NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE PROGRAM IN COMPUTER SCIENCE

DIPLOMA THESIS

ChainTask: A Tool for Automating Web API Functional Testing

Chrysostomos A. Rampidis

Supervisor: **Alexis Delis,** Professor
Kostas Saidis, Visiting Lecturer

Athens

January 2025

Diploma Thesis

ChainTask: A Tool for Automating Web API Functional Testing

Chrysostomos A. Rampidis
Student Number: CS22200028

Supervisor: **Alexis Delis,** Professor
Kostas Saidis, Visiting Lecturer

Committee of inquiry: **Alexis Delis,** Professor
Alexandros Ntoulas, Assistant Professor
Kostas Saidis, Visiting Lecturer

January 2025

ABSTRACT

The World Wide Web (WWW) continues to grow with an ever-increasing rate. Through the use of Web APIs, developing complex and scalable web applications has become an integral part of that growth. It is of critical importance that these Web APIs be tested to ensure their correctness and reliability across a plethora of use cases. Such a task, however, can be tedious and hard work. This is where test automation frameworks come into play, providing a way for developers and testers alike to evaluate a system's alignment to its functional specs.

This thesis introduces a tool, named ChainTask, designed for functional testing of Web APIs. The tool aims to make the process of API testing more straightforward and accessible, allowing users to define test cases in a concise and readable format. By focusing on essential testing tasks, the tool enables users to specify their own testing environments upon which they can perform sequences of automated tests. The tests are driven by a user defined input file which provides a basic structure for defining initial variables, actions to be executed, and interdependencies among those actions, making it easy to write and maintain test cases.

The versatility of ChainTask is demonstrated through experimentation with both an open, public Web APIs, and private typescript SDKs. ChainTask's inner workings attempt to increase the throughput by allowing parallel execution of tasks whenever possible, thus achieving a significant speedup compared to frameworks that use a traditional topological sort approach for handling their Direct Acyclic Graphs (DAGs).

ChainTask, provides foundations for easier and more efficient testing. A future addition of conditional actions can easily turn this tool into a full-fledged DSL for Dev Ops, incorporating features such as build and testing automation.

SUBJECT AREA: Web API Functional Testing Execution

KEYWORDS: Web API, System Integration Testing, Pipelines, Inter-dependency, Functional Testing

Table of Contents

Table of Figures

Prologue

This thesis outlines the creation and execution of a functional testing automation framework, which was specifically created and carried out in the Department of Informatics and Telecommunications at the University of Athens.

ChainTask's development spanned several months and was supervised by Kostas Saidis (Visting Lecturer). Its objective was to tackle the intricacies of automating interdependent web API calls, mainly for web API testing. It was carried out online using remote collaboration platforms, such as Slack and Bitbucket to communicate with the supervisor.

This prologue establishes the context for a comprehensive examination of ChainTask's capabilities and technical implementation.

1. Introduction

As web APIs (Application Programming Interfaces) become more integral to software applications, ensuring their reliability and correctness has become a central challenge. Web APIs are essential for enabling communication between systems, supporting data exchange, and facilitating coordinated operations across services. Given their importance, functional testing of web APIs is critical to verify that endpoints operate as specified, handling requests and producing expected responses. However, web API testing is often a cumbersome process, requiring extensive setup, detailed scripting, and technical expertise, which can complicate testing efforts and lead to inconsistencies in test coverage.

Traditional approaches to web API testing rely heavily on general-purpose testing frameworks that are often overly complex for straightforward functional testing tasks. These frameworks typically require intricate configurations and extensive programming knowledge to create and maintain test cases, making it challenging to achieve both accuracy and ease of use. As web APIs evolve and become more complex, these traditional methods can become burdensome, increasing the likelihood of testing gaps and making it harder to maintain up-to-date test suites.

ChainTask aims to address these challenges by developing a simple tool specifically for functional testing of web APIs. The tool is designed to streamline web API testing by providing a minimalistic approach focused on essential functionality, enabling testers and developers to define requests, expected responses, and validation checks with ease. By reducing the overhead involved in setting up and writing tests, ChainTask simplifies the testing workflow, making it more accessible while maintaining flexibility for different testing scenarios.

ChainTask is a lightweight, focused that improves web API testing workflows by reducing complexity and enhancing readability and maintainability. This thesis discusses the design and implementation of ChainTask, showcasing how it contributes to a more efficient and effective approach to web API testing, enabling developers and testers to ensure web API reliability in a streamlined and manageable way.

The remainder of this thesis includes Section 2 where we make an introduction to software testing methodologies and Section 3 where an introduction to ChainTask is made. Section 4 and 5 dive into the inner workings of ChainTask by analyzing its Design and Implementation while Section 6 presents an evaluation of ChainTask, including real-world use cases. Section 7 presents related works and finally Section 8 reflects on ChainTask discussing what it does well and where it can be improved upon.

2. Software Testing Methodologies

Software testing[1], [2], [3], [4], [5] is a foundational element in the development of reliable and high-quality applications. At its core, testing is about verifying that software functions as intended, identifying defects, and ensuring that users have a seamless experience. Software testing spans a range of activities, from validating that individual components perform correctly to assessing the behavior of the entire system under real-world conditions. Over time, testing practices have grown increasingly sophisticated, evolving from manual, ad hoc procedures to structured, systematic approaches that incorporate automation and formalized testing methodologies. As software systems have become more complex, with components spanning multiple platforms and dependencies, testing has become an indispensable part of the development lifecycle.

Modern testing includes various methodologies and types, such as unit testing, integration testing, system testing, and acceptance testing. Each type addresses a specific aspect of quality assurance. Unit testing verifies the smallest components, integration testing examines interactions between components, system testing evaluates the entire application, and acceptance testing ensures the product meets end-user requirements. The emergence of agile and DevOps practices has pushed testing to be continuous and iterative, ensuring that quality checks are woven into every stage of development. Automated testing frameworks, too, have transformed the testing landscape, making it possible to perform frequent and consistent testing, identify issues early, and maintain code quality even as applications grow in complexity.

One particular area of growing importance within software testing is the testing of web APIs (Application Programming Interfaces). As web APIs facilitate communication and data exchange between applications, ensuring their reliability and accuracy has become essential. Web API testing validates that each endpoint functions as expected, handles data correctly, and responds within acceptable performance limits. The next section will explore web API testing in greater detail, highlighting its unique challenges and discussing approaches to ensure robust web API functionality in today’s interconnected software ecosystems.

Testing methodologies provide structured approaches to ensure software quality and functionality. These methodologies guide the testing process, defining how to conduct tests, what techniques to apply, and the level of access to the system's internal workings.

One widely-used methodology is black-box testing, where testers focus on the system's external behavior without knowledge of the internal code or architecture. This method simulates the user's perspective by feeding inputs into the system and checking whether the outputs match the expected results. The tester remains unaware of the internal logic, ensuring an unbiased evaluation of whether the software meets its requirements. Black-box testing is primarily used for functional testing to validate that the software performs its intended functions.

White-box testing, on the other hand, involves testing the internal structure and logic of the application. The tester has access to the underlying code and uses this knowledge to design test cases that examine specific paths, conditions, and loops in the code. This methodology helps to detect issues such as hidden bugs, logical errors, and security vulnerabilities. White-box testing is particularly useful for unit testing, where individual components or functions are tested in isolation.

Gray-box testing combines aspects of both black-box and white-box testing. In this methodology, testers have partial knowledge of the internal workings of the system while still focusing primarily on the system's external behavior. Gray-box testing is often used when a deeper understanding of the system’s internal logic is needed to create more precise test cases, without delving into the full complexity of the codebase.

Other methodologies, such as agile testing and exploratory testing, are gaining prominence. Agile testing aligns with the iterative nature of agile development, with continuous integration of testing throughout the software lifecycle. Exploratory testing is more flexible and less structured, where testers actively explore the system, learning and adjusting their tests dynamically based on their findings.

Each methodology serves a specific purpose, and the choice of approach depends on the project's goals, the software being tested, and the resources available. By selecting the appropriate testing methodology, teams can ensure comprehensive coverage and more effectively identify defects, improving overall software quality.

3. ChainTask Overview

ChainTask, as a functional testing framework, follows a series of steps in order to properly perform the required tests. ChainTask requires a JSON input file to specify the details of each test it will perform, but it keeps the configuration minimal, focusing on the essential components of functional testing. The input file defines the sequence of actions to be tested and any dependencies between those actions, allowing ChainTask to execute tests in a logical, structured flow. Each action in the file represents a distinct step in the testing sequence, with references to variables and dependencies that ChainTask will interpret to perform the tests. This structure allows testers to set up functional tests quickly, streamlining the configuration process and making ChainTask easy to use.

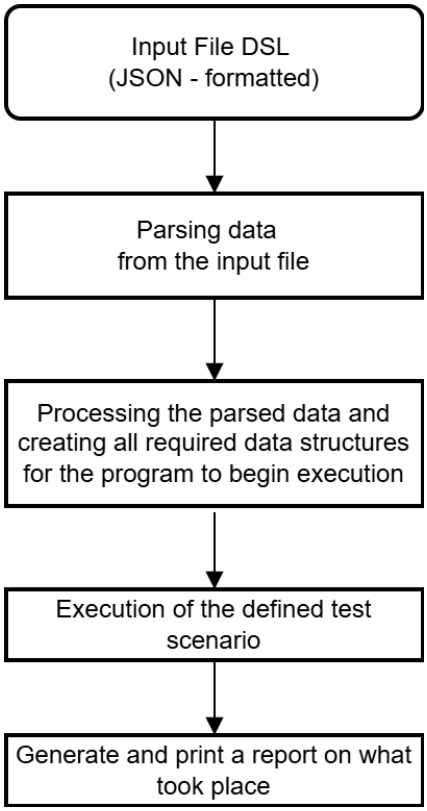


Image 1: A Visual Representation of the ChainTask's execution

Through the input file, a tester is able to quickly make changes to the steps and their dependencies or even the variables defined. This allows for quick scenario generation and assessment of the test system. Having the ability to make such quick and easy changes can be vital when having to perform a plethora of test cases in a short amount of time.

