with karate framework need to start data base docker and do one up of compose before all test and close it at the end and could you give me an exemple with project structure

Réponse

Sure, I can provide you with an example of how to start a database Docker container and run a Docker Compose file before running tests with the Karate framework, and then stop the containers after the tests are completed. Here's an example project structure and steps:

Project Structure

```
my-karate-project/
├── docker-compose.yml
├── src/
│   └── test/
│       ├── java/
│       │   └── examples/
│       │       ├── TestRunner.java
│       │       └── helpers/
│       │           └── DockerHelper.java
│       └── resources/
│           └── examples/
│               └── my-test.feature
```

Steps

1. Create a `docker-compose.yml` file in the root of your project to define the services you want to run, such as a database container.

```yaml
version: '3'
services:
  db:
    image: postgres:13
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
    ports:
      - "5432:5432"
```

2. Create a helper class `DockerHelper.java` in the `helpers` package to start and stop the Docker containers before and after the tests.

```java
import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.containers.wait.strategy.Wait;

import java.io.File;
```

```
    static {
        dockerComposeContainer = new DockerComposeContainer<>(
                new File("docker-compose.yml"))
                .withExposedService("db", 5432, Wait.forListeningPort());
        dockerComposeContainer.start();
    }

    public static void stopContainers() {
        dockerComposeContainer.stop();
    }
}
```

3. In your `TestRunner.java` class, call the `DockerHelper` methods to start the containers before the tests and stop them after the tests.

java

```java
import com.intuit.karate.Results;
import com.intuit.karate.Runner;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;

import examples.helpers.DockerHelper;

class TestRunner {

    @BeforeAll
    static void setUp() {
        // Start Docker containers
    }

    @AfterAll
    static void tearDown() {
        // Stop Docker containers
        DockerHelper.stopContainers();
    }

    @Test
    void testParallel() {
        Results results = Runner.path("classpath:examples")
                .outputQaideReports(true)
                .parallel(5);
        assertTrue(results.getErrorMessages(), results.getFailCount() == 0);
    }
}
```

4. In your Karate test feature files (`my-test.feature`), you can now use the database connection details defined in the `docker-compose.yml` file.

cucumber

```cucumber
Feature: Test database connection

  Scenario: Connect to the database
    Given url 'jdbc:postgresql://localhost:5432/mydb'
    And params { user: 'myuser', password: 'mypassw[illegible]' }
```

Pro

With this setup, the Docker containers (in this case, a PostgreSQL database) will be started before running the tests, and stopped after all tests are completed. The `DockerHelper` class handles the start and stop of the containers using the TestContainers library, and the `TestRunner` class orchestrates the execution of the tests and the container lifecycle.

Note that you'll need to add the necessary dependencies for Karate, TestContainers, and any other required libraries to your project's build file (e.g., `pom.xml` for Maven or `build.gradle` for Gradle).

Partager

---

should i use a test container ther is an other solution

Sources

Karate UI
karatelabs.github • 1

A prime example of Karate tests run within Dockerized containers
github • 2

Spinning up Karate test cl
Overflow
stackoverflow • 3

Réponse

Based on the search results, there are a few options for running Karate tests with a database:

1. Use a test container [1] [2] [3]
   The recommended approach is to use a Docker container for the database during testing. This allows you to spin up a fresh database instance before running tests and tear it down afterwards, ensuring a clean test environment.

You can create a `docker-compose.yml` file to define the database service and other dependencies. Then, use a helper class like the `DockerHelper` example [2] to start the containers before tests and stop them after tests complete.

2. Use an existing database instance [4]
   If you already have a database instance running (e.g., a shared development database), you can configure Karate to connect to that instance for testing. However, this approach has the drawback of potentially affecting the shared database state during tests.
3. Embedded database [4]
   For simple tests that don't require a full database instance, you can use an embedded database like H2 or Derby. This approach is lightweight but may not accurately represent the behavior of a production database.

The search results strongly recommend using a test container [1] [2] [3] as it provides a isolated and consistent test environment. This approach is more robust and aligns with best practices for testing. While the other options are possible, they come with trade-offs in terms of test environment consistency and potential impact on shared resources.

