Artist Portfolio

Developer: Joshua Reyes
Date submitted: 05/05/2024
Reviewer: Parth Thanki
Date Reviewed: 05/05/2024

Major positives

- Extensive Handling of Use Cases: The codebase covers all necessary features for an artist's portfolio. Users can upload audio tracks, clips, and demos while managing and customizing metadata for their role as composers, arrangers, or instrumentalists. Portfolios can be public or private, providing a professional touch.

- Effective Logging: The logging system maintains comprehensive records for every operation, successful or failed. This allows developers to trace event sequences and helps administrators ensure accountability and compliance, making debugging easier.

- Layered Structure of Code: The codebase uses a logical layering system to separate concerns. It includes a controller layer for API requests/responses, a service layer for business logic, and a data access layer. This modular structure improves testability and maintainability.

- Proper Exception Handling: Exceptions are properly managed, offering user-friendly error messages while providing developers with full tracebacks for debugging. This ensures end users receive clear feedback while developers can identify issues quickly.

- Secure File Storage: Uploaded audio files are stored securely, with users having exclusive access to their content. By separating file slots and metadata, content is linked to the correct user portfolios.

- Quite Scalable: The backend design is scalable to handle growth, using SSH for file transfers and a relational database for metadata. This structure supports increasing loads efficiently.

Major negatives

- Poor Validation of Inputs: Despite the system having rich metadata and slot management, the current software lacks rigorous user input validation. Audio file uploads need to be closely scrutinized to ensure they're in supported formats and appropriate lengths, preventing corrupted files from being saved. Metadata fields also need discipline in preventing incomplete or improper data storage.

- Potential Security Issues: Dynamic SQL queries at the data access layer pose a significant threat. These queries, if not properly parameterized, are susceptible to SQL injection attacks that can lead to data manipulation or breaches. Inputs

need thorough sanitization and validation at each level to ensure overall system security.

- Inconsistent Naming of Methods: The ArtistPortfolioDao class has inconsistencies in method naming, reducing code readability and complicating navigation. For instance, "DeleteFilePath" should be renamed for clarity, aligning with other classes. Standardizing naming conventions across the codebase will improve developer experience and code maintainability.

- Lack of Documentation: Core methods lack clear documentation regarding parameters, return values, and edge case handling. This lack of documentation makes it difficult for new developers to grasp the code's implementation and underlying business logic. Good comments, especially for data access methods and business logic, should clarify each purpose and implementation detail.

- Logging and Error Handling Lack Consistency: While logging is generally well-implemented, error handling across different modules and logging practices remain inconsistent. Some logs are too sparse, while others are overly verbose. Certain methods don't handle exceptions properly, resulting in vague errors for users. Standardizing these practices will improve debugging and user experience.

- Rigid Architecture: Although the current architecture is layered, certain areas are rigid. Some methods hardcode dependencies and are tightly coupled with implementation details, making future feature expansion or changes difficult. More abstraction and modularity would increase scalability.

- Proper Feedback: Users lack clear feedback for actions like file uploads or metadata updates. Concise and accurate messages for successful or failed actions will enhance user understanding and engagement with the system.

Unmet Requirements

- Testing of Area:
    - A significant testing gap exists in the front end, making UI validation challenging. Portfolio uploads, updates, or views may have underlying issues that go undetected, possibly explaining the absence of expected error messages or confirmations. Comprehensive tests using tools like Selenium and Cypress are necessary to ensure proper validation and accurate user feedback.

    - The backend tests also fail to cover new features and edge cases, leading to potential errors and unhandled scenarios. Updating tests to include all create, read, update, and delete operations is crucial to verify proper feature behavior. This can be achieved by writing unit tests for methods in the ArtistPortfolioDao and ArtistPortfolio classes and using integration tests to confirm database integrity.

- User Interface Feedback:
    - The current design lacks sufficient feedback mechanisms for portfolio actions, possibly confusing users. For example, when an audio file is uploaded, users should receive immediate confirmation or error messages. Adding frontend alerts will keep users informed of action outcomes, while detailed backend logging can aid in debugging.

- Scalability Issues:
    - Performance bottlenecks may emerge as the number of portfolios and uploads increases. Asynchronous processing for file uploads or metadata updates can significantly improve responsiveness while caching frequently accessed data will reduce the database load.

- Error Reporting:
    - Current error handling doesn't provide sufficient diagnostic information for developers. Improving this involves structured exception types, more detailed logging of exceptions, and attaching error IDs that customer support can reference.

Design Recommendations

- Input Validation
    - Implement robust validation for audio file uploads and metadata. Use a centralized validation layer to ensure data integrity checks are consistent across modules. Specific validation should enforce format adherence and character limits for the audio uploads (see page 10) to avoid data corruption or errors during processing.

- Log Standardization
    - Ensure logging is standardized across classes and components like ArtistPortfolioDao. Include essential context such as timestamps, user IDs,

and error categories. This will facilitate comprehensive debugging of issues like failing to update portfolio sections (page 5). Additionally, standardized logs will help adhere to organizational standards.

- Use Parameterized Queries
    - Replace dynamic SQL queries used in ArtistPortfolioDao (page 2) with parameterized ones to prevent SQL injection attacks. Adopting an Object-Relational Mapping (ORM) framework like Entity Framework or Dapper will further strengthen security.

- Improve Documentation
    - Add comprehensive docstrings to all key functions, especially in classes like ArtistPortfolioDao, to detail their purpose, parameters, and return values. The DeleteFile method (page 1) should have detailed explanations to help new developers understand its functionality and potential edge cases. Provide a high-level overview of each module and class in a separate documentation file.

- Improve Exception Handling
    - Introduce specialized exception types to differentiate errors. For instance, ArtistPortfolioDao should use exceptions to distinguish between errors in updating portfolio info and network failures (page 12). This will lead to more precise error messages and troubleshooting.

Test Recommendations

- Frontend Testing:
    - Consistent UI: Ensure a consistent look and feel across browsers and devices.
    - Functionality Validation: Test buttons, forms, and navigation for functionality with scenarios like metadata updates, file visibility changes, and occupation selection.
    - Error Handling: Verify error messages are displayed for conditions like unsupported file formats or connection problems.

    - Accessibility Testing: Validate screen reader compatibility, keyboard navigation, and color contrast for impaired user accessibility.

- Backend Testing:
    - Unit Tests: Write unit tests for backend functions to handle scenarios like database errors, missing parameters, or incorrect data formats.
    - Validation Tests: Validate incoming data (e.g., file uploads, metadata updates) and ensure errors are raised for invalid inputs.
    - SQL Injection Testing: To identify potential vulnerabilities, create tests that exploit unsanitized data inputs.
    - Performance Testing: Conduct load testing to verify the system can handle concurrent portfolio uploads and retrievals within a 3-second limit.

- Integration Testing:
    - API Functionality: Test the entire API functionality, ensuring data is saved, updated, and retrieved accurately.
    - End-to-End Flows: Simulate real-world workflows, ensuring portfolio uploads, metadata updates, and visibility changes work seamlessly together.
    - Role-Based Testing: Verify users only access, edit, and view portfolios authorized for their roles (e.g., talent role, admin role).
    - Regression Testing: Establish a regression test suite to prevent new changes from breaking existing functionality.

- Continuous Integration Testing:
    - Incorporate automated tests into the CI/CD pipeline to catch potential issues early and maintain a high-quality codebase.