Once the JSON input file is loaded, ChainTask interprets each test step in sequence, managing dependencies and executing actions as specified. The testing process involves performing these actions and capturing their execution results, allowing ChainTask to build an overview of the web API's operational flow. ChainTask concentrates on ensuring that the sequence of actions proceeds as expected, producing an output that reflects whether each action completed successfully. The generated output file provides a structured summary of the test results, identifying any issues, which helps testers pinpoint potential issues.

```
    {
        "action": "getReleases",
        "arguments": [
            "baseUrl",
            "releaseSearchUrl",
            "searchaction1.artists.0.id"
        ],
        "dependsOn": [
            "searchaction1"

        ],
        "actionId": "getReleases1"
    },


    {
        "action": "search",
        "arguments": [
            "baseUrl",
            "artistSearchUrl",
            "search2"
        ],
        "dependsOn": [
        ],
        "actionId": "searchaction2"
    },
```

Image 2: A sample input file

3.1. Automated Web API Testing

Web API testing involves evaluating the functionality, performance, and reliability of a web Application Programming Interface (API) by sending requests to various endpoints and validating the responses. Web APIs enable communication between different software systems, making them critical components in modern software applications. Testing web APIs ensures that they behave as expected when interacting with other applications, which is crucial for both frontend and backend development.

Web API testing can focus on different aspects such as ensuring that the response structure adheres to the web API specification, confirming the data returned is accurate, and validating the behavior under various conditions like high traffic or erroneous input. Since web APIs often act as intermediaries between services, testing includes not only functional aspects but also performance and security. Testing helps identify potential vulnerabilities or points of failure that could affect the entire system.

There are multiple types of web API testing, each addressing different quality criteria. Functional testing verifies that each web API endpoint performs its intended function, returning the expected data and responding accurately to various inputs. Performance testing assesses how well the web API handles high loads and measures response times, helping to identify bottlenecks and optimize web API performance for scalability. Security testing ensures that web APIs are protected against unauthorized access and vulnerabilities, safeguarding sensitive data and maintaining system integrity. Each of these testing types contributes to a comprehensive assessment of web API quality.[1], [2], [5]

API testing often involves various HTTP methods such as GET (to retrieve data), POST (to send data), PUT (to update data), and DELETE (to remove data). By executing these methods with a variety of test cases, testers ensure that the web API performs all CRUD (Create, Read, Update, Delete) operations correctly. Beyond basic functionality, it also involves checking the response codes (e.g., 200 for success, 404 for not found), error handling, and authentication protocols.

Automated web API testing has become increasingly important due to the complex nature of modern web applications. Testing tools like Postman[6], RestAssured[7], and SoapUI[8] help automate these processes, allowing for more efficient testing cycles, especially during continuous integration/continuous deployment (CI/CD) workflows. These tools support scripting to run multiple test cases at once and verify the stability of the web API across multiple environments.

3.2. ChainTask for Automated Functional Testing

ChainTask focuses on functional testing because this type of testing is essential for validating the core behavior of web APIs, ensuring that each endpoint behaves as expected and correctly handles the data it receives and processes. Web APIs are often responsible for key functions within an application, such as data retrieval, processing, and interaction with other services. If even one web API endpoint fails or returns unexpected results, it can disrupt workflows and lead to application-level errors. Functional testing directly addresses these risks by testing each endpoint to verify that it performs its designated function, responds with correct data, and handles errors as expected. ChainTask's emphasis on functional testing, therefore, aligns with the need to create reliable, predictable web API interactions, which are crucial to seamless application performance.

ChainTask achieves this by using a JSON – formatted input file that makes it easy to define test cases for each endpoint in a concise and readable way. Testers can specify requests and expected responses without needing to build complex scripts or configurations, making the testing process faster and more accessible. ChainTask allows testers to validate inputs, outputs, and basic functionality of endpoints, enabling a thorough assessment of web API performance against expected results. By supporting core functional test features like defining web API requests, validating responses, and handling both expected and edge cases, ChainTask ensures that each endpoint can be reliably tested with minimal setup, enabling developers to verify functionality quickly and consistently. This approach not only simplifies functional testing but also integrates it seamlessly into development workflows, helping to catch issues early and maintain consistent web API quality.

4. ChainTask Design

This section focuses on explaining the design of ChainTask. It aims to give the reader a basic understanding of the structure of ChainTask, while expanding on the input requirements and explaining the output that is generated after each run.

4.1. Defining the testing parameters

Each time a user needs to initiate testing of their application, they will need to provide some details regarding what it is they are trying to test. These details must be specified by creating a list of tests that can be performed. This must be done by creating a new class which will contain a list of functions that are available for ChainTask to utilize and call. Every such function is to be considered as a single test action. This class must extend the ChainTask's **AbstractEnvironment** class which is an abstract class essential for the execution of a testing scenario.

The user will then need to define an input file which must follow the structure that is described below in 4.3 The Structure of the Input FileThe Structure of the Input File. The input file should contain information regarding which actions [of the ones listed in the user created class] must be executed, what their dependencies are, and what should be done if an error occurs.

The input file will be parsed upon the program's initiation. The parsed data will be used to initialize the required mechanisms that ChainTask uses internally in order to keep track of the testing's progress. These include:

- The **Scenario** class which is an object that keeps track of the testing's process
- Multiple **Step** entities, one for every single test that is to be made
- Multiple **Node** entities, that are objects that each encapsulates a Step class and are used to make sure that dependencies among Steps are honored.
- The **Environment** class, which will be instantiated through the **EnvironmentFactory** depending on the user provided variables. The

Environment class will also contain a **MethodRegistry** which is a user-defined list of available testing actions.

ChainTask will then proceed with actually carrying out the testing of the specified web API by calling the methods that correspond to the actions specified in the input file. Should an error occur, the **onError** method will be called and the execution will be halted thereafter.

Upon completion – successful or not – of the execution, a report will be generated detailing the actions that were carried out. For each action, a status will be reported along with the time it took to execute. The total execution time will also be displayed. The generated report is discussed in detail in Reporting Results: Generating Execution Reports. A schema of the discussed process is shown in Image 1: A Visual Representation of the ChainTask's execution.

4.2. Parsing the input file

The index.ts file serves as the application's entry point. It initializes and executes a scenario using the definitions specified in a JSON input file. The process commences with the inclusion of the essential **Scenario** and **Parser** classes. These classes are crucial for analyzing the scenario input file and overseeing the execution of the pipeline, respectively.

The code thereafter instantiates an object of the **Parser** type and employs it to retrieve and analyze the scenario details from the designated JSON file (./input.json). The *parseFromFile* function of the **Parser** instance retrieves the file, analyzes the JSON information, and transforms it into a structured **InputScenario** object. This object is then converted into an array of **Step** entities and an **Environment**. The **Environment** is initialized by selecting the most suitable **Environment factory** according to the specifications of the input scenario file.

After initializing the **Steps** and **Environment**, the code utilizes the *Scenario.newScenario* method to create a new **Scenario** instance. This method validates the input data and

returns a **Scenario** object only if all checks are passed. This approach establishes the **Scenario** created will have a validated DAG, and thus will not contain any cycles. The **Scenario** instance is subsequently executed using the *execute* method. This approach guarantees that the **Steps** are carried out in the appropriate sequence, taking into account their interdependencies and managing any asynchronous operations.

The **Scenario** is executed using a promise-based method, which effectively manages both successful and unsuccessful outcomes. Upon the completion of the execution, regardless of its success or failure, the finally block is called. Within this block, the report method of the **Scenario** object is invoked, resulting in the creation of a comprehensive report detailing the execution. This report contains the end state and duration of each **Step.** It also includes a thorough summary of the scenario's efficiency and any encountered faults throughout execution. The significance of this final report resides in its ability to comprehend the result of the **Scenario** and identify any potential problems that may have emerged.

4.3. The Structure of the Input File

The JSON input file provided specifies a structured scenario for carrying out a sequence of steps in a designated environment. The file starts with a section named *variables* that consists of a collection of key-value pairs. These variables are utilized consistently throughout the **Scenario's** execution to furnish essential info for the **Steps**. Variables can encompass several types of data, such as *strings*, *integers*, *URLs*, or other pertinent information that can be utilized by the **Steps** during the execution.

The *environment key*, located after the variables section, indicates the specific sort of **Environment** to be utilized for this scenario. The **Environment** in which the **Steps** will be executed is determined by this environment field in the input file, which is built using a suitable *environmentFactory* method described in the **Parser's** *parseInput* method.

A central element of the **Scenario** is the *steps* array, which comprises a collection of to-be **Step** instances. Every entry defined in the *steps* array of the input file represents a

distinct action that has to be executed and contains several characteristics such as *action*, *arguments*, *dependsOn*, and *actionId*. The *action* property provides the designated name of the function to be executed. The *arguments* property is an array consisting of strings that serve as the parameters to be transmitted to the **Step** at runtime. These *arguments* have the ability to refer to the variables that were defined earlier as the outcomes of prior **Steps** by using dot notation (e.g. CompletedStep.propertyArray.0.value).

The *dependsOn* attribute is an array that enumerates the action IDs (labels) of the **Steps** that must be fulfilled prior to the execution of this **Step**. This dependency management guarantees that **Steps** are carried out in the appropriate sequence, adhering to the progression defined in the input file. The **Scenario** guarantees that required tasks are completed before following actions are performed by identifying dependencies.

The *actionId* (label) property serves as a distinct identifier for each individual **Step**. This identifier is utilized to refer to the **Step** in many sections of the **Scenario**, specifically in the *dependsOn* arrays of other **Steps**. The identifiers guarantee that each **Step** can be clearly distinguished and its outcomes can be accessed as required.

