wiremock

1. Run WireMock as a Standalone JAR

This is the simplest way to get started:

✅ Steps:

1. Download the standalone JAR from WireMock's official site.
2. Run it using the command line:

bash

```
java -jar wiremock-standalone-3.13.1.jar --port 8080
```

3. Place your stub mappings in a `mappings/` folder in the same directory.

🧪 Example Stub (`mappings/mock.json`):

json

```
{
"request": {
"method": "GET",
"url": "/api/mock"
},
"response": {
"status": 200,
"body": "{ \"message\": \"Hello from WireMock\" }",
"headers": {
"Content-Type": "application/json"
}
}
}
```

🐋 2. Run WireMock with Docker

If you prefer containers:

bash

```
docker run -it --rm -p 8080:8080 \
-v $(pwd)/mappings:/home/wiremock/mappings \
wiremock/wiremock
```

This mounts your local stub files and exposes WireMock on port 8080.
More details in the Docker guide.

🧰 3. Use WireMock CLI (Advanced)

For enterprise or cloud-backed setups, WireMock offers a CLI tool to pull and run mock APIs locally:

- Pull mock APIs:
  bash
- Run locally:
  bash

See full instructions in the WireMock CLI documentation.

You can install and run **WireMock** on macOS using Homebrew in just a few steps. Here's the complete guide:

🍺 Step 1: Install WireMock via Homebrew

bash

```
brew install wiremock-standalone
```

This installs the standalone JAR and makes it available via the `wiremock` command.

🚀 Step 2: Run WireMock Locally

🔷 Basic Run (Default Port 8080)

bash

```
wiremock
```

This starts WireMock on http://localhost:8080.

🔷 Run on Custom Port

bash

```
wiremock --port 8081
```

🔷 Run with Stub Mappings

If you have mappings and response files:

bash

```
wiremock --root-dir ./wiremock
```

Your folder should look like:

Code

```
wiremock/
├── __files/
│ └── users.json
└── mappings/
└── get-user.json
```

🔷 Example Mapping (`mappings/get-user.json`)

json

```
{
"request": {
"method": "GET",
"url": "/api/users/42"
},
"response": {
"status": 200,
"bodyFileName": "users.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

🔷 Example Response (`__files/users.json`)

json

```
{
"id": 42,
"name": "Alice",
"email": "alice@example.com"
}
```

🧪 Test It

bash

```
curl http://localhost:8080/api/users/42
```

You should get the mocked JSON response.

🧠 Bonus Tips

- Use `--verbose` to see incoming requests
- Use `--global-response-templating` for dynamic stubs
- Use `--extensions` to load custom Java plugins

🧰 1. Setup: Dependencies

Add these to your `pom.xml`:

xml

```
<!-- REST client -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- WireMock for testing -->
<dependency>
<groupId>com.github.tomakehurst</groupId>
<artifactId>wiremock-jre8</artifactId>
<version>2.35.0</version>
<scope>test</scope>
</dependency>
```

🧪 2. WireMock Test with REST Client

✅ Controller (calls external API):

java

```
@RestController
@RequestMapping("/proxy")
public class ProxyController {

private final RestTemplate restTemplate;

public ProxyController(RestTemplateBuilder builder) {
this.restTemplate = builder.build();
}
```

```
@GetMapping("/data")
public ResponseEntity<String> getData() {
String externalUrl = "http://localhost:9999/external/data";
String response = restTemplate.getForObject(externalUrl,
String.class);
return ResponseEntity.ok(response);
}
}
```

✅ Test Class:

java

```
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 9999) // WireMock runs on port 9999
class ProxyControllerTest {

@Autowired
private TestRestTemplate restTemplate;

@BeforeEach
void setupStub() {
WireMock.stubFor(get(urlEqualTo("/external/data"))
.willReturn(aResponse()
.withHeader("Content-Type", "application/json")
.withBody("{ \"status\": \"success\", \"source\": \"wiremock\" }")));
}

@Test
void testProxyEndpoint() {
ResponseEntity<String> response =
restTemplate.getForEntity("/proxy/data", String.class);
Assertions.assertTrue(response.getBody().contains("wiremock"));
}
}
```

🧪 Result

When you hit `/proxy/data`, your Spring Boot app makes a REST call to the mocked WireMock endpoint (`/external/data`), and returns the stubbed JSON response.

To call a WireMock API from a Spring Boot REST client and deserialize the response from a JSON file into a Java object, follow this complete example:

✅ 1. Setup: WireMock JSON File

Place your mock JSON response in:

Code

```
src/test/resources/__files/mock-response.json
```

Example `mock-response.json`:

json

```
{
"id": 101,
"name": "WireMock User",
"email": "mock@wiremock.io"
}
```

🧪 2. WireMock Stub Configuration

In your test class:

java

```
@BeforeEach
void setupStub() {
WireMock.stubFor(get(urlEqualTo("/api/user"))
.willReturn(aResponse()
.withHeader("Content-Type", "application/json")
.withBodyFile("mock-response.json")));
}
```

✅ Make sure the file is inside `__files/` — WireMock looks there by default.

🧰 3. Java Model Class

java

```
public class User {
private int id;
private String name;
private String email;
```

```
// Getters and setters
}
```

🌐 4. Spring Boot REST Client Call

java

```
@RestController
@RequestMapping("/client")
public class ClientController {

private final RestTemplate restTemplate;

public ClientController(RestTemplateBuilder builder) {
this.restTemplate = builder.build();
}

@GetMapping("/user")
public ResponseEntity<User> getUser() {
String wireMockUrl = "http://localhost:9999/api/user";
User user = restTemplate.getForObject(wireMockUrl, User.class);
return ResponseEntity.ok(user);
}
}
```

🧪 5. Test the Endpoint

java

```
@Test
void testClientUserCall() {
ResponseEntity<User> response =
restTemplate.getForEntity("/client/user", User.class);
Assertions.assertEquals("WireMock User",
response.getBody().getName());
}
```

📁 Project Structure for Docker

Code

```
wiremock-docker/
├── mappings/
│ └── user-mapping.json
├── __files/
│ └── user-response.json
```

- `mappings/` : Contains stub mapping definitions.
- `__files/` : Contains response bodies referenced by `bodyFileName` .

📄 Sample JSON Files

✅ `mappings/user-mapping.json`

json

```
{
"request": {
"method": "GET",
"url": "/api/user"
},
"response": {
"status": 200,
"bodyFileName": "user-response.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

✅ `__files/user-response.json`

json

```
{
"id": 101,
"name": "WireMock User",
"email": "mock@wiremock.io"
}
```

🐋 Docker Command to Run WireMock

bash

```
docker run -it --rm -p 8080:8080 \
-v $(pwd)/mappings:/home/wiremock/mappings \
-v $(pwd)/__files:/home/wiremock/__files \
wiremock/wiremock
```

This mounts your local `mappings/` and `__files/` folders into the container, allowing WireMock to serve the stubbed `/api/user` endpoint.

📁 Project Structure

Code

```
wiremock-docker/
├── mappings/
│ ├── create-user.json
│ ├── get-user.json
│ ├── update-user.json
│ └── delete-user.json
├── __files/
│ ├── user-created.json
│ ├── user-details.json
│ ├── user-updated.json
│ └── user-deleted.json
```

📄 Mapping Files

✅ `create-user.json` (POST `/api/users`)

json

```
{
"request": {
"method": "POST",
"url": "/api/users"
},
"response": {
"status": 201,
"bodyFileName": "user-created.json",
"headers": {
"Content-Type": "application/json"
}
}
```

```
}
```

✅ `get-user.json` (GET `/api/users/1`)

json

```
{
"request": {
"method": "GET",
"url": "/api/users/1"
},
"response": {
"status": 200,
"bodyFileName": "user-details.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

✅ `update-user.json` (PUT `/api/users/1`)

json

```
{
"request": {
"method": "PUT",
"url": "/api/users/1"
},
"response": {
"status": 200,
"bodyFileName": "user-updated.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

✅ `delete-user.json` (DELETE `/api/users/1`)

json

```
{
"request": {
"method": "DELETE",
"url": "/api/users/1"
```

```
},
"response": {
"status": 204
}
}
```

📄 Response Files

user-created.json

json

```
{
"id": 1,
"name": "New User",
"email": "new.user@example.com"
}
```

user-details.json

json

```
{
"id": 1,
"name": "Existing User",
"email": "existing.user@example.com"
}
```

user-updated.json

json

```
{
"id": 1,
"name": "Updated User",
"email": "updated.user@example.com"
}
```

user-deleted.json (optional for 204)

json

```
{
"message": "User deleted successfully"
}
```

📁 Project Structure

Code

```
wiremock-docker/
├── mappings/
│ ├── get-all-users.json
│ └── get-user-by-id.json
├── __files/
│ ├── all-users.json
│ └── user-1.json
```

📄 Mapping Files

✅ `get-all-users.json`

json

```
{
"request": {
"method": "GET",
"url": "/api/users"
},
"response": {
"status": 200,
"bodyFileName": "all-users.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

✅ `get-user-by-id.json`

To simulate dynamic ID matching, use a regex pattern:

json

```
{
"request": {
"method": "GET",
"urlPattern": "/api/users/1"
},
"response": {
"status": 200,
"bodyFileName": "user-1.json",
```

```
"headers": {
"Content-Type": "application/json"
}
}
}
```

You can duplicate this file for other IDs or use `urlPattern` with a regex like `/api/users/[0-9]+` and add logic via response templating if needed.

📄 Response Files

`all-users.json`

json

```
[
{
"id": 1,
"name": "Alice",
"email": "alice@example.com"
},
{
"id": 2,
"name": "Bob",
"email": "bob@example.com"
}
]
```

`user-1.json`

json

