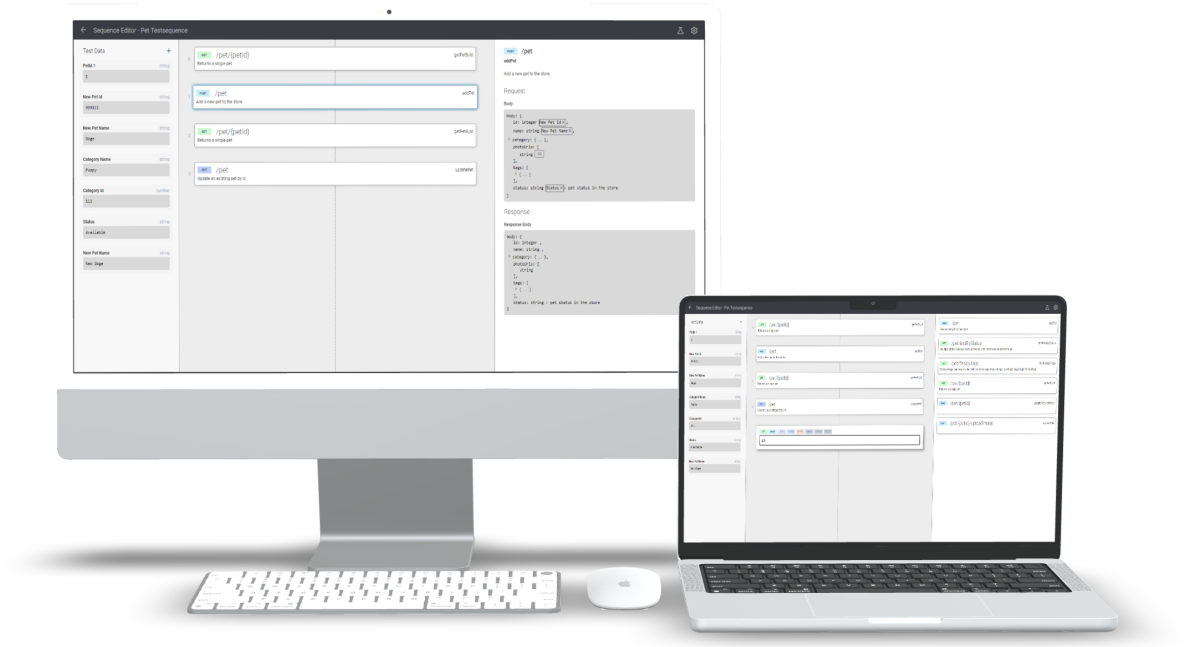
Fachhochschule Nordwestschweiz
Hochschule für Technik

TESTIFI

Interactive Use-Case Generation Tool for Functional REST API Testing

Bachelor Thesis

Windisch, August 2023



Students	Jonas Volken Benjamin Leu
Expert	Romano Roth
Supervisors	Prof. Martin Kropp & Fabian Affolter
Customer	Nejdet Dogru, Testifi GmbH
Project number	23fs_imvs12

Fachhochschule Nordwestschweiz, Hochschule für Technik

Abstract

Software is an integral part of any business, which makes the significance of high-quality software in today's digital age undeniable. However, despite the advancements in software testing, challenges persist in efficiently planning, generating, and executing test cases, particularly for REST API-based applications.

This project addresses the issue by developing a sequence generator tool that enables testers to effortlessly create and execute sequences of requests, streamlining the creation of comprehensive test scenarios. By simplifying the process of connecting response values to subsequent request values, the software seeks to maximize test coverage, improve test quality, and enable testers to focus more on software quality enhancement than the efforts of test construction.

The client for this project is Testifi GmbH, a company dedicated to enhancing software delivery processes through DevOps integrations and AI-automated quality assurance solutions.

The main focus of the project was to find out if the test quality increased by using the sequence generator tool due to more edge cases and more complex scenarios being tested compared to manual API testing, as well as showing if the efficiency improvement can be measured in reduced amount of time necessary for creation sequences.

To answer these questions and develop an application that offers value for Testifi GmbH, a literature review was conducted on the subjects of basic user interface design and user experience concepts for advanced users. Based on the findings, the user interface of the application was outlined and the software implemented.

During development and with the finished product, multiple sets of user tests were conducted with users experienced in working with APIs, to improve the design and software during development, and to gain insights about the effectiveness of the final product. Those tests showed that the main goals of the project could be reached by demonstrating a considerable amount of time saved by using the application, while also outperforming manual testing methods in efficiency and ease of use. Key features like the linking of response values to subsequent request values and the suggestion of such links based on Testifi's Pulse Artificial intelligence (AI) as well as previously created sequences were well received by testers and customer. The literature review also proved to be very valuable as users praised the straight forward design, while never missing any important data.

When Testifi GmbH integrates the end product in their pulse workflow, its ability to create sequences easily and intuitively as well as the potential of the additional link suggestions created by the tool to be used in improving the Pulse AI will be indispensable.

Acknowledgements

We would like to express our gratitude to Prof. Martin Kropp for his support throughout this bachelor's thesis. His guidance and advice helped us to stay on track and ultimately achieve the project's objectives.

Fabian Affolter also played a crucial role in the project by providing guidance and feedback throughout its duration, as well as tremendous support in all UI/UX aspects during several workshops and meetings.

From Testifi GmbH, we would also like to thank Nejdet Dogru and Gabor Bakos for their invaluable inputs, feedback and support, Ece Kilis for her help on the initial high fidelity designs as well as Orhan Nurkan and Gabor Bakors for their participation in user tests.

Throughout the project we were able to gain valuable experience in multiple new fields while reinforcing skills we acquired during our studies.

Contents

1 Introduction

1.1 Client

Testifi GmbH is an international company headquartered in Munich, Germany. Their focus is on optimising software delivery processes through DevOps integrations while providing AI-automated quality assurance solutions. They provide a highly advanced software tool for fully integrated automated testing and aim to provide the ability to test and create automated tests.

1.2 Inital Position

Regardless of product, enterprise, or domain, software is an inevitable part of the industrial operation. The use of quality software, therefore, is one of the most important aspects of business success in our age. Software testing is an essential quality assurance step used in software development. Testing gives feedback to developers about the quality and functionality of their software so they can correct and optimize their product. Furthermore, testing improves the overall stability of the product and enables the developers to easily add and change features. The later an issue is found the more work is required to fix it. Testing manually is simply not sufficient for modern software development that demands speed and continuous development. Therefore, test automation - tests executed automatically via software tools - is a critical success factor for modern software delivery, and therefore, for modern business. Since web applications are everywhere and modern software development methods employ microservices extensively, test automation for REST APIs is the focus of this project.

1.3 Problem Statement

Planning, generation, and execution of test cases are time-consuming and significantly affected by human errors. In addition, while a software is evolving, APIs are also updated according to current needs. Changes in APIs need to be reflected in test cases. When all these processes (creating requests, entering correct parameters and payload data, creating test data sets, following all changes, and refactoring test cases, executing and reporting tests) are performed manually, testers spend most of their time debugging errors and maintaining the tests. They usually end up having less than necessary number of tests. This results in suboptimal test coverage, as well as a lack of transparency regarding the quality and quantity of manually performed tests.

Testifi is already able to analyse the endpoints of APIs, infer the producer-consumer relationship between endpoints, and create a graph. That helps create a sequence of requests to send to the server to check if the logic behind this sequence is implemented. The creation of sequences by AI is very limited, as while links between some parameters are found, often not all required values can be provided by AI. This causes manual execution of multiple requests that represent a sequence, which is time consuming.

The amount of work required to create and maintain tests could be vastly reduced if the testers were supported with automation and could use tools to create the tests instead of writing them. Reducing the effort of writing individual tests or performing them manually would also free up resources to improve the quality and coverage of the executed tests and therefore contribute to the quality and speed of future developments.

1.4 Goals

The vision of this project is to minimize the effort of testers used to create and maintain testing scenarios, so the testers can focus on software quality instead of writing tests.

This project aims to develop a sequence generator that eases test case creation for REST API-based applications by enabling the tester to easily create and execute sequences of requests. These sequences can vary from simple login-logout behavior and complex operations, to cover all necessary usecases the applications might have. The testers can create connections between responses and subsequent requests to reuse returned values and generate a request-response-flow. With the generation of sequences of requests, testers can check if the endpoints are able to work in harmony with each other (integration tests) and if all planned functionalities are implemented (functional tests). At the end of the execution of these tests, the user can see in a report of all requests were executed successfully. As the software is used, it will gain more information about the links between requests and responses. The information learned from interactively created connections can be used to add the missing edges into the graph. Subsequently created sequences can provide a higher number of predefined connections in order to further reduce the manual input required by testers. Ultimately, Testifi can use the new edges to improve it's AI's link detection.

While providing all the functionality needed, the tool should also meet user experience goals. The software provided for the testers should be intuitive and easy to use for users. State of the art concepts for working with graphs and workflows will be researched and used if applicable, while basic user experience concepts as well as concepts concerned with supporting advanced users will be reviewed as well. The user interface will also be in accordance with Testifi's design guidelines to match the other products it will later be implemented with.

Based on the goals, the following research questions are to be answered in this project:

1. Is the test quality increased by using the sequence generator?
   - Are more edge cases / special scenarios tested compared to manual testing?
   - Can more complex scenarios be tested by using the tool?
2. Does using the sequence generator reduce the amount of work necessary to create test sequences?
   - Is there a measurable improvement compared to creating / executing them manually?
   - Does the increasing number of known links reduce the necessary amount of work further?

1.5 Report Structure

The report begins with a literature review about the user interface and user experience theories related to the task at hand, that offers insights in existing concepts that are used as a foundation for the further chapters. After introducing the main terminology specific to the project, the report then continues with the user interface design and user experience concepts that are based on the results of the literature review. Next, the report goes into detail about the application design of the frontend and backend components, before transitioning into the implementation part, which goes into detail in how the final product was implemented. After reporting on the methodology and results of the user tests, the report concludes with an extensive discussion of the project results and a final conclusion.

2 Foundations and Literature Review

This chapter lays the groundwork for understanding the key principles and concepts that underpin effective UI design, as well as providing an overview of the most important user experience concepts for advanced users. It also delves into the exploration of the OpenAPI Specification (OAS) as a foundational standard for describing Hypertext Transfer Protocol (HTTP) Application Programming Interfaces (APIs), essential to our application's development.

2.1 UI Design Principles

2.1.1 Basics

Many of the principles that are used by default in UI design today were introduced when UI research started shortly after the introduction of graphical user interfaces 40 years ago.

In 1996, when GUIs were often sill rudimentary and a lot of potential to visualize was left unused, Ben Shneiderman devised the "visual information seeking mantra":

> Overview first, zoom and filter, then details-on-demand [1].

He uses this mantra when designing new user interfaces that visualize complex and multi-dimensional data that then in-turn can form relationships among its items. Today this principle is still very much applicable when visualising complex data.

To start off users need an *overview* that presents them with an high-level view of the available data. The user can get an idea on what already exists without having to deal with any detail. This creates a experience where they are encouraged to explore the data and dive deeper. By *zooming and filtering* the items in the overview can be reduced to only items that are relevant to the user. Finally by selecting an item the user is presented with the *details on demand*.

Still there are many different techniques to create interfaces that hold true to Shneiderman's mantra. Freitas summarizes different approaches [2] that were established in recent years. These are about providing an overview an detailed information. While in the first approach, overview and detail is separated, in the second approach, overview and detail are sections of the same view.

Shneidermann defines eight golden rules of interaction design [3, pp. 74, 75]. The rules are very general and can be applied to almost any user interface. Only the rules that bear the most importance regarding this project will be discussed here, but the other rules should not be disregarded at all. His first rule is to *strive for consistency*, which can be subdivided into many categories of consistencies. It can be terminology used in the application or the consistent application of layouts fonts and colors.

In the second rule *Cater to Universal Usability*, Shneidermann describes how a system should accompany a wide range of users with different backgrounds. Luckily in this application the target audience is relatively narrow, as the application is to be used by testers and software engineers. These groups posses a good technical understanding, so the focus can be shifted to fulfill their needs.

In the seventh rule called *Support internal locus of control* it is described how the interface should feel to the user like they are in control. The user should feel like they are the initiators of actions and not responding to actions. Especially for advanced or expert users this feeling of being in charge can be very important.

While user interfaces have evolved substantially since the definition of these rules, they still hold up well, as they are not limited to any sort of format of screen but can be applied universally from apps on smartphones to business applications.

2.1.2 Reducing work

One important aspect of creating digital experiences is to reduce the work the user has to do. Users want to have an experience where every interaction with the application brings them a step closer to the goal they have. Cooper [4, p. 271] dissects these interactions into four types of work a user has:

- Cognitive work
- Memory work
- Visual work
- physical work

For each interaction the user has with the application these types of work are like a tax the user pays, referred to *excise* by Cooper [4, p. 272].

The goal of any application should be to minimize these excise tasks and focus on goal-directed tasks that guide the user towards their goal. Where in contrast excise tasks are only done because they are required in order to perform some other goal-directed task.

2.1.3 Existing Solutions

Having laid out a foundation of principals that are important for a good user experience, it makes sense to see how existing solutions function and if there are obvious fields of improvement.

Taking a look at Swagger UI there are a lot of excise tasks as mentioned in section 2.1.2.

The following are some observations of excise tasks a user has to do when using Swagger UI to test a certain endpoint.

Search

A user is presented with a long list of all available endpoints. This provides some good overview for users who want to explore what an API has to offer, but the excise of navigation, more specifically scrolling through the list can be annoying to a user who is looking for a specific endpoint. When applying the visual information seeking mantra described in section 2.1.1, it becomes obvious that the "zooming and filtering" part is missing.

Using endpoints

Before even being able to use an endpoint the user is presented with a lot of excise tasks. First the endpoint has to be selected in the list. Then the user has to confirm by clicking on a "Try it out" button, as shown in fig. 2.1, that their intention actually is to use this endpoint. Then the user can start with the goal-directed task of filling out the parameter fields, before finally being able to send the request.

Navigation

In the previous example one important detail was left out. When a user wants to try out an endpoint it often needs some other information like an id. To obtain this information the API has to be queried first.

In Swagger UI's case this again involves the excise task of scrolling to another endpoint which will deliver the necessary data to input, just to then scroll until the other request is found again and the data can be input there.

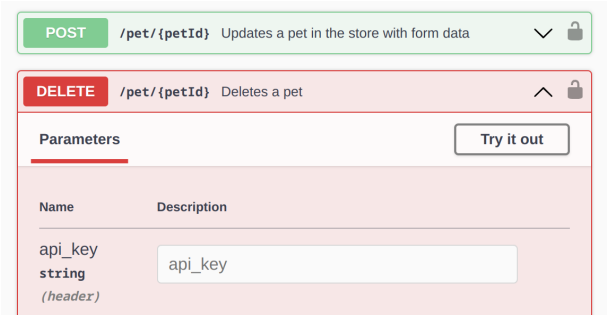


Figure 2.1: Screenshot of Swagger UI including "Try it out" button

2.1.4 Graphs and Workflows

Common ways of visualizing linked or hierarchical data are described by Freitas in the Handbook of Human Computer Interaction [2, p. 23]. These often include drawing graphs where nodes are connected by lines, in a node-link diagram. Another visualisation is the use of a matrix, where each relationship between two nodes can me marked as an entry in the matrix. For only showing relationships this view can be very helpful even if there are a lot of nodes and connections, but when some detail about the items should also be displayed the matrix visualisation falls short.

When using a node-link diagram there are some more considerations to be done. As there are many ways to arrange nodes and links. Freitas mentions different asthetic criteria [2, p. 24] that can be considered, such as edge crossing, where lines should cross as little as possible.

Endpoint Node-Link Diagram

The connections between endpoints as provided by Pulse can be displayed as a graph with the endpoints as nodes and links as edges.

Matrix

Research shows, that matrixes have advantages when displaying very dense graphs with many connections [5]. By not having the issue of edge crossings, the visualisation appears cleaner. Tasks like finding a particular node are a lot quicker in the matrix viusalisation [5, p. 22]. One drawback however is the missing directionality that can be easier to visualize using arrows in a node-link diagram.

Although there is a lot of research done in the field of optimizing graph visualisations, there seems to be no little to no research about the impact of these visualisations regarding the additional work a user has to do to process these visualisations as laid out in section 2.1.2.

2.1.5 UI Design For Advanced Users

As the basic principles are set and further research into graphs and workflows is unnecessary, some additional concepts to cater advanced users can be discussed.

Learning by doing

Research shows that advanced users in particular are usually more motivated to start working with an application immediately instead of learning it first through tutorials or documentation. Users should be supported in exploring an application and learning the interface through trial and error, while being given the possibility to constantly see the result of their work [6, Chapter 1].

Accelerators

Advanced users are generally spending more time working with a product. As they get to know an application better, they want to minimize the time needed for their workflows, which can be achieved by reducing unnecessary steps and context switching, minimizing unnecessary repetitive work and introducing accelerators [7].

An accelerator in the UI/UX world describes a feature that speeds up an interaction or process. It's goal is to provide an alternative method for an action, which is usually faster but possibly more demanding. Accelerators (or shortcuts) are created to be used by advanced users to complete known tasks quickly and efficiently. Other users should be able to ignore them completely, as they should always just be an alternative to the standard way of performing a task.

Common accelerators include:

- Keyboard shortcuts
- Marcos
- Touch gestures

While not all actions require an accelerator, they are best used for features that users tend to use repeatedly [8].

Important Information

To help users find and act upon important information, critical elements have to stand out from their surroundings. This does not need to be done by adding emphasis to that information (e.g. by adding a color), but can also be achieved by removing nonessential elements. For example, removing graphics and visual elements that serve no purpose can make the data left behind stand out [6, Chapter 8].

Removing visual elements can also help the user understand the application better. By removing clutter and only displaying things when they are needed, the feature and function overload can be minimized, without reducing the capability of the application [6, Chapter 6].

Secondary Information

When clutter is removed, some information has to be moved to secondary levels, as it might not be important for the user at all times but is needed to contextualize or provide a deeper level of detail. In these cases it is recommended to ease the transition between primary and secondary information by not switching the context of the application.

Whenever possible, secondary information should be displayed in context with its related primary information. This can be achieved by displaying the it in the same screen or environment [6, Chapter 7], instead opening a new context.

2.2 Open API Standard

As this project heavily relies on OAS definitions, the concepts and definitions of the standard have to be understood before delving into the implementation.

The OAS, formerly known as Swagger is a specification for describing HTTP APIs. The specification was first introduced in 2010 and since evolved to version 3 which was released in 2017 [9].

This literature review will focus on the latest version 3.1 of the specification [10], as there are significant changes made to the format between version 2 and 3 [11].

A baseline understanding of the OAS is required as the whole application will be based on handling a wide range of the features and the flexibility that the specification offers to developers when describing their APIs.

2.2.1 Basic Structure

The specification defines a single file as an OpenAPI document. An OpenApi document includes multiple sections of which only some are required. The base of the document is defined as an OpenAPI Object[10, Chapter 4.8.1]. Documents can also reference external definitions in many cases where for example schemas are described.

Info
Includes information about the API that is described. For example the title of the API is provided here.

Servers
This section contains the addresses where the HTTP requests can be sent to.

Paths
This section contains all Paths the API is exposing. For each path it is defined which HTTP verbs can be used. Response and request schemas can be defined for each Method independently, as a "GET"-Endpoint might return a list of Resources and a "DELETE" Endpoint might not return any data as a resource is deleted. Each of these path items is referred to as an operation, as described in section 2.2.1.

The specification also allows different content formats for requests and responses depending on the content type used respectively.

Parameters are part of the request schema and can be placed in these locations: "query", "header", "path" or "cookie". These parameters are relevant for our application as they should be used as link targets.

Operation
An operation is the equivalent to an actual API-call. It defines how a request and the subsequent response look like[10, Chapter 4.8.10]. Listing 1 shows an example of an operation object, where all data is directly included in the definition without the use of references.