The JSON input file provides a detailed description of a series of actions [that will later be converted into **Steps** at runtime] and their interdependencies, facilitating the construction and execution of intricate flows. The execution of the complete **Scenario** within the stated **Environment** is ensured to be cohesive and orderly by precisely defining each **Step** with its action, needed arguments, dependencies on other actions, and a unique identifier. This methodical technique enables meticulous management of the scenario, guaranteeing that all tasks are executed in the appropriate order and with the requisite information.

4.4. ChainTask’s Usage and Execution

Firstly, one would need to install the required software on their machine. In order to execute JavaScript applications and handle dependencies, it is needed to have Node.js and npm (Node Package Manager) installed. You can acquire and deploy them from the official website of Node.js. In addition, TypeScript is required for writing JavaScript code that is type-safe. To globally install TypeScript using npm, execute the command "npm

install -g typescript" in your terminal. It is advisable to use a code editor such as Visual Studio Code for more convenient code organization and debugging.

To proceed, you must install any required dependencies by using the command "npm install". Additionally, it is necessary to have an input.json file that is present and properly configured. The input.json file should correspond to the user-defined data in the **environmentFactories** and the */environments/* files.

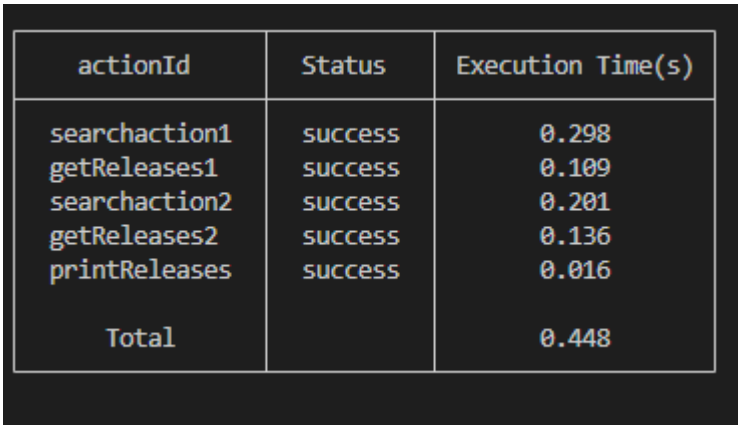
Once you have installed the necessary dependencies and configured the input JSON file, you may proceed to execute the program. The application's starting point is usually specified in a TypeScript or JavaScript file that initializes and runs the **Scenario**. The file responsible in this instance is the index.ts file. To execute this, transpile the TypeScript files into JavaScript by using the "tsc" command, and then run the "node index.js" command. This should initiate ChainTask's execution which involves parsing the input JSON file, initializing the **Environment** and **Steps**, executing the **Scenario**, and ultimately generating a report of the execution.

4.5. Reporting Results: Generating Execution Reports

The reporting mechanism in this web API testing system is an essential element that offers an analysis of a **Scenario** execution. The system records data regarding the execution of each **Step**, encompassing precise details such as the *initiation* and *completion* times and the *status*. Such a reporting system is very useful for monitoring, debugging, and optimizing testing operations.

While a **Scenario** is being executed, ChainTask keeps a *statistics map*, which is simply a log that records the state of each **Step**. The system logs the outcome of each **Step** as either *successful*, *failed*, or *not started*. Each **Step's** *start time* and *end time* are recorded, enabling the determination of the duration of execution. This temporal data is essential for identifying performance bottlenecks and comprehending the dynamics of the pipeline execution.

After the completion of the **Scenario** execution, regardless of whether it is successful or experiences errors, ChainTask produces a comprehensive report. This report is generated by iterating through the *statistics map* and consolidating the recorded data into a well-organized style. This complete report includes the *execution status* and *duration* for each **Step**. The reporting method additionally computes the overall execution time of the entire **Scenario**, offering a comprehensive summary of the scenario's effectiveness.



actionId	Status	Execution Time(s)
searchaction1	success	0.298
getReleases1	success	0.109
searchaction2	success	0.201
getReleases2	success	0.136
printReleases	success	0.016
Total		0.448

Image 3: Sample report

The process of generating a report encompasses various essential elements. Initially, it combines separate **Step** reports into a cohesive summary, emphasizing successful, unsuccessful, and un-run **Steps**. If errors occur, the report can provide descriptive error messages and indicate the particular **Step** at which the issue happened. The user can control this through the *onError* method of the **Environment** which is called when an error is encountered and after which the **Scenario** terminates.

4.6. Implementation with Typescript: Ensuring Robustness and Maintainability

TypeScript, which is a language that extends JavaScript, brings static typing to the JavaScript environment. This provides developers with advanced capabilities to detect problems at an early stage and boost the maintainability of their code. Within ChainTask's scope, TypeScript is of utmost importance as it offers type safety, a critical feature for handling intricate data structures [such as **Steps**, **Environments, and Scenarios** with **directed acyclic graphs (DAGs)**], and guaranteeing the accurate execution of the tests.

The main advantage of using TypeScript for ChainTask is its capability to construct and enforce types using interfaces and type annotations. Entities such as **Step**, **Node**, and **Environment** are precisely specified with distinct properties and methods, guaranteeing that each component of the test scenario conforms to a pre-established structure. Type enforcement is implemented to mitigate typical programming errors, including the passing of incorrect data types to functions.

In addition, TypeScript's incorporation of contemporary JavaScript functionalities, such as async/await, improves the clarity and ease of maintaining the codebase. Asynchronous operations are commonly found in workflow execution, where components frequently need to perform web API calls or involve lengthy computations. TypeScript facilitates the management of these activities in a clear and predictable manner, hence minimizing the occurrence of runtime mistakes and enhancing the comprehensibility and ease of debugging of the code.

TypeScript seamlessly integrates with development tools and editors, offering instantaneous feedback and autocompletion, so greatly expediting the development process. TypeScript's robust generics significantly augment the adaptability and recyclability of the code. **Generics** are employed in ChainTask’s development to define the **Step** and **Node** entities in a manner that allows them to function with any sort of **Environment**. The generic typing feature enables ChainTask to be readily upgraded to handle new environments and testing scenarios, without the need to rewrite current code. This promotes scalability and adaptability.

5. Implementation

ChainTask is a framework for overseeing and carrying out a sequence of interconnected actions within a directed acyclic graph (DAG) structure. The essential element of this framework is the **Scenario** class, which guarantees that operations are carried out in a particular sequence determined by their dependencies. The **Step** interface represents each action and consists of a label, a list of dependents, and an *execute* method that accepts an environment context and returns a promise.

Inside the **Scenario** class, every action is contained within a **Node** interface. This interface consists of methods that allow for checking if the action is prepared to be executed and updating counters when dependent **Nodes** or destination **Nodes** are finished. The **Scenario** class contains a graph property that stores a Directed Acyclic Graph (DAG) of **Nodes**. It also has a *statistics Map* that keeps track of the execution status and timing of each action. Additionally, the class has properties to record the start and finish times of the **Scenario's** execution.

The **Scenario** class constructor initializes the graph and establishes the **Nodes** and their dependencies. This process guarantees that every individual **Step** is included as a vertex in the graph, and that its edges are established according to the interdependencies among the steps. If the graph is acyclic, the *execute* method commences the process by locating the graph’s root nodes and initiating their execution.

Individual **Nodes** are executed by performing the associated **Step** actions and controlling the flow to dependent **Nodes.** The *execute* method also updates the *statistics Map* with the outcome of the execution and invokes the error handling method of the **Environment** in case a **Step** fails, potentially halting any subsequent execution. A report is generated in the end, which includes the status and execution time for each **Step**, as well as the overall execution time of the **Scenario.**

In general, the code offers a strong and reliable structure for handling and carrying out intricate flows that involve activities that depend on each other. By structuring actions in a Directed Acyclic Graph (DAG), it guarantees that dependencies are honored, and actions are carried out in the appropriate sequence. ChainTask incorporates meticulous monitoring of execution progress, error management, and extensive reporting, rendering it suited for any web API pipeline testing scenario, and in general any scenario that depends on the orderly execution of interconnected tasks.

5.1. ChainTask's mechanism

This section focuses on the fundamental elements that are the foundation of ChainTask. Every entity has a crucial function in establishing, overseeing, and carrying out processes inside ChainTask. The entities are intricately built to provide robust, adaptable, and efficient automation of testing, encompassing the representation of individual phases, management of dependencies, handling of execution contexts, and management of environments. We will examine the design, functionality, and interactions of these entities, offering a thorough comprehension of their contributions to the broader system. Through a meticulous analysis of these components, our objective is to demonstrate ChainTask's ability to effortlessly manage intricate and interconnected tasks.

5.1.1. The Steps

The **Step** entity is a core element of ChainTask, specifically created to represent a singular unit of work, for example a web API call. Every individual **Step** is distinguished by a label, which functions as its unique identity within the testing scenario. The *label* is essential for monitoring the progress of the **Step** and handling interdependencies. Furthermore, every **Step** includes a collection of other ***Step*** entities that it relies on. The dependencies guarantee that a **Step** will be executed only when all the **Steps** it depends on have been finished, thus ensuring a predetermined order of execution according to the input file provided.

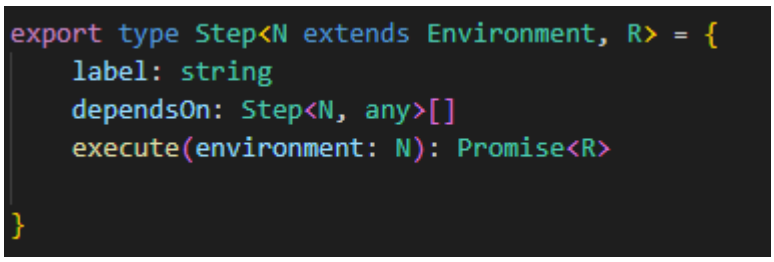


Image 4: The Step type

The primary functionality of a **Step** is contained within its *execute* method. This method is responsible for specifying the activities to be carried out when the **Step** is executed. The **Step's** execution relies on an **Environment** parameter that supplies the required context and resources. The *execute* method yields a promise, enabling the execution of asynchronous activities within the **Step**. The asynchronous aspect of the system is crucial for managing operations that have an uncertain duration, such as network requests or computations that take a long time to finish.