```
{
"id": 1,
"name": "Alice",
"email": "alice@example.com"
}
```

T

If you want to mock multiple user IDs like `/api/users/1`, `/api/users/2`, etc., you have two options:

Option 1: Duplicate Mapping Files

You create separate stub files for each user ID:

Code

```
mappings/
├── get-user-1.json
├── get-user-2.json
...
__files/
├── user-1.json
├── user-2.json
...
```

Each mapping hardcodes the URL and response. This works, but it's repetitive and hard to maintain.

🧪 Option 2: Use `urlPattern` + Response Templating

WireMock lets you match URLs using regex and dynamically generate responses using **response templating**.

✅ Mapping File: `get-user-by-id.json`

json

```
{
"request": {
"method": "GET",
"urlPattern": "/api/users/([0-9]+)"
},
"response": {
"status": 200,
"headers": {
"Content-Type": "application/json"
},
"body": "{ \"id\": {{request.path.[2]}}, \"name\": \"User
{{request.path.[2]}}\", \"email\": \"user{{request.path.
[2]}}@example.com\" }",
"transformers": ["response-template"]
}
}
```

🔍 Explanation:

- `urlPattern` : Matches any `/api/users/{id}` where `{id}` is a number.
- `{{request.path.[2]}}` : Extracts the ID from the path (`/api/users/1` → `1`).
- `transformers` : Enables response templating.

WireMock indexes path segments starting from 0. So `/api/users/1` → `["", "api", "users", "1"]`, and `[2]` gives `"users"` while `[3]` gives `"1"` — adjust accordingly.

🧰 Result

Calling `/api/users/5` will return:

json

```
{
"id": 5,
"name": "User 5",
"email": "user5@example.com"
}
```

No need to create separate files for each user!

🛠️ Enable Response Templating in Docker

To use response templating with Docker, add the `--global-response-templating` flag:

bash

```
docker run -it --rm -p 8080:8080 \
-v $(pwd)/mappings:/home/wiremock/mappings \
wiremock/wiremock --global-response-templating
```

🧱 Project Structure

Code

```
wiremock-docker/
├── mappings/
│ └── get-user-by-id.json
├── __files/
│ └── (optional if using inline body)
├── client/
│ └── TestClient.java
```

📄 WireMock Mapping: `get-user-by-id.json`

json

```
{
```

```
"request": {
"method": "GET",
"urlPattern": "/api/users/([0-9]+)"
},
"response": {
"status": 200,
"headers": {
"Content-Type": "application/json"
},
"body": "{ \"id\": {{request.path.[2]}}, \"name\": \"User
{{request.path.[2]}}\", \"email\": \"user{{request.path.
[2]}}@example.com\" }",
"transformers": ["response-template"]
}
}
```

This uses `urlPattern` to match any numeric ID and `{{request.path.[2]}}` to extract the ID dynamically.

🐋 Docker Command to Run WireMock

bash

```
docker run -it --rm -p 8080:8080 \
-v $(pwd)/mappings:/home/wiremock/mappings \
wiremock/wiremock --global-response-templating
```

💻 Java Test Client

java

```
import org.springframework.web.client.RestTemplate;

public class TestClient {
public static void main(String[] args) {
RestTemplate restTemplate = new RestTemplate();
String url = "http://localhost:8080/api/users/42";
String response = restTemplate.getForObject(url, String.class);
System.out.println("Response: " + response);
}
}
```

🧪 Output

json

```
{
"id": 42,
"name": "User 42",
"email": "user42@example.com"
}
```

Perfect—let's build out a **full CRUD WireMock setup** with:

- ✅ Create (POST)
- 🔍 Read (GET with query param filtering)
- ✏️ Update (PUT)
- ❌ Delete (DELETE)
- ⚠️ Simulated error responses (e.g. 404, 500)

All responses will use **response templating**, and we'll keep it Docker-friendly.

📦 Updated Project Structure

Code

```
wiremock-docker/
├── mappings/
│ ├── get-user-by-id.json
│ ├── get-users-by-query.json
│ ├── post-create-user.json
│ ├── put-update-user.json
│ ├── delete-user.json
│ └── error-user-not-found.json
```

📄 CRUD Stubs

🟢 POST `/api/users`

json

```
{
"request": {
"method": "POST",
"url": "/api/users"
},
```

```
"response": {
"status": 201,
"headers": {
"Content-Type": "application/json"
},
"body": "{ \"id\": 101, \"name\": \"{{jsonPath request.body
'$.name'}}\", \"email\": \"{{jsonPath request.body '$.email'}}\" }",
"transformers": ["response-template"]
}
}
```

🔵 GET `/api/users/{id}`

Same as before, using `urlPattern` and `request.path.[2]`.

🔍 GET `/api/users?name=John`

json

```
{
"request": {
"method": "GET",
"urlPath": "/api/users",
"queryParameters": {
"name": {
"matches": "John"
}
}
},
"response": {
"status": 200,
"body": "{ \"id\": 42, \"name\": \"John\", \"email\":
\"john@example.com\" }"
}
}
```

✏️ PUT `/api/users/{id}`

json

```
{
"request": {
"method": "PUT",
"urlPattern": "/api/users/([0-9]+)"
```

```
},
"response": {
"status": 200,
"body": "{ \"id\": {{request.path.[2]}}, \"updated\": true }",
"transformers": ["response-template"]
}
}
```

❌ DELETE `/api/users/{id}`

json

```
{
"request": {
"method": "DELETE",
"urlPattern": "/api/users/([0-9]+)"
},
"response": {
"status": 204
}
}
```

⚠ Simulated Error Responses

404 Not Found

json

```
{
"request": {
"method": "GET",
"url": "/api/users/999"
},
"response": {
"status": 404,
"body": "{ \"error\": \"User not found\" }"
}
}
```

500 Internal Server Error

json

```
{
```

```
"request": {
"method": "GET",
"url": "/api/users/fail"
},
"response": {
"status": 500,
"body": "{ \"error\": \"Internal server error\" }"
}
}
```

🐋 Docker Run (with templating)

bash

```
docker run -it --rm -p 8080:8080 \
-v $(pwd)/mappings:/home/wiremock/mappings \
wiremock/wiremock --global-response-templating
```

Awesome. Here's a complete **Java test suite** using **Spring Boot**, **Gradle**, and **RestTemplate** to interact with your WireMock-powered CRUD API. It includes:

- ✅ Unit tests for each endpoint
- 🧪 Error response validation
- 🛠️ Configurable WireMock base URL

🧰 Project Setup

`build.gradle`

groovy

```
plugins {
id 'java'
id 'org.springframework.boot' version '3.1.0'
}

dependencies {
implementation 'org.springframework.boot:spring-boot-starter-web'
testImplementation 'org.springframework.boot:spring-boot-starter-
test'
}
```

📁 Directory Structure

Code

```
src/
├── main/
│ └── java/com/example/client/
│ └── UserClient.java
└── test/
└── java/com/example/client/
└── UserClientTests.java
```

💻 UserClient.java

java

```
package com.example.client;

import org.springframework.http.*;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

public class UserClient {
private final RestTemplate restTemplate = new RestTemplate();
private final String baseUrl;

public UserClient(String baseUrl) {
this.baseUrl = baseUrl;
}

public String getUserById(int id) {
return restTemplate.getForObject(baseUrl + "/api/users/" + id,
String.class);
}

public String getUserByQuery(String name) {
return restTemplate.getForObject(baseUrl + "/api/users?name=" + name,
String.class);
}

public String createUser(String name, String email) {
```

```
Map<String, String> payload = Map.of("name", name, "email", email);
return restTemplate.postForObject(baseUrl + "/api/users", payload,
String.class);
}

public String updateUser(int id) {
HttpEntity<Void> entity = new HttpEntity<>(null);
ResponseEntity<String> response = restTemplate.exchange(
baseUrl + "/api/users/" + id, HttpMethod.PUT, entity, String.class);
return response.getBody();
}

public ResponseEntity<Void> deleteUser(int id) {
return restTemplate.exchange(baseUrl + "/api/users/" + id,
HttpMethod.DELETE, null, Void.class);
}

public ResponseEntity<String> getUserWithError(int id) {
return restTemplate.exchange(baseUrl + "/api/users/" + id,
HttpMethod.GET, null, String.class);
}
}
```

🧪 UserClientTests.java

java

```
package com.example.client;

import org.junit.jupiter.api.*;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;

class UserClientTests {
private final UserClient client = new
UserClient("http://localhost:8080");

@Test
void testGetUserById() {
String response = client.getUserById(42);
assertThat(response).contains("\"id\": 42");
}
```

```
@Test
void testGetUserByQuery() {
String response = client.getUserByQuery("John");
assertThat(response).contains("\"name\": \"John\"");
}

@Test
void testCreateUser() {
String response = client.createUser("Alice", "alice@example.com");
assertThat(response).contains("\"name\": \"Alice\"");
}

@Test
void testUpdateUser() {
String response = client.updateUser(42);
assertThat(response).contains("\"updated\": true");
}

@Test
void testDeleteUser() {
ResponseEntity<Void> response = client.deleteUser(42);
assertThat(response.getStatusCodeValue()).isEqualTo(204);
}

@Test
void testUserNotFound() {
ResponseEntity<String> response = client.getUserWithError(999);
assertThat(response.getStatusCodeValue()).isEqualTo(404);
assertThat(response.getBody()).contains("User not found");
}