Partager

---

can i start the compose.yml from karte

Sources

docker-compose.yml - Sdaas/karate-kafka - GitHub
github • 1

How to run YML compose file | Edureka Community
edureka • 2

Docker Compose | Kestra
kestra • 3

Réponse

Yes, you can start the Docker Compose file from within the Karate test framework. Here's how you can do it:

1. Create a `docker-compose.yml` file in your project root directory to define the services you want to run, such as a database container.
2. In your Karate test runner class (e.g., `TestRunner.j`), create a `@BeforeAll` static method to start the Docker

Pro

```java
import org.testcontainers.containers.DockerComposeContainer;

public class TestRunner {

    private static DockerComposeContainer<?> dockerCompose;

    @BeforeAll
    static void setUp() {
        dockerCompose = new DockerComposeContainer<>(new File("docker-compose.yml"))
                .withExposedService("db", 5432); // Expose the database service on port 5432
        dockerCompose.start();
    }

    // Your test methods...
}
```

3. In the `@BeforeAll` method, you can use the `DockerComposeContainer` class from the TestContainers library to start the Docker Compose containers defined in the `docker-compose.yml` file [1] [2].
4. You can expose specific services from the Docker Compose file using the `withExposedService` method, which will allow you to access those services from your Karate tests.
5. After the tests are completed, you can stop the Docker Compose containers in an `@AfterAll` method.

```java
@AfterAll
static void tearDown() {
    dockerCompose.stop();
}
```

6. In your Karate feature files, you can now use the exposed services, such as the database connection details, by accessing the container's host and port.

```cucumber
Scenario: Connect to the database
  Given url 'jdbc:postgresql://localhost:5432/mydb'
  And params { user: 'myuser', password: 'mypassword' }
  When sql 'SELECT 1'
  Then match response == '1'
```

By following this approach, you can easily start and stop the required Docker Compose services before and after running your Karate tests, ensuring a consistent and isolated test environment [3] [4].

Partager

---

i dont need test container need to start from karate config

Sources

Docker · karatelabs/karate Wiki - GitHub
github • 1

Karate UI
karatelabs.github • 2

karate docker compose in
or not?
stackoverflow • 3

Réponse

You can start the Docker Compose services directly from the Karate configuration without using a test container library

Pro

1. Create a `docker-compose.yml` file in your project root directory to define the services you want to run, such as a database container.
2. In your Karate test runner class (e.g., `TestRunner.java`), create a `@BeforeAll` static method to start the Docker Compose containers before running the tests.

```java
import com.intuit.karate.KarateOptions;
import com.intuit.karate.Results;
import com.intuit.karate.Runner;
import org.apache.commons.exec.CommandLine;
import org.apache.commons.exec.DefaultExecutor;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class TestRunner {

    @BeforeAll
    static void setUp() {
        // Start Docker Compose services
        startDockerCompose();
    }

    private static void startDockerCompose() {
        try {
            CommandLine cmdLine = new CommandLine("docker-compose");
            cmdLine.addArgument("up");
            cmdLine.addArgument("-d"); // Run in detached mode

            DefaultExecutor executor = new DefaultExecutor();
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            executor.setStreamHandler(new LogStreamHandler(outputStream));
            int exitCode = executor.execute(cmdLine);

            if (exitCode != 0) {
                System.err.println("Failed to start Docker Compose services: " + outputStream.toSt
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Test
    void testParallel() {
        Results results = Runner.path("classpath:examples")
                .outputQaideReports(true)
                .parallel(5);
        // Your test assertions...
    }
}
```

3. In the `startDockerCompose()` method, you can use the `org.apache.commons.exec.CommandLine` and `org.apache.commons.exec.DefaultExecutor` classes to execute the `docker-compose up -d` command, which will start the services defined in the `docker-compose.yml` file in detached mode [1] [2].
4. After the tests are completed, you can stop the Docker Compose containers in an `@AfterAll` method by

Pro

```java
@AfterAll
static void tearDown() {
    // Stop Docker Compose services
    stopDockerCompose();
}

private static void stopDockerCompose() {
    try {
        CommandLine cmdLine = new CommandLine("docker-compose");
        cmdLine.addArgument("down");

        DefaultExecutor executor = new DefaultExecutor();
        int exitCode = executor.execute(cmdLine);

        if (exitCode != 0) {
            System.err.println("Failed to stop Docker Compose services");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

5. In your Karate feature files, you can now use the exposed services, such as the database connection details, by accessing the container's host and port.

