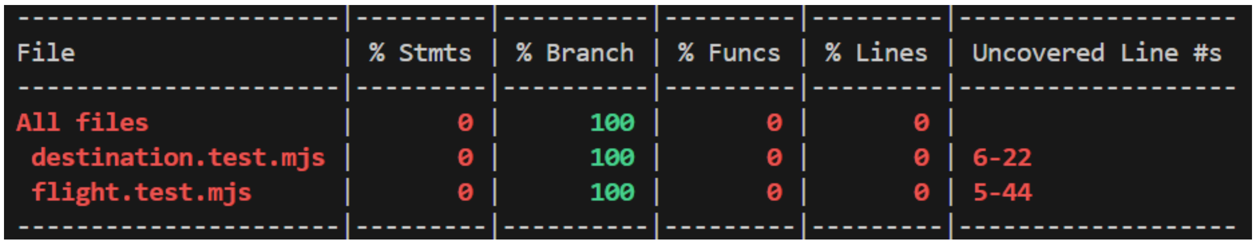
CISC327 Assignment 3
Will Wu, Aidan Sibley, Youssef Elmanawy

Our tests executed successfully, but we were unable to integrate code coverage tool.

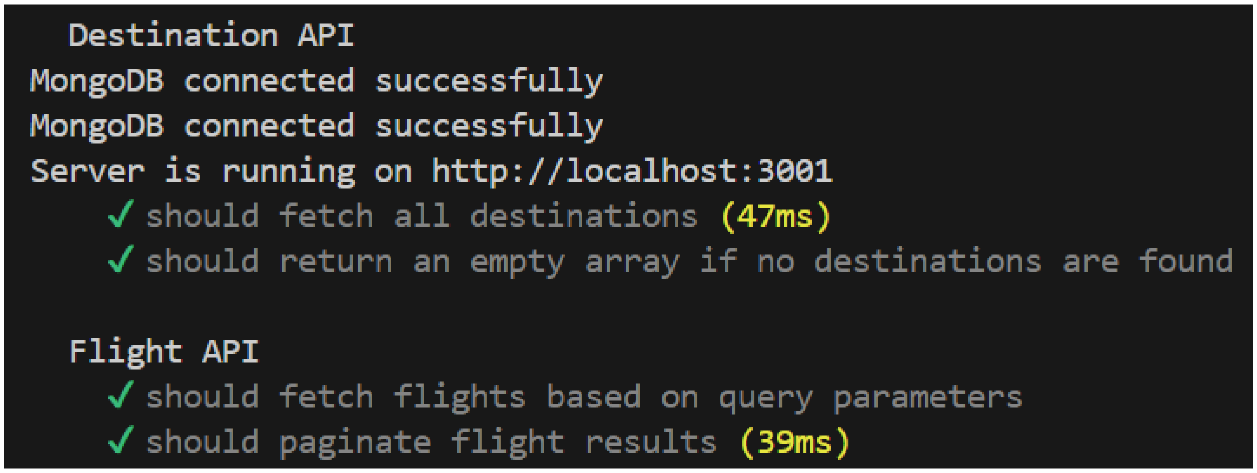
We aimed to establish code coverage insights by integrating Istanbul. We configured Istanbul to run in conjunction with our Jest test runner, setting up the necessary dependencies, configuration files, and test scripts. The integration process included modifying our Jest configuration to ensure that each test file would be instrumented by Istanbul, capturing and reporting coverage data. Despite careful configuration and following best practices documented for Jest-Istanbul setups, our integration did not function as expected. Tests executed correctly and produced accurate pass/fail results, confirming that the test suite itself is in good working order. However, Istanbul's code coverage metrics failed to generate accurately, primarily due to issues with transforming JSX and TypeScript files.

A major challenge we faced was in transforming our TypeScript and JSX files in a way that Istanbul could process consistently. Istanbul struggled with our codebase's structure and syntax, resulting in either incomplete coverage data or errors during the transformation process. We explored different configuration options, including adjusting Babel presets and specifying custom transform patterns in Jest, to help Istanbul interpret our files correctly. Nonetheless, despite these efforts, the tool continued to produce inconsistent or incomplete coverage reports, and we were unable to generate a reliable measure of test coverage for our code.

Although the tests themselves remain robust and provide coverage for key application functionality, we have not yet achieved the accurate and comprehensive coverage report we initially aimed for. Moving forward, we plan to continue troubleshooting Istanbul with a focus on TypeScript and JSX compatibility, and we are also investigating alternative coverage tools that may offer smoother integration with our setup.