@Test
void testInternalServerError() {
ResponseEntity<String> response = client.getUserWithError("fail");
assertThat(response.getStatusCodeValue()).isEqualTo(500);
assertThat(response.getBody()).contains("Internal server error");
}
}
```

🧩 Use Case: Mocking `UserClient` in a Service

Let's say you have a `UserService` that depends on `UserClient` . We'll:

- Create `UserService`
- Write unit tests using `@Mock` and `@InjectMocks`

📁 Updated Structure

Code

```
src/
├── main/java/com/example/service/
│ └── UserService.java
└── test/java/com/example/service/
└── UserServiceTests.java
```

💼 UserService.java

java

```
package com.example.service;

import com.example.client.UserClient;

public class UserService {
private final UserClient userClient;

public UserService(UserClient userClient) {
this.userClient = userClient;
}

public String fetchUserName(int id) {
String json = userClient.getUserById(id);
// naive parsing for demo purposes
return json.contains("\"name\":") ? json.split("\"name\":")
[1].split("\"")[1] : null;
}

public boolean deleteUser(int id) {
return userClient.deleteUser(id).getStatusCodeValue() == 204;
}
}
```

🧪 UserServiceTests.java

java

```
package com.example.service;

import com.example.client.UserClient;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.http.ResponseEntity;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

class UserServiceTests {

@Mock
UserClient userClient;

@InjectMocks
UserService userService;

public UserServiceTests() {
MockitoAnnotations.openMocks(this);
}

@Test
void testFetchUserName() {
when(userClient.getUserById(42)).thenReturn("{ \"id\": 42, \"name\":
\"Alice\" }");

String name = userService.fetchUserName(42);
assertThat(name).isEqualTo("Alice");
}

@Test
void testDeleteUser() {
when(userClient.deleteUser(42)).thenReturn(ResponseEntity.noContent()
.build());

boolean result = userService.deleteUser(42);
assertThat(result).isTrue();
```

```
}
}
```

🧪 Optional: Use `@ExtendWith(MockitoExtension.class)` (JUnit 5)

If you prefer annotation-based setup:

java

```
@ExtendWith(MockitoExtension.class)
class UserServiceTests {
@Mock UserClient userClient;
@InjectMocks UserService userService;
...
}
```

1. ✅ **Jackson-based JSON parsing** for clean, robust data handling
2. 📊 **JaCoCo integration** to generate test coverage reports

🧠 1. Replace Naive Parsing with Jackson

📦 Add Jackson Dependency (if not already included)

groovy

```
dependencies {
implementation 'com.fasterxml.jackson.core:jackson-databind'
}
```

👷 Define a `User` DTO

java

```
package com.example.model;

public class User {
public int id;
public String name;
public String email;

// Getters and setters (or use Lombok if preferred)
}
```

🔧 Update UserService.java

java

```
package com.example.service;

import com.example.client.UserClient;
import com.example.model.User;
import com.fasterxml.jackson.databind.ObjectMapper;

public class UserService {
private final UserClient userClient;
private final ObjectMapper objectMapper = new ObjectMapper();

public UserService(UserClient userClient) {
this.userClient = userClient;
}

public String fetchUserName(int id) {
try {
String json = userClient.getUserById(id);
User user = objectMapper.readValue(json, User.class);
return user.name;
} catch (Exception e) {
return null;
}
}

public boolean deleteUser(int id) {
return userClient.deleteUser(id).getStatusCodeValue() == 204;
}
}
```

✅ Update UserServiceTests.java

No changes needed to the test logic—just ensure the mocked JSON matches the `User` structure.

📊 2. Add JaCoCo for Coverage Reports

🔧 `build.gradle`

groovy

```
plugins {
id 'jacoco'
}

jacoco {
toolVersion = "0.8.10"
}

jacocoTestReport {
dependsOn test
reports {
xml.required = true
html.required = true
}
}
```

📦 Run Tests and Generate Report

bash

```
./gradlew test jacocoTestReport
```

📁 Report Location

Code

```
build/reports/jacoco/test/html/index.html
```

Open that HTML file to view detailed coverage by class, method, and line.

🛠️ 1. Add Lombok to Simplify DTOs

📦 `build.gradle`

groovy

```
dependencies {
compileOnly 'org.projectlombok:lombok:1.18.30'
annotationProcessor 'org.projectlombok:lombok:1.18.30'
testCompileOnly 'org.projectlombok:lombok:1.18.30'
testAnnotationProcessor 'org.projectlombok:lombok:1.18.30'
}
```

✨ Simplified User.java

java

```
package com.example.model;

import lombok.Data;

@Data
public class User {
private int id;
private String name;
private String email;
}
```

`@Data` generates getters, setters, `toString`, `equals`, and `hashCode`.

🤖 2. GitHub Actions CI with JaCoCo

📁 **`.github/workflows/ci.yml`**

yaml

```
name: Java CI with Gradle

on:
push:
branches: [ main ]
pull_request:
branches: [ main ]

jobs:
build:
runs-on: ubuntu-latest

steps:
- name: Checkout code
uses: actions/checkout@v3

- name: Set up JDK
uses: actions/setup-java@v3
with:
distribution: 'temurin'
java-version: '17'
```

```
- name: Grant execute permission for Gradle
run: chmod +x gradlew

- name: Build and test
run: ./gradlew clean test jacocoTestReport

- name: Upload coverage report
uses: actions/upload-artifact@v3
with:
name: coverage-report
path: build/reports/jacoco/test/html
```

This workflow:

- Runs on every push/PR to `main`
- Builds, tests, and generates coverage
- Uploads the HTML report as an artifact

🔒 Configure JaCoCo Thresholds

📄 **`build.gradle`**

Add this block to enforce coverage rules:

groovy

```
jacocoTestCoverageVerification {
violationRules {
rule {
limit {
minimum = 0.80 // 80% minimum coverage
}
}
}
}

check.dependsOn jacocoTestCoverageVerification
```

This ensures that `./gradlew check` will fail if coverage is below 80%.

✅ Full JaCoCo Setup Summary

groovy

```
plugins {
id 'jacoco'
}

jacoco {
toolVersion = "0.8.10"
}

jacocoTestReport {
dependsOn test
reports {
xml.required = true
html.required = true
}
}

jacocoTestCoverageVerification {
violationRules {
rule {
limit {
minimum = 0.80
}
}
}
}

check.dependsOn jacocoTestCoverageVerification
```

🧪 GitHub Actions: Enforce Thresholds

Update your workflow to run `check` instead of just `test`:

yaml

```
- name: Build and verify coverage
run: ./gradlew clean check jacocoTestReport
```

This will fail the build if coverage is below 80%.

To add **API contract testing** to your Spring Boot project, the most widely used and robust approach is to use **Pact**, a code-first contract testing tool. Here's how to

integrate it step-by-step:

🧠 What Is API Contract Testing?

API contract testing ensures that the **consumer** (e.g. frontend or another service) and the **provider** (your Spring Boot API) agree on the structure of requests and responses. It helps catch breaking changes early—without needing full integration tests.

🛠️ 1. Add Pact Dependencies (Gradle)

groovy

```
dependencies {
testImplementation 'au.com.dius.pact.consumer:junit5:4.6.10'
testImplementation 'au.com.dius.pact.provider:junit5spring:4.6.10'
}
```

👥 2. Consumer Test Example

java

```
@ExtendWith(PactConsumerTestExt.class)
@PactTestFor(providerName = "UserService", port = "8081")
public class UserClientContractTest {

@Pact(consumer = "UserClient")
public RequestResponsePact createUserPact(PactDslWithProvider
builder) {
return builder
.given("User creation")
.uponReceiving("A request to create a user")
.path("/api/users")
.method("POST")
.body("{\"name\": \"Alice\", \"email\": \"alice@example.com\"}")
.willRespondWith()
.status(201)
.body("{\"id\": 101, \"name\": \"Alice\", \"email\":
\"alice@example.com\"}")
.toPact();
}

@Test
void testCreateUser(MockServer mockServer) {
```

```
RestTemplate restTemplate = new RestTemplate();
String response = restTemplate.postForObject(mockServer.getUrl() +
"/api/users",
Map.of("name", "Alice", "email", "alice@example.com"), String.class);
assertThat(response).contains("Alice");
}
}
```

🧪 3. Provider Verification

Use `@Provider` and `@PactFolder` to verify against saved contracts:

java

```
@Provider("UserService")
@PactFolder("pacts")
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT)
public class UserServiceContractVerificationTest {

@TestTarget
public final Target target = new HttpTestTarget("localhost", 8080);

@State("User creation")
public void userCreationState() {
// Setup provider state if needed
}
}
```

📁 Pact File Location

After running consumer tests, Pact will generate files in:

Code

```
build/pacts/UserClient-UserService.json
```

📚 Learn More

- Pact Documentation https://copilot.microsoft.com/chats/bvxiG3eF66j9kRqAy2ehD
- Testsigma's Guide to API Contract Testing https://copilot.microsoft.com/chats/bvxiG3eF66j9kRqAy2ehD#:~:text=Testsigma's%20Guide-,to,-API%20Contract%20Testing

- testRigor's Step-by-Step Contract Testing Guide https://copilot.microsoft.com/chats/bvxiG3eF66j9kRqAy2ehD#:~:text=testRigor's%20Step%2Dby%2D-,Step,-Contract%20Testing%20Guide

integrate **API contract testing with Pact** into both your **CI pipeline** and a **Pact Broker** for sharing contracts across teams.

🔁 CI Pipeline Integration with Pact

To automate contract testing in your CI/CD workflow, follow this structure:

✅ Consumer Pipeline Steps

1. **Run Pact consumer tests** to generate contract files.
2. **Publish contracts** to the Pact Broker:
   - Use the Pact Gradle/Maven plugin or CLI.
   - Tag contracts with branch name or version.

bash

