StratoSplit

Software Testing Plan

April 4th, 2025

Client:

General Dynamics Mission Systems

Mentors:

Brian Donnelly, Savannah Chappus

Team Members:

Sam Cain

Nolan Newman

Dallon Jarman

Elliot Hull

Revision 1.0

Table of Contents

Introduction

Effective communication is paramount in defense, public safety, and intelligence operations. In high-stakes environments, rapid and reliable information exchange is critical for mission success. Leveraging General Dynamics Mission Systems (GDMS) Coast Guard distress location system, Rescue 21, the StratoSplit project aims to address the challenge of inefficient communication with radio modems on mobile devices. Our team is developing a Node.js web application, StratoSplit, to simulate audio generation and transmission to a dashboard for real-time monitoring.

In order to ensure our software's effectiveness, we will implement multiple forms of testing within our code. Unit testing forms the first line of defense, running simple tests against each of the functions utilized in our main javascript code. Each user input will be tested with this method, to ensure stability even in the event of invalid inputs. This testing can be automated with Jest and python to run automatically upon submission of code to git. The coverage of the code will similarly be generated automatically. Other modules of the code will be emulated using built in tools, to keep unit tests limited to their specific functions.

Integration testing will use simulation of different parts of our application, such as the web interface, or the audio generator that it must connect to, to ensure robust connections between the large modules our project is made of. Instead of mocking other parts of our code, integration testing will use the real modules, to effectively test if everything works together as expected.

Our usability testing will be conducted with the members of our team, running the application while accessing it from multiple devices and accounts simultaneously, and running through our user use-cases to check for ease of use and how clearly we have labeled our interface. At regular intervals, our mentor will also use our application to provide feedback and find bugs.

Jest is a clear choice to make, as it is both important to our team and to our client that the code we create has zero trust, and is able to detect if any updates lead to breaks in the larger code base. Along with Jest, python's built in testing methods will also be used. With a combination of testing practices, each layered on top of each other, we can help ensure the application is robust against breaking during use, and breaking upon being updated.

Unit Testing

Unit testing refers to a simple testing method that passes values into functions and checks for a specific result. This method's simplicity is also its strength; by running many edge cases though as parameters, unit testing can weed out odd results or inconsistencies in the code and allow hard-to-find bugs to be detected and resolved before they are ever combined with the larger code base. Our aim for a larger testing base would be to reach 70% coverage which will ensure a solid and effective test of the codebase.

Our project utilizes the Jest package to run our unit tests for all javascript files, and this provides a solid foundation to build thorough testing procedures. Jest allows for a comprehensive simulation of different parts of the application, for example; creation of virtual databases, which can then provide the necessary inputs to run tests without needing to worry about the status of the actual database. This siloing of different parts of the project makes it easier to find exactly what files are responsible for any issues that may pop up. Jest also has built in tools that can measure metrics like code coverage, making keeping track of progress with the code testing fairly simple. There is even a feature to generate a html page with code coverage across multiple files. Together these tools form the basis of the code tests.

The basic structure for Jest includes a matching *.test* file for each JavaScript file, in order to run test cases on different parts of the initial, matching file. For our projects JavaScript files, one example of the unit tests include passing in server requests as the units, and checking for web codes that indicate success or failure. A successful request would return *200*, and therefore a success, passing the test. However, the tests also pass through parts of the page that don't exist, and expect a *404* code, meaning that the result wasn't found. This is also important, since a false negative is just as important to detect as a false positive.

Another specific example would be the audio processor interface, which is responsible for writing the audio data to the buffer. Jest allows for recreating a virtual form of the data buffer, to easily measure if data is being written correctly as expected. These functions are relatively easy to test with unit testing, since they include making sure data can be passed through as a parameter and be written into another variable. There are many values that could be passed through this function, so the tests simply check if the values passed equal the resulting values.

For the client file, unit testing is able to pass through different status codes for UI elements on the webpage, to a mocked implementation of the screen. These mocked implementations can then check for changes in the unit tests, to make sure that the functions affecting the client's view are being properly affected. Sending commands for

toggling the channel buttons can be checked by reading the new values of the functions output.