The format of a request can be defined directly inside the operation, or these might be references to generic definitions in a schema by using a reference object [10, Chapter 4.8.23].

```
{
  "tags": [
    "pet"
  ],
  "summary": "Updates a pet in the store with form data",
  "operationId": "updatePetWithForm",
  "parameters": [
    {
      "name": "petId",
      "in": "path",
      "description": "ID of pet that needs to be updated",
      "required": true,
      "schema": {
        "type": "string"
      }
    }
  ],
  "requestBody": {
    "content": {
      "application/x-www-form-urlencoded": {
        "schema": {
          "type": "object",
          "properties": {
            "name": {
              "description": "Updated name of the pet",
              "type": "string"
            },
            "status": {
              "description": "Updated status of the pet",
              "type": "string"
            }
          },
          "required": ["status"]
        }
      }
    }
  },
  "responses": {
    "200": {
      "description": "Pet updated.",
      "content": {
        "application/json": {},
        "application/xml": {}
      }
    }
  }
}
```

Listing 1: Excerpt of Operation Object Example. Adapted from the OAS [10, Chapter 4.8.10.2]

Schemas
The schema section allows the author of the definition to define reusable schemas that can be referenced from other parts of the specification. This allows for flexible constructs and re-use of the same schemas across different paths and operations.

Schemas can also reference other schemas to create hierarchical as well as recursive data structures.

Security
The security section allows APIs to describe what security mechanisms are used for individual operations. This allows developers to even specify which permissions a user of the API would require to access a certain operation [10, Chapter 4.8.27].

2.2.2 Data Types

Data Types in the OpenAPI specification are based on the JavaScript Object Notation (JSON) Schema Specification. Since the latest version all JSON Schema Types are fully supported by the OAS [11].

Hierarchical structures as described in section 2.2.1 are possible due the definitions in the JSON Schema Specification that defines the keywords used for applying sub-schemas with logic [12].

OAS recognizes all keywords of a JSON schema, which are *allOf*, *anyOf* and *oneOf*. These keywords allow for conditional dymamic sub-schemas, according to Drotteboom [13, p. 56] they can be compared to logical gates and are to be interpreted the following way:

- allOf (AND) All the subschemas must be valid
- anyOf (OR) At least one of the subschemas must be valid
- oneOf (XOR) Exactly one subschema must be valid

By also introducing a discriminator the OAS can be used to get additional context on the subschema that is to be expected [10, Chapter 4.8.25]. The schema type can still be resolved based on it. The discriminator is a field in the top data structure that defines which schema is to be expected in the inner data.

2.2.3 Relevant features

It is challenging to create an application that can handle every feature of the OAS. Implementing all features would require to implement a lot of edge cases and combinations of features. Instead the focus should lay on supporting the most common use cases and specifications found in the real world. These are the key focuses chosen based on the real world usage.

Media Types
Cloudflare, a company offering content delivery network (CDN) services published statistics regarding the usage of API and web traffic in general.

This offers some insight on the most common used media types in APIs. Their findings show that JSON compared to Extensible Markup Language (XML) is used in 97% of requests [14].

Therefore, this application should focus on supporting JSON requests.

Response codes
Most APIs are built to respond with a successful HTTP status code in 97% of cases [14], as documentation tools such as OAS help developers avoid creating requests that cannot be handled by an application. In contrast, HTML web traffic has a lower success rate of 91%, as this traffic

stems mostly from end users, who might use an invalid link or could have mistyped a web url. Therefore, a response with a 20X status code can be considered successful in an API test while all other codes can be considered failed.

Data structures

As the OAS focuses on documenting Representational State Transfer (REST)ful APIs. We can assume that APIs will respond with singular resources in a standard JSON format. All common features of the JSON format should be implemented such as nested objects and arrays.

Methods

According to cloudflare, "The vast majority of API traffic is the result of POST or GET requests (98% of all requests)." [14] This shows that these methods should be supported by the application.

3 Key Concepts and Terminology

In this section, some of the main terminology that will be referenced in this paper is introduced. Further terminology is described in the glossary at section 10.

Application
In the context of this paper, the *application* is the whole 'Sequence Generator Tool'. It consists of the individual components, i.E. the frontend, the backend, and also any configuration or scripts that are necessary to run either the frontend or backend.

Project
A project in this context is an instance that wraps all data for a specific REST API. It corresponds with one import of an OAS definition into the already existing Pulse application and is based on the OAS definition as well as the link file created by Pulse during such an import. Multiple sequences can be created in a project.

Sequence
A sequence describes a list of related endpoints. It is the main entity of the application as it consists of endpoints to test, the order in which they are to be executed, the links between them as well as test data to be used in the requests.

Endpoint
An endpoint in a project or sequence represents any HTTP endpoint that was imported from the OAS definition during project creation. Endpoints can be added to sequences and their respective response values can be linked to request values of subsequent endpoints.

Link
A link represents a connection between a request value (an endpoint parameter or a field in it's request body) and a source for that value. The source can either be a manually created test data field, or a field in a preceding request's response body.

Link Suggestion
As Pulse already tries to find possible links in the API through AI, and there potentially were already manual links created for an API, possible links can be suggested to the user.

4 User Interface Design

This chapter explains the user interface design of the application. The final design is the result of multiple iterations with initial user tests, feedback from the customer, and workshops with Fabian Affolter.

The application's user interface consists of two main screens. The overview screen for a project shows a list of the created sequences, lets the user filter them and provides actions for the sequences. When a sequence is selected, the editor screen is presented to the user, which lets them change the endpoints in the sequence, add links and verify their changes.

4.1 Overview Screen

The first screen a user sees, when the application is started, is the sequence overview of a project, as shown in fig. 4.1. It displays a complete list of the sequences in the project and presents multiple possibilities to filter those sequences.

The user can choose endpoints that are included in the sequence they are looking for as well as adding keywords that must be included either in the title or description of the sequence.

Both filter actions are performed through the search field. When the user types, they can either select an endpoint from the appearing drop-down, that shows the projects' endpoints that match the typed term, or they can hit enter to add the typed text as a search term.

For each sequence, the user has the possibility to edit, verify, or delete it. The main focus of the application however is on the edit part.

Further details on how these user interactions work are provided in section 5.5.

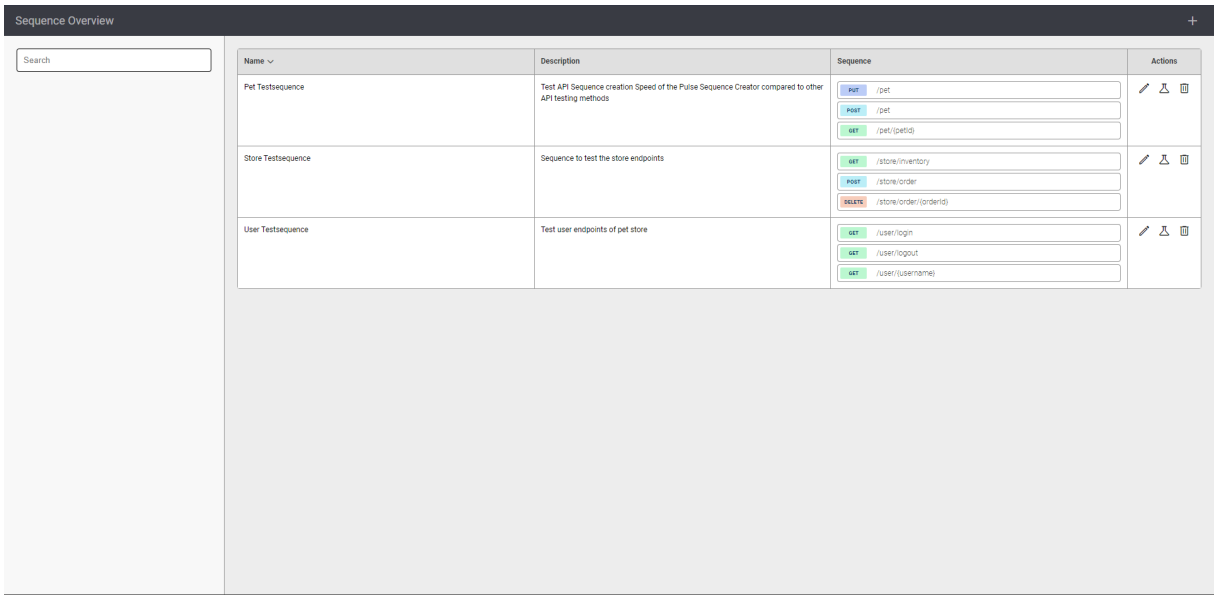


Figure 4.1: Screenshot of the overview screen in the final product

4.2 Editor

When a selected sequence is edited or a new sequence is created, the editor screen visible in fig. 4.2 is opened.

It consists of three sections, the first being the test data column, which lets the user add, edit, and delete test data fields that can be used as sources for links. The second column shows the sequence flow, which consists of endpoints that will be executed one after another. In this column an endpoint can be selected to show its details in the third column and add or edit links.

Endpoints can be moved and deleted from the sequence directly from the sequence flow, by either hovering over them to make the respective buttons visible or using keyboard shortcuts. Moving endpoints is also possible via drag-and-drop. More information about the keyboard shortcuts can be found in section 5.3.

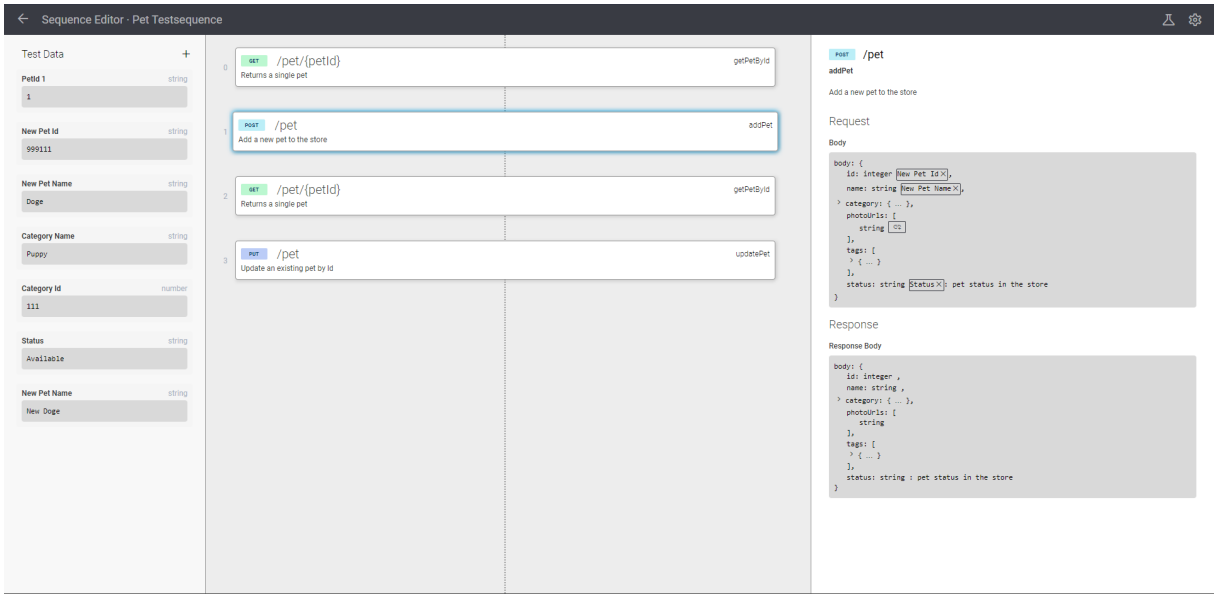


Figure 4.2: Screenshot of the editor screen in the final product with an endpoint selected

The second column also lets the user add endpoints to the sequence by clicking on the 'plus' icon between any endpoints or at the end of the sequence, which opens a selector and displays all available endpoint in the third column. In the selector, the user can search for text that appears in the *path* or *operationId* attributes of the endpoint and use the method filter to only show selected HTTP verbs. This behavior is showcased in fig. 4.3.

In the header bar, the user can go back to the overview with the 'arrow' icon on the left, verify the sequence with the 'science' icon on the right, or edit the name and description using the 'cogwheel' icon.

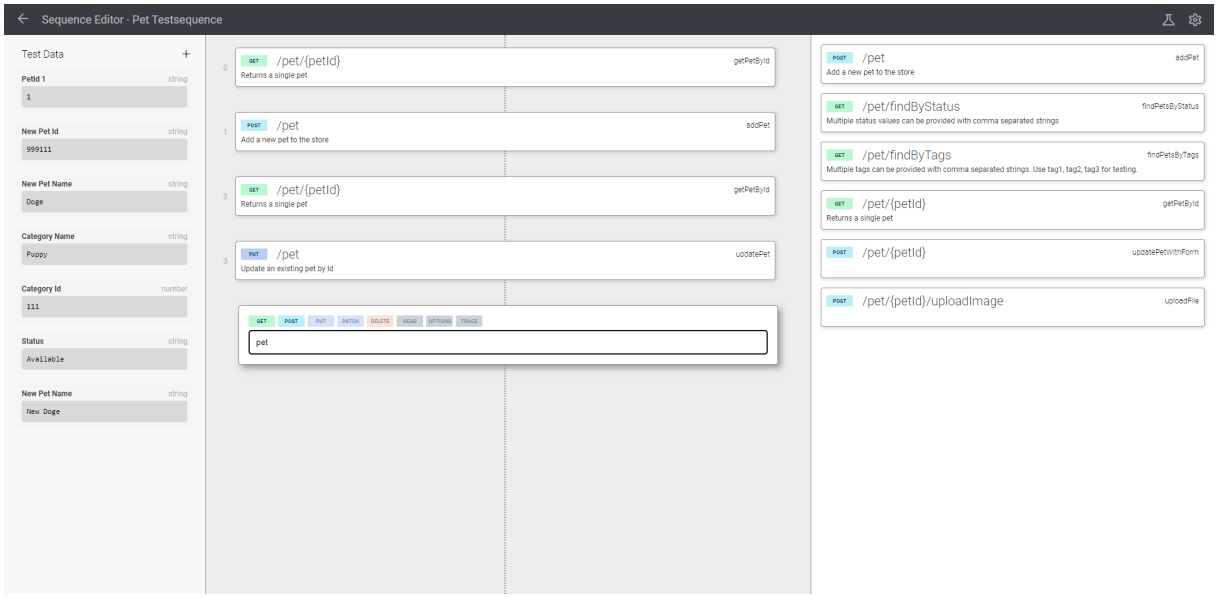


Figure 4.3: Screenshot of the editor screen in the final product when adding a new endpoint

4.3 Low-Fidelity Design

The low-fidelity design visible in figs. 4.4 and 4.5 was created through multiple iterations with input from Testifi employees Nejdet Dogru, Gabor Bakos, and Ece Kilis as well as the coaches Martin Kropp and Fabian Affolter. The iterations were used to get an understanding of what the customer wants and needs as well as developing a concept of user interaction with the application. The low-fidelity design represents the initial idea of the user interface structure without adding too much detail on design aspects and overall look and feel.

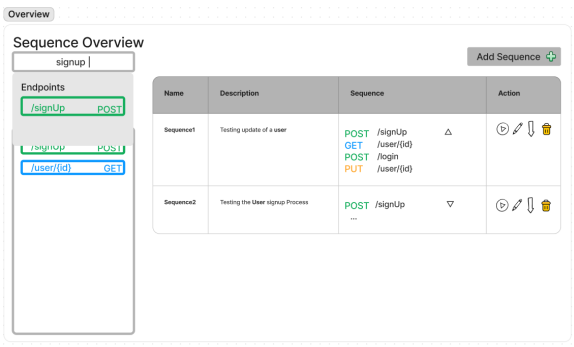


Figure 4.4: Low-Fidelity Design: Overview Screen

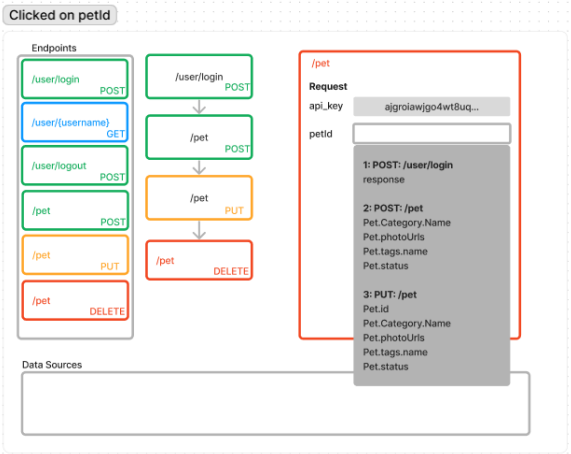


Figure 4.5: Low-Fidelity Design: Editor Screen

4.3.1 Initial User Tests and Workshops

After the creation of an initial low-fidelity design, user tests were run in order to understand if the design is intuitive and provides all expected functionality. The methodology and results of these tests are documented in section 8.1 of the report.

The results of these tests, as well as workshops with Fabian Affolter, lead to numerous changes on the design during multiple iterations.

An example of such a change is, that in early versions the sequence overview screen distinguished between list-operations and row-operations. Even though deleting a sequence in the list would be considered a list operation, and therefore warrant an option for the user to delete multiple sequences at once, it was decided, based on user feedback, that the delete button should be grouped with the sequence actions as a column action. As the creation of a complex sequence is time consuming, avoiding accidental deletion of sequences bears more advantages to the user than an easy process to delete multiple sequences. Furthermore, deletion of many sequences at once can't be considered a use case that would be often encountered.

Figure 4.6 shows that to further secure the user from accidental deletion of a sequence, upon clicking the delete button, a popup will open for the user to confirm or cancel the operation.

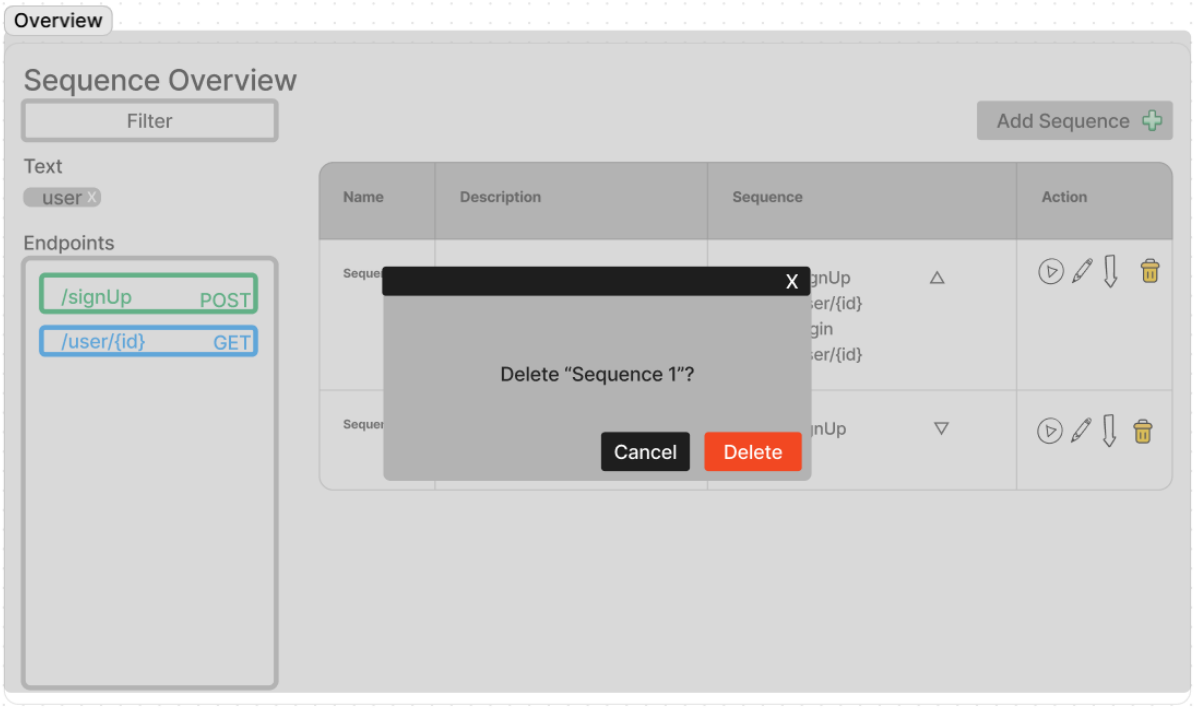


Figure 4.6: Low-Fidelity Design: Delete Sequence Pop-Up

As there was also confusion around the 'plus' button to add a sequence to the list, which was also considered a list-operation, the button first received an 'Add Sequence' label in further iterations, which was removed again as it represented the only remaining list-operation and therefore was moved to the header bar.