The **Step** entity not only manages the execution sequence and performs operations, but it also interfaces with the **Environment** to retrieve and set runtime variables. The integration is streamlined by utilizing methods offered by the **Environment**, such as *getActionResult* and *setActionResult*. These approaches enable a **Step** to retrieve the outcomes of preceding **Steps** and save its own outcomes for later use. The dynamic interaction with the **Environment** allows for the execution of tasks that are aware of the context, meaning that the actions taken in a particular **Step** can be influenced by the results of previous **Steps**.

The **Step** entity's design facilitates the creation of complex flows that are driven by dependencies. The **Step** entity guarantees the correct sequence and fulfillment of relevant preconditions for each job by imposing a strict execution order based on dependencies and allowing for asynchronous actions. The execution model described here is crucial for effectively managing intricate flows, in which operations rely on each other and must be carried out in a meticulously planned sequence to attain the intended result. The **Step** entity is essential for ensuring the effective and dependable execution of tests in the specified **Environment**.

5.1.2. The Nodes

The **Node** object is an essential element in this functional testing system, serving as a vertex in the directed acyclic graph (DAG) of a **Scenario** and containing information about its execution state and dependencies. Every **Node** includes a reference to a **Step** entity, which specifies the precise task to be performed. This reference enables the **Node** to access the **Step's** *label, dependencies,* and *execution* method, thus incorporating the **Node** into the larger scenario structure.

```
type Node<N extends Environment> = {
    step: Step<N, any>
    dependencyFinishedCounter: number
    destinationsFinishedCounter: number
    readyToRun(): boolean
    dependencyFinished(): void
    destinationFinished(): void
}
```

Image 5: The Node type

Each **Node** keeps track of the progress of its dependencies and destinations by maintaining counters, in addition to referencing a **Step**. The *dependencyFinishedCounter* field keeps track of the number of finished dependencies for a given **Node**, while the *destinationsFinishedCounter* variable keeps track of the number of future **Nodes** (destinations) that have confirmed the completion of their execution. These counters are crucial for identifying the readiness of a **Node** to execute and for controlling the flow of execution across the DAG.

The **Node** entity encompasses many methods for controlling its execution state. The *readyToRun* method verifies if all dependencies have completed, utilizing the *dependencyFinishedCounter* to ascertain if the **Node** is ready to be executed. The *dependencyFinished* method increases the *dependencyFinishedCounter*, indicating that a dependency has been finished. Similarly, the *destinationFinished* method increases the *destinationsFinishedCounter*, indicating that a **Node** dependent on the current **Node's** execution has finished executing. These approaches guarantee that the execution of

each **Node** is coordinated with the completed processes of its dependent **Nodes** and the acknowledgements from its destination **Nodes**.

By incorporating the **Node** entity into the **Scenario**, it becomes possible to exert precise control over the order in which **Steps** are executed. **Nodes** guarantee the proper sequence and timing of execution by monitoring the fulfillment of dependents and destinations, ensuring that each **Step** is completed in the appropriate order and only when all required prerequisites are satisfied. This method facilitates the dependable and effective execution of intricate processes, where the interdependencies between jobs necessitate meticulous supervision. The **Node** entity is crucial for maintaining the integrity of the scenario execution process, ensuring that tasks are carried out in a coordinated and regulated manner.

5.1.3. The Scenario

The **Scenario** class is specifically designed to manage the execution of a sequence of interdependent activities that are structured in a directed acyclic graph (DAG). This class guarantees the sequential execution of **Nodes**, taking into account their interdependencies. It also includes features for monitoring the progress of execution, managing failures, and providing useful reports.

```
export class Scenario<N extends Environment> {
    private graph: DirectedGraph
    private statisticsMap: Map<Step<N, any>, { status: string, startTime: number, endTime: number }>
    private startTime: number
    private endTime: number
    private environment: N
    private errorOccured: boolean
```

Image 6: The Scenario type

The class encompasses various essential attributes. The *graph* property represents a Directed Acyclic Graph (DAG) that captures the activities to be executed. Each vertex in the graph corresponds to an individual **Node** and each edge corresponds to a

dependency association between two vertices. The *statisticsMap* is utilized to monitor the execution status of each action, documenting whether it has notStarted, succeeded, or encountered failure, alongside the start time and end time of its execution. The *startTime* and *endTime* **Scenario** properties capture the precise beginning and completion times of the execution of the **Scenario**. The *environment* property denotes the specific circumstances or resources required for carrying out tasks, supplying essential data and procedures. The *errorOccurred* property is used to signal whether any action has met an error during execution, which is essential for halting further execution of the **Scenario** if necessary.

The **Scenario** class is equipped with a static function called *newScenario*, which generates a new instance of the class. This technique verifies whether the given actions constitute a valid Directed Acyclic Graph (DAG). If the condition is true, it will generate a new instance of the **Scenario** class; otherwise, it will return undefined. This is used in order to make sure that no **Scenario** entities containing cycles can exist. In this context, we need to make sure that no cycles exist in order to be certain that no interdependent loops can be formed, where a group of **Steps** would be cyclically executed indefinitely.

The **Scenario** class's private *constructor* creates a new instance of a **Scenario**, taking as parameters a specified **Environment** and an array of **Steps**. During the initialization process, a DAG is created. Each **Step** is encapsulated in a **Node** entity and is added as a vertex in the graph, while its edges are established depending on the given dependencies of each **Step**. Additionally, the *constructor* initializes the *statistics map* which monitors the progress and timing of each **Step's** execution and later logs the start and end times along with the **Step's** status. Initially, all **Steps** are assigned the *notStarted* status by default.

The class contains multiple methods for controlling the execution flow. The *canExecute* method determines the validity of the graph as a DAG by confirming the absence of cycles. If the graph contains cycles, it will stop execution, indicating that the **Scenario** cannot be executed. The *execute* method initiates the execution process by initially validating the graph as a legitimate DAG. If the condition is met, the procedure detects the root **Nodes** (those that do not have any dependencies) and initiates their execution

using the *startAllRoots* method. This function establishes promises[9] to handle the resolution or rejection depending on the execution result of these starting **Nodes**.

The *executeNode* method is responsible for carrying out the execution of individual **Node**. The **Node** carries out the operation specified in the *step* property, records the execution outcome in the **Scenario's** *statistics map*, and controls the progression to following dependent **Nodes**. In the event of a failed **Step** execution, the method will activate the *errorOccurred* flag on the **Scenario** entity and invoke the **Environment's** error handling function, ceasing any additional execution.

The **Scenario's** *report* function produces a comprehensive report of the execution, displaying the status and duration of each **Step**, as well as the overall execution time. This function generates a report in a structured table style, offering a concise and thorough summary of the execution of the **Scenario**.

5.1.4. The Environment

The **Environment** object is a crucial element in this functional testing system, serving as the context and resources required for executing the **Steps**. This entity specifies the interface and methods that **Steps** utilize to interact with the execution context. The essential methods for handling the test's execution state and interactions are *getMethodRegistry*, *getActionResult*, *setActionResult*, and *methodRun*.

```
export type Environment = {
    getMethodRegistry(): MethodRegistry
    getActionResult<R>(actionId:string) : R
    setActionResult(actionId:string,result:any):void
    methodRun(method: string, args: any[]):any
}
```

Image 7: The Environment type

The *getMethodRegistry* function retrieves a registry containing the available methods inside the given context. This registry contains both standard methods and any custom methods registered for the particular context. The feature enables **Steps** to dynamically access and invoke methods while they are being executed, creating a flexible and extendable execution framework. Dynamic method invocation is essential for adjusting to various requirements and situations.

The *getActionResult* method returns the outcome of a previously executed **Step** by utilizing its label. This feature enables **Steps** to get data returned by previously executed **Steps**, hence easing the flow of data and the management of dependencies within the testing scenario. The *getActionResult* method allows **Steps** to make informed decisions by providing a means to retrieve prior results and base their actions on the outcomes of their dependents.

The *setActionResult* method saves the outcome of a **Step's** execution and links it to the label of the **Step**. The utilization of this strategy is crucial in preserving the current condition of the test, guaranteeing that the outcomes of each **Step's** execution is saved and can be retrieved by succeeding **Steps**. It facilitates the transmission of data throughout the testing scenario, enabling subsequent **Steps** to build upon the results of previous ones.

The *methodRun* method executes a specified method from the method registry using the provided arguments. This method is essential for carrying out the tasks specified in each phase, offering a uniform approach to invoke methods and manage their execution. It guarantees that every **Step** may carry out its specified actions within the context of the **Environment**, making use of the resources and methods that are accessible.

5.1.5. The AbstractEnvironment

The **AbstractEnvironment** entity is a fundamental element in this automated functional testing system. Its purpose is to handle the runtime **Environment** and offer crucial services to **Steps**. It serves as a base class that implements core functionalities shared across various specific **Environments**. These features encompass the management of

variables during program execution, the processing of method calls, and the facilitation of retrieving and storing the output of a **Step**.

```
export abstract class AbstractEnvironment implements Environment {
    protected runtimeVariables:Record<string,any>

    constructor(){
        this.runtimeVariables={}
    }
    getActionResult<R>(actionId: string): R {
        return this.runtimeVariables[actionId] as R
    }
    setActionResult(actionId: string, result: any): void {
        this.runtimeVariables[actionId]=result
    }
    methodRun(method: string, args: any[]){
        if(this.getMethodRegistry()[method])
            return this.getMethodRegistry()[method](...args)

        console.log(`Method '${method}' is not implemented.`)
        throw new Error(`Method '${method}' is not implemented.`);

    }

    getRuntimeVariables(){
        return this.runtimeVariables
    }
    abstract getMethodRegistry(): MethodRegistry
}
```

Image 8: The AbstractEnvironment class

The fundamental function of **AbstractEnvironment** is to preserve a log of runtime variables, which are essential for the dynamic execution of tests. These variables serve the purpose of storing intermediate outcomes and execution states, guaranteeing that each step has the necessary access to the data it requires. The *getActionResult* and *setActionResult* methods facilitate the retrieval and storage of results, respectively, enabling a seamless flow of information throughout a **Scenario's Steps**.

