Dispute Resolution Engine

Testing Policy

Introduction of the Project

Alternate Dispute Resolution (ADR) provides a way for parties to negotiate disputes without involving the judicial system. Conventional litigation processes are often costly, time-consuming, and can vary significantly across different companies and domains. By automating the processes involved, using custom workflow generation and NLP enhancements, the Dispute Resolution Engine (DRE) aims to drastically increase the speed and cost-effectiveness of ADR. DRE is designed to handle a large number of disputes simultaneously, offering a more efficient and scalable solution compared to traditional methods.

Testing Policy Introduction

The purpose of this Testing Policy is to define the approach and guidelines for testing the Dispute Resolution Engine (DRE). Testing ensures that all functionalities perform as expected and meet the required standards in terms of performance, reliability, usability, security, and maintainability. This policy outlines the procedures for unit, integration, and end-to-end (E2E) testing to confirm that the system operates seamlessly in a production environment.

Unit and Integration Testing Policy

Unit Testing

Objective

The objective of unit testing is to verify that individual components of the Dispute Resolution Engine work as expected in isolation. Each unit (function or method) will be tested to ensure it produces the correct output for a given input, meeting both functional and technical requirements.

Automation

Unit tests are automated using GitHub Actions as part of the continuous integration pipeline. This ensures that unit tests are triggered on every merged pull request, providing rapid feedback on code quality and identifying potential issues early in the development cycle.

Coverage

Unit tests aim to cover at least 80% of the codebase, focusing on critical logic paths and edge cases. Code coverage reports will be generated automatically as part of the CI process.

Frameworks

Frontend Unit Tests: Implemented using Jest to test the React-based user interface components.
Backend Unit Tests: The API, written in Go, will be tested using go test (gotest), ensuring core business logic and API functionality are working as expected.

Integration Testing

Objective

The goal of integration testing is to ensure that the different modules of the Dispute Resolution Engine work together as expected. This testing focuses on verifying data flow between services, APIs, and modules.

Automation

Integration tests are automated using GitHub Actions. Upon a successful run of unit tests, integration tests are triggered to ensure that all components work harmoniously after code changes.

End-to-End (E2E) Testing

Objective

The purpose of E2E testing is to validate the entire system from the user's perspective, ensuring that user flows and real-world scenarios work correctly across both frontend and backend services.

Framework

Cypress is used for integration and E2E testing. Cypress allows us to simulate user interactions with the system from the front-end interface to the back-end, validating the complete workflow.

Environment

E2E tests are run post deployment on the production server. This ensures that the system behaves correctly when all components are deployed in a production setting.

Testing Processes

Running Unit Tests

Unit tests for both frontend and backend are automatically triggered by GitHub Actions upon every code commit. These tests can also be run manually using the following commands:
Frontend: npm run test (Jest)
Backend: go test ./... (gotest)

Running Integration Tests

Integration tests are executed automatically after unit tests pass. Cypress handles the integration testing workflow, testing the communication between various parts of the application. To run integration tests manually:

- yarn cypress run

Running E2E Tests

End-to-end tests are executed automatically through GitHub Actions. Manual execution of E2E tests can be done using Cypress with:

- yarn cypress open or cypress run

Quality Assurance Tests

Reliability

The Dispute Resolution Engine must maintain high uptime due to the nature of the services provided. Clients require continuous access to the system to respond promptly to time-sensitive communications. A reliable system with high uptime not only benefits clients but also increases overall throughput.

- Stimulus Source: System Users/Clients
- Stimulus: Attempt to access the Dispute Resolution Engine
- Response: The system should maintain high uptime to ensure continuous access.
- Response Measure: 99.9% system uptime, fewer than 1 hour of downtime per year
- Environment: Production environment
- Artefact: Dispute Resolution Engine

Quantification

Fault-Tolerance

The system must be capable of handling external errors or unexpected conditions without failing. For example, if the system encounters a timeout due to a delayed response from an external service, it should still maintain its uptime and not crash or hang indefinitely.

- Test Specification
    - Timers and timeouts should be introduced to gracefully handle delayed or failed responses from external systems. Tests should ensure that, after a timeout, the system resumes in the expected state without impacting other operations.
- Example
    - A timer is set for a dispute workflow, and if the backend service does not respond within a predefined period, the system will handle it by logging the error, notifying the user, and remaining operational.

Recoverability

This aspect of reliability focuses on the ability to restore a stable state after a failure. The system should use mechanisms like preambles (to set up a known state before operation) and postambles (to clean up or reset the system to a stable state after operation).