As a direct result of the initial user tests, the 'play' button on the overview screen became the 'verify' button, matching the functionality on the editor screen and removing the confusion around the 'play' icon, and the spacing between icon buttons was increased, to avoid users clicking the wrong button by mistake.

Based on user feedback, drag-and-drop functionality for endpoint moving was introduced on the editor screen. Save and cancel buttons and the possibility to directly edit the sequence name were introduced based on the feedback, but later dropped in favor of an auto-save feature, which promised a bigger improvement in user experience.

The linking drop-down was removed in a later iteration and replaced by visualizing the linking sources directly in the respective endpoint tiles in the flow, as shown in fig. 4.7, which allowed better scalability while also creating a strong visual connection between the value and the endpoint it originates from.

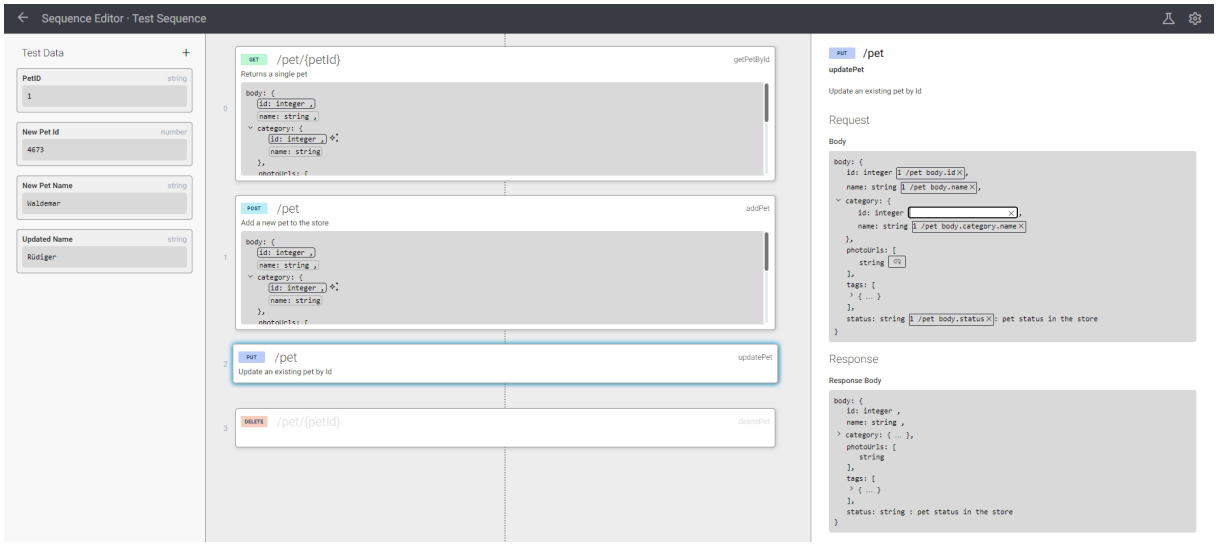


Figure 4.7: Screenshot of the editor screen in the final product when selecting a link source

The sequence name and description will usually be set upon creation and not changed often. During design iteration, how these values were displayed and updated was changed multiple times. In the final design, in order to keep a clean editor view, the name of the sequence is only displayed in the header bar, while the description is hidden on the editor view and might only be changed by opening the settings pop-up, which also provides an option to update the name.

The "Data Sources" section was later renamed to "Test Data" to make its purpose as data used in the sequences more clear and was moved to the left of the editor screen as it takes a prominent role during sequence creation. In fig. 4.7, the position of test data as well as the design during link selection (where test data values are available as link sources) can be found on the left side of the screen.

4.4 High-Fidelity design

The initial high-fidelity design was created in cooperation with UI design intern Ece Kilis, who worked at Testifi during the first part of the project.

After multiple workshops and a user test, the high-fidelity design of the editor was completely reworked, in order to be more appealing to the user. In the reworks, the design philosophy was redone as well as multiple functionality changes were introduced to the design concept of the editor.

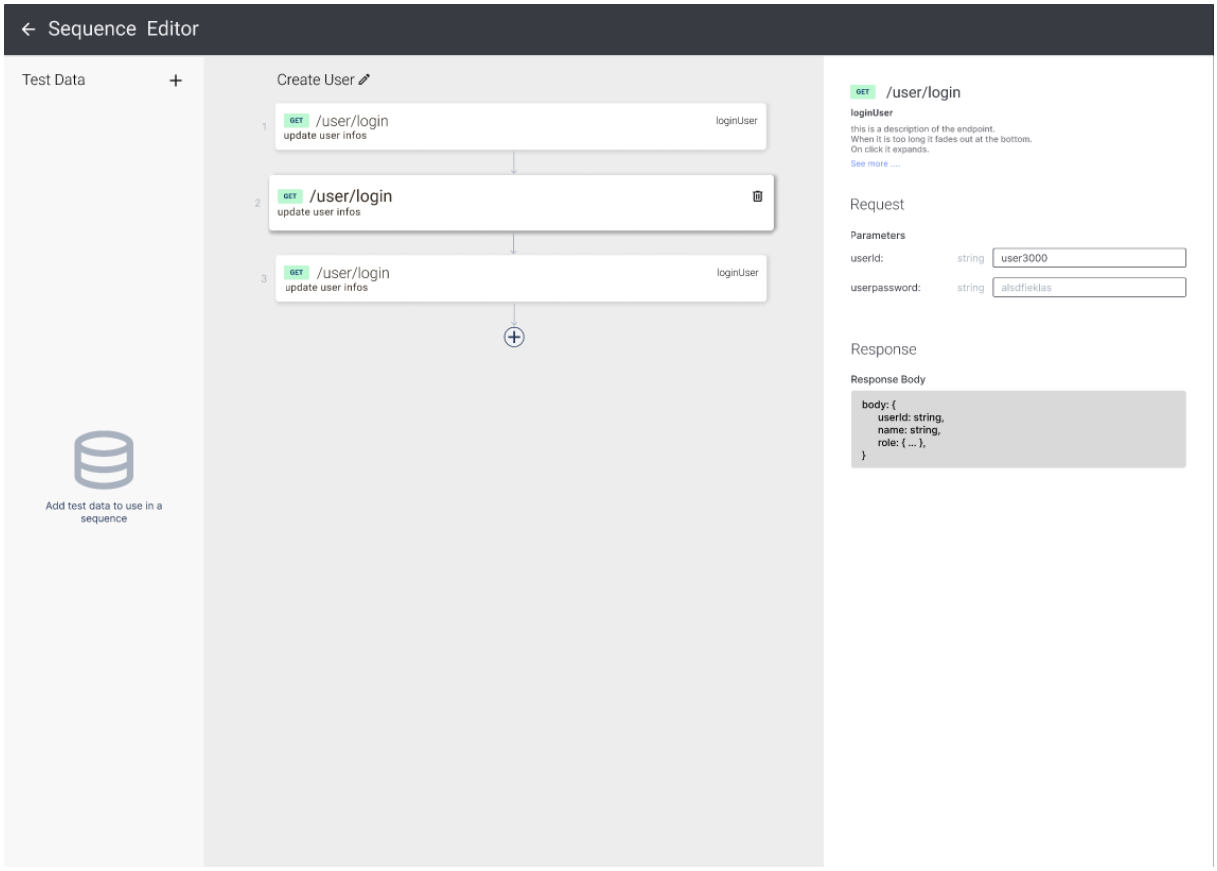


Figure 4.8: High-Fidelity Design: Editor Screen

Compared to the low-fidelity design, the high-fidelity design shown in fig. 4.8 contains a lot more details and incorporates the planned design completely instead of focusing only on functions and positioning. The color design was updated according to the Testifi design guide and text, spacing and position was defined to be used in the final UI of the application.

In the high-fidelity design, not every screen or user flow was represented, as it is was mainly used to communicate an idea of the look and feel of the final application. For more development efficiency, it was decided that the final details are directly implemented and then discussed and changed if needed. With this faster feedback loop, the necessity for complete high-fidelity designs for every page was cut out.

4.4.1 Testifi Design Guide

Testifi provided a design guide that defined fonts, text sizes, colors and effects to be used in the design. The final design largely adheres to this guide, while some colors were added or altered. An example of this are the colors of the HTTP badges for the endpoints. Lighter colours were chosen, which fit better into the design and do not interfere with each other when displayed next to each other (as on the endpoint selector shown in fig. 4.3).

5 User Experience

This chapter discusses the user experience concept of the frontend application. First, how the findings of section 2.1 are applied is discussed. Secondly, the available keyboard interactions are documented in detail and finally, user flow diagrams outline the main actions the user is expected to perform.

5.1 User Guidance

Based on literature review findings documented in section 2.1, key areas to improve the user experience for advanced users were identified and integrated into the user interface and user interaction designs. Still the user should feel like they are in control, as defined in the eight golden rules. The experience needs to be consistent by re-using elements such as the *endpoint-tile* across the application.

5.1.1 Overview and Detail

When applying Shneidermans mantra discussed in section 2.1.1 to our goals the following structure can be devised.

Sequence Overview
Overview: Get an overview of a project by listing the sequences of the project while only showing limited information about each sequence on the overview screen.

Zoom and Filter: Filter the sequences by their attributes and contents. Allowing users to use both endpoints to be included as well as text terms to filter the sequences satisfies this requirement.

Details on demand: By switching to the editor screen for a selected sequence, the user has access to all details needed, which reflects Shneiderman's first approach for data on demand.

Sequence Editor
Once on the editor screen for a specific sequence, the same approach can be employed again:

Overview: For an existing sequence, an overview of endpoints in the sequence and available test data is displayed. For a new sequence, the selector directly shows an overview of available endpoints. No endpoint details are visible.

Zoom and Filter If a new endpoint is added, the endpoint selector provides multiple filtering and search capabilities to filter the overview of available endpoints. This is explained in great detail in section 5.1.6.

Details on demand By selecting an endpoint in the sequence, the user can access the endpoint details on demand, without disturbing the overview. This reflects Shneiderman's second approach for data on demand, which is described in section 2.1.1.

5.1.2 Learning By Doing

As documented in section 2.1.5, it is a good idea to encourage the user to learn about the application by exploring its possibilities on their own and providing a way to see if their actions are successful or not.

One way to support this in the Pulse sequence generator tool is the 'verify' feature, which let's the user execute a created sequence from the overview or editor screen. While on the overview screen, only the success of the whole sequence is visualized, the editor screen provides an execution result for every endpoint, which helps the user find errors and improve the sequence.

5.1.3 Accelerators

As described in section 2.1.5, accelerators are a great way to support advanced users of an application in reaching their goals faster with alternatives to the standard workflow. In this case, these accelerators should provide advanced users with a faster way to interact with the frontend and therefore reduce the overall time to create a sequence.

The frontend application has two main accelerators to improve the user experience for advanced users. One is the possibility to reorder the endpoints in a sequence by drag-and-drop instead of using the move buttons that appear on hover for each endpoint. To provide an alternative to mouse control for certain actions, the application also supports an variety of keyboard shortcuts, which are documented in section 5.3.

5.1.4 Important Information

Section 2.1.5 provides insights in the significance of highlighting important information, but also emphasises that this does not always mean that more highlighting has to be added, as it's also a possibility that unimportant elements can be hidden. This helps the user find important information on first sight, but can also improve the efficiency by guiding the user to appropriate actions and help them understand the application better.

As also learned in section 2.1.5, whenever secondary information is hidden because it is not considered important, it should always be provided to the user in a way that doesn't remove the context with the primary element which relates to the opened information.

During creation of the UI design of the application, these aspects were considered and lead to multiple changes in the positioning of elements and actions.

One example for both these concepts is the endpoint details on the editor screen. While the endpoint description shown on the details can be helpful for the user to understand the purpose of the endpoint and sometimes provide important information about request and response value, they can also be cluttered with unnecessary information. As the OAS does not limit the length of the description, they can be very long, depending on the level of detail they provide. To make all this information available to the user while not taking attention away from the actually necessary fields, like request parameters and body, the detail view shows the description at the top but collapses if it's too long as visible in fig. 5.1. Figure 5.2 shows that the user can expand it as needed, but per default, the focus is on the fields the user has to fill out.

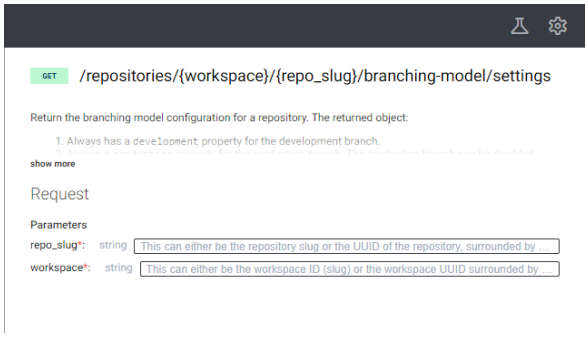


Figure 5.1: Example of collapsed endpoint details

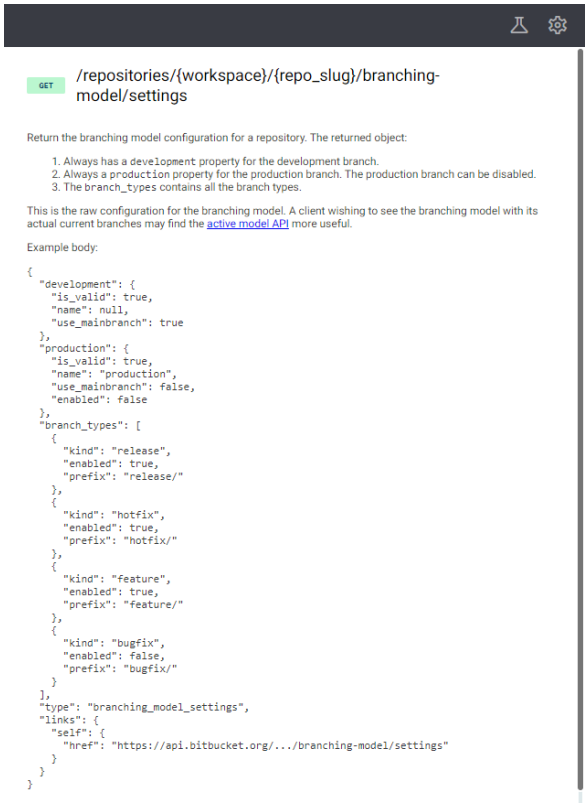


Figure 5.2: Example of expanded endpoint details

As seen on fig. 5.1 there are also different levels of information highlighting for the request parameters. While the parameter name, as the most important value, is best visible, the type (e.g. string) as well as the description in the input field are displayed in a more subtle grey color, which is done to guide the users' focus.

To further reduce clutter that distracts the user, some actions are only visible if the user hovers on the right location. This includes the buttons to move or delete endpoints in a sequence and the button to add a new endpoint at a specific location.

5.1.5 Secondary Information

As also learned in section 2.1.5, whenever secondary information is hidden because it is not considered important, it should always be provided to the user in a way that doesn't remove the context with the primary element which relates to the opened information.

This principle was used in the right column of the application, which can show the details of the selected endpoint as well as a list of endpoints to add to the sequence if the endpoint selector is used (as seen in fig. 5.4).

Figure 5.3 shows how when an endpoint is selected, the focus of the user is not disturbed by a modal or new screen appearing, but instead only on third of the screen changes, and the context is preserved, as the selected endpoint is still clearly indicated.

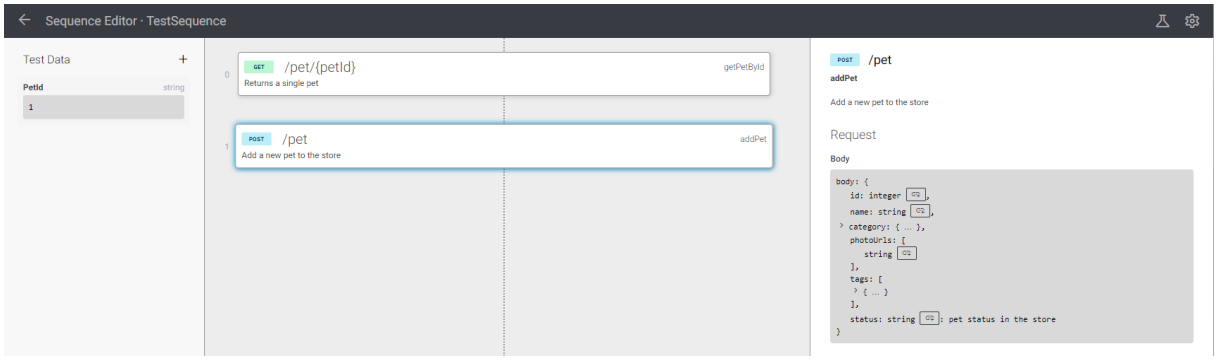


Figure 5.3: Screenshot of the editor screen showing the endpoint details

5.1.6 Reducing Work

To create an experience that reduces work and excise tasks as described in section 2.1.2, many considerations went into the design of the application. As discovered in section 2.1.3, especially navigational excise is an issue that needs to be dealt with.

By only displaying the list of available endpoints when the user is actively trying to add a new one, we can reduce the amount of data shown on the screen and thereby reduce the visual overload.

Further the list can be filtered as shown in fig. 5.4, allowing users to first get an overview, then filter and then select to get a detailed view as discussed in section 2.1.1.

The navigational excise is greatly improved compared to Swagger UI's approach described in section 2.1.3. By separating the task of selecting the correct endpoint from the task of linking the data between endpoints, navigational paths become shorter.

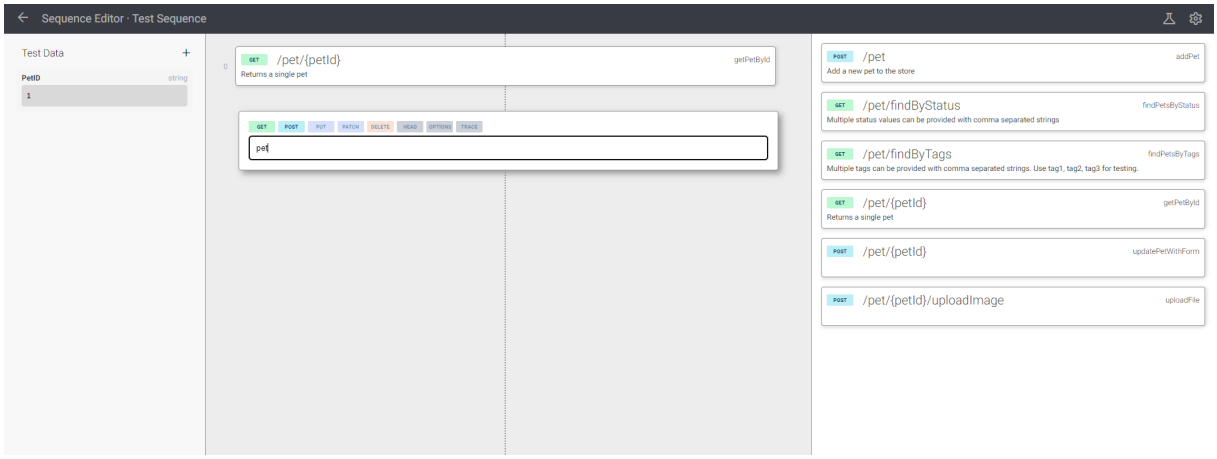


Figure 5.4: Screenshot of the editor screen showing the endpoint selector with active filters and a search term

5.2 Autosave

Users create sequences in a normal work setting. Ideally users are able to fully focus on the task at hand and are able to create sequences without distractions. Often these ideal conditions are not found in a real-world setting where distractions like meetings, incoming e-mails or chat messages might disrupt the task at hand. An autosave functionality lets the user to keep working

on the sequence creation after being distracted for a longer period of time. In some cases users might even switching the device used and want to pick up the work where they left off. Therefore a autosave features takes off the additional cognitive load of remembering to always save the progress.