In addition to our core JavaScript codebase, our project includes a small, yet critical, Python module responsible for audio packetization and transmission. For this component, we use Python's unittest framework to test key functionality such as the 'create_rtp_packet' function, which builds RTP headers and appends audio payloads. We validate header formatting, boundary values for sequence numbers and timestamps, and response to invalid input. Other functions such as 'start_streams' and 'stop_streaming' are tested with mocks to ensure correct threading behavior and global state management. We do not intend to unit test the full 'stream_audio' function due to its reliance on sockets and real-time audio transmission and processing, yet we plan to isolate and verify its core logic through these mock ups. These tests can be integrated into our Jest-based pipeline via subprocess checks, ensuring reliability across both languages and machines.

Input Field	Valid Inputs	Invalid Inputs
User-provided email	Must include "@" and a valid domain, like ".com" or ".edu"	Any address that lacks the "@" symbol, lacks a valid domain, or lacks a username
Team Name	Must have something in the field	A blank field
Audio stream generator - file input	.mp3 file	Any other input file type
Audio stream generator - duration input	Integer number between 1 and 1000 seconds	Any input other than positive numbers, or numbers above 1000

For user inputs, checks must be implemented in case invalid inputs are provided. For user email validation, a third party service is used, called Hanko. This service sanitizes inputs all on its own, and prevents a user from inputting a false email. If the user returns a value that isn't allowed, it will reject the data, and ask for the user to provide a valid email.

In the admin panel, the user is able to set a team name. This name can be nearly any value, however, the user must provide a value of some sort to be viewed. An empty field will be rejected, and the user will be asked to enter a valid name. Advanced symbols like emojis will not be supported, but any other special characters or numbers are allowed, as these characters can often be useful to differentiate different teams from

each other, and allow the administrator to set their own system for identifying the teams from each other.

For the access panel for the audio stream generator, .mp3 files must be provided to be played. As a first line of defense against invalid files being passed through, the application will check if the file name has ”.mp3” as a suffix. However, since this filename can be changed relatively easily, additional checks will be put in place for error detection, and reflect files that were either named incorrectly or corrupted in a way that makes them unplayable.

Additionally, the audio generator can play the audio files for different durations. The duration field will only allow for positive integers, and these numbers have to be below 1000. Since this input takes in seconds for the audio generator to play, only whole numbers and positive numbers will make any sense. Any other value will simply be rejected, and the audio generator will not play the audio files.

Integration Testing

Integration testing verifies that all the crucial components of our audio streaming dashboard communicate with one another as intended. Because our application is a blend of several technologie such as Express.js, MongoDB, Hanko Authentication, Node.js, and Python. It is crucial to verify proper data passing between these modules. This involves verifying interactions such as API calls, authentication, database functionality, and audio streaming. In so doing, we ensure that every layer properly receives and interprets the data it requires, thus minimizing the potential for data corruption, performance bottlenecks, or unanticipated failures.

There are several primary objectives of integration testing in this regard. First, we need to confirm data exchange: sending the right data from the frontend to the backend, from the backend to the database, and from the Python audio processor to the frontend. This requires that data be formatted properly, processed without errors, and sent to the right places. Second, we must confirm the authentication pipeline by ensuring that Hanko natively integrates with Express.js, protecting secured routes as intended and only allowing authorized users to see sensitive endpoints. Third, we must confirm the audio streaming pipeline by ensuring that the system can process multiple concurrent streams of audio without data corruption or slowdown. Fourth, we must test database interactions to ensure MongoDB updates and queries occur correctly and respond to errors gracefully. Finally, we must ensure error handling mechanisms catch invalid requests, authentication errors, and database disconnection without affecting system stability as a whole.

Testing begins with API endpoint testing to ensure Express.js routes behave as expected with Hanko, MongoDB, and the Python audio stream generator. This stage entails checking that endpoints receive the correct parameters, output the correct responses, and respond correctly to unexpected inputs. After this, we do authentication flow testing to confirm correct processing of valid and invalid Hanko tokens and proper processing of user sessions, including correct login and logout. Then we perform database integration testing by performing multiple operations in MongoDB with the help of Express.js to make sure data is being stored, fetched, updated, and deleted as required. After that, we perform concurrency testing where we stream audio from multiple users concurrently to make sure the system can handle multiple streams in parallel without crashing, being too latency-prone, or losing data.