Moreover, the **AbstractEnvironment** class includes the *methodRun* method, which enables steps to dynamically call methods that have been registered in the **Environment's** method registry. The ability to dynamically invoke actions is crucial for supporting a diverse variety of tasks and ensuring that ChainTask can adjust to various situations and needs. The **MethodRegistry**, which is required by specific subclasses of **AbstractEnvironment**, specifies the accessible actions and their related implementations. The *methodRun* method initially verifies that the method actually exists in the **MethodRegistry** and only proceeds if the method requested is valid.

The **AbstractEnvironment** entity guarantees consistency and reusability across many specialized **Environments**, such as the later discussed examples of *MusicBrainz* or *Butterfly*, by abstracting these fundamental features. Concrete subclasses have the ability to expand upon the functionality of the **AbstractEnvironment** class by adding environment-specific configurations and methods. At the same time, they may utilize the common logic that is already defined in the base class. This design facilitates a modular and adaptable structure, allowing for the effective and versatile execution of intricate procedures across many sectors.

5.1.6. The Environment Factory

The **EnvironmentFactory** entity is essential as it generates instances of various **Environments** required for executing web API pipelines. This entity serves as a facility that sets up **Environments** by providing the necessary context and variables specified in the scenario input file. The **EnvironmentFactory** entity contains a factory method for each type of **Environment** that is supported, which configures the **Environment** with the necessary settings and resources.

```
export type EnvironmentFactory<N extends Environment> = (variablesMap: Record<string, any>) => N
```

Image 9: The EnvironmentFactory type

The utilization of the factory pattern by the **EnvironmentFactory** guarantees the presence of modularity and flexibility inside ChainTask. By separating the process of creating and configuring **Environments** from the fundamental logic of the testing execution, the system can effortlessly accommodate different workflows (e.g. different web APIs) and their distinct demands. This approach enables the creation of **Environments** that may adapt to the individual requirements of a situation, hence enhancing scalability and maintainability.

The **Environment** factories are defined by the end user discreetly in a separate file. Each entry corresponds to a user-defined **Environment**. Thus a new instance of a class, ideally extending the **AbstractEnvironment** class, is returned. Each such class should be defined separately, in its unique file, under the */environments/* directory, in order to maintain a high level of code readability. Every such user-defined class must include all the static variables that may be defined in the input file as class properties, while also providing the definition of the *getMethodRegistry* method which should return a **MethodRegistry** typed object.

```
export const EnvironmentFactories: Record<string, EnvironmentFactory<any>> = {
    Butterfly: (variables: Record<string, any>) => {
        return new Butterfly(variables)
    },
    MusicBrainz: (variables: Record<string,any>) =>{
        return new MusicBrainz(variables)
    },
}
```

Image 10: Defining Environment Factories

```
export class MusicBrainz extends AbstractEnvironment {
    search: string = ''
    baseUrl: string = ''
    artistSearchUrl: string = ''
    search2: string = ''
    search3: string =''
    searchNum: number = 0
    releaseSearchUrl: string = ''

    constructor(dataInput: Record<string, any>) {
        super()
        for (const key in this) {
            if (dataInput[key] == undefined && this[key]==undefined)
                throw new TypeError()

            if (dataInput[key] != undefined)
                this[key] = dataInput[key]
        }

    }
    getMethodRegistry(): MethodRegistry {
        return {
            search: (baseUrl: string, artistSearchUrl: string, search: string) => {···
            },
            getReleases: (baseUrl: string, releaseSearchUrl: string, artistId: string)=>{···
            },
            printReleases: (artist1: string, releases1:any[],artist2: string, releases2:any[]) => {···
            },
            onError:(stepId:string,error:any) =>{···
            }
        }
    }


}
```

Image 11: Example of user-defined Environment

5.1.7. The MethodRegistry

The **MethodRegistry** entity is a crucial component in the **Scenario** execution. It serves as a repository for methods that can be triggered dynamically by **Steps** during their execution. The purpose of this registry is to enhance the adaptability and versatility of the execution environment, enabling a diverse set of activities to be carried out according to the specific requirements of the testing scenario.

```
export type MethodRegistry= Record<string,Function> & {
    onError(stepId:string,error:any):void
}
```

Image 12: The Method Registry type

The **MethodRegistry** is essentially a map of strings to functions, where each key represents a method name and the matching value is the implementation of the function. This architecture enables the **Environment** to get and execute methods based on their names, facilitating dynamic and adaptable execution. The main function of the **MethodRegistry** is to serve as a centralized repository of possible actions that **Steps** can employ during their execution.

An essential component of the **MethodRegistry** is the incorporation of the *onError* method. This particular approach is specifically developed to address errors that arise while a **Step** is being executed. The *onError* method requires two parameters: *stepId*, which serves as an identifier for the **Step** where the issue occurred, and *error*, which offers specific information regarding the error. This approach facilitates tailored error management inside the pipeline, empowering the user to define suitable error handling for various exceptions that may come up.

To access the **MethodRegistry**, one can uses the *getMethodRegistry* method of the **Environment** entity. This access mechanism guarantees that **Steps** can dynamically

access and execute the methods they need, depending on the current state and context of the testing scenario. The **MethodRegistry** facilitates the management and access to methods in a standardized manner, hence enabling the execution of various activities within the pipeline. This enhances the adaptability and reliability of the execution framework.

The **MethodRegistry's** design facilitates effortless expansion and customization. Additional functionality and developing workflow requirements can be accommodated by adding new methods to the registry as necessary, allowing the **Environment** to adapt and support them. The ability to be extended is essential for adapting to intricate and diverse test scenarios, in which the tasks to be carried out can be altered depending on the particular context and objectives of the execution process.

Essentially, the **MethodRegistry** entity is crucial in ChainTask as it serves as a centralized and expandable library of methods that may be triggered dynamically by **Steps**. It provides support for versatile and dynamic execution, enables customizable error handling through the *onError* method, and improves the adaptation of the **Environment** to various testing flow requirements. The **MethodRegistry** guarantees that the execution **Environment** is equipped with the requisite actions to properly perform and manage all required processes.

5.1.8. The Parser

The **Parser** entity is a crucial element in ChainTask. Its primary function is to transform a scenario’s specifications from JSON files into executable **Steps** within a designated **Environment**. This object is essential for connecting static scenario definitions and dynamic testing execution by accurately interpreting and organizing the input data.

The **Parser** class contains a *parseFromFile* function that retrieves scenario data from a JSON file. This approach utilizes the *Node.js fs* module to retrieve the contents of the file and subsequently converts the JSON data into a well-organized format. The parsed data is encapsulated by the **InputScenario** type, which encompasses variables, steps, and the environment type.

After the JSON data is parsed, the **Parser** proceeds to handle it using the *parseInput* method. This method accepts the parsed **InputScenario** and an **EnvironmentFactory**, which is responsible for generating the suitable **Environment** instance according to the stated *environment* in the input file. The **EnvironmentFactory** utilizes the given variables to initialize the corresponding **Environment** object, guaranteeing that it is configured with the essential context and resources for executing the **Steps**.

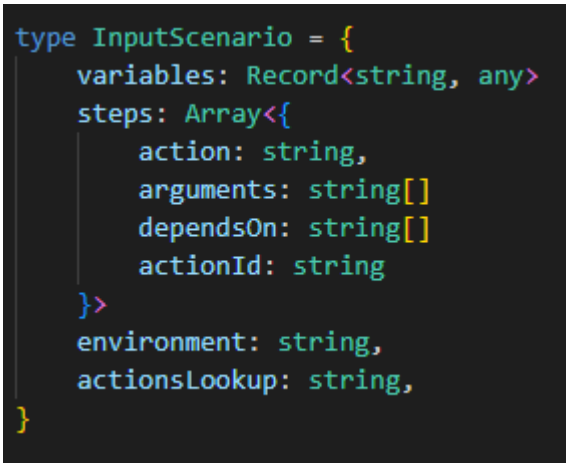


Image 13: The InputScenario type

The *parseInput* function of the **Parser** class subsequently creates an array of **Step** entities using the *step* definitions of the scenario. Every **Step** in the scenario comprises of an *action*, *arguments*, *dependencies*, and a *label*. The **Step** entity contains and defines the exact unit of work to be executed. The *execute* function of the **Step** class is designed to dynamically resolve arguments from the **Environment**, handle any nested property lookups, including arrays, and invoke the corresponding action using the *methodRun* method of the **Environment**. This technique guarantees that every **Step** has the ability to access and utilize the data generated by preceding **Steps** or described in the **Environment** in a dynamic manner.

In order to handle dependencies, the **Parser** keeps track of a mapping between labels and **Step** entities. This mapping enables the **Parser** to resolve dependencies by establishing links between each **Step** and the **Steps** it relies on. The dependencies in the **Scenario** are specified as an array of labels, which are then translated by the **Parser** into references to the associated **Step** entities. The linkage is essential for establishing the

directed acyclic graph (DAG) that determines the sequence in which the **Steps** are executed.

The **Parser** plays a crucial role in ChainTask by interpreting scenario specifications in a JSON formatted file and creating executable **Step** objects in a suitable **Environment**. This process guarantees the conversion of static scenario data into dynamic and executable test flows, which include clearly stated dependencies and context-aware actions.

5.2. Exploiting Promises to Improve Throughput

The most essential component of the **Scenario** execution process is the *execute* method within the **Scenario** class. This method starts off by verifying the validity of the Directed Acyclic Graph (DAG), confirming the absence of cycles that could potentially result in infinite loops or execution faults. The validation is conducted by utilizing the *canExecute* method, which uses graph theory algorithms to identify cycles in the graph structure. Here, the *getCycles* method is used, which is defined in the graph's library that is utilized by the **Scenario**.