Any action performed by the user will immediately be saved to the backend.

5.3 Keyboard Interaction

The keyboard shortcuts are available in different parts of the editor application and can have different assignments based on the situation they are used in. The shortcuts are detrimental in providing advanced users additional speed and support a smoother workflow. In some cases, they can also reduce the harm of context switching, as advanced users can confirm or decline a confirmation pop-up with one click.

5.3.1 Endpoint Flow

The endpoint flow in the center of the editor screen lets the user add, remove, move and select endpoints. While the user selected an endpoint there, the following actions are available, as table 5.1 shows.

Keys	Action
⇤⇥	Select next endpoint (downwards, looping)
↓	Select next endpoint (downwards, looping)
↑	Select previous endpoint (upwards, looping)
⇧ + ↓	Move endpoint downwards
⇧ + ↑	Move endpoint upwards
Del.	Delete endpoint

Table 5.1: Endpoint selection keyboard shortcuts

5.3.2 Link Selection

As seen in table 5.2, when the user selects a linking target, all linking sources become visible. While the user can type in the input to filter the available fields, they can also use tab to navigate through the displayed response values of the previous requests and select a link source.

Keys	Action
⇤⇥	Navigate through displayed response body link sources
Space	Select link source
↵	Select link source

Table 5.2: Link selection keyboard shortcuts

5.3.3 Dialog

Some scenarios will prompt the user with a confirmation dialog. Whenever a dialog is opened, the user can confirm or close it with the keyboard shortcuts shown in table 5.3.

Keys	Action
Esc	Cancel dialog
Space	Confirm dialog
↵	Confirm dialog

Table 5.3: Dialog keyboard shortcuts

5.4 Usage Of Node-Link Diagrams And Matrixes

In section 2.1.4 it was explored which methods can be employed to visualize linked data in a graph. After exploring these possibilities, the decision was made to abandon these approaches.

The use of node link diagrams is beneficial for applications where nodes are all equivalent and their positioning on the canvas shows the proximity to neighbours. In this case the positioning of the nodes should also convey information about the nodes position in a chain of requests.

Also the matrix visualisation has the issue that it is difficult to incorporate detail information about each request without using too much space, as well as the additional navigation that can occur in both the x-axis and the y-axis. We found that the model can be visually simplified to a sequence, a one dimensional structure.

This makes it simpler to, at a glance, reason about what the order of requests is without having to follow any lines in a complex graph or matrix structure. Of course there are some trade-offs to this approach. When links are created between endpoints that are far apart in the sequence, the navigational excise increases significantly.

5.5 User Flow

The user flow diagrams represent user actions on a high level without showing any design details. They are supposed to provide an understanding of the user's interaction with the application for certain tasks.

Only the basic user actions are shown, while others (such as moving an endpoint, removing an existing link, etc.) do not require additional explanation at this point.

5.5.1 Filter Sequences

To filter the sequence list, a shown in fig. 5.5, the user can choose endpoints that are included in a sequence and add keywords that must be included either in the title or description of the sequence.

5.5.2 Delete Sequence

To delete a sequence, the user can click on the 'trash' icon in the actions column. As this is a rare action and accidental deletion should be prevented, a pop-up will ask the user to confirm the deletion before it is executed, as visible in fig. 5.6.

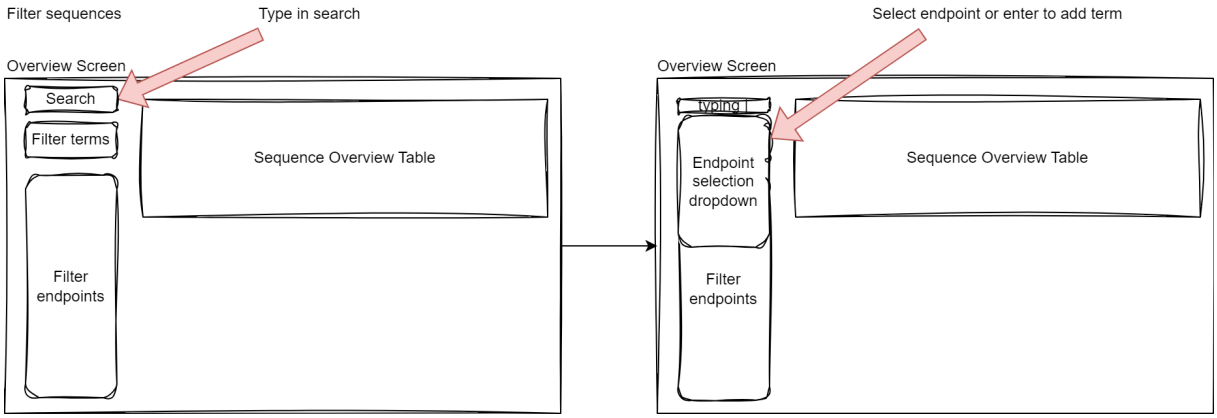


Figure 5.5: The user can add search terms and endpoints to filter the sequence list

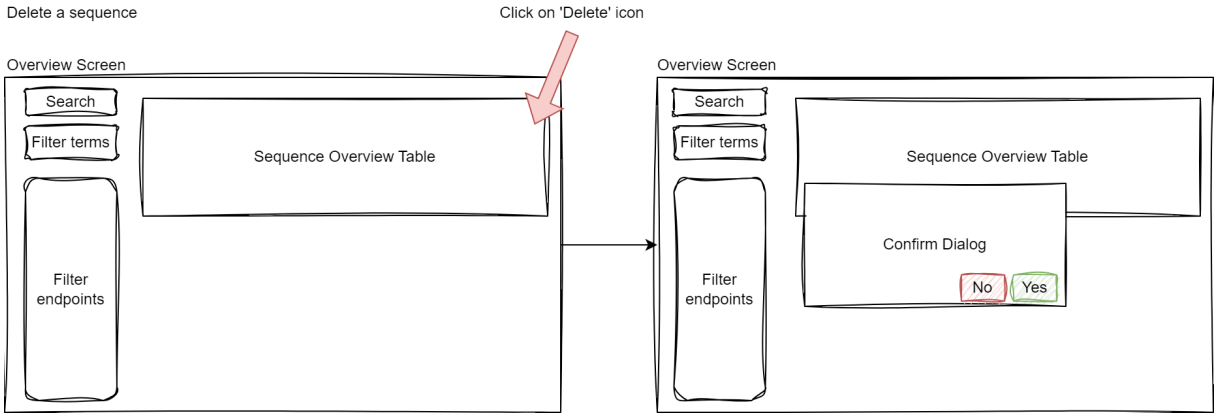


Figure 5.6: Sequences can be deleted by clicking on the trash icon and confirming the action in a pop-up

5.5.3 Add Or Edit Sequence

By clicking on the 'plus' icon in the header bar on the sequence overview screen, the user is redirected to the editor where a pop-up opens and prompts the user for a sequence name and description. While the name is mandatory, the description is optional and only used on the sequence overview page. Once the name is entered, the user can click create to start editing the new sequence or cancel to get back to the overview screen.

To edit an existing sequence the user can click on the 'edit' icon in the sequence overview table. Figure 5.7 show both of these actions.

When the selected sequence is empty or newly created, the editor screen directly shows the endpoint selector to filter the available endpoints, which are displayed on the right side of the screen. If the sequence already contains endpoints, those can be selected to show details, or a new endpoint can be added by opening the endpoint selector between endpoints or at the end of the sequence.

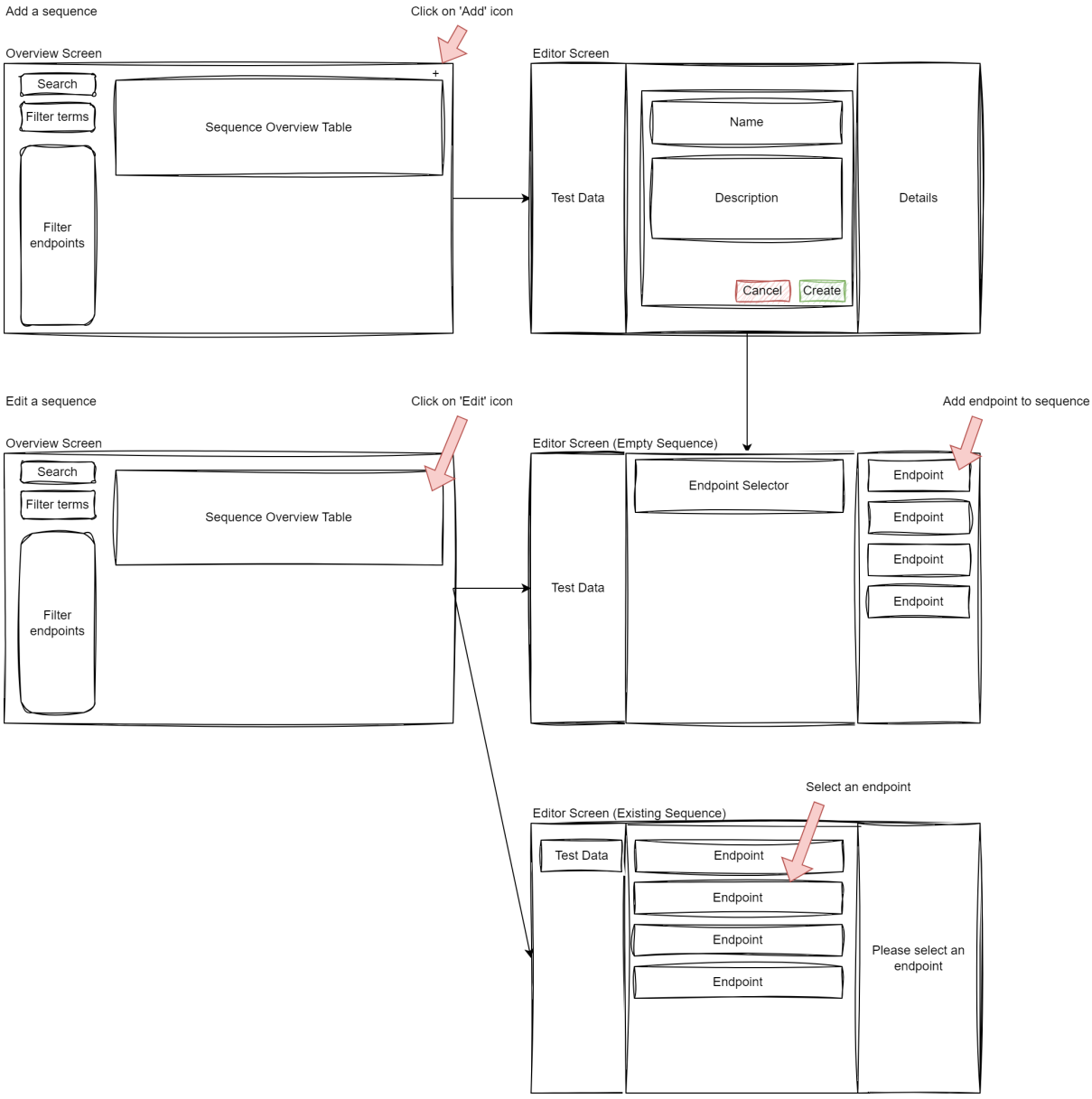


Figure 5.7: Adding a new sequence or editing an existing one.

5.5.4 Edit Sequence Details

If the name or description of a sequence needs to be changed, the user can always open the same pop-up as when adding a new sequence by clicking on the 'cog-wheel' icon in the editor's header bar and update the fields, as fig. 5.8 shows.

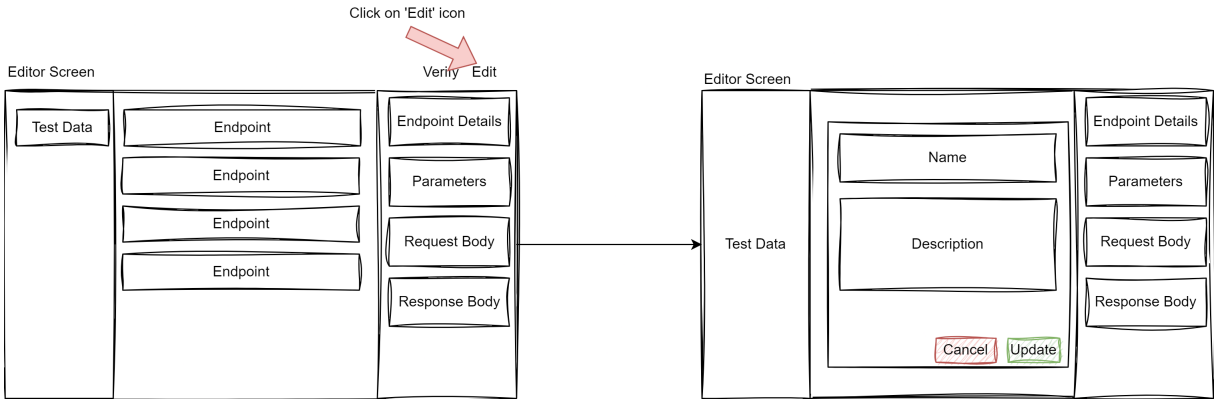


Figure 5.8: Editing a sequence by opening a pop-up to change the name and description.

5.5.5 Add Link

When endpoints are added to the sequence, the user can link values from endpoint response body objects or created test data to endpoint parameters or request body object fields to reuse returned data in subsequent requests.

Figure 5.9 shows that this can be done by selecting the target field first (either a parameter or a field in the request body object of the target endpoint) and the selecting the source field from available test data or response objects of the endpoints preceding the target in the sequence.

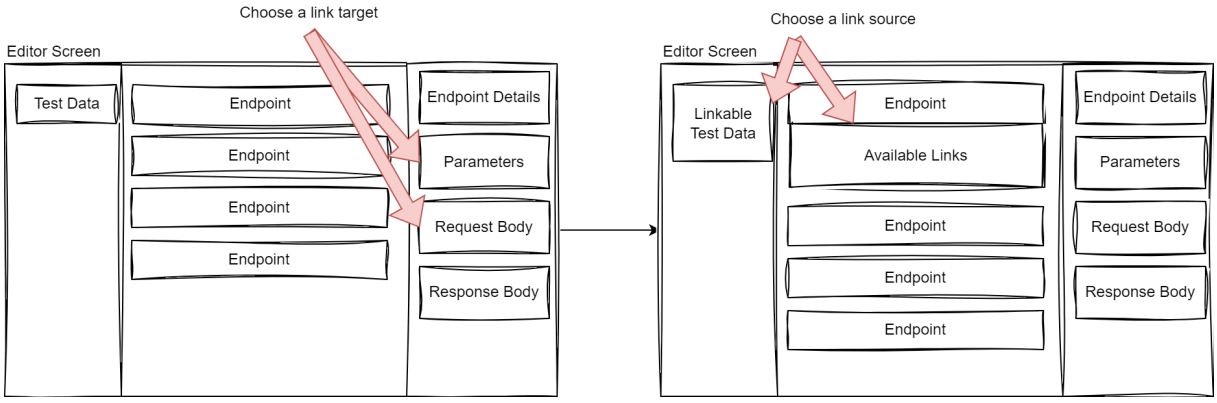


Figure 5.9: Choosing a target and source field for a link

5.5.6 Verify Endpoint

Once one or more endpoints are added, the user can check the sequence by clicking on the 'verify' icon in the editors header bar. As fig. 5.10 shows, the verification result for each endpoint will then appear next to the endpoints in the sequence.

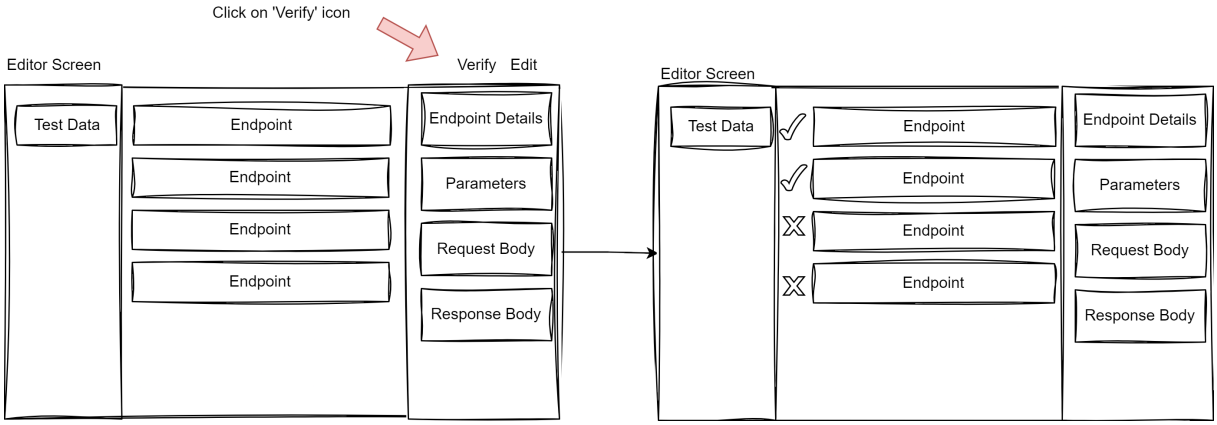


Figure 5.10: Verifying the sequence

6 Application Design

This chapter describes the design of the application and it's components, as well as their interaction. First, the software components and used technologies are introduces, before going into greater detail of each applications architecture. Figure 6.1 gives a broad overview of the interaction between the applications.

Most of the base technologies were defined by Testifi, as their other products are also based on Angular for frontend applications and Python with Flask for backend.

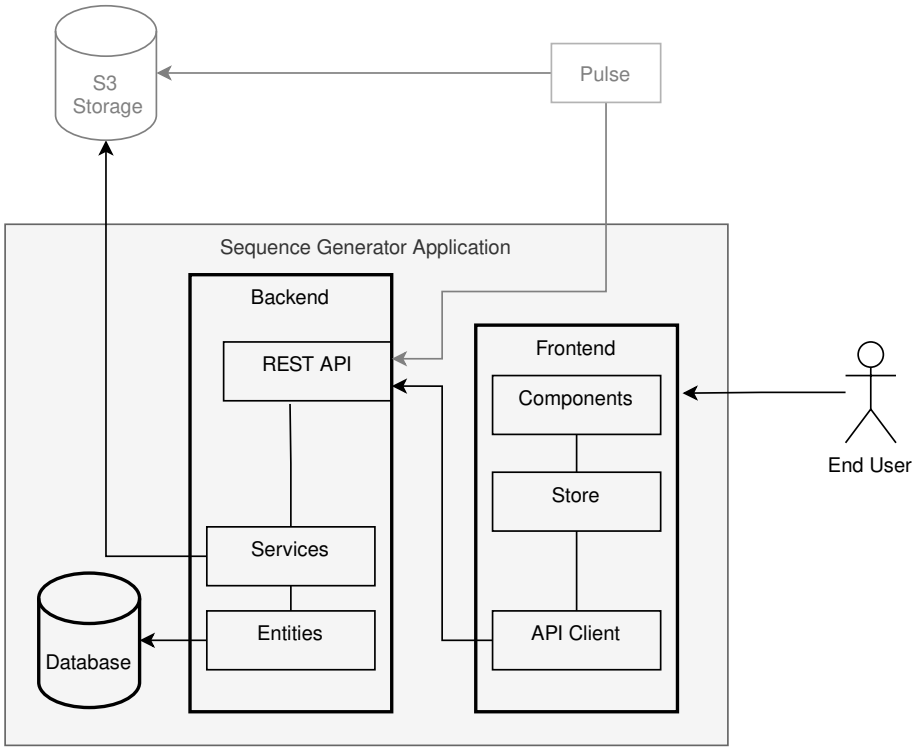


Figure 6.1: System overview

6.1 Frontend

The frontend of the application is an Angular 16 web application. It provides all user interaction and is designed to be used with any modern browser on standard notebook or personal computer screens. Usage with mobile phones or tablets is not supported.

6.1.1 Technologies

Angular

Angular is a single-page web application framework built on TypeScript. It is component based for scalability and provides a collection of libraries that cover a wide range of features [15].

While mainly developed by Google, Angular is open-source and free to use. For the frontend of this project, Angular 16.0.3 is used, extended by some commonly used libraries.