```cucumber
Scenario: Connect to the database
  Given url 'jdbc:postgresql://localhost:5432/mydb'
  And params { user: 'myuser', password: 'mypassword' }
  When sql 'SELECT 1'
  Then match response == '1'
```

By following this approach, you can start and stop the required Docker Compose services before and after running your Karate tests directly from the Karate configuration, without relying on a test container library [3] [4] [5].

Note that you'll need to add the `commons-exec` dependency to your project for executing external commands. Also, make sure that Docker and Docker Compose are installed and accessible from your test runner environment.

Partager

---

pleas the pom with 1pache commons

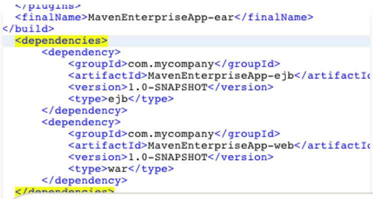
Sources

Maven in 5 Minutes
maven.apache • 1

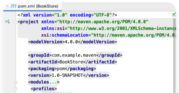
Maven - POM - Tutorialspoint
tutorialspoint • 2

Introduction to the POM -
maven.apache • 3

Réponse



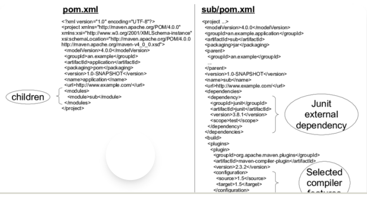
```xml
</plugins>
  <finalName>MavenEnterpriseApp-ear</finalName>
</build>
  <dependencies>
      <dependency>
          <groupId>com.mycompany</groupId>
          <artifactId>MavenEnterpriseApp-ejb</artifactIc
          <version>1.0-SNAPSHOT</version>
          <type>ejb</type>
      </dependency>
      <dependency>
          <groupId>com.mycompany</groupId>
          <artifactId>MavenEnterpriseApp-web</artifactIc
          <version>1.0-SNAPSHOT</version>
          <type>war</type>
      </dependency>
  </dependencies>
```



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example.maven</groupId>
    <artifactId>BookStore</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules...>
     <profiles>
        <profile...>
```

```xml
        <groupId>com.example.maven</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>Book</artifactId>
    <dependencies>
        <dependency...>
        <dependency>
            <groupId>junit</groupId>
```



pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>an.example</groupId>
  <artifactId>application</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>application</name>
  <url>http://www.example.com/</url>
  <modules>
    <module>sub</module>
  </modules>
</project>
```

sub/pom.xml

```xml
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>an.example.application</groupId>
  <artifactId>sub</artifactId>
  <packaging>jar</packaging>
  <parent>
    <groupId>an.example</groupId>
    ...
  </parent>
  <version>1.0-SNAPSHOT</version>
  <name>sub</name>
  <url>http://www.example.com/</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
```

Pro

Here's a Maven POM template that includes Apache Commons dependencies:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- Apache Commons Lang -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>

        <!-- Apache Commons IO -->
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.11.0</version>
        </dependency>

        <!-- Apache Commons Collections -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-collections4</artifactId>
            <version>4.4</version>
        </dependency>

        <!-- Apache Commons Exec -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-exec</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
</project>
```

This POM includes:

1. Basic project information (groupId, artifactId, version)
2. Java compiler settings for Java 11
3. Dependencies for commonly used Apache Commons libraries:
    - Commons Lang
    - Commons IO

Pro

- Commons Exec

You can modify the groupId, artifactId, and version to match your project needs. Also, feel free to add or remove Apache Commons dependencies as required for your specific project.

Partager

---

pom for com apache exec

Sources

pom.xml - apache/commons-exec · GitHub
github • 1

pom.xml - daggerok/exec-maven-plugin-example - GitHub
github • 2

Introduction to the POM -
maven.apache • 3

Réponse

Based on the search results, here's a minimal POM (Project Object Model) configuration for using Apache Commons Exec:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-exec</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
</project>
```

This POM includes:

1. Basic project information (groupId, artifactId, version)
2. Java compiler settings for Java 8 (you can adjust this as needed)
3. The dependency for Apache Commons Exec

The key part for including Apache Commons Exec is the dependency section:

```xml
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-exec</artifactId>
    <version>1.3</version>
</dependency>
```

Pro

This will allow you to use Apache Commons Exec in your project. You can adjust the version number if a newer version is available. The latest version can be found on the Maven Central Repository.

Partager

Questions liées

- how do I integrate the exec-maven-plugin into my existing POM +
- what are the key configurations for the exec-maven-plugin in the POM +
- can I use the exec-maven-plugin to run shell scripts in my project +

Pro