - Ensures valid inputs for id, name, and version.
  - Returns appropriate HTTP status codes for invalid inputs (400 Bad Request) or when the entity is not found (404 Not Found).
- **Processing Logic**:
  - Retrieves the entity from the repository.
  - Updates its name and version.
  - Handles optimistic locking using the provided version to prevent concurrent updates from overwriting each other.
  - Returns the updated entity as a TestDTO.
- **Error Handling**:
  - **OptimisticLockException**: Indicates that the entity was updated by another process.
  - **General Exception**: Captures other errors during the update process and logs them.

13 Quarkus Implementation

Below is the key functionality of the updateTestEntity method:

1. **Logging**: Logs actions, warnings, and errors for better traceability.
2. **Validation**: Ensures the integrity of input parameters.
3. **Database Interaction**:
   - Fetches the entity using testEntityRepository.findById.
   - Updates the entity via testEntityRepository.updateTestEntity.
4. **Response Mapping**:
   - Maps the updated entity to a TestDTO for a clean API response.
5. **Error Handling**:
   - Uses specific and generic exceptions to handle edge cases.

14 Infrastructure Setup

1. **Amazon VPC**:
    - Define a VPC with public and private subnets to securely deploy the VM instance hosting the Quarkus application.
2. **VM Instance**:
    - Host the Quarkus application on an EC2 instance.
    - Enable secure access through private subnets.
3. **Database**:
    - Deploy a database instance in the private subnet.
    - Use Flyway for schema migration and version control.

15 Scenarios Demonstrated

- How to handle API input validation.
- Implementing optimistic locking to ensure data consistency.
- Using a structured DTO for clean API responses.
- Designing and deploying a microservice in a cloud environment.

This structure provides a robust and practical example of a modern cloud-based microservice using Quarkus.

16 Flyway Overview for Trainees

Flyway is a tool for managing database migrations in a versioned and consistent manner. It is often used in software projects to apply and manage changes to the database schema. Flyway ensures that the same set of database changes (migrations) are applied across different environments (e.g., development, testing, production) in a reliable and predictable way.

16.1 Key features of Flyway:

1. **Versioned Migrations**: Flyway supports versioned database migrations using SQL scripts. Each migration is associated with a version number and a description, making it easy to track the changes over time.
2. **Automated Execution**: Flyway can be integrated into CI/CD pipelines to automatically apply migrations to databases as part of the deployment process.
3. **Backward Compatibility**: Flyway ensures that migrations can be rolled back or forward without breaking the system.
4. **Cross-Database Compatibility**: Flyway supports multiple relational databases like PostgreSQL, MySQL, Oracle, SQL Server, etc.

16.2 Flyway Workflow

1. **Migration Scripts**: Write SQL migration scripts (e.g., V1__create_table.sql, V2__add_column.sql).
2. **Migration Execution**: Flyway will execute the scripts in order, applying changes to the database.
3. **Tracking Migrations**: Flyway uses a metadata table (flyway_schema_history) to track which migrations have been applied.

16.3 Flyway Migration Script Example

In Flyway, the above SQL scripts would be written in migration files. Here's an example of how the migration script would look:

- **V1__create_table.sql**:

This will:

1. Apply the SQL script to the database.
2. Track the migration in the flyway_schema_history table.

16.4 Practical Usage

1. **Development**:
    - Add new migration scripts for schema changes (e.g., adding columns or indices).
2. **Deployment**:
    - Flyway ensures the same migrations are applied across staging, testing, and production environments.
3. **History Tracking**:
    - Flyway records applied migrations in a metadata table (flyway_schema_history).

17 This approach ensures database changes are automated, consistent, and version-controlled.

18 Quarkus Property Management: Example with `greetingProperty`

Quarkus allows developers to manage application properties using the `application.properties` or `application.yml` file. These properties enable fine-grained control over the application's behavior and configuration.

Below is an example of how a custom property, `greetingProperty`, can be defined, accessed, and used within a Quarkus application.

18.1 1. Defining the Property

In the `application.properties` file, you can define the custom property as follows:

```
custom.greetingProperty=Hello, Quarkus!
```

Here, `custom.greetingProperty` is the key, and `Hello, Quarkus!` is its value.

18.2 2. Injecting the Property

To use the property in your application, you can inject it using the `@ConfigProperty` annotation provided by Quarkus.

Example:

```
import org.eclipse.microprofile.config.inject.ConfigProperty;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/greeting")
public class GreetingResource {

   @ConfigProperty(name = "custom.greetingProperty")
   String greetingProperty;

   @GET
   @Produces(MediaType.TEXT_PLAIN)
   public String getGreeting() {
      return greetingProperty;
   }
}
```

18.3 3. Providing Default Values

You can provide a default value in case the property is not defined in the `application.properties` file.

Example:

```
@ConfigProperty(name = "custom.greetingProperty", defaultValue = "Hello, World!")
String greetingProperty;
```

If `custom.greetingProperty` is not defined, the default value `Hello, World!` will be used.

18.4 4. Retrieving Properties Programmatically

You can also retrieve properties programmatically using the `Config` interface from MicroProfile Config API.

Example:

```
import org.eclipse.microprofile.config.Config;
import org.eclipse.microprofile.config.ConfigProvider;

public class GreetingService {

    public String getGreeting() {
        Config config = ConfigProvider.getConfig();
        return config.getValue("custom.greetingProperty", String.class);
    }
}
```

18.5 5. Profiles for Environment-Specific Properties

Quarkus supports environment-specific profiles to enable different configurations for development, testing, and production.

Example:

```
# Default profile
custom.greetingProperty=Hello, Quarkus!

# Development profile
%dev.custom.greetingProperty=Hello, Developer!

# Production profile
%prod.custom.greetingProperty=Hello, User!
```

You can activate a profile by setting the `quarkus.profile` property:

```
-Dquarkus.profile=dev
```

18.6 6. Using Properties for Conditional Logic

Properties can be used to control application logic. For instance, conditional behavior based on `greetingProperty`:

Example:

```
if ("Hello, Developer!".equals(greetingProperty)) {
   // Execute development-specific logic
} else {
   // Execute default logic
}
```

18.7 Conclusion

Quarkus properties, like `greetingProperty`, offer powerful flexibility for configuring applications. By combining property injection, default values, profiles, and dynamic retrieval, developers can create adaptable and maintainable applications for various environments.