```
./gradlew pactPublish \
-Dpact.consumer.version=$GIT_COMMIT \
-Dpact.tag=$BRANCH_NAME
```

✅ Provider Pipeline Steps

1. **Fetch contracts** from the Pact Broker using `@PactBroker` annotation.
2. **Run provider verification tests**.
3. **Publish verification results** back to the broker.

bash

```
-Dpact.verifier.publishResults=true \
-Dpact.provider.version=$GIT_COMMIT
```

✅ Deployment Gate: `can-i-deploy`

Use Pact CLI to check if a service version can be safely deployed:

bash

```
pact-broker can-i-deploy \
--pacticipant UserService \
--version $GIT_COMMIT \
--broker-base-url http://your-broker-url
```

This prevents breaking deployments by verifying compatibility across services.

📘 Full CI/CD strategy: Pact Nirvana Guide

🌐 Setting Up a Pact Broker

You can either self-host or use PactFlow for a managed solution.

🐳 Self-Hosted Broker (Docker Compose)

yaml

```
version: "3"
services:
postgres:
image: postgres
environment:
POSTGRES_USER: postgres
POSTGRES_PASSWORD: password
POSTGRES_DB: postgres

pact-broker:
image: pactfoundation/pact-broker
ports:
- "9292:9292"
environment:
PACT_BROKER_DATABASE_URL:
"postgres://postgres:password@postgres/postgres"
```

Run with:

bash

```
docker-compose up -d
```

Visit http://localhost:9292 to access the broker UI.

📘 Full setup guide: Sharing Pacts with the Pact Broker

🧩 Best Practices

- **Tag contracts** with environment or feature branch names.
- Use **webhooks** to trigger provider builds when contracts change.
- Integrate `can-i-deploy` checks before every release.
- Visualize service relationships via the broker's network graph.

To add **ZAP (Zed Attack Proxy) testing** to your workflow, you'll be integrating a powerful open-source security scanner that helps identify vulnerabilities in your web application. Here's how to set it up for both **local testing** and **CI/CD automation**, plus how to use it alongside your existing Spring Boot and WireMock setup.

⚙️ Step 1: Install OWASP ZAP

You can install ZAP via:

- **Official site**: ZAP Getting Started Guide
- **Package managers**:
  - macOS: `brew install zap`
  - Linux: `snap install zaproxy`
  - Windows: Download installer from the site

Or run it via Docker:
bash

```
docker run -u zap -p 8080:8080 -i owasp/zap2docker-stable zap.sh -
daemon -port 8080
```

🕷️ Step 2: Run a Basic Scan

You can use ZAP's **Quick Start** tab to scan a URL:

1. Launch ZAP GUI.
2. Enter your app's URL (e.g. http://localhost:8080) in the Quick Start tab.
3. Click **Attack**.
4. ZAP will spider the site and run an active scan.

This is ideal for manual testing or exploratory scans during development.

🤖 Step 3: Automate ZAP in CI/CD

Use ZAP's CLI or Docker in your pipeline:

🔧 Example GitHub Actions Workflow

yaml

```
jobs:
zap_scan:
runs-on: ubuntu-latest
steps:
- name: Run ZAP Baseline Scan
run: |
docker run -v $(pwd):/zap/wrk/:rw -t owasp/zap2docker-stable zap-
baseline.py \
-t http://your-app-url \
-g gen.conf -r zap_report.html
```

This runs a passive scan and generates an HTML report.

🔐 Step 4: Integrate with Spring Boot + WireMock

If you're mocking endpoints with WireMock, you can:

- Run ZAP against the **mock server** to test how your client handles malicious responses.
- Use ZAP's **fuzzer** to test input validation on mocked endpoints.
- Combine with **JUnit** to automate security assertions post-scan.

📊 Step 5: Generate Reports

ZAP can export results in:

- HTML
- XML
- JSON

Use `zap-cli` or Docker to extract reports:

bash

```
zap-cli report -o zap_report.html -f html
```

🧠 Bonus: Advanced Features

- **Authentication testing**: Simulate login flows.
- **Session management**: Test cookie handling and token expiration.
- **Scripting**: Use ZAP scripts to automate custom attack scenarios.
- **Integration with Jenkins**: Add ZAP as a post-build step.

Here's a complete setup for integrating **OWASP ZAP testing** into your **GitHub Actions pipeline** and configuring it for **Spring Boot apps**, including report generation and deployment gating.

🚀 GitHub Actions Pipeline for ZAP

You can use either the **baseline scan** (passive, safe) or the **full scan** (active, aggressive). Here's a full scan example that fails the build on high-severity findings:

📄 **`.github/workflows/zap_scan.yml`**

yaml

```
name: OWASP ZAP Full Scan

on:
push:
```

```
branches:
- main

jobs:
zap_scan:
runs-on: ubuntu-latest
name: Run ZAP Full Scan
steps:
- name: Checkout code
uses: actions/checkout@v4

- name: ZAP Full Scan
uses: zaproxy/action-full-scan@v0.12.0
with:
target: 'http://localhost:8080'
token: ${{ secrets.GITHUB_TOKEN }}
rules_file_name: '.zap/rules.tsv'
cmd_options: '-a'
fail_action: true
artifact_name: 'zap_scan_report'
```

📘 Full guide: ZAP Full Scan GitHub Action

📁 Sample `.zap/rules.tsv` File

This file lets you ignore or fail specific alerts:
tsv

```
10035 IGNORE (Strict-Transport-Security Header Not Set)
10098 FAIL (Cross-Domain Misconfiguration)
```

Place it in your repo at `.zap/rules.tsv`.

🧰 Spring Boot Configuration Tips

To make your Spring Boot app ZAP-friendly:

- **Disable CSRF** for test profiles if needed.
- Use a **test-specific profile** with relaxed security settings.
- Ensure the app is **served locally** during CI (e.g., via `mvn spring-boot:run` or Docker).
- Expose endpoints on `localhost:8080` for ZAP to scan.

📘 Spring Boot config reference: Externalized Configuration

📊 Report Access

ZAP will attach an HTML report to the GitHub Actions run. You can:

- View it in the **Artifacts** section.
- Parse it for alerts and severity.
- Use it to gate deployments (via `fail_action: true`).

🧠 Bonus: Pact + ZAP Synergy

You can run ZAP against your **WireMock mocks** or **Spring Boot provider** after Pact verification:

1. Run Pact provider tests.
2. Start the app or mock server.
3. Trigger ZAP scan.
4. Fail build if ZAP finds critical issues.

Here's a **Docker Compose setup** that runs a full-stack environment with **Spring Boot**, **WireMock**, and **OWASP ZAP** for local security and contract testing. This setup lets you simulate external APIs, run your app, and scan it for vulnerabilities—all in one go.

🚢 Docker Compose Structure

📁 Project Layout

Code

```
project-root/
├── docker-compose.yml
├── spring-app/
│   ├── Dockerfile
│   └── target/app.jar
├── wiremock/
│   ├── __files/
│   └── mappings/
└── zap/
└── zap-config.conf (optional)
```

⚙️ `docker-compose.yml`

yaml

```
version: '3.8'

services:
spring-app:
build:
context: ./spring-app
container_name: spring-app
ports:
- "8080:8080"
depends_on:
- wiremock
environment:
- SPRING_PROFILES_ACTIVE=docker
- WIREMOCK_BASE_URL=http://wiremock:8081

wiremock:
image: wiremock/wiremock:2.35.0
container_name: wiremock
ports:
- "8081:8080"
volumes:
- ./wiremock/__files:/home/wiremock/__files
- ./wiremock/mappings:/home/wiremock/mappings

zap:
image: owasp/zap2docker-stable
container_name: zap
depends_on:
- spring-app
command: >
zap-full-scan.py -t http://spring-app:8080
-r zap-report.html
-g zap-config.conf
volumes:
- ./zap:/zap/wrk
```

🐳 spring-app/Dockerfile

Dockerfile

```
FROM openjdk:17-jdk-slim
COPY target/app.jar app.jar
```

```
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Make sure your Spring Boot app is built and the JAR is placed in `spring-app/target/app.jar`.

🧪 WireMock Setup

Place your stubbed responses in:

- `wiremock/__files/` → JSON response bodies
- `wiremock/mappings/` → WireMock mapping files

Example mapping file:
json

```
{
"request": {
"method": "GET",
"url": "/api/users"
},
"response": {
"status": 200,
"bodyFileName": "users.json",
"headers": {
"Content-Type": "application/json"
}
}
}
```

🔐 ZAP Config (Optional)

You can customize ZAP behavior with `zap/zap-config.conf`, e.g., exclude endpoints or set scan thresholds.

🧠 How It Works

- Spring Boot app starts and connects to WireMock for external API stubbing.
- ZAP scans the running app on http://spring-app:8080.
- ZAP generates a report (`zap-report.html`) in the `zap/` folder.

Here's the complete setup you asked for: a **Docker Compose environment** that runs **Spring Boot**, **WireMock**, and **OWASP ZAP**, plus a **GitHub Actions workflow** to automate it, and **Pact integration** for contract testing.