To turn these plans into concrete action, our Integration Test Cases begin by verifying user authentication and access to protected routes, where integration points are the frontend, Hanko Auth, and Express.js middleware. We verify a login request is successful with valid Hanko credentials and that the Express.js middleware correctly verifies the session token, rejecting unauthorized requests and granting only authenticated users protected route access. The second test scenario validates the audio stream request and processing chain, highlighting integration points between the frontend, Express.js, and the Python audio processor. Here we are making requests to trigger audio streams, validating that Express.js forwards those requests to the Python service, and that the service returns valid streamable URLs. We also validate concurrency in the system by having multiple users request a different stream. The third test case addresses MongoDB data persistence and retrieval with focus on Express.js and MongoDB integration. We create, query, update, and delete stream records in the database and validate error scenarios such as invalid queries or DB connection issues to yield appropriate HTTP status codes and not take down the system.

Verification and validation across all test cases will involve testing HTTP status codes to verify good responses (i.e., 200 for a successful response, 401 or 403 for unauthorized access, and 500 for server faults). We will also verify returned data structures like JSON schemas and stream URLs, and test the performance of systems so that we have no noticeable lag or system crashes during load testing. Any errors caught during this process—ranging from failed database queries to authentication rejections—will be logged and analyzed to enhance stability.

We plan to rely on two primary frameworks for verification: Jest for JavaScript-based assertions and mocking, and Python’s built-in unit test for testing audio-processing logic. This approach allows us to thoroughly validate the integrity of our Express.js routes with Jest while simultaneously ensuring the Python audio components function correctly through unittest. Both of these methods of testing have built-in methods for returning the results of tests in a format that allows us to go to the

specific test, and fix the problem. These systems also include ways to check coverage, and we will be aiming for about 70% testing coverage at least, at the direct request of our client.

Through this integration test plan, we expect to give confidence that our audio streaming dashboard plays together well on all components. We will validate authentication, database transactions, and audio processing under real-world scenarios, thereby enhancing our confidence in an expandable and stable system. In the future, future steps involve implementing the in-depth test cases, incorporating automated testing into a CI/CD pipeline such as GitHub Actions, and expanding edge-case testing based on initial results to refine overall performance and reliability.

Usability / End-user Testing

Usability testing is a form of non-functional software testing focused on evaluating how intuitive, efficient, and satisfying a software system is for end users. It explores how real users interact with the application, identifies barriers to accomplishing tasks, and ensures that the software interface meets the needs and expectations of its target audience. The goal is not only functional success but user satisfaction, workflow efficiency, and overall ease of use.

For our project, which is a web-based administrative audio console, usability is particularly important. The application is designed for system operators, admins, or coordinators who need to manage, monitor, and configure multiple audio streams with minimal technical training. Therefore, it’s crucial that the interface is clear, responsive, and resistant to error.

Our application has several unique characteristics that shape our usability strategy. First, real-time responsiveness is essential. Users need to see immediate feedback, such as toggles changing state or traffic highlights appearing as audio streams are activated or deactivated. Second, our user base is diverse. Some users may be technically proficient, while others are team leads or managers with limited technical background. The system must be easy to understand regardless of experience level. Third, the system may be used in high-pressure environments where admins must act quickly. Complex workflows could cause delays in urgent scenarios. Finally, the interface should have a low learning curve. It should be operable by someone with minimal training within a few minutes of first exposure.

To ensure our software delivers a smooth and intuitive user experience, we structured our usability testing into three core sessions: a Client Review Session, a Mentor Review Session, and Acceptance Testing with Realistic Scenarios. Regarding the timeline, we intend to hold the Client Review session on April 14, 2025, the Mentor

Review Session on April 11, 2025, and finally the Acceptance Testing on April 25, 2025. These sessions span internal evaluation, expert review, and actual usage simulation. Each is tailored to uncover pain points from a unique perspective, ensuring that we capture a holistic understanding of usability effectiveness and areas for improvement.

The Client Review Session involves real users - especially those resembling operators or administrators and familiar with the current system we intend to replace. They will interact with the software in a semi-guided format. These users are asked to complete common tasks such as toggling audio channels, adjusting sliders for volume and panning, using mute and clear buttons, saving configurations, and loading them back into memory. Users are observed without interruption, and their reactions, hesitations, and comments will be documented.

This session provides insight into first-time use experiences. It allows us to identify parts of the interface that may not be as intuitive as expected. Some users may struggle with less-visible elements such as the configuration dropdown, or misunderstand buttons with similar icons. This test is also an opportunity to analyze user confidence, and observe whether the interface promotes self-recovery from minor errors without outside help.