Notable libraries used in the frontend are:

- Angular CDK (16.0.3): The Component Dev Kit is a set of behavior primitives for building UI components [16] and is mainly used for pop-ups and overlays in this project.
- NgRx (16.0.1): NgRx Store provides reactive state management for Angular apps inspired by Redux [17]. NgRx is used for the applications' state management.
- RxJS (7.5.0): RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code [18]. RxJS is used all through the frontend application. NgRx also uses RxJS for the reactive state management.
- Marked (5.1.1): Marked is a markdown compiler that parses markdown from different flavours and outputs HTML code.
- Sass (1.59.1): Sass is a library that provides Cascading Style Sheets (CSS) language extensions [19]. In this project the .scss extension an its syntax are used, because it is a superset of CSS, which allows developers to also write standard CSS code.

6.1.2 Style Guide

To keep the application clean and adhere to state of the art coding standards, the official Angular style guide [20] was chosen as the policy to follow. The guide lays out the default structure an Angular application is supposed to have including folder structure, naming conventions and rules for components, directives and services.

The guide suggests to apply the single responsibility principle to all components, services and other elements. There should also not be more than 400 lines of code in a file and no more than 75 lines in a function.

Files and Structure

For the file names, it is best practise to use the type of file as a suffix before the actual filename extension, such as *.component.ts*, *.module.ts*, or *.service.ts*. Connected files should show the association by sharing the file name: *example.component.html/.ts/.scss* [20, Chapter Naming].

As for the folders, it is recommended to keep a clean structure from the start, to always have the application ready for extension without restructuring. The style guide lays out some simple rules [20, Chapter Application structure and NgModules] to adhere to from the start in order to achieve this, of which the most important are:

- All of the application's original code goes into the *src* folder.
- Third party scripts are not stored in *src*.
- A flat structure should be kept as long as possible, while a folder should not contain more than 7 files.
- Each component should have it's own folder where the standard *.html/.ts/.scss* files are kept.
- A separate folder per feature can be created to group associated components.
- Only one asset should be kept in a file. Each component and service is in its own file.

6.2 Backend

The backend of the application consists of a Python 3.10 application. All data the frontend displays is provided to it through a REST API that runs with Flask. Furthermore, the backend is responsible for persisting the data in a MongoDB database, parse the used OAS and communicate with a Simple Storage Service (S3) bucket over Secure File Transfer Protocol (SFTP).

The application follows a standard Model View Controller (MVC) architecture and uses a Object–relational mapping (ORM) layer to interact with the database. The application is divided into a controller, models, and services. The services are called from the REST controller and then use the models to interact with the database.

6.2.1 Technologies

Python

Python is a high-level interpreted programming language, which offers dynamic typing and dynamic binding and aims to be attractive for both the development of object-oriented applications and scripting. In addition to it's standard library, a wide range of third party libraries, modules and packages are available [21].

Notable libraries used in the backend are:

- Flask (2.3.1): a simple http server for python [22].
- Spectree (1.1.2): a library to generate OpenAPI documentation and for validating requests and responses [23].
- Odmantic (0.9.2): ORM for MongoDB using pydantic for model definition and validation [24].
- Pydantic (1.10.7): a library for data validation using type hints [25].
- Prance (23.6.21.0): a library for parsing OAS documents using the openapi-spec-validator library [26].
- Openapi-spec-validator (0.5.7): a library that validates documents against the OAS [27].
- Pysftp (0.2.9): a library to interact with file servers via SFTP [28].

Flask

Flask is a simple open source web-framework that provides HTTP endpoints in a python application, which is aimed at both beginners and advanced developers. While similar frameworks offer a wide array of libraries, Flask was created as slim as possible while providing the possibility to integrate third party libraries [29].

S3

S3 is a storage solution provided by Amazon Web Services, which claims to offer industry-leading scalability, data availability, security, and performance. It provides a simple REST API for reading and writing files into so-called storage buckets, as well as different access possibilities such as SFTP [30].

Testifi uses this storage solution to store schema files as well as linking files that are created by Pulse, as described in section 6.5.1.

Mongo DB

MongoDB is a document-based database solution which was developed with the main goal of providing ease of application development and scaling [31]. It provides a lot of flexibility when creating tables, as the schema does not have to be defined explicitly.

The backend stores all its data in the MongoDB in the structure as described in section 6.4.

OpenAPI Generator

OpenAPI Generator is a tool that can generate clients for different programming languages as well as specific frameworks.

The generator is open source and its generated code is not under any license restrictions [32].

By periodically running the OpenAPI generator it can be assured that the frontend always uses the latest version of the API as changes result in a updated client for the frontend.

Referencing the generated models in the frontend ensures that changes to the model (such as deleting a property) have to be reflected on the frontend as well. Should the property be referenced anywhere in the frontend, its build would fail and show an appropriate error message.

The generator is run by *docker-compose* when the local development environment is started. Then it can be run on-demand during development if changes were made to the API controllers in the backend.

6.3 REST API

The API exposed by the backend follows a RESTful approach. Endpoints are grouped by their respective resources, and are nested if they belong to a parent resource. For example a sequence belonging to a project is accessed as follows: */project/:projectId/sequence/:sequenceId*

Individual resources can then be modified using the default HTTP verbs. These principals are applied across the whole API to ensure that the API is simple to understand.

To create new resources, the *POST* verb is used. For retrieving data without modification the *GET* verb is used. Deleting resources is possible using the *DELETE* verb. To modify existing resources the *PATCH* verb is used for updates to sequence and sequence endpoint, where only certain values are updated (e.g. the position of an endpoint in the sequence). In the case of test data, the *PUT* verb is used because when a test data set is updated, all values are sent and saved. This is adhering to the method definitions documented in HTTP RFC 2616 [33, Chapter 9] for *POST*, *GET*, *DELETE* and *PUT*, as well as RFC 5789 'PATCH Method for HTTP' [34] for *PATCH*.

Depending on the resource not all Create Read Update Delete (CRUD) operations are supported. Table 6.1 shows which resources support which CRUD operations, as indicated by an 'x'.

	Create	Read	Update	Delete
Project		x		
Endpoint		x		
LinkSuggestion		x		
Sequence	x	x	x	x
SequenceEndpoint	x	x	x	x
Link	x	x		x
TestData	x	x	x	x

Table 6.1: Resources and their supported CRUD operations

The creation of projects, endpoints and link-suggestions is not directly possible. All three of these resources are created during the import process and should not change as they might be used in different sequences in the same project. While link-suggestions can also be added later, when users add new links, the creation in this case is always tied to the creation of a new link. As projects and endpoints are never changed after the import process, caching can be enabled

for the *GET* request concerning them to improve loading times of the screens. This is especially beneficial for large OAS definitions, where the loading of endpoints can tak considerable time.

Sequences can be deleted and updated freely by the user through the overview- or editor screen. The same is true for the sequence's elements: SequenceEndpoints, Links and TestData. While a link can be "updated" from a user experience standpoint, technically speaking it's always deleted and a new link is created, which is why the link controller does not offer an update capability.

Richardson Maturity Model

The best known metric to classify a web API is the Richardson maturity model. It was introduced by Leonard Richardson in 2008 [35].

The model categorizes REST APIs in four levels [36, pp. 18–20]:

- *Level 0:* A single URI and only one HTTP verb is used (usually *POST*).
- *Level 1:* Different resources have different URIs, but there is still only one HTTP verb used.
- *Level 2:* Additionally to level 1, different HTTP verbs are used for different actions.
- *Level 3:* In addition to level 2, representations contain URI links to other resources.

The API of this project aspires to reach level 2 of Richardsons' classification. As the communication between frontend and backend is mostly CRUD actions, the default way to implement it is to use a separate URI for each resource and offer the HTTP verbs *POST*, *GET*, *DELETE* and *PUT* [36, pp. 55–58].

Some slight deviations from this approach are necessary however, as not all resources are implementing *CREATE*, *UPDATE* and *DELETE* and some resources use *PATCH* for updates instead of *PUT*. The maturity level 2 can be considered reached anyway however, as the API uses different URIs for each resource, the correct HTTP verbs and provides correct response codes.

6.4 Data Design

The application uses a format that is different from the format specified in the OAS to store information about a specific API definition.

A different format was chosen, to better suit the needs and requirements of the application. The OAS offers a lot of flexibility for describing APIs as discovered in section 2.2. Endpoints are grouped by their Paths but apart from belonging to the same logical feature differ vastly in their input and output.

For the purposes of our application a view from someone consuming the API is more appropriate. When consuming an API the groups of paths should be viewed as a whole but more importantly the single method that can be used on one path, as a request to the API also has to follow this.

Therefore the chosen data design focuses on the individual operations (methods) on an endpoint as a single entity (Which is what Swagger UI does as well).

The application should not have to deal with resolving each endpoint based on the schema each time. Also when placing these endpoints in a sequence it always references a single operation.

During the import phase data is mapped onto the data structures as described in section 6.5.1.

Database Schema

Based on the requirements the database schema in fig. 6.2. Relationships between the Collections are not strictly enforced by the database itself but on an application level, as there is no concept of foreign keys in MongoDB. When using the editor view to edit a sequence, SequenceEndpoint entities are created for each endpoint that is added to the sequence. TestData entities can be created and belong to a specific Sequence entity. Then Link entities are created that link TestData or SequenceEndpoint entities to SequenceEndpoint entities. This structure separates the data that is consistent across a project and the dynamic data creating in the sequence creation process.

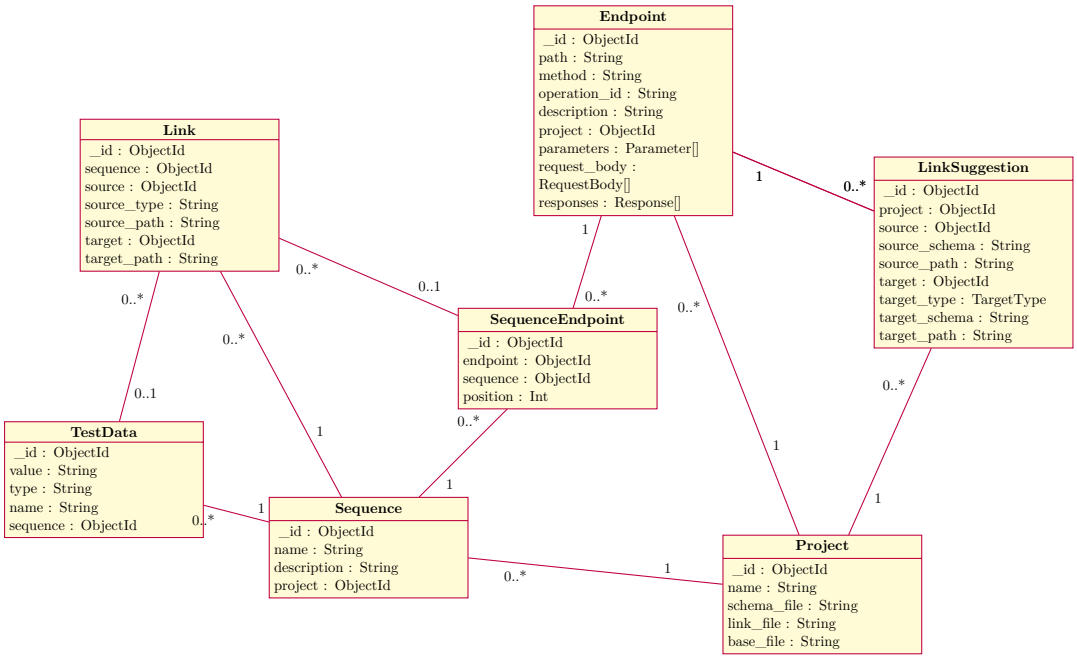


Figure 6.2: Database schema

6.5 Data flow

This chapter describes the data flow between the application and external data sources.

6.5.1 Import

Schemas and links created by Pulse can be imported into the application. An endpoint is exposed that provides the possibility for an external application to import specific files. In listing 2 an example on how a new project can be imported is provided.

```
GET http://ai-docker.testifi.io:8000/api/v1/projects/import?
schema_file=0a06cc81-e9ee-426a-931d-09dc41a6352d.json&
link_file=https:__api-petstore.testifi.io_api_v3.json
```

Listing 2: HTTP request to import a project

The sequence diagram in fig. 6.3 describes the import process across all involved systems. The integration of pulse into the import process is not in the scope of this project, but demonstrates how the application can be integrated into the existing systems of the customer.

1. The user imports a OAS definition into Pulse.
2. Pulse analyses the OAS definition and creates links.
3. Pulse uploads the generated links and the definition to a S3 bucket.
4. Pulse calls the backend to start the project import.
5. The backend reads the link and OAS definition from the S3 bucket.
6. The backend processes the OAS definition an link file and parses them into entities that can be used by the application. Figure 6.4 shows how the file contents are mapped to entities.
7. The backend stores the created entites as documents in the MongoDB.

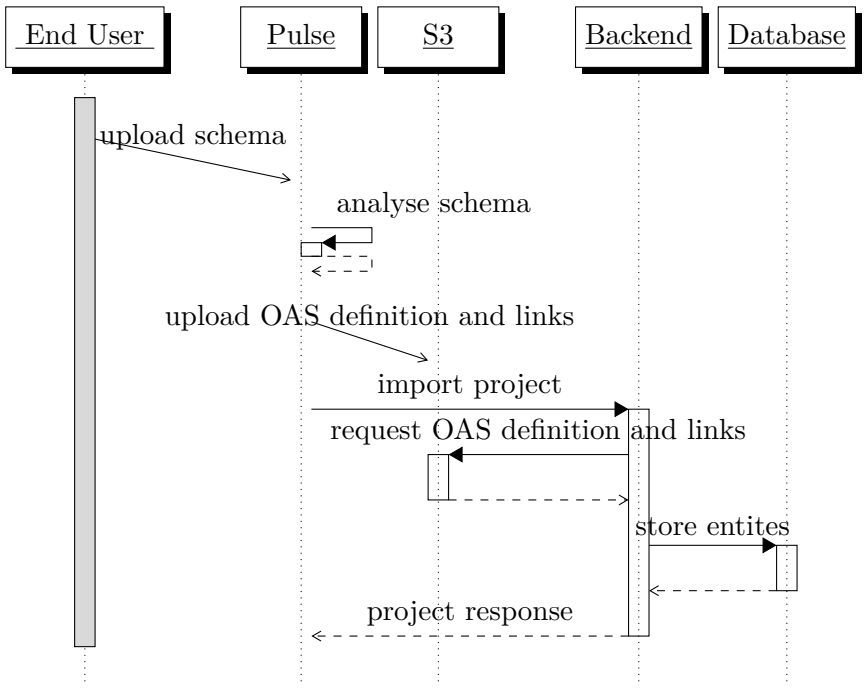


Figure 6.3: Import data sequence diagram

In fig. 6.4 the flow of the data during the import process is visualized. Data flows from the S3 bucket into the backend where the files are parsed and processed into entities before finally being persisted into the database.

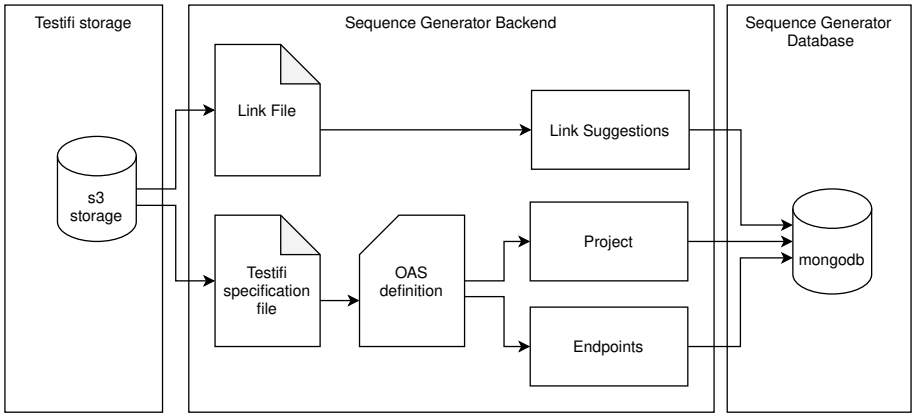


Figure 6.4: Import Data flow

6.5.2 View Project

The sequence diagram in fig. 6.5 describes the steps necessary for a user of Pulse to view a project in the sequence editor. This flow shows how the application can be incorporated into the existing Pulse application, it is however not the goal to integrate it during the project. As a precondition for this flow, it is necessary that the project is already imported into the application.

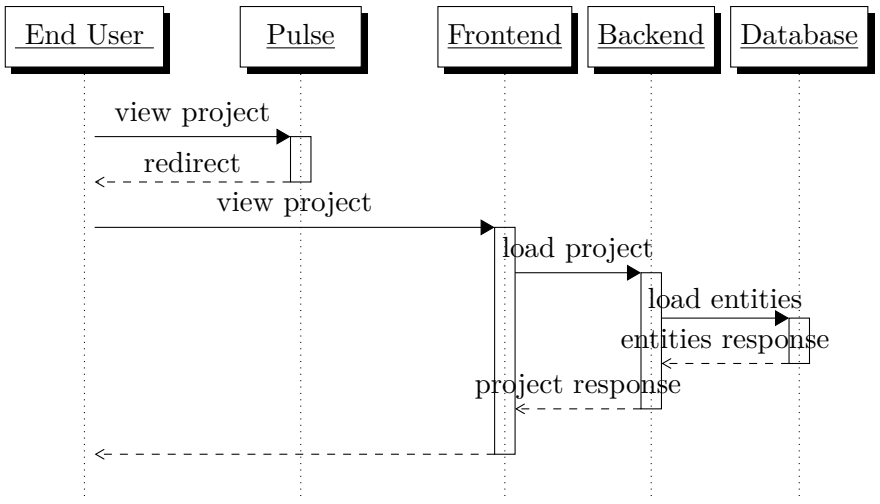


Figure 6.5: View project sequence diagram

1. The user opens Pulse and selects a button or link to open the sequence editor application
2. pulse redirects the user to the frontend including a project identifier
3. the frontend requests the project data from the backend
4. the backend requests the project data from the database
5. the requested data is returned from the database to the backend and then to the frontend
6. the project is displayed to the user

6.5.3 Create Link

If a new link is created the client sends a request to the backend. First, the link is inserted into the links document in the MongoDB as well as the link suggestion document. This way, the user will immediately have the new suggestion available, should they reuse the same endpoint in this or any other sequence of the project.

The link is then also saved to the link-file on S3 to make sure that future projects can profit by having the suggestion in their link-file from the start.

The following steps are performed when a user creates a new link in the frontend. The sequence is also visualized in fig. 6.6.

1. Check if created link is between endpoints.
2. Resolve defined path to a schema.
3. Check if the link suggestion already exists in the database.
4. Write link suggestion to database.
5. Load link file from S3.
6. Insert link in testifi format into the file.
7. Overwrite the link file.