1. Docker Compose Setup

docker-compose.yml

yaml

```
version: '3.8'

services:
spring-app:
build:
context: ./spring-app
container_name: spring-app
ports:
- "8080:8080"
depends_on:
- wiremock
environment:
- SPRING_PROFILES_ACTIVE=docker
- WIREMOCK_BASE_URL=http://wiremock:8081

wiremock:
image: wiremock/wiremock:2.35.0
container_name: wiremock
ports:
- "8081:8080"
volumes:
- ./wiremock/__files:/home/wiremock/__files
- ./wiremock/mappings:/home/wiremock/mappings

zap:
image: owasp/zap2docker-stable
container_name: zap
depends_on:
- spring-app
command: >
zap-full-scan.py -t http://spring-app:8080
-r zap-report.html
-g zap-config.conf
volumes:
- ./zap:/zap/wrk
```

🐋 2. Spring Boot Dockerfile

📄 **spring-app/Dockerfile**

Dockerfile

```
FROM openjdk:17-jdk-slim
COPY target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

🧪 3. WireMock Stubs

Place your stub files in:

- `wiremock/__files/` → JSON response bodies
- `wiremock/mappings/` → Mapping definitions

Include metadata for Pact generation:

json

```
{
"request": {
"method": "GET",
"url": "/api/users"
},
"response": {
"status": 200,
"bodyFileName": "users.json",
"headers": {
"Content-Type": "application/json"
}
},
"metadata": {
"wireMockPactSettings": {
"provider": "UserService"
}
}
}
```

🔐 4. ZAP Config (Optional)

📄 **zap/zap-config.conf**

text

```
# Example: exclude static assets
```

```
exclude=.*\.css
exclude=.*\.js
```

🤖 5. GitHub Actions Workflow

📄 **.github/workflows/full-stack-test.yml**

yaml

```
name: Full Stack Security & Contract Test

on:
push:
branches: [main]

jobs:
test-stack:
runs-on: ubuntu-latest
services:
docker:
image: docker:20.10.16
options: --privileged

steps:
- name: Checkout code
uses: actions/checkout@v4

- name: Set up Docker Compose
run: docker-compose up --build -d

- name: Wait for services
run: sleep 30

- name: Run ZAP scan
run: docker exec zap cat /zap/wrk/zap-report.html

- name: Upload ZAP report
uses: actions/upload-artifact@v4
with:
name: zap-report
path: zap/zap-report.html

- name: Pact Verification
run: |
```

```
docker exec spring-app ./gradlew pactVerify
```

🔗 6. Pact Integration with WireMock

Use WireMock Pact Extension to generate contracts from WireMock stubs:

- Add `wiremock-pact-lib` and `wiremock-pact-extension-junit5` to your test dependencies.
- Register the extension in your test class.
- After tests, extract `ServeEvent`s and save Pact JSON.

Example:
java

```
@RegisterExtension
static WireMockPactExtension pactExtension = new
WireMockPactExtension(
WireMockPactConfig.builder()
.setConsumerDefaultValue("MyConsumer")
.setProviderDefaultValue("UserService")
.setPactJsonFolder("build/pacts")
.build()
);
```

📘 Full guide: WireMock + Pact Integration

Here's the **final piece** of your full-stack testing setup: a local shell script to run everything, Slack/Teams notifications for ZAP results, and Pact Broker integration for deployment gating.

🧨 1. Shell Script to Run Full Stack Locally

📄 run-full-stack.sh

bash

```
#!/bin/bash

echo "🔧 Building Spring Boot app..."
./gradlew clean build -x test

echo "🐳 Starting Docker Compose stack..."
docker-compose up --build -d

echo "⏳ Waiting for services to start..."
```

```
sleep 30

echo "🕷️ Running ZAP scan..."
docker exec zap cat /zap/wrk/zap-report.html > zap/zap-report.html

echo "📊 ZAP report saved to zap/zap-report.html"

echo "📦 Running Pact verification..."
docker exec spring-app ./gradlew pactVerify

echo "✅ All tests complete."
```

Make it executable:
bash

```
chmod +x run-full-stack.sh
```

📣 2. Slack/Teams Notification for ZAP Results

Use Zapier or a custom Node.js notifier like ZaptecUnifiedNotifier.

📄 `notify-zap.js` (Node.js)

js

```
const fs = require('fs');
const axios = require('axios');

const webhook = process.env.SLACK_WEBHOOK_URL;
const report = fs.readFileSync('zap/zap-report.html', 'utf8');

axios.post(webhook, {
text: `🛡️ ZAP Scan Completed\n\n${report.substring(0, 500)}...`,
})
.then(() => console.log("✅ Slack notification sent"))
.catch(err => console.error("❌ Slack notification failed", err));
```

Run with:
bash

```
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/... node notify-
zap.js
```

🚦 3. Pact Broker Deployment Gating

Use the `can-i-deploy` CLI to check compatibility before deploying:

✅ Check if safe to deploy

bash

```
pact-broker can-i-deploy \
--pacticipant UserService \
--version $GIT_COMMIT \
--to-environment production \
--broker-base-url http://your-broker-url
```

✅ Record deployment

bash

```
pact-broker record-deployment \
--pacticipant UserService \
--version $GIT_COMMIT \
--environment production \
--broker-base-url http://your-broker-url
```

📘 Full guide: Can I Deploy – Pact Docs

Makefile, a dashboard idea, and GitHub Actions integration. This will give you a clean dev experience, CI/CD automation, and visibility across your team.

🛠️ 1. Makefile to Wrap Everything

📄 **`Makefile`**

makefile

```
.PHONY: build up zap pact notify all

build:
./gradlew clean build -x test

up:
docker-compose up --build -d

zap:
sleep 30
docker exec zap cat /zap/wrk/zap-report.html > zap/zap-report.html
```

```
pact:
docker exec spring-app ./gradlew pactVerify

notify:
SLACK_WEBHOOK_URL=$(SLACK_WEBHOOK_URL) node notify-zap.js

all: build up zap pact notify
```

Run it with:
bash

```
make all SLACK_WEBHOOK_URL=https://hooks.slack.com/services/...
```

📊 2. Dashboard for ZAP + Pact Results

You could build a lightweight dashboard using Spring Boot + Thymeleaf or Node.js + Express. Here's the concept:

🧩 Features

- View latest ZAP scan results (HTML report embedded)
- Pact verification status per service
- "Can I Deploy" status for each environment
- Slack/Teams webhook logs

📁 Folder Structure

Code

```
dashboard/
├── public/
│ └── zap-report.html
├── views/
│ └── index.ejs
├── server.js
```

🧪 Sample Express Server

js

```
const express = require('express');
const app = express();
const fs = require('fs');

app.set('view engine', 'ejs');
```

```
app.get('/', (req, res) => {
const zapReport = fs.readFileSync('public/zap-report.html', 'utf8');
res.render('index', { zapReport });
});

app.listen(3000, () => console.log('Dashboard running on
http://localhost:3000'));
```

⚙️ 3. GitHub Actions Integration

📄 **.github/workflows/full-stack.yml**

yaml

```
name: Full Stack Test

on:
push:
branches: [ main ]

jobs:
test-stack:
runs-on: ubuntu-latest
env:
SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}

steps:
- uses: actions/checkout@v3

- name: Set up JDK
uses: actions/setup-java@v3
with:
java-version: '17'

- name: Run Makefile
run: make all SLACK_WEBHOOK_URL=${{ secrets.SLACK_WEBHOOK_URL }}
```

Let's embed both functional and non-functional testing into your full-stack setup—cleanly separated, automated, and CI/CD-friendly.

✅ Functional Tests

These validate correctness of business logic and API behavior.

🔷 1. Unit Tests (JUnit + Mockito)

Already part of your Spring Boot build. Just ensure they run separately:

bash

```
./gradlew test
```

🔷 2. Integration Tests

Use Spring Boot's `@SpringBootTest` with embedded WireMock or test containers.

java

```
@SpringBootTest
@AutoConfigureWireMock(port = 8089)
public class UserServiceIntegrationTest {
@Test
void shouldReturnUserDetails() {
stubFor(get("/users/123").willReturn(okJson("{ \"id\": 123, \"name\":
\"Esin\" }")));
// Call service and assert response
}
}
```

Run with:

bash

```
./gradlew integrationTest
```

Add this to your `build.gradle`:

groovy

```
sourceSets {
integrationTest {
java.srcDir file('src/integrationTest/java')
resources.srcDir file('src/integrationTest/resources')
compileClasspath += sourceSets.main.output +
configurations.testRuntimeClasspath
runtimeClasspath += output + compileClasspath
}
}

task integrationTest(type: Test) {
testClassesDirs = sourceSets.integrationTest.output.classesDirs
classpath = sourceSets.integrationTest.runtimeClasspath
```

```
}
```

🧪 Non-Functional Tests

These validate performance, security, and reliability.

🔶 1. Security: ZAP Scan

Already integrated. You can enhance it with threshold checks:

bash

```
docker exec zap zap-cli alerts -l High
```

Fail the build if high-risk alerts are found:

bash

```
if [ "$(docker exec zap zap-cli alerts -l High | wc -l)" -gt 0 ];
then
echo "❌ High-risk vulnerabilities found!"
exit 1
fi
```

🔶 2. Performance: JMeter or Gatling

🧪 JMeter Example

bash

```
jmeter -n -t tests/performance.jmx -l results/performance.jtl
```

Add a threshold check:

bash

```
if grep -q "KO" results/performance.jtl; then
echo "❌ Performance test failed"
exit 1
fi
```

🧪 Gatling (Scala-based)

bash

```
./gradlew gatlingRun
```

🧵 Makefile Update

Add these targets:
makefile

```
unit:
./gradlew test

integration:
./gradlew integrationTest

security:
docker exec zap zap-cli alerts -l High

performance:
jmeter -n -t tests/performance.jmx -l results/performance.jtl

functional: unit integration pact
nonfunctional: zap security performance

all: build up functional nonfunctional notify
```

Here's how to complete your full-stack testing setup with:

✅ Functional & Non-Functional Testing Integration

🧪 Functional Tests

- **Unit Tests**: JUnit + Mockito (already configured)
- **Integration Tests**: Spring Boot + WireMock
- **Contract Tests**: Pact (consumer + provider verification)

🔐 Non-Functional Tests

- **Security**: OWASP ZAP (Docker + CI)
- **Performance**: JMeter and Gatling

🧰 JMeter Setup

📘 Guide: JMeter Test Plan Manual

🗂️ Folder Structure

Code