After the graph is validated as *acyclic*, the execution process continues by finding all root **Nodes**, which are **Nodes** that do not have any dependencies. These root **Nodes** are independent and can be executed promptly. The *startAllRoots* method is responsible for initiating the execution of these root **Nodes**. The *executeNode* method is invoked for every root **Node**, overseeing the execution of the **Step** contained in each such **Node**.

The *executeNode* method is specifically built to manage the asynchronous execution of each individual **Step**. The process begins by recording the start time and calling the *execute* function of the associated **Step**, while providing the **Environment** context as a parameter. This **Environment** encompasses all essential facts and functions that are necessary for the **Step** to execute all of its actions. The function returns a promise, enabling us to efficiently manage asynchronous operations. Upon successful completion of the **Step**, the method proceeds to handle the subsequent dependencies.

Upon the completion of a **Step**, the *executeNode* method modifies the state and execution outcomes in the **Environment** and marks the **Step** as finished (*success* or *fail*) in the *statistics map*. Next, it obtains all the subsequent **Steps** that are dependent on this completed **Step** and were waiting for it to finish. The *dependency counter* of each subsequent **Step** is updated, and if all the dependencies for a **Step** are resolved, the *executeNode* method is called recursively for this dependent **Step**.

An essential characteristic of this approach is its resilient error handling. If a **Step** encounters an error, the *executeNode* method captures the error and calls a predefined, user-defined error handling method that is specified in the **Environment**. This procedure may involve recording the error or executing necessary cleanup tasks. Additionally, this strategy guarantees a smooth and controlled termination of the **Scenario** execution, thus preventing any subsequent **Steps** from being executed and thereby avoiding the possibility of triggering a series of failures. This behavior is an aspect that could easily be modified such that if an error were to occur during a **Steps's** execution, the rest of the **Scenario** could be executed normally.

During the execution, the **Scenario** class keeps a comprehensive record of the execution *status*, *start time*, *finish time*, and any encountered outputs or errors for each **Step**. This information is essential for generating execution reports, which offer valuable insights into the performance of the process, enabling the identification of bottlenecks and the diagnosis of errors.

5.3. Comparing Execution Methods: Efficiently managing Javascript pseudo – asynchronous execution.

The **Scenario's** *execution* method implemented achieves faster execution times compared to using a standard topological sort order[10] by leveraging concurrent execution of independent **Steps**. When employing this method, execution begins by identifying and immediately starting the execution of root **Nodes** - **Steps** with no dependencies. This approach enables a pseudo-parallel execution of these independent **Steps**, significantly reducing the overall execution time as multiple tasks can progress simultaneously.

This is pseudo-parallel execution is achieved thanks to the inner workings of Javscript. Its asynchronous nature through features like Promises and callbacks can give the illusion of parallelism, especially in handling I/O operations. These asynchronous operations are scheduled via the event loop, allowing JavaScript to continue executing other code while waiting for tasks such as file reads or network requests to complete. However, this isn't truly multithreading, as JavaScript still processes one operation at a time within its main thread. The asynchronous tasks merely allow for non-blocking behavior, where the event loop manages tasks and executes them when they are ready, which can resemble parallel execution.

ChainTask utilizes this mechanism by using Promises to execute an action as soon as it has all of its dependencies fulfilled. This multi – Promise approach takes advantage of the inner workings of Javascript which yield a parallel – like execution of the Nodes in the DAG.

Conversely, using a topological sort order involves pre-computing a complete linear ordering of all **Steps** based on their dependencies before any execution begins. This requirement can delay the start of execution, as it necessitates processing the entire dependency graph to determine the correct order. During this time, opportunities for concurrent execution may be missed, especially when there are multiple independent **Steps** that could be executed simultaneously.

Looking at Image 11: Sample report, we can see that the execution time of a **Scenario** was 0.448 seconds. If we were to sum up the execution times of each **Step** we would get 0.76 seconds which would be the assumed time if the **Steps** were executed sequentially by using the topological sort method.

By dynamically managing dependencies and executing **Steps** as soon as their prerequisites are met, the core method optimizes resource utilization and speeds up the testing completion. Each **Step’s** completion triggers the execution of its dependent **Steps** immediately, considering dependency limitations are met, maintaining a continuous flow of operations and further enhancing the efficiency of the execution process. This method ensures that the system can handle complex tasks with numerous interdependencies

more effectively than the traditional topological sort approach, leading to improved performance and faster overall execution times.

6. ChainTask Evaluation

6.1. Use Case: MusicBrainz, a public web API

The MusicBrainz web API serves as a gateway to the MusicBrainz database, granting users access to an extensive compilation of music metadata. Its purpose is to streamline the process of accessing and utilizing comprehensive data about music entities, including artists, albums, tracks, and other related information. This web API follows RESTful principles, enabling developers to interact with it using regular HTTP queries and receive responses in either XML or JSON format.

The web API facilitates a wide range of inquiries and activities. An essential feature is the capability to execute search queries. Users have the ability to conduct searches for entities such as artists, albums, and tracks by utilizing particular query parameters. To search for an artist by name, you can utilize a URL structure that incorporates the entity type (artist) and the search query. This capability is especially beneficial when you require the MusicBrainz Identifier (MBID) for an entity, which can subsequently be utilized for more comprehensive searches.

Lookup queries are a crucial component of the MusicBrainz web API. After obtaining an entity's MBID, you can conduct a lookup to obtain extensive details about that entity. For instance, while making a lookup request for an artist, one can request further information such as their whole list of released albums, connections to other entities, and diverse metadata such as tags and ratings. This feature enables a comprehensive exploration of the data linked to certain music entities, rendering it indispensable for applications that want intricate music knowledge.

The web API also provides support for browse requests, enabling you to fetch a collection of entities associated with a certain entity. For example, you have the ability to peruse all albums linked to a specific artist. This functionality facilitates the development of applications that require the presentation of interconnected data, such as a comprehensive catalogue of an artist's releases or all the recordings inside a particular release group.

To obtain more detailed information regarding the MusicBrainz web API and its functionalities, please refer to the official MusicBrainz API documentation.

The JSON input file provided specifies a scenario to be executed in the MusicBrainz environment. The sample scenario involves searching for two artists, getting their releases, and displaying the results. When wanting to run the code, the user needs to type the **tsc** command which will use the tsconfig.json configuration file to create a dist directory with the compiled TypeScript files. Then, the user can type the **node ./dist/index.js** command in order to run the produced JavaScript index file. Upon running the code, the scenario execution progresses in the following manner:

The scenario input file begins with the variables section, which establishes the essential variables for the subsequent processes. As an example, the search query is set to *"led zeppelin"* and the second search query is set to *"Metallica"*. The baseUrl is "https://musicbrainz.org/ws/2/". It serves as the web API endpoint for MusicBrainz. The variables *artistSearchUrl* and *releaseSearchUrl* define the endpoints used for searching artists and their releases, respectively.

Image 14: MusicBrainz, Input file variables

Subsequently, the *environment* field is explicitly designated as "MusicBrainz". This command instructs ChainTask to utilize the environment factory in order to generate a

suitable instance of the MusicBrainz **Environment**. This **Environment** is designed to manage the precise web API requests and answers that are necessary for the **Steps**.

Image 15: MusicBrainz, Environment selector

A crucial element of the scenario input file is the *steps* array, which specifies a sequence of actions to be executed. The initial action, labeled as *actionId* "searchaction1", performs a search for the term *"led zeppelin"*. The web API request is constructed using the *baseUrl*, *artistSearchUrl*, and *search* variables. Since this **Step** does not have any dependencies (*dependsOn* is an empty array), it can be executed immediately.

Image 16: MusicBrainz: Test Steps definition

After the completion of "*searchaction1*", it obtains the artist info from the MusicBrainz web API. The outcome is saved in the **Environment** with the "*searchaction1*" as the key, encompassing the roster of artists that correspond to the search keyword. The subsequent action, "*getReleases1*", is contingent upon "*searchaction1*". To retrieve the releases of "*led zeppelin*", another web API request is constructed using the *baseUrl*, *releaseSearchUrl*, and the *ID* of the first artist supplied by "*searchaction1*". It is guaranteed that the execution of "*getReleases1*" will only occur once "*searchaction1*" has properly gathered artist data due to the dependencies array.

```
export class MusicBrainz extends AbstractEnvironment {
    search: string = ''
    baseUrl: string = ''
    artistSearchUrl: string = ''
    search2: string = ''
    search3: string =''
    searchNum: number = 0
    releaseSearchUrl: string = ''

    constructor(dataInput: Record<string, any>) {
        super()
        for (const key in this) {
            if (dataInput[key] == undefined && this[key]==undefined)
                throw new TypeError()

            if (dataInput[key] != undefined)
                this[key] = dataInput[key]
        }

    }
    getMethodRegistry(): MethodRegistry {
        return {
            search: (baseUrl: string, artistSearchUrl: string, search: string) => {
                const headers: HeadersInit = {
                    'Accept': 'application/json',
                }
                const opts: RequestInit = {
                    method: 'GET',
                    headers,
                };
                return new Promise<any>(async (resolve, reject) => {
                    const query = baseUrl + artistSearchUrl + search
                    // console.log(query)
                    // console.log(opts)
                    return fetch(query, opts).then(res => res.json()).then(res => {
                        resolve(res)
                    })
                })
            },
```

```
            getReleases: (baseUrl: string, releaseSearchUrl: string, artistId: string)=>{
                const headers: HeadersInit = {
                    'Accept': 'application/json',
                }
                const opts: RequestInit = {
                    method: 'GET',
                    headers,
                };
                return new Promise<any>(async (resolve, reject) => {
                    // reject("some reason")
                    const query = baseUrl + releaseSearchUrl + artistId
                    return fetch(query, opts).then(res => res.json()).then(res => {
                        resolve(res)
                    })
                })
            },
```

```
            printReleases: (artist1: string, releases1:any[],artist2: string, releases2:any[])
                return new Promise<any>(async (resolve, reject) => {
                    console.log("Releases for "+artist1)
                    var distinctReleases:any[]=[]
                    releases1.forEach(element => {
                        if(distinctReleases.includes(element['title']))
                            return

                        distinctReleases.push(element['title'])
                    });

                    distinctReleases.forEach(element => {
                        console.log(element)
                    });
                    console.log("")
                    console.log("")
                    console.log("Releases for "+artist2)
                    distinctReleases=[]
                    releases2.forEach(element => {
                        if(distinctReleases.includes(element['title']))
                            return

                        distinctReleases.push(element['title'])
                    });

                    distinctReleases.forEach(element => {
                        console.log(element)
                    });

                    resolve("")
                })
            },
            onError:(stepId:string,error:any) =>{
                console.log("error occured")
                console.log(stepId,error)
                return null
            }
        }
    }


}
```

Image 17: MusicBrainz, User defined Environment

At the same time, a comparable procedure takes place for the band "*Metallica*". The **Step** "*searchaction2*" performs an artist search for "*Metallica*" using the same *baseUrl* and *artistSearchUrl*, but with the *search2* variable. Similar to "*searchaction1*", this **Step** does not have any dependencies and can be executed immediately. After the execution of "*searchaction2*", it retrieves the artist data for "Metallica".

