Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №3
Технології розроблення програмного забезпечення
ДІАГРАМА РОЗГОРТАННЯ. ДІАГРАМА КОМПОНЕНТІВ.
ДІАГРАМА ВЗАЄМОДІЙ ТА ПОСЛІДОВНОСТЕЙ
CI-сервер

Виконала:
Студент групи ІА-22
Вдовиченко А.Х.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Діаграма розгортання, діаграма компонентів, діаграма послідовностей
Мета: Проаналізувати тему, створити діаграму розгортання та діаграму компонентів

Теоретичні відомості

Діаграма розгортання (Deployment Diagram)

Діаграма розгортання відображає фізичну структуру системи, вказуючи, на якому обладнанні або середовищах виконуються її компоненти. Основними елементами діаграми розгортання є вузли (nodes), що представляють фізичне обладнання або програмне забезпечення, яке може містити інші елементи системи. Вузли поділяються на два типи: пристрої (наприклад, сервери, комп'ютери або мобільні пристрої) і середовища виконання (такі як веб-сервери або операційні системи). Кожен вузол може містити програмні артефакти, такі як виконувані файли, бібліотеки, бази даних або інші компоненти, необхідні для функціонування системи.

Діаграма розгортання також містить зв'язки між вузлами, що представляють собою канали обміну даними, такі як HTTP-з'єднання або інші протоколи комунікації. Ці зв'язки дозволяють зрозуміти, як вузли взаємодіють між собою і яким чином забезпечується потік інформації між компонентами системи. Діаграми розгортання часто використовуються для планування фізичного розташування компонентів на етапах розгортання та налагодження системи, що допомагає уникнути проблем з продуктивністю і забезпечити відповідність системи вимогам до мережевої інфраструктури та обробки навантажень.

Існують два типи діаграм розгортання: описові та екземплярні. Описові діаграми зображають структуру системи загалом, вказуючи на необхідне обладнання та програмні вимоги. Екземплярні ж конкретизують дані про окремі вузли (наприклад, про певний сервер), що важливо на етапі фінального розгортання системи.

Діаграма компонентів (Component Diagram)

Діаграма компонентів відображає програмні модулі системи та їхні взаємозв'язки, що дозволяє спроектувати систему як набір взаємозамінних частин. Основними елементами діаграми є компоненти (components), що представляють собою незалежні модулі системи. Кожен компонент може взаємодіяти з іншими за допомогою інтерфейсів, що визначають функціональні можливості модуля. Наприклад, в системі управління даними окремими компонентами можуть бути модулі для обробки даних, інтерфейси взаємодії з користувачем, сервіси бази даних тощо.

Діаграма компонентів використовується для моделювання структури програмного коду та опису архітектури системи. Вона дозволяє розробникам визначити, які частини програми можна створювати незалежно, а також забезпечує можливість повторного використання коду за рахунок чіткого розмежування функціональних частин. Компоненти можуть також розподілятися між різними фізичними вузлами в діаграмі розгортання, що забезпечує гнучкість у розробці розподілених або компонентно-орієнтованих систем.

Розробка діаграми компонентів є ключовою для забезпечення масштабованості і зручності обслуговування, особливо у великих проєктах. Вона дозволяє відстежувати залежності між модулями і знижувати ризик конфліктів під час інтеграції.

Діаграма послідовностей (Sequence Diagram)

Діаграма послідовностей описує порядок взаємодії між об'єктами системи у рамках певного сценарію або процесу. Вона допомагає візуалізувати послідовність обміну повідомленнями між об'єктами під час виконання операцій, що є особливо корисним для розуміння алгоритмів та процесів взаємодії в системі. На діаграмі послідовностей зображаються об'єкти системи (actor), повідомлення, які вони обмінюються, і порядок цих повідомлень. Основні елементи включають учасників (actors), повідомлення (messages), що передаються між ними, та часові лінії, які відображають порядок виконання дій.