```
tests/
└── jmeter/
```

```
└── performance.jmx
```

🧪 Run JMeter in CLI

bash

```
jmeter -n -t tests/jmeter/performance.jmx -l results/performance.jtl
```

✅ Add to Makefile

makefile

```
performance-jmeter:
jmeter -n -t tests/jmeter/performance.jmx -l results/performance.jtl
```

🚀 Gatling Setup with Gradle

📘 Guide: Gatling Gradle Plugin Docs

📦 `build.gradle`

groovy

```
plugins {
id 'io.gatling.gradle' version '3.14.3.5'
}

gatling {
simulationsDir = file("src/gatling/scala")
}
```

🗂️ Folder Structure

Code

```
src/
└── gatling/
└── scala/
└── simulations/
└── UserSimulation.scala
```

🧪 Run Gatling

bash

```
./gradlew gatlingRun
```

✅ Add to Makefile

makefile

```
performance-gatling:
./gradlew gatlingRun
```

🤖 GitHub Actions Integration

📄 `.github/workflows/full-stack.yml` (Additions)

yaml

```
- name: Run JMeter Performance Test
run: jmeter -n -t tests/jmeter/performance.jmx -l
results/performance.jtl

- name: Run Gatling Performance Test
run: ./gradlew gatlingRun

- name: Upload Gatling Report
uses: actions/upload-artifact@v4
with:
name: gatling-report
path: build/reports/gatling
```

Here's a complete guide to setting up **JMeter** and **Gatling** for performance testing, with Gradle and CI/CD integration:

🧪 JMeter Setup

📘 Reference: JMeter Test Plan Manual

✅ Steps to Build a Test Plan

- **Create Test Plan**: File → New Test Plan
- **Add Thread Group**: Right-click → Add → Threads → Thread Group
- **Add HTTP Sampler**: Inside Thread Group → Add → Sampler → HTTP Request
- **Add Listeners**: Add → Listener → View Results Tree, Summary Report, etc.
- **Optional Elements**: Config Elements, Assertions, Timers
- **Run Test**: GUI (for debugging) or CLI (for real load)

🧪 CLI Execution

bash

```
jmeter -n -t tests/jmeter/performance.jmx -l results/performance.jtl
```

✅ Makefile Target

makefile

```
performance-jmeter:
jmeter -n -t tests/jmeter/performance.jmx -l results/performance.jtl
```

🚀 Gatling Setup with Gradle

📘 Reference: Gatling Gradle Plugin Docs

✅ `build.gradle` Configuration

groovy

```
plugins {
id 'io.gatling.gradle' version '3.14.3.5'
}

gatling {
simulationsDir = file("src/gatling/scala")
}
```

📁 Folder Structure

Code

```
src/
└── gatling/
└── scala/
└── simulations/
└── UserSimulation.scala
```

🧪 Run Gatling

bash

```
./gradlew gatlingRun
```

✅ Makefile Target

makefile

```
performance-gatling:
```

```
./gradlew gatlingRun
```

🤖 GitHub Actions Integration

📄 **`.github/workflows/performance.yml`**

yaml

```
jobs:
performance:
runs-on: ubuntu-latest
steps:
- name: Checkout Code
uses: actions/checkout@v4

- name: Run JMeter Performance Test
run: jmeter -n -t tests/jmeter/performance.jmx -l
results/performance.jtl

- name: Run Gatling Performance Test
run: ./gradlew gatlingRun

- name: Upload Gatling Report
uses: actions/upload-artifact@v4
with:
name: gatling-report
path: build/reports/gatling
```

Here's your complete, production-grade full-stack testing setup—integrating:

- ✅ Functional tests (unit, integration, contract)
- 🔐 Non-functional tests (security via ZAP, performance via JMeter & Gatling)
- 🧪 CI/CD automation with GitHub Actions
- 📊 Threshold enforcement
- 📣 Slack/Teams notifications
- 🧰 Local Makefile for developer convenience
- 🐳 Docker Compose for full-stack orchestration
- 🌐 Optional dashboard for visibility

🧱 Project Structure

Code

```
project-root/
├── docker-compose.yml
├── Makefile
├── run-full-stack.sh
├── spring-app/
│ ├── Dockerfile
│ └── target/app.jar
├── wiremock/
│ ├── __files/
│ └── mappings/
├── zap/
│ ├── zap-config.conf
│ └── zap-report.html
├── tests/
│ ├── jmeter/
│ │ └── performance.jmx
│ └── gatling/
│ └── scala/
│ └── simulations/
│ └── UserSimulation.scala
├── dashboard/
│ ├── server.js
│ ├── views/index.ejs
│ └── public/zap-report.html
├── notify-zap.js
└── .github/
└── workflows/full-stack.yml
```

🐳 docker-compose.yml

yaml

```
version: '3.8'

services:
spring-app:
build:
context: ./spring-app
container_name: spring-app
ports:
- "8080:8080"
depends_on:
```

```
- wiremock
environment:
- SPRING_PROFILES_ACTIVE=docker
- WIREMOCK_BASE_URL=http://wiremock:8081

wiremock:
image: wiremock/wiremock:2.35.0
container_name: wiremock
ports:
- "8081:8080"
volumes:
- ./wiremock/__files:/home/wiremock/__files
- ./wiremock/mappings:/home/wiremock/mappings

zap:
image: owasp/zap2docker-stable
container_name: zap
depends_on:
- spring-app
command: >
zap-full-scan.py -t http://spring-app:8080
-r zap-report.html
-g zap-config.conf
volumes:
- ./zap:/zap/wrk
```

🛠️ Makefile

makefile

```
.PHONY: build up unit integration pact zap security performance-
jmeter performance-gatling notify all

build:
./gradlew clean build -x test

up:
docker-compose up --build -d

unit:
./gradlew test

integration:
```

```
./gradlew integrationTest

pact:
docker exec spring-app ./gradlew pactVerify

zap:
sleep 30
docker exec zap cat /zap/wrk/zap-report.html > zap/zap-report.html

security:
@echo "Checking for high-risk ZAP alerts..."
@if [ $(shell docker exec zap zap-cli alerts -l High | wc -l) -gt 0
]; then \
echo "❌ High-risk vulnerabilities found!"; exit 1; \
else echo "✅ No high-risk vulnerabilities."; fi

performance-jmeter:
jmeter -n -t tests/jmeter/performance.jmx -l results/performance.jtl

performance-gatling:
./gradlew gatlingRun

notify:
SLACK_WEBHOOK_URL=$(SLACK_WEBHOOK_URL) node notify-zap.js

functional: unit integration pact
nonfunctional: zap security performance-jmeter performance-gatling

all: build up functional nonfunctional notify
```

🧪 notify-zap.js

js

```
const fs = require('fs');
const axios = require('axios');

const webhook = process.env.SLACK_WEBHOOK_URL;
const report = fs.readFileSync('zap/zap-report.html', 'utf8');

axios.post(webhook, {
text: `🛡️ ZAP Scan Completed\n\n${report.substring(0, 500)}...`,
})
```

```
.then(() => console.log("✅ Slack notification sent"))
.catch(err => console.error("❌ Slack notification failed", err));
```

🤖 GitHub Actions: `.github/workflows/full-stack.yml`

yaml

```
name: Full Stack Test & Security Scan

on:
push:
branches: [ main ]

jobs:
test-stack:
runs-on: ubuntu-latest
env:
SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}

steps:
- uses: actions/checkout@v4

- name: Set up JDK
uses: actions/setup-java@v3
with:
java-version: '17'

- name: Run Full Stack Tests
run: make all SLACK_WEBHOOK_URL=${{ secrets.SLACK_WEBHOOK_URL }}

- name: Upload ZAP Report
uses: actions/upload-artifact@v4
with:
name: zap-report
path: zap/zap-report.html

- name: Upload Gatling Report
uses: actions/upload-artifact@v4
with:
name: gatling-report
path: build/reports/gatling
```

🌐 Dashboard (Optional)

server.js

js

```
const express = require('express');
const fs = require('fs');
const app = express();

app.set('view engine', 'ejs');
app.use(express.static('public'));

app.get('/', (req, res) => {
const zapReport = fs.readFileSync('public/zap-report.html', 'utf8');
res.render('index', { zapReport });
});

app.listen(3000, () => console.log('Dashboard running at
http://localhost:3000'));
```

views/index.ejs

html

```
<!DOCTYPE html>
<html>
<head><title>Security & Contract Dashboard</title></head>
<body>
<h1>🛡️ ZAP Security Report</h1>
<div><%= zapReport %></div>
</body>
</html>
```

🧠 Final Touches

- ✅ Thresholds enforced for ZAP alerts
- ✅ Slack notifications for scan results
- ✅ Performance tests via JMeter + Gatling
- ✅ Pact contract verification
- ✅ CI/CD automation via GitHub Actions
- ✅ Local orchestration via Docker Compose + Makefile
- ✅ Optional dashboard for visibility

I can’t create or share downloadable ZIP files directly—but I can guide you to set it up locally in seconds. Here’s how to generate the full project structure on your machine:

⚡ Quick Setup Script

Save this as setup.sh and run it from an empty directory:

bash

```
#!/bin/bash

mkdir -p project-root/{spring-app,wiremock/{__files,mappings},zap,tests/{jmeter,gatling/scala/simulations},dashboard/views,.github/workflows}