The **Step** "*getReleases2*" is contingent upon the completion of "*searchaction2*". The web API request is built using the *baseUrl*, *releaseSearchUrl*, and the *ID* of the first artist

obtained from the "*searchaction2*" to retrieve the releases of "*Metallica*". It is guaranteed that the execution of "*getReleases2*" will only occur after the successful retrieval of artist data by "*searchaction2*" due to the dependencies array.

The **Step** "*printReleases*" is contingent upon both "*getReleases1*" and "*getReleases2*". This **Step** consolidates the outcomes of the preceding **Steps**, including the name of the initial artist from "*searchaction1*", the collection of releases from "*getReleases1*", the name of the initial artist from "*searchaction2*", and the collection of releases from "*getReleases2*". Subsequently, it outputs this data, presenting a unified summary of the releases for both "*Led Zeppelin*" and "*Metallica*".

ChainTask guarantees the correct execution sequence of each **Step** by organizing the **Scenario** in this manner. It ensures that dependencies are respected and that all required data is accessible before moving on to the next **Step**. This meticulous and systematic process ensures that the final outcome precisely represents the releases of the searched artists, relying on the data obtained from the MusicBrainz web API.

```
Releases for Led Zeppelin
Led Zeppelin III
Good Times Bad Times / Communication Breakdown
Led Zeppelin
Babe, I'm Gonna Leave You / Dazed and Confused
Led Zeppelin / Near the Beginning
Living Loving Maid (She's Just a Woman) / Bring It On Home
Led Zeppelin II
Dusty in Memphis / Led Zeppelin
Heartbreaker / Bring It On Home
Whole Lotta Love / Living Loving Maid


Releases for Metallica
Dave 'Em All
Ride the Lightning
1982-11-30: Mabuhay Gardens, San Francisco, CA, USA
Kill 'Em All
Whiplash
Power Metal
Creeping Death
No Life 'til Leather
Ride The Lightning
Megaforce Demo
Live Metal Up Your Ass
1982-10-18: Old Waldorf - San Francisco, California, USA
Jump in the Fire
1983-04-09: L'amours/The Ritz, New York, NY, USA
1984-12-20: Lyceum Ballroom, London, UK
KUSF
1983-04-16: The Showplace, NJ, Dover, USA
1983-12-18: Agora Ballroom, Cleveland, OH, USA
```

Image 18: Sample MusicBrainz output

6.2. Use Case: The Butterfly API, a private API with a Typescript SDK

In order to demonstrate ChainTask's versatility, the Butterfly API is introduced, which is a private SDK that is a versatile data platform specifically created for the purpose of overseeing and organizing digital libraries, institutional repositories, and electronic archives. It is capable of serving several users simultaneously. The software provides a versatile and adaptable data structure, enabling the categorization of items into groups of limitless complexity. The API is designed to be independent of any specific datastore, offering a versatile abstraction layer that supports different data access patterns. The system provides strong user authentication and authorization capabilities, including support for local databases and single sign-on over CAS or LDAP. Other notable

capabilities encompass comprehensive text search functionality, customizable processes triggered by server-side events, and strict adherence to open data best practices. The RESTful interface guarantees smooth interaction with a wide range of data sources. Additional information can be accessed on the official Niovity Butterfly webpage.

The JSON input file provided specifies a scenario to be executed within the Butterfly environment. This scenario is intended to authenticate into a system, validate functionalities, fetch and handle store objects, and do a search using provided models. When wanting to run the code, the user needs to type the **tsc** command which will use the tsconfig.json configuration file to create a dist directory with the compiled TypeScript files. Then, the user can type the **node ./dist/index.js** command in order to run the produced JavaScript index file. Below is a comprehensive explanation of the sequence of events that would occur if you were to execute the code:

The scenario begins with the variables section, which establishes the essential variables for the subsequent processes. The login and password required to access the system are set. The *storeName* is assigned the value "*uoadl*" and the *storeId* is assigned the value "graduate_*theses*", which serves to identify the particular store and its corresponding ID that will be read and altered. The *baseUrl*, which serves as the API endpoint for the system, is "https://pergamos-staging.niovity.com/bui". Furthermore, the "*models*" array consists of two model identifiers, namely "*modela*" and "*modelb*", which will be utilized in a subsequent search operation, but for our purposes are fake.

Image 19: Butterfly API, Input file variables

The environment is designated as "*Butterfly*", signifying that the scenario will take place within the Butterfly user-defined environment. The **Environment** is instantiated using the suitable factory method, which initializes the **Environment** with the given variables and establishes the required context for the actions to be executed.

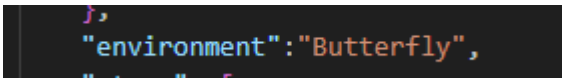


Image 20: Butterfly API, Environment selector

A crucial element of the scenario input file is the *steps* array, which delineates a sequence of events to be executed in a predetermined order. Every entry in the *steps* array possesses distinct characteristics, including *action*, *arguments*, *dependsOn*, and *actionId*. The execution commences with the *login* **Step**, which is defined by the *actionId* "*login*". This **Step** involves utilizing the provided *login* and *password* to get access to the system. Since this **Step** does not have any dependencies (*dependsOn* is an empty array), it can be executed immediately. Successfully completing this **Step** establishes a session and obtains an authentication token for future actions.

```
"steps": [
    {
        "action": "login",
        "arguments": [
            "username",
            "password"
        ],
        "dependsOn": [],
        "actionId":"login"
    },
    {
        "action": "hasCapabilityToLoadObject",
        "arguments": [
            "storeName",
            "storeId"
        ],
        "dependsOn": [
            "login"
        ],
        "actionId":"hasCapabilityToLoadObject"
    },
    {
        "action": "getStoreObject",
        "arguments": [
            "storeName",
            "storeId"
        ],
        "dependsOn": [
            "hasCapabilityToLoadObject"
        ],
        "actionId":"getStoreObject"
    },
    {
        "action": "getAllowedChildrenTypes",
        "arguments": [
            "storeName",
            "storeId"
        ],
        "dependsOn": [
            "getStoreObject"
        ],
        "actionId":"getAllowedChildrenTypes"
    },
```

Image 21: Butterfly API, Test Steps definition

After a successful login, the next **Step**, "*hasCapabilityToLoadObject*", checks if the user who signed in has the ability to load objects from the specified store. This **Step** is contingent upon the "*login*" **Step**, guaranteeing that the capability check is executed only following a successful login. The access rights are verified using the arguments *storeName* and *storeId*.

```
export class Butterfly extends AbstractEnvironment{
    username: string = ''
    password: string = ''
    private api: StandardButterflyAPI
    baseUrl: string = ''
    storeName: string = ''
    storeId: string = ''
    models: Array<string> = []

    constructor(dataInput: Record<string, any>) {
        super()
        for (const key in this) {
            if (dataInput[key] == undefined && this[key]==undefined)
                throw new TypeError()

            if (dataInput[key] != undefined)
                this[key] = dataInput[key]

        }
        // this.api=API.initializeAPI(this.baseUrl,undefined,()=>null)
        const token=undefined
        this.api= createStandardButterflyAPI(this.baseUrl, fetch, { token, onTokenRefresh: () => null })


    }
```

```
    getMethodRegistry(): MethodRegistry {
        return {
            login: (username: string, password: string) => {
                return this.api.login(username, password)
            },
            getStoreObject: (storeName: string, id: string) => {
                return this.api.getStoreObject(storeName, id)
            },
            getAllowedChildrenTypes: (storeName: string, id: string) => {
                return this.api.getAllowedChildrenTypes(storeName, id)
            },
            getAncestors: (storeName: string, id: string) => {
                return this.api.getAncestors(storeName, id, { includeSelf: false })
            },
            hasCapabilityToLoadObject: (storeName: string, id: string) => {
                const objectCapabilities = this.api.getCapabilities(storeName, id, defaultObjectSpecificCapabilitiesFactory).then((capabilities) => {
                    return capabilities.hasCapabilityToLoadObject()
                })
                return objectCapabilities
            },
            search: (storeName: string, id: string, models: string[]) => {
                const searchPayload = {
                    start: 0,
                    count: 20,
                    models: models,
                    query: {
                        filters: {
                            eq: {
                                parents: `${storeName}:${id}`//storeName+":"+id //
                            }
                        }
                    }
                }
                return this.api.search(searchPayload)
            },
            onError:(stepId:string,error:any) =>{
                return null
            }
        }
    }
}
```

Image 22: Butterfly API, User defined Environment

After verifying the capability, the "*getStoreObject*" **Step** receives the store object that is provided by the *storeName* and *storeId*. This phase is contingent upon the "*hasCapabilityToLoadObject*" **Step**, which guarantees that the retrieval of the store object only takes place once the system has confirmed the user's ability to load the object. Completing this **Step** successfully retrieves the required store data for subsequent activities.