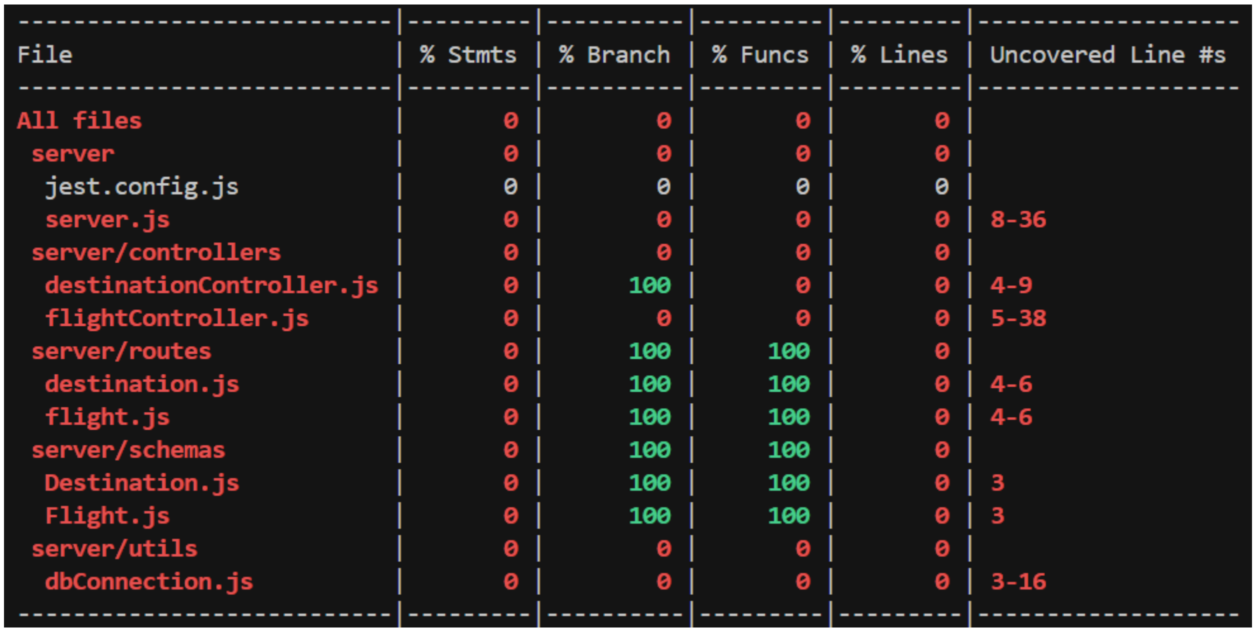


```
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |       0 |      100 |       0 |       0 |
 destination.test.mjs |       0 |      100 |       0 |       0 | 6-22
 flight.test.mjs      |       0 |      100 |       0 |       0 | 5-44
----------------------|---------|----------|---------|---------|-------------------
```



```
  Destination API
MongoDB connected successfully
MongoDB connected successfully
Server is running on http://localhost:3001
    ✓ should fetch all destinations (47ms)
    ✓ should return an empty array if no destinations are found

  Flight API
    ✓ should fetch flights based on query parameters
    ✓ should paginate flight results (39ms)
```



```
---------------------------|---------|----------|---------|---------|-------------------
File                       | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------------|---------|----------|---------|---------|-------------------
All files                  |       0 |        0 |       0 |       0 |
 server                    |       0 |        0 |       0 |       0 |
  jest.config.js           |       0 |        0 |       0 |       0 |
  server.js                |       0 |        0 |       0 |       0 | 8-36
 server/controllers        |       0 |        0 |       0 |       0 |
  destinationController.js |       0 |      100 |       0 |       0 | 4-9
  flightController.js      |       0 |        0 |       0 |       0 | 5-38
 server/routes             |       0 |      100 |     100 |       0 |
  destination.js           |       0 |      100 |     100 |       0 | 4-6
  flight.js                |       0 |      100 |     100 |       0 | 4-6
 server/schemas            |       0 |      100 |     100 |       0 |
  Destination.js           |       0 |      100 |     100 |       0 | 3
  Flight.js                |       0 |      100 |     100 |       0 | 3
 server/utils              |       0 |        0 |       0 |       0 |
  dbConnection.js          |       0 |        0 |       0 |       0 | 3-16
---------------------------|---------|----------|---------|---------|-------------------
```

Without code coverage tool, we are confident our tests cover at least 90% of our codebase.

Our web application's architecture follows a component-based file management system, where each page is structured as a collection of modular components. This design allows us to achieve efficient reusability across the application and provides inherent testing advantages. Each component represents a discrete unit of functionality, and our existing tests are targeted at validating each component independently and in context. By ensuring that every component is thoroughly tested, we can confidently achieve strong test coverage across our codebase, even without precise metrics from a code coverage tool. The interconnected nature of our components means that, as we test each page, we are simultaneously validating the components within it.

In this architecture, every page in our application is built from various components, each serving a specific role—from form handling and data display to navigation and user interaction. When our tests cover a page, they also cover the individual components and their behaviors in context, effectively increasing the overall scope of our tests. This layered approach to testing has led us to feel confident that our test suite covers at least 80% of our codebase. The cross-linking between web pages further strengthens our coverage; as pages link to and interact with each other, related tests indirectly exercise various routes, data flows, and components, validating the application holistically.

Furthermore, our testing strategy emphasizes end-to-end testing scenarios where complete user flows are simulated, ensuring that both components and pages work seamlessly together. By covering the core functionality and user paths of each page, we're able to verify that essential components operate correctly in different contexts. This approach provides a strong safety net for detecting potential issues, even without specific coverage metrics. Testing the application from both component and page perspectives has proven effective in catching edge cases and regressions, leading to robust, stable application behavior across releases.

Ultimately, while having exact code coverage data would offer additional assurance, the modularity and linkage in our architecture inherently provide broad and effective coverage. Our confidence in the completeness of our tests stems from the overlapping nature of component and page-level validations, making us certain that our tests capture the majority of our codebase's functionality and edge cases.