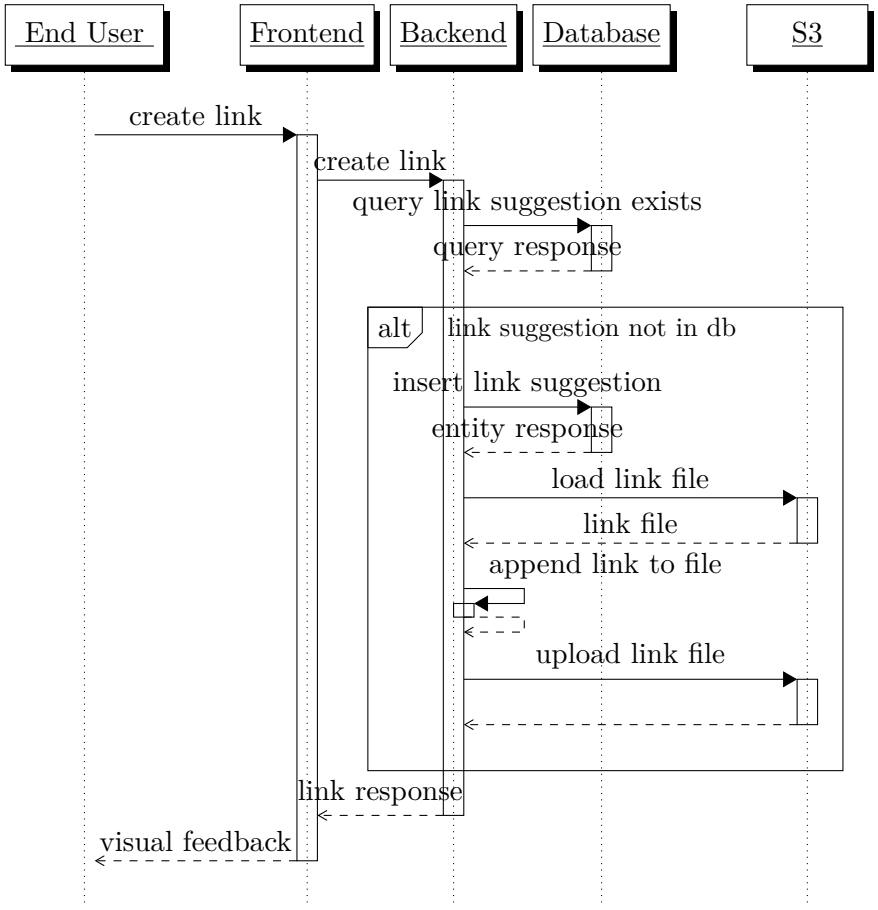


Figure 6.6: Create link sequence diagram

7 Implementation

This chapter details the implementation phase of the project. It provides insights in the development setup as well as the frontend and backend applications and their connecting API.

7.1 Deployment

To run the application Docker Compose is used. To keep the setup as simple as possible the development and the production deployments are kept as similar as possible.

7.1.1 Development Setup

Figure 7.1 shows a schematic overview of the development setup. It shows how a developer interacts with the application during the development of new features.

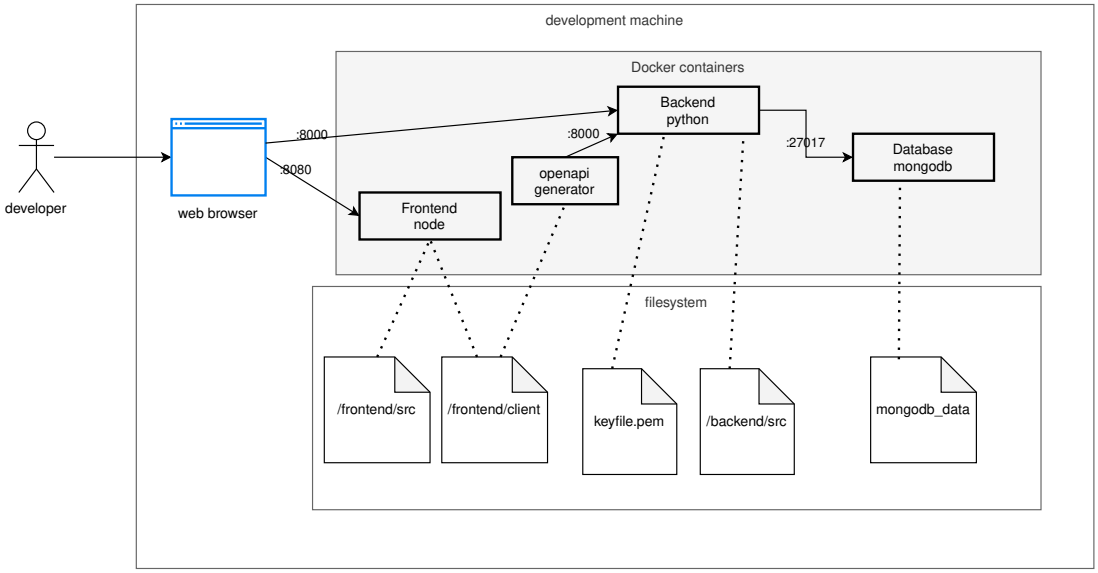


Figure 7.1: Overview of the local application setup

Apart from Docker a developer does not have to install any special runtimes or tools like node or python to run the application.

When launching the application with Docker Compose all necessary components are started to fully run the application. However, a simple change needs to be done to start the application successfully: the *BASE_PATH* value has to be manually set to *http://localhost:8080* in the file `frontend‣src‣app‣app.module.ts`

The frontend container is a Node.js runtime that installs all dependencies as defined in the `package.json` file and runs the application using the angular-cli. To achieve this the `frontend‣src` directory is mounted into the container. The `frontend‣client` directory which contains the code generated by the OpenAPI Generator is mounted into the container as well.

For the backend a conainer running python is used that installs all dependencies from the `requirements.txt` file. Similarly to the frontend the `backend‣src` directory is mounted into the container.

By directly mounting the host filesystem into the containers the code can be edited in an integrated development environment (IDE) and changes can be observed without the need to restart any container.

Only if changes to either dependencies of the backend or frontend are made, a rebuild of the respective docker container is necessary, as the installation of dependencies is done when building the dockerimages from the Dockerfiles located in both the backend and frontend.

To keep the development and production setup as close as possible a multi-stage Docker image is used in the frontend[37]. The Multi-Stage image provide a base that is shared across the environments.

A container running the OpenAPI Generator is started as well. The container uses the endpoint that exposes the OAS document to create the API client directly in `frontend‣client`.

7.1.2 Production Setup

In contrast to the development setup the application code is not mounted into the container but copied into the docker image during the build phase for the frontend and backend, as shown in fig. 7.2.

The deployment was done by the customer directly by cloning the git-repository on a server and starting the application with Docker Compose.

During the docker build process the frontend application is bundled into assets, that can be served by any static web server by using the *ng build* command from the Angular cli. Caddy is used to serve the static assets and as a reverse proxy. It forwards requests starting with */api/* to the backend. For all other requests the static assets of the frontend will be served.

The rest of the production setup is identical to the development setup described in section 7.1.1.

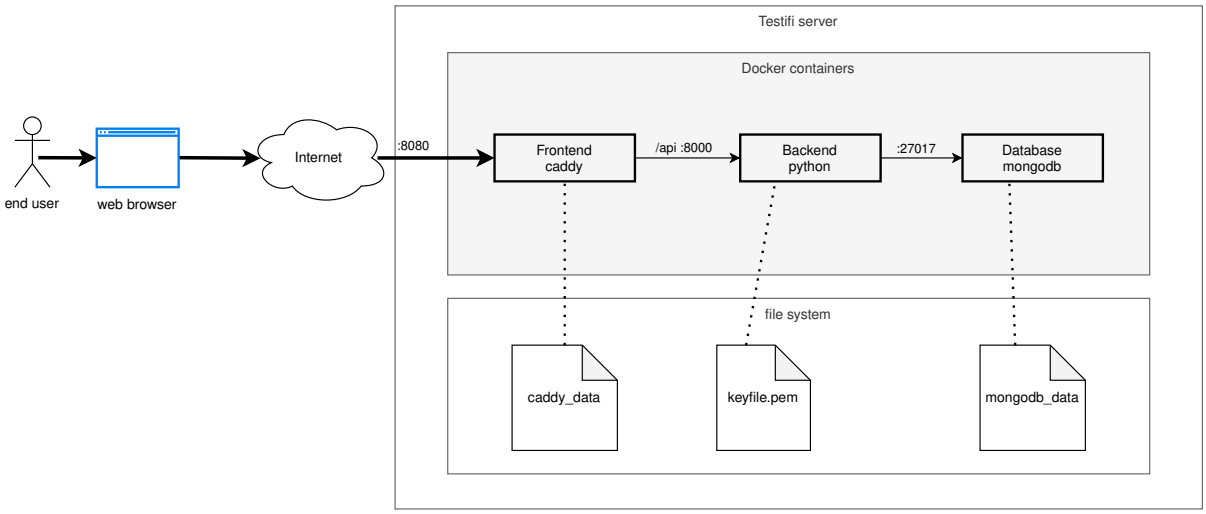


Figure 7.2: Overview of the deployed application setup

7.2 Frontend

7.2.1 Folders

The folder structure shown in fig. 7.3 follows the Angular style guide introduced in section 6.1.2. Some files were omitted for brevity. The root folder holds the config files generated by Angular (such as angular.json, package.json) as well as additional config created manually (such as the Dockerfile and Caddyfile).

While usually all code goes into the `src` folder, in this project another folder called `client` exists, which contains a automatically generated API client that provides functions to call all

endpoints exposed by the backend. This is in accordance with the proposed structure in the Angular style guide.

The `src` folder contains `app`, `assets`, and `environments` folders according to the Angular default structure.

The `src‣app` directory contains all components (split by feature into `project`, `sequence-editor`, `sequence-overview` and `shared`) as well as the `store`s, models and services (in `shared`).

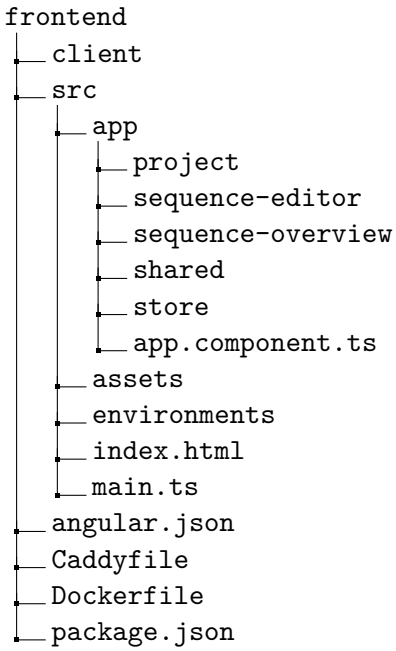


Figure 7.3: top-level folder structure of the frontend

7.2.2 Pages

Angular does not have a concept of pages itself, as everything displayed is regarded as a component. But as angular supports the concept of routing these routes define what components are loaded at each instance. Because the routing in angular is hierarchical it is most logical to also represent it as shown in fig. 7.4. Fragments of the path denoted with a colon(:) represent path variables. The asterisks(*) matches any character. The left hand side describes the path or URL entered in the browser address bar. On the right the corresponding Loaded component is listed.

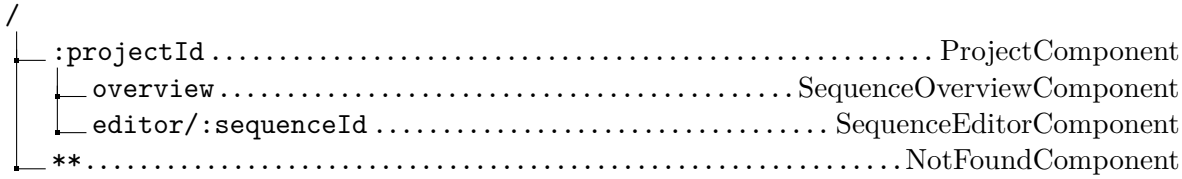


Figure 7.4: routes defined in the frontend

7.2.3 Components

As any standard Angular app, the main component in this projects' frontend application is the *app* component, which implements the router. In this case, the main routing path ':projectId' implements another wrapping component *project* which then leads to either *sequence-overview* or *sequence-editor*.

Project

The *project* component is a wrapper that is used to hydrate the store with data that is used both in the *sequence-overview* as well as the *sequence-editor* component. It loads the requested project and the endpoints and link suggestions associated with it.

Overview

Figure 7.5 shows the Hypertext Markup Language (HTML) hierarchy of the Angular components on the overview page. As the overview page's main responsibility is to display and filter the existing sequences in the project, there are two sub-components: The *sequence-overview-filter* component provides the possibility to search for sequences that match certain criteria while the *sequence-overview-content* component displays the respective results.

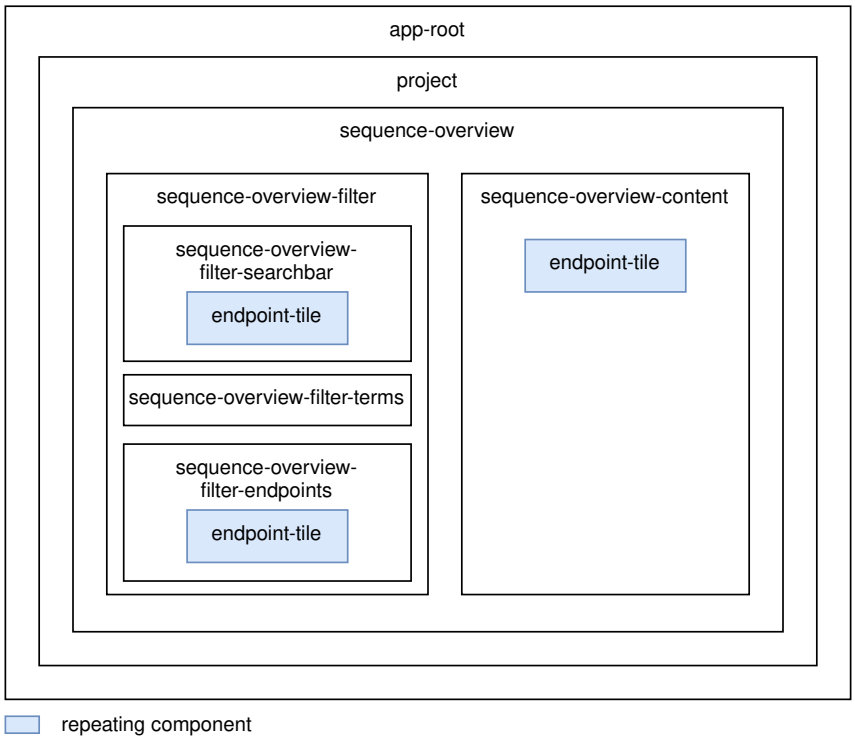


Figure 7.5: Hierarchy of angular components on the overview page

Overview Filter

The *sequence-overview-filter* component consists of a search-bar, to search for endpoints or enter text terms to the filter. The added endpoints are displayed in the *sequence-overview-filter-endpoints* component, which makes use of the shared *endpoint-tile* component, while the added text terms are displayed in the *sequence-overview-filter-terms* component.

Overview Content

The *sequence-overview-content* component is responsible for displaying the list of sequences provided by the store. For each sequence it displays the name, description, endpoints that the sequence contains as well as providing actions for the user.

Editor

Figure 7.6 shows the HTML hierarchy of the Angular components on the editor screen. On the editor screen, the user manages a sequence by adding test data, endpoints and links.

The *sequence-editor-test-data* component is always shown to the user. In the linking mode (described in section 7.2.5) of the application the test data entries can be selected as a link source.

The *sequence-editor-flow* displays all Endpoints of the selected sequence. When a new endpoint should be added the *endpoint-selector* component is displayed as well as the *sequence-editor-endpoints* component on the right side. For each link that exists starting or ending from the selected endpoint an *app-sequence-editor-flow-link-svg* component is displayed that is responsible for drawing a svg line between the components.

If a new sequence is created or an existing is updated, the editor opens a pop-up to edit the name and description of the sequence.

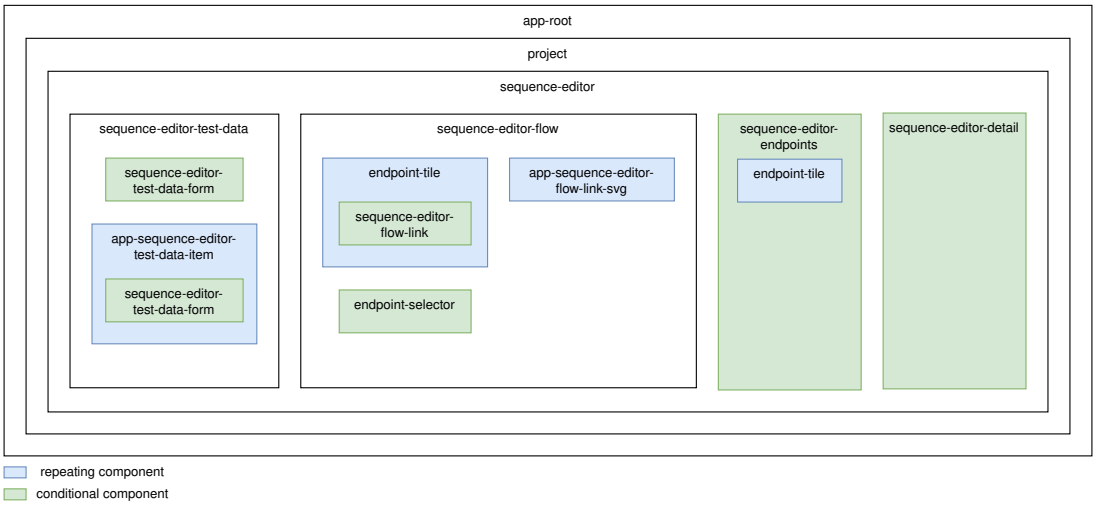


Figure 7.6: Hierarchy of angular components on the editor page

Editor Test Data

The *sequence-editor-test-data* component gives the user the possibility to add and edit test-data values, which can be used as input data in endpoints that are added to the sequence by linking them. The component offers simple CRUD operations when link selection is disabled and offers its values as possible sources for links if the selection is enabled.

Editor Flow

The central *sequence-editor-flow* component offers an overview over the existing endpoints added to the sequence with options to manipulate them by adding a new endpoint, remove one or change the order. During link selection, it offers the linkable response values for each endpoint preceding the link target.

Editor Detail

If an endpoint is selected, the editor *sequence-editor-detail* component displays the endpoints description, parameters and request and response body. It offers the user the possiblity to start adding a link by selecting the link target (a parameter or request body value).

The description of the endpoint can potentially contain markdown syntax, as this is a supported feature of the OAS [10, Chapter 4.5]. To support this the *marked* library is used to parse the mardown into HTML that the browser can display. As the description might contain potentially

unsafe HTML, such as *script* tags, but also other elements such as tables that should be displayed properly and not be escaped by Angular.

To achieve this behaviour of safely rendering potentially unsafe HTML the *innerHTML* property is used. This contents will then still be sanitized by the Angular *DomSanitizer* before being passed to the browser to interpret [38].

Editor Endpoints

Figure 7.6 also shows the HTML hierarchy if the user chooses to add another endpoint to a sequence.

The *endpoint-selector* component is displayed when the user clicks on the '+' icon anywhere in the sequence. It provides HTTP method badges and a search field to let the user filter the available endpoints in the project. When the component is shown, the *sequence-editor-detail* component is replaced by the *sequence-editor-endpoints* component that shows the available *endpoint-tile*s filtered according to the criteria set in the selector. This state of the frontend application is also visible in fig. 5.4.

Editor Schema View

The editors *schema-view* component serves the purpose of recursively displaying the schema of a request or response body while implementing . It is used in both the *sequence-editor-detail* as well as *endpoint-tile* component. Figure 5.4 shows both uses of the component in the scenario of a link selection.

Uniquely, the component uses itself as a child component for each child property if the model it is displaying is either an object or an array. Listing 3 shows an excerpt of the HTML template that covers the object case and uses itself as a child component. This lets it display JSON objects without any depth restrictions. Possible infinite loops in recursive schemas are handled by the parser as described in section 7.3.3.

```
<span *ngIf="model.type === 'object'">
   ...
     <div class="object-item" [class]="{hidden: !isExpanded}"
       *ngFor="let p of model.properties; let last = last; ">
               <schema-view
                 [name]="p.name"
                 [path]="path + '.' + p.name"
                 [model]="p.model"
                 ...
               ><span *ngIf="!last">,</span>
       </schema-view>
         <span *ngIf="filterSuggestions(model, p)"
           class="ai-suggestion material-symbols-outlined">auto_awesome</span>
     </div>
   ...
 </span>
```

Listing 3: Excerpt of SchemaViewComponent (▸ frontend ▸ src ▸ app ▸ sequence-editor ▸ schema-view ▸ schema-view.component.html)

Shared Components

There are some shared components that are used in both the overview and editor screen.

- The *not-found* component is used whenever a link that the router cannot resolve is requested and shows the user a 404 page.
- The *endpoint-tile* displays information about an endpoint and is used in multiple locations.
- The *http-method-badge* displays a HTTP verb with it's associated color for recognizably. It is used in the endpoint tile and other places across the application.
- The *confirm-dialog* offers a pop-up to ask the user to confirm their action. It is used if the user wants to delete or move an endpoint that contains links or deletes a sequence on the overview.

