AS02: Testing, Security & Expanded Application 🚨

Introduction:

In this assignment, you will further develop the `musicFinder` application by:

- Implementing unit tests,
- Performing static and dynamic security testing, and
- Expanding the application's functionality.

The goal is to introduce testing methodologies and security practices which are crucial in modern software development.

▼ Task 2.1: Unit Testing with `JUnit5` [10 marks]

Goal:

You will write unit tests to verify the functionality of the `musicFinder` application's **API layer** for fetching song lyrics, and handle error cases with invalid inputs are provided.

Instructions:

1. **Set Up `JUnit`**:
    - Ensure `JUnit5` is added as a dependency in your `pom.xml`:
2. **Write Unit Tests for the API Layer**:
    - Test the `/song/{artist}/{name}` endpoint to verify that:
        - **Valid inputs** return the correct lyrics.
        - **Invalid inputs** (e.g., unknown artists / songs, or null, or other relevant edge cases) return the appropriate error messages.
    - Ensure the proper HTTP status codes are returned (e.g., `200 OK` for valid responses and `404 Not Found` for errors).
    - Use Mock if necessary.
3. **Skeleton Code**:

- Place your tests under `src/test/java/com/example/musicFinder/controller/` in a class such as `SongControllerTest.java`.
- Sample code:

```
public class SongControllerTest {
    @Test
    public void testFetchLyrics_ValidSong() {
        // Add code to test valid artist/song requ
est
    }

    @Test
    public void testFetchLyrics_InvalidSong() {
        // Add code to test invalid artist/song re
quest
        // and error handling as well
    }
}
```

- You can include a new exception class (if required).

4. **Run Tests**:
   - Use Maven to run the tests locally.
5. **Integrate with GitHub Action**:
   - Ensure the unit tests run automatically in the CI/CD pipeline whenever code is pushed.

Submission:

- `JUnit` **test cases** in the `test` directory for testing the API layer and error handling.
- **GitHub Action workflow** showing successful test runs as part of the build process.

▼ Task 2.2: Static Code Analysis with `SonarQube` [10 marks]

Goal:

Integrate `SonarQube` to perform static code analysis on the `musicFinder` application.

You'll identify code quality issues, security vulnerabilities, and technical debt, and take steps to improve the application based on the findings.

Instructions:

1. **Set Up `SonarQube`**:
    - If you are using a `SonarQube` **Cloud** instance, follow the integration steps to link your GitHub repository to `SonarQube`.
    - If using a **local SonarQube instance**, install `SonarQube` locally and configure it to scan your codebase.
2. **Analyse the Codebase**:
    - Run `SonarQube` to scan the `musicFinder` application.
    - Focus on identifying:
        - **Code Smells** (inefficient or non-standard coding practices).
        - **Security Vulnerabilities** (e.g., missing input validation).
        - **Duplicated Code** and **Complexity**.
3. **Review and Fix Issues**:
    - Identify at least **TWO major code smells** or **security vulnerabilities**.
    - Refactor the application code to resolve these issues, ensuring the next SonarQube scan reflects improvements.
4. **Integrate `SonarQube` with GitHub Action**:
    - Modify your GitHub Actions pipeline to trigger a SonarQube scan on every push.
    - Run SonarCloud if appropriate.

💡 **Submission:**

- `SonarQube` **Report** detailing the issues found and the steps taken to resolve them, (i) initial report, and (ii) after report - add reports folder to repository.
- GitHub Action workflow with `SonarQube` **configuration.**

▼ Task 2.3: Dynamic Security Testing with `OWASP ZAP` [10 marks]

Goal:

Use `OWASP ZAP` to perform dynamic security testing on the `musicFinder` application.

This task will help identify web vulnerabilities, such as **SQL injection** or **Cross-Site Scripting (XSS)**, that could compromise the security of your application.

Instructions:

1. **Set Up** `OWASP ZAP`:
   - Install `OWASP ZAP` locally or use the Docker image for ZAP.
2. **Run a Security Scan**:
   - Perform a **full scan** of the `musicFinder` application's endpoints (e.g., `/song/{artist}/{name}`).
   - Focus on identifying common vulnerabilities like **SQL injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF) - if any.**
3. **Review and Fix Vulnerabilities**:
   - Address at least **one critical vulnerability** found during the scan (e.g., ensuring input validation on the song and artist fields).
   - Update the application code to mitigate these vulnerabilities.
4. **Generate the** `OWASP ZAP` **Report**:

- After fixing the vulnerabilities, rerun the scan and generate a new report to confirm that the issues are resolved.

5. **Integrate `OWASP ZAP` with GitHub Action**:
   - Modify your GitHub Actions pipeline to trigger a SonarQube scan on every push.

> 💡 **Submission:**
>
> - `OWASP ZAP` **Reports** showing the vulnerabilities found (initial report) and the latest report after you've performed the necessary steps to mitigate them.
> - Updated GitHub repository with changes reflecting the fixes for the security vulnerabilities.

▼ Task 2.4: Expanded Application Functionality: Artist Information via Wikipedia API [10 marks]

Goal:

Expand the functionality of the `musicFinder` application by adding a new feature that fetches artist information (e.g., biography) using the **Wikipedia API**.

This feature will allow users to retrieve details about the artist without needing an API key.

Instructions:

1. **Add the Wikipedia API Endpoint**:
   - Create a new **controller** method in `src/main/java/com/example/musicFinder/controller/ArtistController.java`.
   - The endpoint should handle requests to `/artist/{name}` and fetch artist information using Wikipedia's API:

```
https://en.wikipedia.org/api/rest_v1/page/summary/{name}
```

2. **Skeleton Code**:
    - You can add the following into your controller:

```
@GetMapping("/artist/{name}")
public ResponseEntity<String> getArtistInfo(@PathV
ariable String name) {
    String url = "https://en.wikipedia.org/api/res
t_v1/page/summary/"
        + name;
    ResponseEntity<String> response = restTemplat
e.getForEntity(url,
        String.class);
    return ResponseEntity.ok(response.getBody());
}
```

3. **Update `index.html`**:
    - Modify the `index.html` file to include a section that displays the artist information.
4. **Test the Endpoint**:
    - Test the new `/artist/{name}` endpoint by making a `GET` request to it.
    - For example:

```
GET /artist/Coldplay
```

    - Ensure the response contains a summary of the artist retrieved from Wikipedia.
5. **Error Handling**:
    - Ensure the application responds correctly when the artist does not exist on Wikipedia, returning an appropriate error message and HTTP status code (`404 Not Found`).
6. **Add a run a `curl` command in your GitHub Action**:
    - Modify your GitHub Actions to trigger a `curl` command to test your new API.

Submission:

- **Updated** `index.html` with the expanded functionality to fetch artist information using Wikipedia's API.
- Add a `curl` command inside your GitHub Action workflow for the new API.