Завдяки діаграмам послідовностей можна точно визначити основний потік виконання функцій та альтернативні сценарії, які виникають при певних умовах. Наприклад, в системі інтернет-магазину діаграма послідовностей може відображати процес оформлення замовлення: від вибору товарів і заповнення інформації про доставку до підтвердження замовлення і надсилання його на склад. Кожен з цих кроків представлений як окреме повідомлення на діаграмі, що дозволяє чітко побачити, як дані передаються між компонентами системи і в якому порядку відбуваються операції.

Використання діаграм послідовностей є важливим етапом у розробці систем, які мають складні алгоритми або потребують точного контролю над послідовністю операцій. Вони дозволяють спростити тестування і налагодження, оскільки допомагають передбачити можливі збої в логіці роботи системи та зрозуміти взаємодію компонентів на рівні процесів.

Хід роботи

Схема послідовностей

Діаграма послідовностей зображує процес створення нового проєкту та запуску збірки з використанням вебсайту, сервера, бази даних і Docker. Користувач взаємодіє з вебсайтом для створення проєкту та ініціації збірки. Сервер обробляє запити, зберігає дані в базі даних і керує процесом клонування репозиторію та запуску контейнера в Docker. Під час виконання збірки сервер надсилає логи в реальному часі на вебсайт. Діаграма також передбачає альтернативний сценарій, коли запуск контейнера завершується помилкою — у такому випадку система оновлює статус збірки як "FAILURE" і повідомляє користувача про проблему. Таким чином, діаграма ілюструє як основний, так і винятковий потоки виконання процесу.