The subsequent **Step**, "*getAllowedChildrenTypes*", ascertains the permissible types of subordinate objects that can be present within the obtained store object. This **Step** is contingent upon the "*getStoreObject*" **Step**, guaranteeing that the retrieval of the child type only takes place after successfully obtaining the store object. This **Step** utilizes the *storeName* and *storeId* parameters to request the specified information.

The "*getAncestors*" **Step** collects information about the ancestors of the store object after obtaining the permitted child types. This **Step** is contingent upon the successful execution of the "*getAllowedChildrenTypes*" **Step**, guaranteeing that it will only be executed after the child types have been retrieved successfully. This **Step** once again utilizes the inputs *storeName* and *storeId* to execute this action.

The last **Step**, labeled as "*search*", conducts a search within the designated store by utilizing the models given in the *models* array. This **Step** is contingent upon the "*getAncestors*" **Step**, which guarantees that the search is performed only after collecting all essential hierarchical information about the store item. The **Step** utilizes the *storeName*, *storeId*, and *models* array as parameters to perform the search operation.

Although this use case is not very complex, since it is basically a sequence of actions, it provides an insightful understanding into the inner workings of ChainTask. ChainTask guarantees the correct execution sequence of each **Step** by organizing the **Scenario** in this manner. It ensures that dependencies are respected and that all required data is present before moving on to the next **Step**. This systematic technique ensures that all actions are run in the proper order and that every **Step** is run only after the previous one has finished. This sequence of actions, although basic, demonstrates the seamless integration of an external SDK with ChainTask. Of course, the input file and the user-

defined *Butterfly* class can be enhanced to support intricate test flows with several interdependent API calls.

6.3. Versatility and Ease of Use with ChainTask

The above examples allows us to understand how ChainTask can work with both a public web API and a private Typescript SDK. The ease of configuration through the input file can support quick definition of test scenarios while also providing the ability to easily perform changes to existing test scenarios thus demonstrating the versatility and ease of use of ChainTask. These can come in handy for both developers and testers seeking to streamline their approach to functional web API testing through the use of automation tools.

7. Related Work

There are several testing frameworks available for automated web API testing. Some of them include Postman[6], Playwright[11], and Apache Airflow[12]. While not all frameworks provide the same functionality as ChainTask, the one that most closely compares is Apache Airflow. According to the official documentation, Apache Airflow uses DAGs to allow individual steps to be executed only after their dependencies have finished executing. Apache Airflow uses a typical topological sort while ChainTask utilizes a smarter approach, parallelizing any tasks that can be executed at the same time.

Gradle[13] is a build tool that utilizes DAGs to store its dependency graph. Gradle is buil to handle “tasks”. Among those tasks, there are dependencies which Gradle has to resolve in order to complete the build process. A DAG makes this dependency resolution much easier since one can easily detect cycles – and thus abort the build – and also by using topological sort, one is able to find the order in which the dependencies must be visited.

Bazel[14] is a build tool that uses Directed Acyclic Graphs (DAGs) to manage and optimize the build process. In Bazel, a DAG represents the relationships between build tasks or targets, where each node in the graph represents a specific build task (like compiling a file or linking a library), and each edge represents a dependency between tasks. These dependencies are strictly acyclic, meaning there are no cycles, which ensures that tasks are executed in the proper sequence without circular dependencies.

Robot Framework[15] is an open source automation framework for test automation and robotic process automation (RPA). Its human-friendly and versatile syntax uses keywords and supports extending through libraries in Python, Java, and other languages. It integrates with other tools for comprehensive automation without licensing fees, bolstered by a rich community with hundreds of third party libraries.

8. Conclusions & Future work

ChainTask effectively showcases the ability to build, analyze, and execute web API functional testing pipelines in a dynamic manner, leveraging a resilient architecture that guarantees meticulous control over execution sequences and dependencies. Through the use of a lightweight, focused JSON – formatted input file, ChainTask is able to efficiently and effectively perform web API functional testing while providing both developers and testers with a tool that reduces complexity and enhances readability and maintainability.

This system is based on a modular design that divides the tasks of establishing tests to be performed, controlling execution contexts, and orchestrating the execution. The modularity of ChainTask improves its capacity to be maintained and scaled, making it easier to expand and maintain. The practical implementation of ChainTask is demonstrated by its capacity to manage real-life situations that consist of several interconnected actions and asynchronous activities, such as web API calls or SDKs. The system is specifically built to parse and execute functional testing scenarios that are user defined.

ChainTask demonstrates proficiency in managing execution environments, allowing the flow of data between actions, and enabling dynamic method invocation. Utilizing promises for asynchronous operations guarantees the sequential execution of each action in the appropriate sequence and only when all essential prerequisites are fulfilled. This execution model ensures the dependable and effective completion of intricate test scenarios, while also preserving the integrity and coordination of the overall execution process.

The main drawback of ChainTask is the lack of support of conditional actions for each Step. In other words, the system is unable to make decisions based on the outcomes of the various actions. One major improvement would be to be able to support exactly that. The user would be able to specify conditional cases in the input file, for example with onSuccess/onFailure callbacks, and ChainTask would execute the corresponding action depending on the step’s outcome. This would support a greater number of potential test cases and would increase ChainTask’s usability.

Enhancing the error management and recovery mechanisms of ChainTask can also lead to considerable improvements. At present, ChainTask manages problems by halting execution and recording the problem, giving the user access to it by calling the *onError* method, but it may be expanded to incorporate more advanced approaches like automatic retries, fallback actions, and thorough error logging. Implementing these enhancements would enhance the system's resilience and enable it to automatically recover from temporary failures without the need for operator intervention.

Another aspect that can be enhanced is the advancement of graph visualization tools. Utilizing a directed acyclic graph (DAG) visualization can enhance users' comprehension of the interdependencies and execution sequence of their processes in an easier to understand manner. This can be accomplished by including graph visualization frameworks that offer interactive interfaces for users to visually create, change, and troubleshoot their tests. These tools will not only improve the ease of use but also increase the accessibility of the system for non-technical users.

ChainTask can also get advantages from enhancing performance, particularly when handling extensive and intricate procedures. This may entail optimizing resource allocation or introduce a concept of priority among actions. Performance profiling and optimization can guarantee that the system effectively handles larger workloads and increased complexity.

By focusing on these specific areas, ChainTask has the potential to enhance its capabilities, adaptability, and ease of use, hence accommodating a broader array of applications and user requirements. ChainTask has the potential to become a full-fledged Dev Ops DSL[16] that could be used for building and testing automations alike.

In conclusion, ChainTask demonstrates a simple yet effective solution for automated functional testing execution within various environments, providing a flexible and scalable framework that can adapt to a wide range of applications.

References

[1] J. Jain, “Introduction to API Testing,” *Learn API Testing Norms, Practices, and Guidelines for Building Effective Test Automation*, pp. 1–9, 2022, doi: 10.1007/978-1-4842-8142-0_1

[2] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, and D. Mishra, “RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions,” *Applied sciences*, vol. 12, no. 9, p. 4369, 2022, doi: 10.3390/app12094369

[3] J. C. Garcia, J. O. O. Hernandez, J. C. P. Arriaga, and H. J. L. Riano, “Advances in Web API testing: A Systematic Mapping Study,” in *2023 Mexican International Conference on Computer Science (ENC)*, IEEE, 2023, pp. 1–8. doi: 10.1109/ENC60556.2023.10508648

[4] A. Martin-Lopez and J. C. Alonso, “Testing of RESTful Web APIs,” in *Service-Oriented Computing – ICSOC 2022 Workshops*, J. Troya, R. Mirandola, E. Navarro, A. Delgado, S. Segura, G. Ortiz, C. Pautasso, C. Zirpins, P. Fernández, and A. Ruiz-Cortés, Eds., Cham: Springer Nature Switzerland, pp. 411–413. doi: 10.1007/978-3-031-26507-5_43

[5] S. Bonfanti, A. Gargantini, and P. Salvaneschi, *Testing Software and Systems: 35th IFIP WG 6.1 International Conference, ICTSS 2023, Bergamo, Italy, September 18-20, 2023, Proceedings*, 1st 2023., vol. 14131. Springer, 2023.

[6] “Postman Automated Testing,” *Postman*. Available: https://www.postman.com/automated-testing/

[7] “REST Assured,” *REST Assured*. Available: https://rest-assured.io/

[8] “SoapUI Running Functional Tests,” *SoapUI Docs*. Available: https://www.soapui.org/docs/test-automation/running-functional-tests/

[9] M. Frisbie, “Promises and Async/Await,” *Professional JavaScript for web developers*, pp. 383–433, Oct. 2023, doi: 10.1002/9781394193240.ch11

[10] C. Pang, J. Wang, Y. Cheng, and H. Zhang, “Topological sorts on DAGs,” *Information processing letters*, vol. 115, no. 2, pp. 298–301, Feb. 2015, doi: 10.1016/j.ipl.2014.09.031

[11] “Playwright API Testing,” *Playwright Documentation*. Available: https://playwright.dev/docs/api-testing

[12] “Apache Airflow DAGs,” *Apache Airflow Docs*. Available: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

[13] “Gradle Task Graphs,” *Gradle Docs*. Available: https://docs.gradle.org/current/userguide/build_lifecycle.html#task_graphs

[14] “Bazel Docs,” *Concepts and terminology*. Available: https://docs.bazel.build/versions/4.2.1/build-ref.html

[15] “ROBOT FRAMEWORK,” *Robot Framework*. Available: https://robotframework.org/

[16] A. Wąsowski, A. Wasowski, and T. Berger, *Domain-Specific Languages*. Springer, 2023. doi: 10.1007/978-3-031-23669-3