- Test Specification
    - Every test should be designed to bring the system back to a known stable state, regardless of the test's success or failure. This ensures that subsequent tests or operations are not affected by any unexpected states.
- Example

- If a service failure occurs during a dispute resolution process, the system will automatically trigger a rollback mechanism to restore the system to a consistent and stable state, allowing the user to retry or proceed with their task without interruptions.

Solution

Given that the Dispute Resolution Engine operates within a Docker Compose environment, the following practices will help achieve the desired reliability standards:

1. Container Orchestration
   - Docker Compose allows services to be run in isolated containers. Ensuring that services can be restarted automatically upon failure (e.g., using Docker's restart: on-failure policy) is crucial for maintaining uptime.
2. Logging and Monitoring
   - Logging mechanisms should capture system errors and failures in real time, and monitoring tools should be used to track uptime. Alerts can be configured to notify administrators if the system approaches downtime thresholds.
3. Scalability
   - To handle unexpected spikes in load or resource consumption, the system should be scalable within the Docker Compose environment, enabling the addition of more container instances without downtime.

Performance

Scalability and performance are critical for the Dispute Resolution Engine (DRE). Disputes are common, and the system must handle many cases simultaneously, along with multiple active users per case. High performance is essential for scalability, as a well-performing system can accommodate more users effectively.

- Stimulus Source: System Users/Clients
- Stimulus: High volume of disputes and multiple active users per case
- Response: The system should handle multiple cases simultaneously and perform efficiently for each user.
- Response Measure: System supports 1000 concurrent users and 500 disputes without performance degradation
- Example: Using JMeter, we confirmed the ability for our system to handle up to 1000 concurrent users at about 46 requests per second with 0.036% errors occurring
- Environment: Production environment
- Artefact: Dispute Resolution Engine API, Backend & Deployment

Quantification

Performance and scalability are critical to ensuring the Dispute Resolution Engine (DRE) can handle a high volume of simultaneous disputes and users. To evaluate the system's performance under real-world conditions, we used Apache JMeter to simulate high user load, focusing on the system's ability to support 1000 concurrent users without significant degradation in performance. These tests measure the system's throughput, error rate, and data transfer rates, providing valuable insights into how the system handles concurrent operations.

Test Metrics

The following metrics were gathered during three test runs under simulated load conditions:

Run	Error %	Throughput	Received KB/sec	Sent KB/sec
Run 1	0.00%	50.35247	1110.25	6.69
Run 2	0.00%	82.79516	1825.6	11
Run 3	0.00%	68.41349	1508.15	9.09

Key Performance Indicators (KPIs)

1. Error Rate
    a. Across all test runs, the system maintained a 0.00% error rate, demonstrating excellent stability and reliability even under heavy load. This indicates that the Dispute Resolution Engine handled all requests successfully without failures or unexpected issues.
2. Throughput
    a. Run 1: 50.35 requests per second
    b. Run 2: 82.80 requests per second
    c. Run 3: 68.41 requests per second
    d. Throughput measures the number of requests the system can handle per second. While the values fluctuate between test runs due to varying conditions, all runs showed acceptable levels of throughput, ensuring that the system can manage large volumes of user requests effectively.
3. Data Transfer Rates
    a. Received Data
        i. The system demonstrated the ability to handle high incoming data, receiving between 1110.25 KB/sec (Run 1) and 1825.6 KB/sec (Run 2), showcasing its ability to process incoming traffic efficiently.
    b. Sent Data
        i. Outgoing data rates ranged from 6.69 KB/sec to 11 KB/sec, which reflects the system's capability to send the necessary response data without creating bottlenecks.

Results Analysis

- Error-Free Operation
    - The 0.00% error rate across all test runs is an excellent indicator of the system's reliability and ability to process a large number of requests concurrently. This is critical in maintaining performance and user satisfaction, especially when handling high-priority, time-sensitive disputes.
- Scalability
    - The fluctuations in throughput between runs reflect the system's varying load-handling capabilities. The highest throughput of 82.80 requests/sec (Run 2) demonstrates that the system can handle high loads efficiently. Even at its lowest throughput of 50.35 requests/sec, the system maintains adequate performance levels to meet the expected requirements for concurrent users and disputes.
- Data Handling
    - The system can efficiently manage and process large volumes of incoming and outgoing data, with incoming data rates reaching 1825.6 KB/sec. This shows that the Dispute Resolution Engine can support large datasets (e.g., documents, case details) without overwhelming the system.

Usability

The Dispute Resolution Engine is designed to be user-friendly, with simple and intuitive interfaces to facilitate easy navigation. The goal is to enable users to operate the system effectively without extensive training or technical knowledge, catering to users from a wide range of domains.