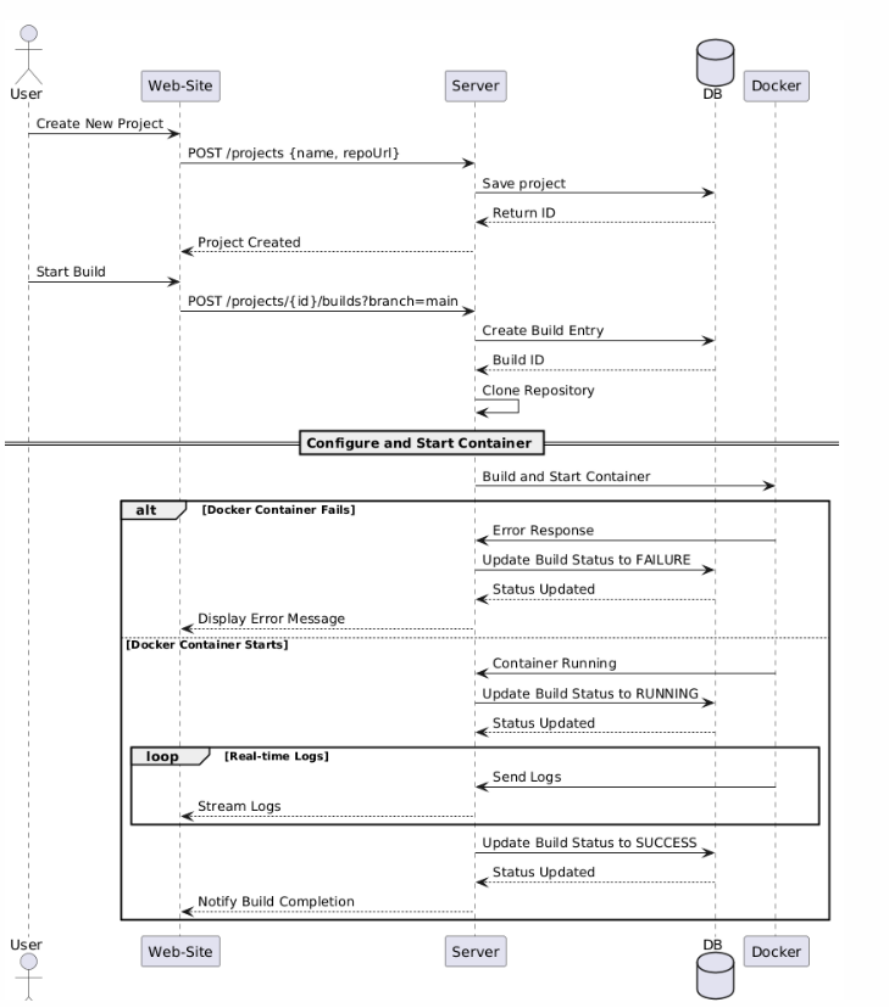

Рисунок 1.1 – Діаграма послідовностей для процесу запуску збірки

Діаграма Розгортання

Ця діаграма представляє архітектуру системи CI-сервера. Вебсайт надсилає HTTP-запити до сервера, де розташований файл CI-server.war для обробки бізнес-логіки. Сервер додатків надсилає SQL-запити до серверу баз даних (DB Server). Також у нас присутній окремий сервер, на якому у нас запускаються всі збірки на докері. (Рис 1.2.)

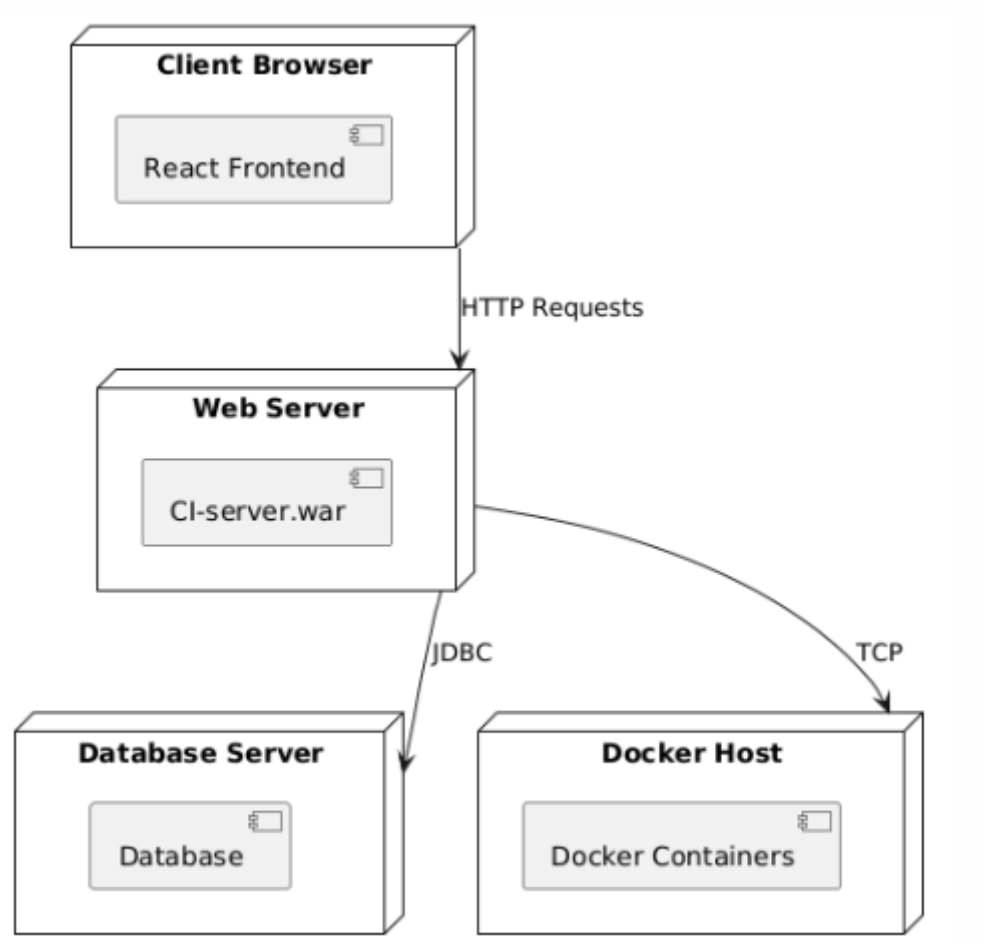

Рисунок 1.2 – Діаграма розгортання CI-серверу

Діаграма компонентів

Ця діаграма показує компоненти системи CI серверу. Клієнт взаємодіє з сервером через HTTP інтерфейс. Сам сервер в свою чергу складається з декількох компонентів. Controller – обробник запитів, що передає інформацію до сервісів, де реалізована бізнес логіка. І компоненти, що відповідають за зв’язок з базою даних та докером. Зв’язок встановлюється через JDBC та TCP інтерфейси відповідно. (Рис 1.3)