# Sample files
echo 'FROM eclipse-temurin:17-jdk\nCOPY target/app.jar /app.jar\nCMD ["java", "-jar", "/app.jar"]' > project-root/spring-app/Dockerfile
echo '<response>Mocked response</response>' > project-root/wiremock/__files/response.xml
echo '{}' > project-root/wiremock/mappings/example.json
echo 'ZAP config placeholder' > project-root/zap/zap-config.conf
echo '<html>ZAP Report</html>' > project-root/zap/zap-report.html
echo 'JMeter test plan placeholder' > project-root/tests/jmeter/performance.jmx
echo 'class UserSimulation extends Simulation {}' > project-root/tests/gatling/scala/simulations/UserSimulation.scala
echo '<html><%= zapReport %></html>' > project-root/dashboard/views/index.ejs
echo 'console.log("Dashboard")' > project-root/dashboard/server.js
echo 'console.log("Notify ZAP")' > project-root/notify-zap.js
echo 'name: Full Stack Test & Security Scan\non:\n push:\n branches: [ main ]\n\njobs:\n test-stack:\n runs-on: ubuntu-latest\n steps:\n - uses: actions/checkout@v4' > project-root/.github/workflows/full-stack.yml

echo '✅ Project scaffolded in ./project-root'
```

Then run:

bash

```
chmod +x setup.sh
./setup.sh
```

🧰 Next Steps

- Drop your actual Spring Boot JAR into `spring-app/target/app.jar`
- Replace placeholders with your real WireMock mappings, ZAP config, and test plans
- Run `make all` from `project-root` to kick off the full stack

Perfect—let's integrate regression testing into your full-stack setup. Since your stack already includes unit, integration, contract, and performance tests, regression testing will slot in as a repeatable suite that verifies critical flows haven't broken due to recent changes.

🧪 Regression Test Strategy

Regression tests typically:

- Cover end-to-end flows across multiple components
- Use stable test data and assertions
- Run after builds and before deployments
- Can be implemented using JUnit, REST-assured, or Gatling (for behavioral flows)

📁 Suggested Structure

Add this to your project:

Code

```
project-root/
└── tests/
└── regression/
├── RegressionTest.java # JUnit-based
└── regression-flow.json # Optional test data
```

🧬 Sample RegressionTest.java (Spring Boot + REST-assured)

java

```
package regression;

import io.restassured.RestAssured;
import org.junit.jupiter.api.Test;

import static org.hamcrest.Matchers.*;
```

```
public class RegressionTest {

static {
RestAssured.baseURI = "http://localhost:8080";
}

@Test
void userFlowShouldSucceed() {
RestAssured
.given()
.header("Accept", "application/json")
.when()
.get("/api/users/42")
.then()
.statusCode(200)
.body("name", not(emptyString()))
.body("email", containsString("@"));
}

@Test
void shouldHandleLegacyRequestGracefully() {
RestAssured
.given()
.header("X-Legacy-Client", "true")
.when()
.get("/api/legacy-endpoint")
.then()
.statusCode(200)
.body("status", equalTo("ok"));
}
}
```

🛠️ Add to `Makefile`

makefile

```
regression:
./gradlew regressionTest
```

Make sure your `build.gradle` includes:

groovy

```
sourceSets {
regression {
java.srcDir file('src/regression/java')
resources.srcDir file('src/regression/resources')
compileClasspath += sourceSets.main.output +
configurations.testRuntimeClasspath
runtimeClasspath += output + compileClasspath
}
}

task regressionTest(type: Test) {
testClassesDirs = sourceSets.regression.output.classesDirs
classpath = sourceSets.regression.runtimeClasspath
shouldRunAfter test
}
```

🤖 GitHub Actions Update

Add this step to `.github/workflows/full-stack.yml`:

yaml

```
- name: Run Regression Tests
run: make regression
```

✅ Summary

You now have:

- A dedicated regression test suite
- JUnit + REST-assured coverage for critical flows
- CI integration via GitHub Actions
- Local trigger via `make regression`

Let's extend your regression testing setup with:

📸 1. Regression Snapshots

Snapshot testing captures expected outputs (e.g. JSON responses) and compares them on each run. If the output changes unexpectedly, the test fails.

✅ Setup with JSONassert

📁 Structure

Code

```
tests/
└── regression/
├── RegressionSnapshotTest.java
└── snapshots/
└── user-42.json
```

user-42.json

json

```
{
"id": 42,
"name": "Alice",
"email": "alice@example.com"
}
```

RegressionSnapshotTest.java

java

```
package regression;

import org.junit.jupiter.api.Test;
import org.skyscreamer.jsonassert.JSONAssert;
import org.springframework.web.client.RestTemplate;

import java.nio.file.Files;
import java.nio.file.Paths;

public class RegressionSnapshotTest {

@Test
void userResponseShouldMatchSnapshot() throws Exception {
String actual = new
RestTemplate().getForObject("http://localhost:8080/api/users/42",
String.class);
String expected =
Files.readString(Paths.get("src/regression/resources/snapshots/user-
42.json"));
JSONAssert.assertEquals(expected, actual, true); // strict match
}
}
```

🧪 2. Database State Verification

Use Spring Boot's `@Sql` or `JdbcTemplate` to verify DB state before/after tests.

📄 DatabaseRegressionTest.java

java

```
package regression;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jdbc.core.JdbcTemplate;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
public class DatabaseRegressionTest {

@Autowired
JdbcTemplate jdbc;

@Test
void userShouldExistInDatabase() {
Integer count = jdbc.queryForObject("SELECT COUNT(*) FROM users WHERE
id = 42", Integer.class);
assertThat(count).isEqualTo(1);
}
}
```

✅ Optional: Use `@Sql` for setup/teardown

java

```
@Sql(scripts = "/setup.sql", executionPhase = BEFORE_TEST_METHOD)
@Sql(scripts = "/cleanup.sql", executionPhase = AFTER_TEST_METHOD)
```

🛠️ Makefile Update

makefile

```
regression-snapshot:
./gradlew regressionSnapshotTest

regression-db:
```

```
./gradlew regressionDbTest

regression: regressionTest regression-snapshot regression-db
```

🤖 GitHub Actions Update

yaml

```
- name: Run Regression Snapshot Tests
run: make regression-snapshot

- name: Run Database Regression Tests
run: make regression-db
```

🧠 Want More?

Would you like:

- Snapshot auto-update mode (with approval)?
- DB state diffing and rollback?
- Visual dashboard for regression deltas?

Let's complete your regression testing system with:

✅ 1. Snapshot Auto-Update with Approval

This allows snapshots to be updated when expected changes occur—but only with explicit approval.

📁 Structure

Code

```
snapshots/
├── user-42.json # Approved snapshot
├── user-42.actual.json # Latest test output
```

📄 Updated RegressionSnapshotTest.java

java

```
@Test
void userResponseShouldMatchSnapshot() throws Exception {
String actual =
restTemplate.getForObject("http://localhost:8080/api/users/42",
String.class);
```

```
Path expectedPath =
Paths.get("src/regression/resources/snapshots/user-42.json");
Path actualPath = Paths.get("src/regression/resources/snapshots/user-
42.actual.json");

Files.writeString(actualPath, actual);

String expected = Files.readString(expectedPath);
try {
JSONAssert.assertEquals(expected, actual, true);
} catch (AssertionError e) {
System.err.println("❌ Snapshot mismatch. Review and approve
changes:");
System.err.println("diff user-42.json user-42.actual.json");
throw e;
}
}
```

✅ Approving Changes

bash

```
cp user-42.actual.json user-42.json
```

You can automate this with a `make approve-snapshots` target.

🧪 2. DB State Diffing and Rollback

Use Flyway or Liquibase for migrations, and verify DB state before/after tests.

📄 DatabaseStateVerifier.java

java

```
@Test
void verifyUserStateBeforeAndAfterUpdate() {
Map<String, Object> before = jdbc.queryForMap("SELECT * FROM users
WHERE id = 42");

// Simulate update
restTemplate.put("http://localhost:8080/api/users/42", Map.of("name",
"Updated"));

Map<String, Object> after = jdbc.queryForMap("SELECT * FROM users
WHERE id = 42");
```

```
assertThat(after.get("name")).isNotEqualTo(before.get("name"));
}
```

✅ Rollback Strategy

Use `@Transactional` with `@Test` to auto-rollback:

java

```
@Transactional
@Test
void testWithRollback() {
// DB changes here will be rolled back after test
}
```

📊 3. Visual Dashboard for Regression Deltas

Extend your existing dashboard:

📄 **views/index.ejs**

html

```
<h2>🧪 Regression Snapshot Diff</h2>
<pre><%= snapshotDiff %></pre>

<h2>🗃️ DB State Changes</h2>
<pre><%= dbDiff %></pre>
```

📄 **server.js**

js

```
const snapshotDiff = fs.readFileSync('snapshots/user-42.actual.json',
'utf8');
// You can use a diff lib to compare with user-42.json

const dbDiff = 'User name changed from Alice to Updated'; //
placeholder

res.render('index', { zapReport, snapshotDiff, dbDiff });
```

🧠 Final Touches

Would you like:

- GitHub Action to auto-approve snapshots on tagged releases?
- Slack alert when snapshot or DB state changes?
- Integration with Pact Broker to show regression impact?

Brilliant. Here's how we can take it further and make the whole system feel seamless and scalable:

🚀 GitHub Actions: Snapshot Approval on Tagged Releases

You can auto-approve snapshots only when a release is tagged—ensuring manual review during development but automation in CI.

📄 **`.github/workflows/approve-snapshots.yml`**

yaml