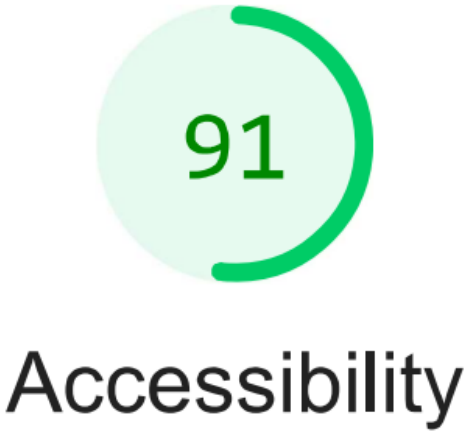
- Stimulus Source: System Users/Clients
- Stimulus: Attempt to navigate and use the system
- Response: The system should provide a simple and intuitive user interface requiring minimal training.
- Response Measure:
    - Lighthouse reports above 90/100, average request responses under 2 seconds
    - Use component libraries that are WAI-ARIA compliant, ensuring accessibility
- Environment: Production environment
- Artefact: Dispute Resolution Engine Frontend

Quantification

DRE aspires to be accessible to cater to as many users as possible, to quantify this we used the Lighthouse, which allows us to test, in a repeatable manner, how compliant we are to the various characteristics that define an accessible web app.

Target

- Above 90/100 for the overall score.
- Actual score:



Full Lighthouse Report

- **Failed Items**
  - Form elements lacking associated labels
  - Links without discernible names
  - Heading elements not in sequentially-descending order
- **Passed Items**
  - 23 audits passed, including proper use of ARIA attributes, sufficient colour contrast, and valid language attributes
  - Not Applicable Items
  - 31 items were marked as not applicable to this particular web application
- **Passed Items that Required Manual Inspection**
  - Interactive controls are keyboard focusable
  - Interactive elements indicate their purpose and state
  - The page has a logical tab order
  - Visual order on the page follows DOM order
  - User focus is not accidentally trapped in a region
  - The user's focus is directed to new content added to the page
  - HTML5 landmark elements are used to improve navigation
  - Offscreen content is hidden from assistive technology
  - Custom controls have associated labels
  - Custom controls have ARIA roles

Security

The Dispute Resolution Engine handles sensitive information, including user credentials and confidential dispute communications. Ensuring the protection of this data is crucial to maintain user trust and uphold the integrity of the system.

- Stimulus Source: Malicious Actors/Unauthorized Users
- Stimulus: Attempt to access sensitive information
- Response: The system should ensure the protection of personal and dispute-related information.
- Response Measure:
    - Dispute activity to be logged for auditability.
    - Minimizing attack vectors that could bypass access controls.
    - Implementing secure policies for all incoming requests.
- Environment: Production environment
- Artefact: Dispute Resolution Engine Frontend & API

Quantifications (OWASP Top 10)

- **A01 - Broken Access Control**
    - Ensure 100% of access control checks are implemented on all APIs, achieving a 0% unauthorized access rate.
    - We address this by using the Gatekeeper Architectural pattern (implemented as a middleware) that filters and enforces security policies to incoming requests.
- **A02 - Cryptographic Failures**
    - All sensitive data must be encrypted in transit and at rest, with 100% compliance to approved cryptographic algorithms.
    - We facilitate an HTTPS connection to all clients. Data is encrypted before being sent over the internet and is decrypted on Arrival.
- **A03 - Injection**
    - Achieve 100% compliance with input validation measures to prevent injection attacks, ensuring all user inputs are sanitized and validated.
    - We make use of GORM's Postgres library in our API. "*GORM uses the database/sql's argument placeholders to construct the SQL statement, which will automatically escape arguments to avoid SQL injection*" - Source
- **A04 - Insecure Design**
    - Conduct threat modeling for all new features, aiming for 0% design vulnerabilities in system architecture.
    - We adhere to the following secure design principles:
        - *Least Privilege and Separation of Duties:* Implement a least privilege access model for all users, ensuring that 100% have only the access necessary for their roles while enforcing separation of duties to mitigate risks.