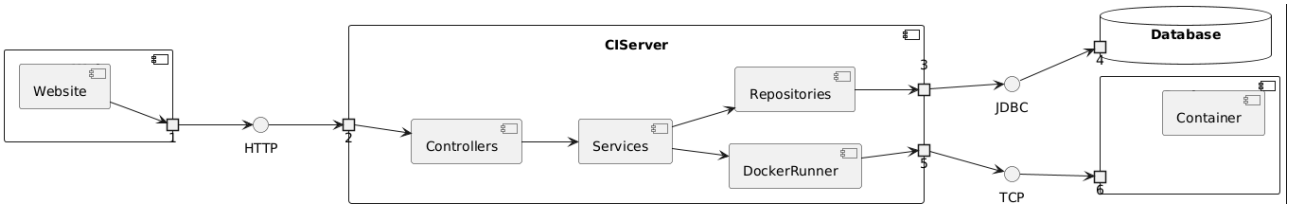

Рисунок 1.2 – Діаграма компонентів CI-серверу

Висновки

У цій лабораторній роботі було створено діаграми розгортання, компонентів і послідовностей для CI-серверу. Діаграми відображають архітектуру систему та взаємодію між її компонентами.

Вихідний код

```
package org.example.service;

import io.quarkus.websockets.next.OpenConnections;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import org.example.domain.Build;
import org.example.domain.BuildStatus;
import org.example.domain.Project;
import org.example.repository.BuildRepository;
import org.example.service.cosumers.LogFileConsumer;
import org.example.service.cosumers.WebSocketConsumer;
import org.testcontainers.containers.ContainerLaunchException;
import org.testcontainers.images.builder.ImageFromDockerfile;

import java.time.Instant;
import java.util.Map;
import java.util.Optional;
import java.util.UUID;

import static
org.example.service.utility.ServiceConstants.WORKSPACE_DIR_PATTERN;

@ApplicationScoped
public class BuildService {

    private final OpenConnections connections;
    private final BuildRepository buildRepository;
    private final GitService gitService;
    private final YamlWorkflowProcessor yamlWorkflowProcessor;
    private final DockerService dockerService;

    @Inject
    public BuildService(BuildRepository buildRepository, OpenConnections
connections,
                        GitService gitService, YamlWorkflowProcessor
yamlWorkflowProcessor, DockerService dockerService) {
        this.buildRepository = buildRepository;
        this.connections = connections;
        this.gitService = gitService;
        this.yamlWorkflowProcessor = yamlWorkflowProcessor;
        this.dockerService = dockerService;
    }

    public Optional<Build> findById(Long id) {
        return buildRepository.findByIdOptional(id);
    }

    @Transactional
    public void runProjectBuild(String id) {
        Build build =
findById(Long.parseLong(id)).orElseThrow(IllegalArgumentException::new);
        Project project = build.getProject();
        var consumer = new WebSocketConsumer(id, connections).andThen(new
LogFileConsumer(id, build.getLogFilePath()));
        var tmpProjectDirName = "%s-%s".formatted(project.getName(),
UUID.randomUUID());
        var clonedProjectDir = gitService.cloneRepository(tmpProjectDirName,
project.getRepositoryUrl(), build.getBranch());

        Map<String, Object> workflowData =
yamlWorkflowProcessor.extractWorkflowData(clonedProjectDir);
        String command =
```

```
yamlWorkflowProcessor.getCommandFromWorkflowData(workflowData);
        Map<String, String> withVariables =
yamlWorkflowProcessor.getWithVariablesFromWorkflowData(workflowData);

        ImageFromDockerfile image =
dockerService.configureImage(withVariables.get("language"),
withVariables.get("version"));
        var workingDir = WORKSPACE_DIR_PATTERN.formatted(tmpProjectDirName);
        try (var container = dockerService.configureContainer(image,
clonedProjectDir, workingDir, consumer, command)) {
            container.start();
            build.setBuildStatus(BuildStatus.SUCCESS);
            build.setEndTime(Instant.now());
            buildRepository.persist(build);;
        } catch (ContainerLaunchException launchException) {
            build.setBuildStatus(BuildStatus.FAILURE);
            build.setEndTime(Instant.now());
            buildRepository.persist(build);;
        };
    }
}
```

```
package org.example.web.controller;