In addition to user testing, we plan to use think-aloud techniques where participants are encouraged to verbalize their thought process while using the system. This often reveals confusion, assumptions, and usability issues that might not be captured through observation alone. We will also conduct short interviews after each session to gather impressions on the overall ease of use, satisfaction, and perceived usefulness of features.

The Mentor Review Session offers a more informed perspective. Our mentor, who is familiar with graphical design and animation, explores the system freely and highlights UI/UX inconsistencies, inefficiencies, or risks. The mentor evaluates both form and function; how accessible features are, whether they are logically grouped, and whether workflow expectations are met without excessive clicks or backtracking.

Our mentor may also test against edge cases, intentionally using features out of sequence or providing invalid input to test the interface’s robustness. Their feedback tends to influence larger structural changes or help identify subtler design flaws that may not be noticed during client testing.

Because the mentor understands our architectural and technical constraints, their recommendations help balance ideal usability with practical implementation limits. This session also supports identification of high-priority issues that might affect both user experience and maintainability of the application. By combining UX principles with real

development awareness, the mentors review bridges the gap between design goals and production feasibility.

The Acceptance Testing with Realistic Scenarios is conducted after we've made the necessary revisions and believe the application is ready to use. This test involves simulated use cases like creating teams, assigning channels, starting and stopping audio transmission, and using the admin panel to manage users. These scenarios are drawn from actual tasks that would occur in live deployment, ensuring our test reflects real operational complexity.

The objective is to verify that end-to-end processes can be completed smoothly under real-world conditions. We measure success based on task completion rates, user confidence scores, and absence of critical breakdowns in the workflow. The acceptance test is comprehensive, testing a complete usage cycle rather than isolated features. This allows us to validate the system's readiness for full deployment and determine whether final refinements are necessary.

In all usability sessions, we will collect both quantitative and qualitative data. This includes task success rates, time on task, observed errors, and post-session interviews. The interviews often reveal pain points not immediately apparent during the test, such as mental models that did not match our design assumptions or unexpected workflow interpretations

We will employ usability scoring methods like the System Usability Scale and Likert-based satisfaction surveys to obtain numerical comparisons between sessions. These provide benchmarks we can use across versions and help quantify user experience improvements over time. Aggregation of scores and analysis of trends will help prioritize interface enhancements and focus efforts on features that impact usability most.

These sessions are essential in shaping an application that feels polished, intuitive, and responsive under real usage. With a foundation of direct user insight, we can deliver a system that is both powerful and easy to use.

Our goal with usability testing is not only to eliminate functional bugs but to enhance the overall experience for the user. These sessions are essential in shaping an application that feels polished, intuitive, and responsive under real usage. With a foundation of direct user insight, we can deliver a system that is both powerful and easy to use - meeting our goals for accessibility and operational performance in demanding environments.

Conclusion

The StratoSplit testing plan is a layered and rigorous framework designed to validate reliability, performance, and usability of our audio streaming application in a high-stakes operational environment. Each stage of the plan - unit testing, integration testing, and usability testing - has been selected and executed with the specific characteristics and demands of our project in mind.

Unit testing serves as our foundation, enabling us to isolate and verify the correctness of individual functions and components in a controlled setting. These tests give us rapid feedback during development, help eliminate logic errors early, and support continuous delivery by ensuring that new changes do not break existing functionality.

Integration testing addressed a more complex concern: the interaction between major subsystems such as Express.js, MongoDB, Hanko Authentication, and the Python-based audio processing layer. Given the multi-technology nature of our application, ensuring that data flowed smoothly between services was critical. Through mock simulations, real API interactions, and concurrency testing, we validated the resilience of our infrastructures and the security of our access controls.

Usability testing ensured that all technical success translated into a product that was accessible, efficient, and comfortable for our target users. Through structured review sessions with clients, mentors, and simulated real-world scenarios, we captured critical feedback and identified areas for improvement. Our emphasis on end-user experience guarantees that our application does not just work, but works well for those who rely on it.

Together, these testing methodologies formed a complete quality assurance cycle. By addressing correctness, compatibility, and end user satisfaction in parallel, we will have prepared StratoSplit for robust deployment in environments where dependability and efficiency are paramount. Our testing plan has not only increased our team’s confidence in the software but has also created a feedback loop that will continue to guide development, refinement, and future scaling of our software.

Through systematic validation and thoughtful iteration, we believe this software testing strategy lays the groundwork for a stable, effective, and user-friendly system that aligns with the goals of General Dynamics Mission Systems, the Rescue 21 team, and the broader mission it supports.