7.2.4 Services

To support certain functionality that is not bound to specific components, the frontend application also uses three services.

Hotkey Service

As there are multiple keybindings to register in order to provide keyboard support for the application, the Hotkey Service provides a centralized place to register new event listeners for key presses.

In its *addShortcut* method, one or more keys (If multiple keys need to be pressed, like ⇧+↓ to move an endpoint) can be provided to get an observable.

This service is inspired by a sample implementation from Basal [39].

Keybinding Service

The Keybinding Service makes use of the Hotkey Service to register all used key press events and subscribes to the returned observables in order to dispatch the associated action once a keybinding fires an event.

It also supports multiple keybinding modes, in order to choose the correct action at the correct time (e.g. When a dialog is open, only the Esc key to close it and ↵ key to confirm it are allowed keys that trigger an action). The mode is stored in the editor state and updated by certain user actions. Those modes and associated key actions reflect the keyboard interaction concept described in section 5.3.

The following modes are available:

- `endpointSelection`: In this default mode, the user can change the selected endpoint by tabbing through them or using the arrow keys to move up and down. By using ⇧ + the up or down arrow the endpoint can also be moved and with the Del. key an endpoint can be deleted.
- `linkSelection`: Once a user selected a link target, only the ⇤⇥ key will allow the user to go through possible link sources in the endpoints, which can then be selected with Space or ↵.
- `dialog`: When a dialog is opened, the user can use the Esc key to cancel the dialog or the ↵ key to confirm.

Verification Service

When a sequence is created, the user usually needs instant feedback if the endpoints with the linked values actually work or not. The Verification Service provides a quick way to check this, by executing the sequence.

It gets all endpoints that make up the sequence and verifies them one by one by filling out the request parameters and body by getting the linked values, sending the request and saving the necessary values for future requests. That way, the user can see, if their sequence is successfully executed and if not, which endpoint request failed in the sequence.

If the user needs to check the response for a request, details about the payload of the request and its response, as well as possible errors, are available through the browsers developer console as shown in fig. 7.7.

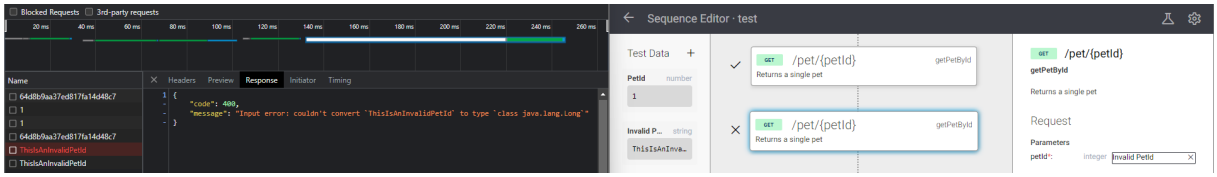


Figure 7.7: The application and the developer tools side-by-side

7.2.5 Store

The frontend application uses NgRx stores for global state management. NgRx store is a RxJS powered library inspired by Redux [17].

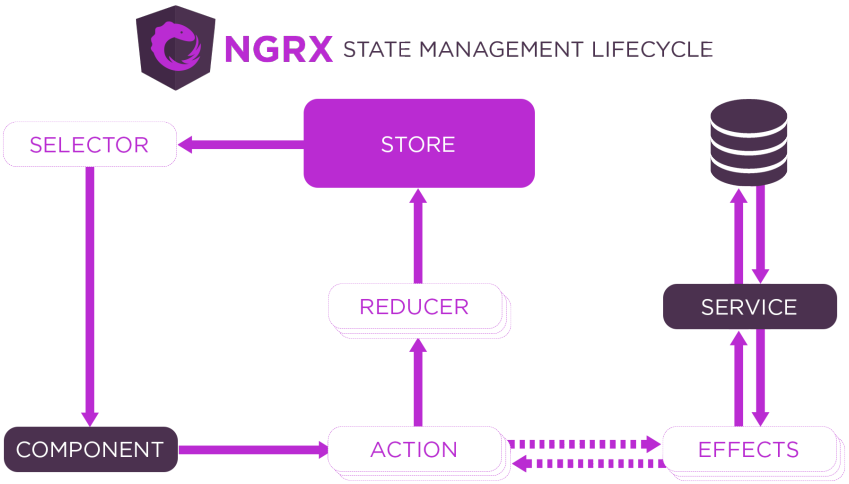


Figure 7.8: Lifecycle of application state in NgRx. Graphic from ngrx.io [40].

A store consists of an observable of state and an observer of actions. It has a set of actions that can be triggered from anywhere in the application. Actions are handled by reducers, which update the state and can trigger effects, which in turn can dispatch new actions. Through selectors, the state data can be accessed and subscribed to from anywhere in the application. A typical NgRx lifecycle is shown in fig. 7.8.

As a simple example of a store action implementation, the *Add Endpoint* action in the *endpoint-in-sequence* store can be examined. In the actions definition itself, the two actions *Add Endpoint* and *Endpoint added* and their respective properties are defined as shown in listing 4.

```
export const EndpointInSequenceActions = createActionGroup({
  source: "Endpoints in Sequence",
  events: {
    ...
    'Add Endpoint': props<{ projectId: string, sequenceId: string,
        endpoint: SequenceEndpointRequest }>(),
    'Endpoint Added': props<{ endpoint: IEndpointInSequence }>(),
    ...
  },
});
```

Listing 4: Excerpt of EndpointInSequenceActions (▸ frontend ▸ src ▸ app ▸ store ▸ endpoint-in-sequence.actions.ts)

In the reducer only the *Endpoint Added* action is triggering changes on the state. As shown in listing 5 it adds the new endpoint to the sequence and updates the position of all following endpoints, while also resetting the *verificationStatus* of all endpoints in the sequence.

```
export const endpointInSequenceReducer = createReducer(
  adapter.getInitialState({ selectedEndpointId: '' }),
  ...
  on(EndpointInSequenceActions.endpointAdded, (state, { endpoint }) =>
    adapter.addOne(
      endpoint,
      adapter.map(
        (entity) => ({
          ...entity,
          verificationStatus: undefined,
          position:
            entity.position >= endpoint.position
              ? entity.position + 1
              : entity.position,
        }),
        state
      )
    )
  ),
  ...
);
```

Listing 5: Excerpt of endpoint in sequence store (▸ frontend ▸ src ▸ app ▸ store ▸ endpoint-in-sequence.reducer.ts)

Add Endpoint is not triggering anything in the reducer as the state should only be updated if the action is successful. To achieve this, *Add Endpoint* passes the reducer and starts an effect.

As seen in listing 6, the effect selects the necessary information and sends a request to the API. Only upon a successful response, it dispatches the *Endpoint Added* action, which passes the reducer and updates the state. This ensures that only changes that were accepted by the backend are applied to the frontend state.

```
addEndpoint$ = createEffect(() => {
  return this.actions$.pipe(
    ofType(EndpointInSequenceActions.addEndpoint),
    concatMap(({ projectId, sequenceId, endpoint: endpointRequest }) =>
      this.service
        .createSequenceEndpoint({
          sequenceId: sequenceId,
          projectId: projectId,
          sequenceEndpointRequest: endpointRequest,
        })
        .pipe(
          map((endpoint) =>
            EndpointInSequenceActions.endpointAdded({ endpoint })
          )
        )
    )
  );
});
```

Listing 6: Excerpt of add endpoint effect (`frontend‣src‣app‣store‣endpoint-in-sequence‣endpoint-in-sequence.effects.ts`)

Editor Store

The editor store provides a state for the editor screen. It contains general data like the selected endpoint, the position where a new endpoint should be added, the endpoint filter for the *sequence-editor-endpoints* component, etc. as well as data about linking if the link selection is in progress. An excerpt of the editor state can be seen in listing 7. It is purely used for state management and does not contain any effects.

```
export interface EditorState {
  selectedEndpointId: string;
  showEndpointSelection: boolean;
  endPointFilter: { textFilter: string; methodFilter: IHttpMethod[] };
  addingPosition: number;
  ...
  showLinkingSelection: boolean;
  selectedLinkingTargetPath: string;
  endpointLinkFilter: string;
  ...
  hotkeyMode: HotkeyMode;
}
```

Listing 7: Excerpt of EditorState (`frontend‣src‣app‣store‣editor‣editor.reducer.ts`)

The editor state could have been split in different stores, to separate it by mode (e.g. an endpoint selector store and a link selector store). As the editor components would need to select data from both stores most of the time anyway, it was decided to keep all editor exclusive data in the central editor store.

Dialog Store

The dialog store provides actions to open and close a confirm dialog. Through this store, a dialog can easily be opened from anywhere in the application (e.g. from a component or a service). As the action to be executed after confirmation is already provided upon opening the dialog, it is not necessary to handle the outcome anywhere outside the store.

Router Store

The router store is a default store of NgRx. It provides query and route specific selectors that are needed if any url data is used in another store, but can also be used by components.

Entitiy Stores

To support simple CRUD operations for database entities, multiple entitiy-stores are used.

- Endpoint
- Endpoint in Sequence
- Link
- Link Suggestion
- Sequence
- Test-Data

While most of their used actions and selectors are standard, in some cases they have additional functionality.

For example if an endpoint in a sequence is added (as seen in listing 5) or deleted, the reducer also updates the positions of all endpoints after the changed one to avoid positioning conflicts. The behavior of the custom *move endpoint* action in the store is similar.

Another example for custom behavior is that the endpoint in sequence effect of deleting (As shown in listing 8) or moving an endpoint dispatches a delete for each link related to the affected endpoint. To realize this, the link store provides a custom selector that selects links by endpoint in sequence ID.

7.2.6 API Access

To communicate with the backend via its API, an Angular Service is used. The service abstracts away the logic required to create the HTTP-requests. Because this service is fully dependent on the API definition provided from the backend, the service is auto-generated using the OpenAPI generator as described in section 6.2.1.

To configure the base path used by the client, the *BASE_PATH* injection token can be provided.

Project Service

The generator creates a single angular-service that contains a function for each endpoint. The name of the function is derived from the *operationId*. Internally the service uses the *HttpClient* from Angular to handle the requests. A Observable will be returned from the service that completes as soon as a response from the server is received. The method signatures are fully typed to ensure that only valid requests are sent.

Models

The client generates models for the inputs and outputs of the service function. These models are used in the application. For easier handling the models are first aliased in our application. This removes the need to import directly from the generated client in all components that use a particular model. If there should ever be the need to override or manually change a generated model, this can be accomplished by changing the definition in the frontend.

```
deleteEndpoint$ = createEffect(() => {
  return this.actions$.pipe(
    ofType(EndpointInSequenceActions.deleteEndpoint),
    concatLatestFrom(() => this.store.select(selectRouteParam('projectId'))),
    concatLatestFrom(() => this.store.select(selectRouteParam('sequenceId'))),
    concatMap(([[{ endpoint }, projectId], sequenceId]) => {
      this.store.dispatch(
        LinkActions.deleteLinksByEndpointInSequenceId({
          endpointInSequenceId: endpoint.id,
        })
      );
      return this.service
        .deleteSequenceEndpoint({
          projectId: projectId as string,
          sequenceId: sequenceId as string,
          sequenceEndpointId: endpoint.id,
        })
        .pipe(
          map(() => EndpointInSequenceActions.endpointDeleted({ endpoint }))
        );
    })
  );
});
```

Listing 8: Excerpt of Delete Endpoint effect (`frontend‣src‣app‣store‣endpoint-in-sequence‣endpoint-in-sequence.effects.ts`)

Listing 9 contains the SequenceResponse class that the backend uses to construct API responses. Based on this model the OpenAPI Generator then generates the typescript interface as seen in listing 10.

```
class SequenceResponse(BaseModel):
    id: str
    name: str
    description: str
    endpoints: List[str]
```

Listing 9: Excerpt of sequence model (`backend‣src‣Model‣Sequence.py`)

```
export interface SequenceResponse {
    description: string;
    endpoints: Array<string>;
    id: string;
    name: string;
}
```

Listing 10: Excerpt of generated sequence response (`frontend ▸ client ▸ model ▸ sequenceResponse.ts`)

7.3 Backend

7.3.1 Folder structure

The backend uses a simple folder structure as shown in fig. 7.9. The entrypoint of the backend is the `app.py` file. This file bootstraps the whole application and defines all REST endpoints, it can be considered the controller of the application. The `Models` directory contains all data structures. Such as the entites as well as Data Transfer Objects (DTOs). The `Service` directory holds the services responsible for the entity manipulations.

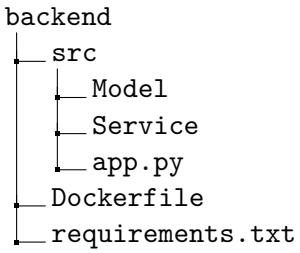


Figure 7.9: Top-level folder structure of the backend

7.3.2 REST API

Flask is a simple framework to provide HTTP endpoints in a python application. To automatically check if the requests and responses of the API are valid, the spectree library is used. All requests are validated by spectree to ensure that only valid inputs can be sent to the API.

Spectree also automatically provides a OAS endpoint containing the current API definition and a Swagger UI endpoint for conveniently viewing all provided endpoints in a web browser.

All endpoints are currently exposed under the base-path */api/v1/*. By using a version in the path name, the API can be changed for future improvements, without breaking backwards-compatibility. Changed endpoints can be exposed under *api/v2/* and then the old ones can be marked as deprecated but still kept intact for a period of time before removing them.

By providing endpoints for all actions in the frontend all changes are always persisted and there is no danger of loosing changes made in the frontend only.

The complete documentation of all available endpoints is provided in the appendix C.

7.3.3 Parsing

The document is parsed in a two step process, as the data model of the application requires all endpoint data to be fully resolved without any references on schemas.

First the prance library is used to verify that the provided document is valid. On the validated document the schemas are not yet resolved. For each schema a custom annotation called *x-testify-schema*, in compliance with the OAS [10, Chapter 4.9], is added that indicates the name of the schema. Then prance is used again with the *ResolvingParser* to resolve all schemas in all endpoints. The parser is configured to only resolve the schemas to a certain depth of recursive schemas to prevent infinite loops during execution.

Finally the resulting data is mapped onto the the schemas defined in section 6.4.

Then the link file is parsed as well and the values are mapped to match the schema of the application. For each link the source and target endpoint are resolved using the already created entities. Then the links are mapped to a link-suggestion entity.

7.3.4 Data Access

The Application data is stored in a MongoDB. To access the database the library odmantic, which acts as a ORM layer, is used. Odmantic is build on-top of pydantic, which is a Python library to validate and define objects.

The database schema can be descirbed using ordinary python classes. The listing 11 shows how the sequence entity is defined.

By extending the *Model* class from Odmanic the class is registered to be used for database access. In the sub-class *Config* further customization is possible. In this case the name of the collection is defined as "sequences".

```python
class SequenceEntity(Model):
    name: str
    description: str
    project: ProjectEntity = Reference()

    class Config:
        collection = "sequences"
```

Listing 11: Excerpt of SequenceEntity (`backend‣src‣Model‣Sequence.py`)

As described in section 6.5, data is read and written from a S3 bucket. Because of technical limitations on the side of the customer the storage bucket is not accessed via the S3 API, but instead a SFTP connection is used.

The SFTP connection is established inside the application. To ensure a stable connection it is checked when file access is required and re-established if necessary. For importing a project the connection is always required as the schemas are fetched from the S3 bucket, therefore the request will fail if the connection cannont be established or if a file is not present on the bucket. When writing links back to the bucket and the connection cannot be established the request will still succeed without writing the file, as it improves the user experience by showing the user that the link was still created successfully inside the application.

7.3.5 Linking Endpoints

When users create new links these are written back to the links file that was provided on the import of the project, and added to the database as new link-suggestions. This process is described in section 6.5.3 and visualized in fig. 6.6.

As Pulse doesn't use the concept of test data like this application, only links between endpoints are relevant as suggestions and written to the file.

8 User Tests

The user tests chapter focuses on the testing phase, including initial and final user tests. Methodologies, results, and the impact of user feedback on the project are discussed.

8.1 Initial User Tests

Initial user tests were conducted with Gabor Bakos, who at the time worked in quality assurance for Testifi GmbH. Njedet Dogru and Fabian Affolter were also present to support during the tests and contribute with their own insights.

The goal of these user tests was to evaluate the usability and functionality of the sequence overview and editor views and gather user feedback and insights to identify potential issues and areas for improvement in the design and user experience concept of the application.

8.1.1 Methodology

The tests were conducted on the low-fidelity design in Figma shown in fig. 8.1 and fig. 8.2. The participant was encouraged to share his thoughts, ask questions and express any uncertainties he had while using the application.

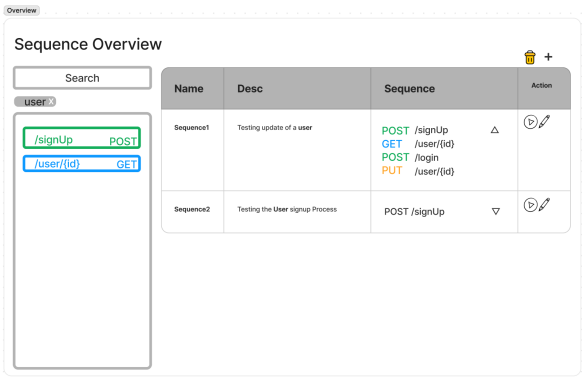


Figure 8.1: Low-Fidelity Design: Overview Screen used for user tests

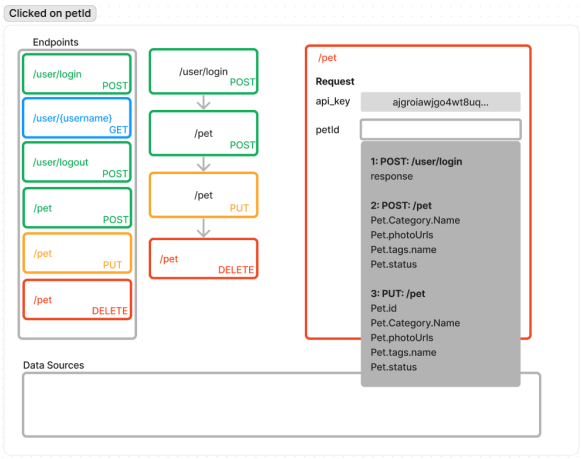


Figure 8.2: Low-Fidelity Design: Editor Screen used for user tests

The tests followed a structured approach:

1. Participant is introduced to the application's sequence overview screen, where he is asked to provide an initial impression of the design and it's components.
2. Participant is asked about his idea of interaction with various elements, such as play, edit, delete and add buttons, or the collapse arrow for sequence endpoints.
3. Participant is asked to explore the sequence filtering feature and explain what results he would expect based on inputs.
4. Participant is guided to the editor view and asked about the components and their suspected purpose.
5. Participant is asked to explore features (endpoint flow, data sources, link visualization).

8.1.2 Results

The user tests provided valuable insights into the strengths and areas for improvement in the application's low fidelity design and functionality.

Overview Screen

- First impressions positive, particularly on the "Action" column of the sequence table.
- The functionality of the play button was unclear.
- It was suggested to improve spacing between icon buttons to prevent accidental clicks was suggested.
- The search/filtering feature for sequences was not clear.
- The participant was surprised that not all endpoints used in the sequence were listed in the filter on the left.
- Actions associated with plus, edit and delete buttons were unclear.
- It was suggested that execution results of the last run should be displayed.
- It was discussed on which screen the popup for the creation of a new sequence should open and which fields should be mandatory.

Editor Screen

- Confusion arose on why endpoints that were not seen on the overview screen were available on the editor.
- It was suggested that in addition to the arrow-buttons, endpoint should be movable via drag-and-drop.
- The participant was uncertain about editing and deleting endpoints, especially if these actions affected the endpoints themselves or only copies in the sequence.
- The concept of reusing response values for parameters of subsequent requests was noted positively.
- The participant suggested to add save and cancel buttons to the editor.
- Clarification was required on the purpose of "Data Source" and the significance of the endpoint colors.
- Questions were raised about the drop down to select linking sources and it's feasibility in large sequences.
- Editing the sequence name directly in the header bar was suggested

How these results influenced the final design is documented in section 4.3.1.