import jakarta.inject.Inject;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.UriBuilder;
import jakarta.ws.rs.core.UriInfo;
import org.example.domain.Project;
import org.example.service.BuildService;
import org.example.service.GitService;
import org.example.service.ProjectService;
import org.example.web.dto.BuildDto;
import org.example.web.dto.ProjectCreateDto;
import org.example.web.dto.ProjectDto;
import org.example.web.mapper.BuildMapper;
import org.example.web.mapper.ProjectMapper;
import org.jboss.resteasy.reactive.RestResponse;

import java.util.List;

@Path("/projects")
public class ProjectResource {

    private final ProjectService projectService;
    private final ProjectMapper projectMapper;
    private final BuildMapper buildMapper;
    private final GitService gitService;

    @Inject
    public ProjectResource(ProjectService projectService, ProjectMapper
projectMapper, BuildMapper buildMapper, GitService gitService) {
        this.projectService = projectService;
        this.projectMapper = projectMapper;
        this.buildMapper = buildMapper;
        this.gitService = gitService;
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
```

```
    public RestResponse<ProjectDto> createProject(ProjectCreateDto projectDto,
@Context UriInfo uriInfo) {
        var project = projectService.save(projectMapper.toEntity(projectDto));
        var uri =
uriInfo.getAbsolutePathBuilder().path(String.valueOf(project.getId())).build();
        return RestResponse.seeOther(uri);
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public RestResponse<List<ProjectDto>> getProjects() {
        var list =
projectService.findAll().stream().map(projectMapper::toDto).toList();
        return RestResponse.ok(list);
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public RestResponse<ProjectDto> findById(@PathParam("id") Long id) {
        var project = projectService.findById(id);
        return project
                .map(projectMapper::toDto)
                .map(RestResponse::ok)
                .orElse(RestResponse.notFound());
    }

    @GET
    @Path("{id}/branches")
    @Produces(MediaType.APPLICATION_JSON)
    public RestResponse<List<String>> getAllRemoteBranches(@PathParam("id") Long
id) {
        var project = projectService.findById(id);
        return project
                .map(Project::getRepositoryUrl)
                .map(gitService::getAllRemoteBranches)
                .map(RestResponse::ok)
                .orElse(RestResponse.notFound());
    }



    @GET
    @Path("{id}/builds")
    @Produces(MediaType.APPLICATION_JSON)
    public RestResponse<List<BuildDto>> findAllBuilds(@PathParam("id") Long id)
{
        var builds = projectService.getAllBuilds(id);
        return builds
                .map(list -> list.stream().map(buildMapper::toDto).toList())
                .map(RestResponse::ok)
                .orElse(RestResponse.notFound());
    }

    @POST
    @Path("{id}/builds")
    @Produces(MediaType.APPLICATION_JSON)
    public RestResponse<BuildDto> createNewBuild(@PathParam("id") Long id,
@QueryParam("branch") String branch,
                                                 @Context UriInfo uriInfo) {
        var build = projectService.createBuild(id, branch);
        System.out.println(uriInfo.getBaseUri());
        var uri =
uriInfo.getBaseUriBuilder().path(BuildResource.class).path(String.valueOf(build.
getId())).build();
```

```
            return RestResponse.seeOther(uri);
        }

    }
```