```
name: Approve Snapshots on Release

on:
push:
tags:
- 'v*'

jobs:
approve:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3
- name: Auto-approve snapshots
run: |
cp snapshots/*.actual.json snapshots/
echo "✅ Snapshots approved for release ${{ github.ref }}"
```

📣 Slack Alerts for Snapshot or DB Changes

Use a Slack webhook to notify your team when regression changes are detected.

📄 notifySlack.sh

bash

```
#!/bin/bash
curl -X POST -H 'Content-type: application/json' \
--data '{"text":"⚠ Regression snapshot mismatch detected in user-
42.json"}' \
$SLACK_WEBHOOK_URL
```

Trigger this from your test failure handler or GitHub Action.

🔗 Pact Broker Integration: Regression Impact View

If you're using Pact, you can annotate consumer contracts with regression metadata.

📄 `pact.json` (extended)

json

```
{
"consumer": { "name": "UserService" },
"provider": { "name": "UserAPI" },
"interactions": [...],
"metadata": {
"regressionSnapshot": "user-42.json",
"lastVerified": "2025-08-14"
}
}
```

Then display this in your dashboard alongside ZAP and performance metrics.

🧩 Unified Dashboard Sections

Let's organize your dashboard like this:

Section		
Source		
Status Indicator		
🔐 ZAP Security Scan	`zap-report.html`	✅ / ⚠ / ❌
📈 Performance Metrics	JMeter/Gatling reports	✅ / ⚠ / ❌
🧪 Snapshot Diff	`user-42.actual.json vs user-42.json`	✅ / ⚠ / ❌
🗃️ DB State Diff	JDBC snapshot comparison	✅ / ⚠ / ❌
🤝 Pact Contracts	Pact Broker metadata	✅ / ⚠ / ❌

Perfect. Here's a full-stack blueprint to implement everything—automated snapshot approval, Slack alerts, Pact metadata, and a unified dashboard—all orchestrated via GitHub Actions.

🧪 1. Snapshot Approval on Tagged Releases

📄 **`.github/workflows/approve-snapshots.yml`**

yaml

```
name: Approve Snapshots on Release

on:
push:
tags:
- 'v*'

jobs:
approve:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3
- name: Auto-approve snapshots
run: |
for f in snapshots/*.actual.json; do
base=$(basename "$f" .actual.json)
cp "$f" "snapshots/$base.json"
echo "✅ Approved $base.json"
done
```

📣 2. Slack Alerts for Regression or DB Changes

📄 notifySlack.sh

bash

```
#!/bin/bash
MESSAGE=$1
curl -X POST -H 'Content-type: application/json' \
--data "{\"text\":\"$MESSAGE\"}" \
$SLACK_WEBHOOK_URL
```

📄 alert-on-diff.sh

bash

```
#!/bin/bash
for f in snapshots/*.actual.json; do
base=$(basename "$f" .actual.json)
diff=$(diff "snapshots/$base.json" "$f")
if [ -n "$diff" ]; then
./notifySlack.sh "⚠ Snapshot mismatch: $base.json"
fi
done
```

Add this to your CI pipeline after tests.

🔗 3. Pact Metadata Extension

📄 pact-with-metadata.json

json

```
{
"consumer": { "name": "UserService" },
"provider": { "name": "UserAPI" },
"interactions": [...],
"metadata": {
"regressionSnapshot": "user-42.json",
"lastVerified": "2025-08-14",
"performance": {
"avgResponseTimeMs": 120,
"thresholdMs": 200
},
"securityScan": {
"zapScore": "PASS",
"lastScan": "2025-08-14"
}
}
}
```

Upload this to your Pact Broker and pull it into your dashboard.

📊 4. Unified Dashboard

📄 **views/index.ejs**

html

```
<h2>🧪 Snapshot Diff</h2>
<pre><%= snapshotDiff %></pre>

<h2>🗃️ DB State Changes</h2>
<pre><%= dbDiff %></pre>

<h2>🔐 ZAP Security Scan</h2>
<pre><%= zapReport %></pre>

<h2>📈 Performance Metrics</h2>
<pre><%= perfMetrics %></pre>

<h2>🤝 Pact Metadata</h2>
<pre><%= pactMetadata %></pre>
```

📄 **server.js**

js

```
const snapshotDiff = fs.readFileSync('snapshots/user-42.actual.json',
'utf8');
const dbDiff = fs.readFileSync('db-diff.txt', 'utf8');
const zapReport = fs.readFileSync('zap-report.html', 'utf8');
const perfMetrics = fs.readFileSync('perf-summary.txt', 'utf8');
const pactMetadata = fs.readFileSync('pact-with-metadata.json',
'utf8');

res.render('index', { snapshotDiff, dbDiff, zapReport, perfMetrics,
pactMetadata });
```

🧠 Bonus: Artifact Upload in GitHub Actions

📄 **.github/workflows/test.yml** (snippet)

yaml

```
- name: Upload Snapshot Diffs
uses: actions/upload-artifact@v3
with:
name: snapshot-diffs
path: snapshots/*.actual.json
```

```
- name: Upload ZAP Report
uses: actions/upload-artifact@v3
with:
name: zap-report
path: zap-report.html
```

Awesome. Here's a powerful, modular `Makefile` to orchestrate your entire regression pipeline locally—snapshot approval, Slack alerts, dashboard refresh, and artifact prep.

🛠️ `Makefile`

makefile

```
# Paths
SNAPSHOT_DIR = snapshots
ACTUAL_SUFFIX = .actual.json
SLACK_SCRIPT = ./notifySlack.sh
DASHBOARD_SCRIPT = ./refreshDashboard.sh

# Approve all actual snapshots
approve-snapshots:
@echo "✅ Approving snapshots..."
@for f in $(SNAPSHOT_DIR)/*$(ACTUAL_SUFFIX); do \
base=$$(basename $$f $(ACTUAL_SUFFIX)); \
cp $$f $(SNAPSHOT_DIR)/$$base.json; \
echo "Approved: $$base.json"; \
done

# Alert on snapshot diffs
alert-on-diff:
@echo "🔍 Checking for snapshot mismatches..."
@for f in $(SNAPSHOT_DIR)/*$(ACTUAL_SUFFIX); do \
base=$$(basename $$f $(ACTUAL_SUFFIX)); \
diff $$(dirname $$f)/$$base.json $$f > /dev/null || \
$(SLACK_SCRIPT) "⚠️ Snapshot mismatch detected: $$base.json"; \
done

# Refresh dashboard content
refresh-dashboard:
@echo "📊 Refreshing dashboard..."
```

```
@$(DASHBOARD_SCRIPT)

# Run full regression pipeline
regression-pipeline: approve-snapshots alert-on-diff refresh-
dashboard
@echo "🚀 Regression pipeline complete."
```

📄 refreshDashboard.sh

bash

```
#!/bin/bash
echo "Updating dashboard data..."

cp snapshots/user-42.actual.json dashboard/snapshotDiff.json
cp db-diff.txt dashboard/dbDiff.txt
cp zap-report.html dashboard/zapReport.html
cp perf-summary.txt dashboard/perfMetrics.txt
cp pact-with-metadata.json dashboard/pactMetadata.json

echo "✅ Dashboard updated."
```

🧪 Usage

bash

```
make regression-pipeline
```

This will:

- Approve all `.actual.json` snapshots
- Alert Slack if any diffs are found
- Refresh the dashboard with latest test artifacts

Love that mindset. Here's how we can turn your regression pipeline into a bulletproof, enterprise-grade system—complete with CI/CD triggers, release automation, and local orchestration.

🧪 1. `make test` : Run Full Test Suite + Generate Artifacts

📄 Add to `Makefile`

makefile

```
test:
@echo "🧪 Running full test suite..."
@./gradlew clean test
@echo "📁 Generating regression artifacts..."
@mkdir -p snapshots
@cp build/test-results/*.json snapshots/ # adjust as needed
@echo "✅ Tests complete."
```

🚀 2. `make release`: Tag, Approve, Upload

📄 Add to `Makefile`

makefile

```
release:
@echo "🚀 Creating release..."
@git tag -a v$(shell date +%Y%m%d%H%M) -m "Release $(shell date)"
@git push origin --tags
@make approve-snapshots
@make upload-artifacts
@echo "✅ Release tagged and snapshots approved."

upload-artifacts:
@echo "📦 Uploading artifacts..."
@gh release create $(shell git describe --tags --abbrev=0) \
--notes "Automated release" \
snapshots/*.json zap-report.html perf-summary.txt pact-with-
metadata.json
```

Requires GitHub CLI (`gh`) installed and authenticated.

🧠 3. GitHub Action: Trigger Makefile on Push or Tag

📄 `.github/workflows/ci.yml`

yaml

```
name: CI Pipeline

on:
push:
branches: [ main ]
```

```
tags: [ 'v*' ]

jobs:
regression:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3
- name: Set up JDK
uses: actions/setup-java@v3
with:
java-version: '17'
- name: Run Makefile pipeline
run: |
make test
make regression-pipeline
```

🧩 4. Optional: `make verify` for Local Snapshot Diffing

makefile

```
verify:
@echo "🔍 Verifying snapshots..."
@for f in $(SNAPSHOT_DIR)/*$(ACTUAL_SUFFIX); do \
base=$$(basename $$f $(ACTUAL_SUFFIX)); \
diff $$f $(SNAPSHOT_DIR)/$$base.json || echo "❌ Diff found in
$$base"; \
done
```

📦 Final Structure

Code

```
project-root/
├── Makefile
├── notifySlack.sh
├── refreshDashboard.sh
├── snapshots/
├── dashboard/
├── pact-with-metadata.json
├── zap-report.html
├── perf-summary.txt
├── db-diff.txt
```

```
├── .github/workflows/ci.yml
```