        - *Zero Trust:* Adopt a Zero Trust approach, where 100% of access requests are authenticated and authorized, continuously logged for tracking anomalies.
        - *Security-in-the-Open:* Ensure that all code undergoes security reviews and uses secure coding practices, targeting 0% vulnerabilities in the deployed application.
- **A05 - Security Misconfiguration**
    - Perform regular security audits to ensure that no misconfigurations exist, targeting 0% security misconfigurations in the environment.
    - We have ensured that no default configurations on any service are left unchanged. This includes our databases, proxies, and services.
- **A06 - Vulnerable and Outdated Components**
    - Maintain an inventory of all third-party components and libraries, ensuring that 100% are up to date and patched.
    - All our used technologies are updated to the latest version and are checked by GitHub's Dependabot's "vulnerable package scanning" utility.
- **A07 - Identification and Authentication Failures**
    - Implement robust authentication mechanisms, achieving a 0% failure rate in user identification and authentication processes.
    - Aside from using OTP to verify user's emails upon signup, we use JSON Web Tokens (JWTs) to securely transmit information between parties. JWTs are utilized for maintaining user sessions and ensuring that only authenticated users can access protected resources.
- **A08 - Software and Data Integrity Failures**
    - Ensure that 100% of software updates are verified for integrity before deployment, preventing unauthorized changes.
    - Docker ensures software integrity by utilizing immutable containers that cannot be altered after deployment, alongside image verification mechanisms (such as Docker Content Trust) to authenticate and sign images, preventing unauthorized changes before deployment.
- **A09 - Security Logging and Monitoring Failures**
    - Implement comprehensive logging and monitoring practices, achieving 100% of critical security events logged and monitored for anomalies.
    - All container resource metrics, API system logs and Dispute procedural activities are logged continuously and stored to allow for the auditability and traceability of security anomalies.
- **A10 - Server-Side Request Forgery**
    - Ensure all server-side requests are validated and properly authenticated, aiming for 0% vulnerabilities related to server-side request forgery.
    - We

We also conducted multiple vulnerability scans to assess potential security risks. The following are the most significant findings.

Assessment Results Summary

- Nikto scan results:
  - Target: http://capstone-dre.dns.net.za/
  - Detected missing anti-clickjacking X-Frame-Options header.
  - 6544 items checked with no CGI directories found and no critical errors reported.
- Nmap scan results:
  - Target: capstone-dre.dns.net.za (196.29.59.166)
  - Detected open ports:
    - 80/tcp (http - Golang net/http server)
    - 443/tcp (ssl/http - Golang net/http server)
  - Identified valid SSL certificate.
- ZAP (OWASP Zed Attack Proxy) Report:.
  - Key risks identified:
    - High Risk: Cross Site Scripting (Reflected), SQL Injection.

Solutions

To address the Nikto find of a missing header, we simply added it. A more pressing matter was the 2 high risk alerts by the ZAP App. The SQL Injection turned out to be valid but inconsequential as the ORM library we use to interact with our Postgres database, GORM, addresses that automatically escape arguments, as previously mentioned. Cross site scripting required adding extra sanitisation to certain fields to prevent arbitrary code injection.

Maintainability

The Dispute Resolution Engine must be easy to update and extend, allowing the addition of new legal processes, dispute types, and other features. This adaptability ensures the system remains relevant and aligned with the latest legal standards and practices.

- Stimulus Source: Development Team
- Stimulus: Requirement to update or extend the system
- Response: The system should allow for easy addition of new legal processes, dispute types, and other features.
- Response Measure:
    - Frontend uses a component library.
    - Backend API is fully extensible with trivial addition/removal of endpoints.
- Environment: Development environment
- Artefact: Dispute Resolution Engine

Quantification

When modifying the codebase, and more importantly the API (which contains the most volatile aspects of the project's features) we aim to minimise the amount of lines of code that need to be modified or added (in the case of feature addition, to abide by any legal/policy updates).

Observations

1. Integrated EXTERNAL OpenAI API
    a. Generating summaries of archived disputes
    b. +104 LoC
    c. 1 hour spent
2. Integrated new INTERNAL service
    a. Added handler and business logic for dispute details endpoint
    b. +76 LoC
    c. 30 minutes spent

When we integrated the OpenAI API into the system to provide summaries for archived disputes, we had to incorporate an external service into our existing structure. Despite this being a new type of integration, we followed the same structure used throughout our API to manage requests, provide responses, and handle errors. This approach ensured that the API remained consistent and maintainable, demonstrating the system's ability to integrate new functionalities seamlessly and minimise the modifications required to existing files.

Links to Tests (GitHub)

All test scripts and reports are stored in the project repository:
Frontend Unit Tests (Jest)
Backend Unit Tests (Go)
Integration and E2E Tests (Cypress)

Test Reports

Detailed test reports are generated for each test suite and stored in the CI/CD pipeline. These reports include coverage data, success/failure rates, and performance metrics. Specific test reports can be accessed via the following links:
Latest Jest Report
Latest Go Unit Test Report
Latest Cypress Test Report
Latest E2E Test Report