8.2 Final User Tests

With the finished application, another set of tests was conducted to evaluate the speed, usability and effectiveness of the pulse sequence creator tool in comparison to other API testing methods, namely manual testing using Postman and Swagger UI. These tests were run with Testifi's Orhan Nurkan, who is very skilled in working with APIs and therefore was the perfect test user to confirm how well the user interface design works for advanced users.

The primary objective of this test was to asses the tool's performance, identify bugs and UI issues as well as finding potential user experience improvements and future features.

8.2.1 Methodology

Used Tools

Three different tools were used to complete the same test sequence and compare the results:

- Pulse Sequence Creator: The primary tool under evaluation, used to create sequences in the provided project. Available in a test deployment at `http://ai-docker.testifi.io:8000/`
- Postman: Used for manual creation of API requests to the `https://api-petstore.testifi.io/api/v3` endpoints.
- Swagger UI: Utilized the "Try it out" feature to manually send requests to the `https://api-petstore.testifi.io/` endpoints.

Test Procedure

Participants were instructed to complete the following steps in each tool while noting the time taken, obstacles, annoyances, and problems encountered:

- Create requests for the sequence.
- In Pulse Sequence Creator, create test data.
- In Pulse Sequence Creator, create links between requests.
- Run the requests, verifying in Pulse Sequence Creator.
- In Postman and Swagger UI, copy result values for new requests with links.
- Verify that every request returns a 20X status and check verification status in Pulse Sequence Creator.

1. Get the pet with the ID 1
2. Create a new pet with
   - Random ID
   - Name “Dog”
   - Category: ID: 111, Name: “Puppy”
   - Status: “available”
3. Get the created pet by ID
4. Update the pet name
   - ID of the created pet
   - New Name
   - Category ID and Name of the creation request
   - Status of the creation request
5. Create an order
   - Random ID
   - Pet ID of the created pet
   - Quantity 1
   - ShipDate “2023-08-31T07:14:54.756Z”
   - Status “approved”
   - Complete “false”
6. Get the created order
7. Delete the created order

Additionally, they were encouraged to autonomously work with other cases in the Sequence Generator Tool to get a better understanding of the application and cover a broader set of use cases.

8.2.2 Results

The results of the tests performed by Orhan Nurkan were predominantly positive. While some very good feature ideas were suggested for further improvement, the software already outperforms the alternatives it was tested against.

Speed Comparison

The participant noted that the Pulse Sequence Creator was faster in setting up the sequence as compared to manual methods like Postman and Swagger UI.

In the tested sequence, creation in Swagger UI took 14% more time for the test user, while Postman took 5% longer than with the sequence generator tool. It was noted however, that the additional effort to use one of the other tools increases exponentially with longer and more complex sequences, as the creation of JSON objects with the correct data was the most time consuming task in Swagger UI and Postman. The test user, who has extensive experienece with both, Swagger and Postman, used the request bodies provided by Swagger UI while creating the sequence in Postman, as creating them from scratch would have taken even more time. But even with this method, the process was significantly slower than using the sequence generator.

Also, while the Sequence Generator Tool is designed to let the user update the sequence and re-verify, in manual tests the whole sequence would have to be retested step-by-step. So in scenarios where sequences change a lot, or where APIs are retested continuously, the applications is immensely valuable to save time.

Usability and Efficiency

The sequence generator tool's interface allows quicker sequence creation due to allowing users to easily link response values of preceding requests to parameters and request body values. Its UI supports the user in keeping focus on the task at hand and allows to create long sequences, without losing track of what was already done and what needs to be completed.

Edge Case and Complex Scenario Testing

As mentioned above when running the tests manually, changes in test data require the whole sequence to be checked again. The participant expressed that with the linking feature of the sequence generator tool, it is very easy to change test data and verify the sequence again, so more test cases with different sets of test data can be covered easily, which makes proper testing of an API 5-10 times faster with this tool. As a result, more edge cases and more complex scenarios can be covered by sequences created in the tool and can be easily rerun after API changes.

He also mentioned however, that with the possibility to export test data sets and import them in other sequences, this could be vastly improved as it could even be a possibility that the same sequence could be run with dozens of test data sets to test edge cases.

Known Links

The link suggestion feature allowed the user to be more efficient in creating correct connections between endpoints. He especially remarked that when working with multiple IDs in a sequence (as in the example sequence, where petId, categoryId and orderId fields wer present) it was very helpful to have an indication which ID was connected to which input.

As used links were also saved as suggestions, if the same endpoint is used again in the sequence, or a similar sequence is created, association of the fields is very easy because of the suggestions and sequence creation is sped up every time because more fields have suggested links.

User Experience

While in Swagger UI, all endpoints are available with description, parameters, request and

response bodies, etc., it offers a similar or greater level of detail for the endpoint information as the sequence generator tool. However, description and parameters of related endpoints always have to be searched for in the extensive endpoint list, while this application provides information about preceding endpoints easily accessible through the endpoint details.

Using Postman, information on the endpoints even needed to be sourced elsewhere and it didn't support the user in finding the necessary parameters, body structure or even endpoint URLs. The sequence generator tool outperforms both applications in user experience when it comes to the creation of request sequences.

Feedback and Recommendations

The participant found Pulse Sequence Creator efficient and user-friendly for creating and verifying sequences of API requests. The presentation of the endpoint details was described as extraordinary, the visuals were clear, understandable and orderly. The filtering in the endpoint selection was praised in particular. The final application was found very useful for API testing, even with some small improvements and possible future features being suggested by the the participant after the test:

- It would be useful to be able to edit the sequence name and description directly from the overview screen (additional edit button).
- Automatic generation of test data values (e.g. IDs) would improve user experience.
- For simple cases, it would sometimes be faster to just enter the request value directly instead of creating a test data entry and link it.
- The possibility to export and import test data would be useful if multiple sequences need similar data.
- An undelete option for deleted test data would be helpful.

8.2.3 Conclusion

The final user test demonstrated the effectiveness and efficiency of the application in testing API request sequences. If offered a smoother and faster experience compared to manual testing methods like Postman and Swagger UI. With the possibility to link response data of preceding requests to request parameters and the suggestion of possible links from Pulse, the creation of an interconnected sequence of requests was found efficient. The tool also showed promise in encouraging testing of edge cases and complex scenarios.

Overall, the test validated the value of the Pulse Sequence Creator Tool in improving API testing workflows and user experience and shows great potential for integration in Pulse.

Through the test, some minor UI updates were suggested and bigger possible new features were mentioned, which could increase the value of the application even further, especially when it comes to testing with different data sets and continuous automated testing.

9 Discussion

This chapter provides a summary and interpretation of the project results, outlines the encountered limitations and gives an outlook on what could be done additionally.

The goal of the project was to minimize the effort for testers to create and maintain testing scenarios by creating a sequence generator tool that enables the user to easily create and execute sequences of REST requests. The created software should provide an intuitive and easy to use user experience based on the current state of the art UI/UX concepts while taking Testifi's design guidelines into account. Additionally, as created links are saved as suggestions for future sequences, creation should become even more efficient over time while the collected data on links can also be used by Testifi to improve the Pulse AI.

9.1 Results summary

Literature Review

It transpired early in the literature review that it was necessary to pivot the focus from graph and workflow specific UI research to focusing more on catering to advanced users. Therefore most of the literature review conducted focused on gaining a better understanding of how to design an application tailored to advanced users, while basic design concepts were reiterated as well.

The literature review yielded great insights used in the creation of low-fidelity design prototypes and the subsequent implementation. Advanced users are encouraged to explore the application instead of reading documentation, can make use of multiple accelerators and find important information easily on all interfaces of the application while secondary information is a way designed to keep the user focused. Work for the user was reduced by identifying and eliminating excise tasks. The used concepts provided a solid base to build the design and application on.

Implementation

Initial user tests based on the low-fidelity design provided helpful insights of where the user interface design is less intuitive and easy to understand. Based on those findings, multiple areas of user interaction could be improved.

All features needed to perform the required task with the application are available in the software. The front end is implemented in Angular and the backend in Python, as required by the customer. Both layers are supported by state of the art libraries and follow an architecture that allows easy extension and adaption of the features. Some ideas for additional features that were introduced during the project, made it to the final version.

User Tests

The final user tests produced a good understanding of how well the application reached its goals, outlined in the project agreement. The results show that the application was usable as intended and provided a big improvement to the efficiency of creating sequences. The user interface was perceived as more suited to test APIs and specifically request sequences. The primary objective of these tests was to gauge the speed, usability, and effectiveness of the tool, as well as to identify potential areas for improvement and future improvement.

In terms of speed, the application excelled in setting up sequences compared to manual methods. While there are considerable time savings when comparing the application to Swagger UI and Postman respectively, it was recognized that the efficiency gains would escalate substantially for longer and more complicated sequences. The tool's ability to handle sequence updates and changes to test data also offered a remarkable advantage. Usability and efficiency were

highlighted by the tool's capacity to easily link response values to parameters and request body values, streamlining sequence creation.

The application showcased its ability to handle edge case and complex scenario testing by providing rapid verification of diverse test data, which enables thorough coverage of different scenarios. The test user found the link suggestion feature highly beneficial, particularly in managing multiple IDs in a sequence. The tool's superior user experience in comparison to Postman and Swagger UI was noted by the test user, as it provided detailed endpoint information while also offering easy creation of request sequences. The filtering of endpoints was praised in particular.

9.2 Interpretation

Literature and UI Design

Initial literature review in the area of user interface design for graph and workflow manipulation indicated that it might not fit the use case at hand, as most research in this direction was aimed a solutions manipulating complicated graphs with many edges or big workflows with parallel paths.

As the needs for this application did not warrant either, but instead only required a one directional step-by-step flow, graphs and workflow manipulation was not further considered as viable research targets and the focus was instead shifted to more basic UI concepts with an emphasis on advanced users.

This switch proved to be important for the further development of the UI/UX concept and designs, as the literature review provided central concepts that helped tailor a user interface that covers all required features while remaining intuitive.

Using proven design principles like Shneiderman's "visual information seeking mantra" as described in section 2.1 and combining them with elements aimed explicitly at advanced users like accelerators (section 2.1.5) could be confirmed as a good method to create a user interface.

Especially methods used to separate important information from details without losing necessary information, documented in section 2.1.5, were successfully applied, as user tests showed that the needed information was readily available even though the user interface was kept clean to minimize possible distractions. Secondary information did not take attention away from the task at hand, because the sequence flow is always the central feature of the editor screen.

Decisions like dropping the distinction between list- and row operations on the overview screen, as documented in section 4.3.1, show however, that adaption of known and proven concepts for the specific requirements can be successful as well, which shows the importance of user tests and feedback to determine where deviating from the default path can add value to the design.

Implementation

During development of the frontend, the Angular style guide introduced in section 6.1.2 was constantly kept in mind. It can be concluded that the final application reaches its goal of adhering to these guidelines. As seen in fig. 7.3, the folder structure respects the folder-by-feature approach with the split of *sequence-editor* and *sequence-overview* and their respective components in separate folders, as well as keeping shared components and services in the *shared* folder. The generated *client* is kept outside *src* according to the standard.

The naming conventions and other rules, like the 400 line per file and 75 line per function limit and the single responsibility principle were respected and adhered to whenever possible. All in all, the final frontend application can be considered a good example of how an Angular application should be built.

As for the backend, the decision to introduce a data design that is different from the OAS standard as described in section 6.4 proved to be valuable as it was and remains easily expandable for new features. Parsing the files once and then delivering only requested datasets to the frontend also enabled us to speed up loading times and introduce caching for certain parts that don't change often.

User Tests
The user tests were a success. While tests with more users and with more diverse qualifications could have brought even more detailed results, having test users that were exactly at the experience level and qualification the application is aimed at has proven to be ideal.

The tests also outlined the importance of the link suggestions already provided by the Pulse AI and the added suggestions based on previously created links. This demonstrates that the product can be integrated with Pulse and contribute significantly to Testifi's product suite.

9.3 Research Questions

The first main research question *"Is the test quality increased by using the sequence generator?"* was split into the two sub questions *"Are more edge cases / special scenarios tested compared to manual testing?"* and *"Can more complex scenarios be tested by using the tool?"*.

Both questions could be positively answered after the final user tests as they have shown the advantages of the application compared to manual testing in efficiency and subsequently volume of possible tests. With the ability to change test data for sequences testing of the same sequence with different data sets to cover edge cases is easily done. Complex scenarios are easier to implement because of the constant user feedback through verification and once set up, changing the sequence or testing it with different data is a simple task.

The second question *"Does using the sequence generator reduce the amount of work necessary to create test sequences?"* was also split into the two subquestions *"Is there a measurable improvement compared to creating / executing them manually?"* and *"Does the increasing number of known links reduce the necessary amount of work further?"*.

The answer to first sub-question is clearly positive as well, as the final user test results in section 8.2.2 show considerable time savings in the creation of even simple sequences. For complex sequences, or if the tests have to be run multiple times, these time savings are to be expected even more considerable. For the second sub-question it can be stated, that even the availability of preexisting suggestions from the Pulse AI helps the user a lot by guiding them in the correct direction. A growing number of suggestions for the user is beneficial to the efficiency and therefore improves the time saving compared to other tools even further.

9.4 Limitations

9.4.1 User Tests

It should be noted that in the context of this project, both the initial and the final user tests were only run users that are skilled in working with APIs and used to working with web applications. As the target audience of the application are software testers with some experience, they were more than suitable for the tests.

9.4.2 Implementation

As the main focus during implementation was providing all required features with an optimized user experience and the software adapted rapidly during iterations, software tests in the front-

and backend had a low priority during development. When integrating the software with Pulse, to make it production ready, extensive unit test suites in both the Angular and Python part of the application would be required.

9.5 Outlook

9.5.1 Research

Even though advanced users were the appropriate target group for the conducted user tests, testing with other user groups, like users that have minimal to no experience in working with APIs could provide further insights in how the software could be improved for these users. Furthermore, letting experienced users test the application over a prolonged time period could provide an understanding on how to improve their experience with additional accelerators.

Further research into how to support new users in learning the application without outside help could improve the necessary adaption time.

9.5.2 Implementation

During the implementation and as a result of the final user tests, multiple improvements and features arose that have the potential to improve the application, but were out of scope and weren't be implemented due to time constraints.

Integration in Pulse
From the start it was defined in the scope that the software would be standalone and not integrated in Pulse in this project. It is also clear however, that for a productive use of the software, integration with pulse as described in section 6.5 massively improves the overall user experience. While in the current version, each project has to be imported by providing the name of a OAS definition and a link file on S3, it would be a possible step for Pulse to automatically create the project and redirect the user once the files were created and imported by the application.

Improved verification
The verification feature of the application works well and provides the user with the result of each request when run on the editor screen. If there is an error however, or if the user wants to check the actual response to a request, they have to open the browser's developer console and check the actual request there. While this is totally practical, it would be an improvement if the user could see the results of the verification directly in the application.

Another possible improvement for the verification would be to trigger it automatically on certain events. In some scenarios, as the deletion or move of an endpoint, automatically re-verifying the sequence would save the user a click.

Improved behaviour when moving endpoints
When endpoints in a sequence are moved or deleted, the user is warned that all corresponding links are removed as well. This was agreed upon with the customer during the implementation phase, as all links would have to be reevaluated on correctness after moving an endpoint anyway. This could be improved however by specifically only remove links that are invalid (due to the source endpoint not being executed before the target anymore) and mark other links to be rechecked by the user.

User support
While the user tests have shown that the overall understanding and intuitiveness of the application are good, there is a potential to help the user with further visual support. As currently the keyboard shortcuts are only mentioned in the documentation and not in the front end itself,

only users that take the time to read about the application will know and use them. This could be improved by providing a user interface that lists all available keyboard shortcuts.

Also, further shortcuts could be introduced to improve the experience for advanced users. Some possible actions that could be triggered by shortcuts are:

- Add Endpoint after the currently selected
- Verify the sequence
- Select Test Data as link source

Another help for new users that could improve their experience would be the addition of tooltips for buttons and other elements, that provide additional information.

Test Data

During the final user tests, multiple possible improvements around the handling of test data transpired. While it is possible to change test data in a sequence to test multiple scenarios, it would be beneficial to either allow copying sequences in order to make them available with different test data sets to assist in continuous testing, or make different sets of test data available for a sequence, so it can be run multiple times with different data easily. Another suggested solution for this was a feature to import and export test data sets in order to manage them elsewhere and allow easy switching.

10 Conclusion

To get a deeper understanding of how to achieve the main goals of the software by providing an application that is intuitive and supports the user in achieving their tasks in the most efficient way, literature about basic user interface design principles as well as user experience concept for advanced users were reviewed. Based on this foundation, first low-fidelity designs were created in multiple iterations while continually consulting with the customer as well as design experts. With initial user tests, design flaws in these designs were eliminated and the concept progressed into high-fidelity designs and a user experience concept which outlined the user interaction with a focus on advanced users. In parallel to the design taking shape, the application was already developed iteratively. Agile processes (Scrum) allowed the development and design to run simultaneously while always adapting and improving. The scope of the project also saw minor adaptions during development, based on customer needs, which was easily manageable due to the agile processes.

Final user tests confirmed the achievement of the main goals of the product by demonstrating that the application outperforms manual API testing methods on efficiency and ease of use. The user interface was well received by both the test users and customer.

Due to the user interface design and user experience concept, the software is intuitive to advanced users and provides considerable improvements for API testing. Especially the design of the linking feature was received well and the helpful link suggestions, soured both from the Pulse AI and links created while working with the software, were essential to the success of the project and commended by test users and the customer.

When integrated in Pulse, the Sequence Generator Tool will also provide Testifi with the ability to use it to source new endpoint links that can be used to improve their AI.

Bibliography

[1] B. Shneiderman, „The eyes have it: A task by data type taxonomy for information visualizations“, in *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996, pp. 336–343. DOI: 10.1109/VL.1996.545307.

[2] C. M. D. S. Freitas, „Information visualization techniques“, in *Handbook of Human Computer Interaction*, J. Vanderdonckt, P. Palanque, and M. Winckler, Eds. Cham: Springer International Publishing, 2020, pp. 1–44, ISBN: 978-3-319-27648-9. DOI: 10.1007/978-3-319-27648-9_18-1. [Online]. Available: https://doi.org/10.1007/978-3-319-27648-9_18-1.

[3] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd. USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0201694972.

[4] A. Cooper, R. Reimann, D. Cronin, and C. Noessel, *About Face: The Essentials of Interaction Design*, 4th. Wiley Publishing, 2014, ISBN: 1118766571.

[5] M. Ghoniem, J.-D. Fekete, and P. Castagliola, „A comparison of the readability of graphs using node-link and matrix-based representations“, in *IEEE Symposium on Information Visualization*, 2004, pp. 17–24. DOI: 10.1109/INFVIS.2004.1.

[6] K. Kaplan. „8 design guidelines for complex applications“. (Nov. 2020), [Online]. Available: https://www.nngroup.com/articles/complex-application-design/.

[7] A. Furness. „Designing for advanced users“. (Feb. 2020), [Online]. Available: https://uxdesign.cc/designing-for-advanced-users-cfc95c058cbd.

[8] A. Harley. „Accelerators allow experts to increase efficiency“. (Dec. 2019), [Online]. Available: https://www.nngroup.com/articles/ui-accelerators/.

[9] „Swagger about“. (2023), [Online]. Available: https://swagger.io/about/.

[10] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, and U. Sarid. „Openapi specification v3.1.0“. (Feb. 2021), [Online]. Available: https://spec.openapis.org/oas/latest.html.

[11] J. Wagner. „What's the difference between openapi types 2.0, 3.0, and 3.1?“ (Sep. 2020), [Online]. Available: https://blog.stoplight.io/difference-between-open-v2-v3-v31.

[12] A. Wright, H. Andrews, B. Hutton, and G. Dennis. „Json schema: A media type for describing json documents“. (Jun. 2022), [Online]. Available: https://json-schema.org/draft/2020-12/json-schema-core.html#name-keywords-for-applying-subsch.

[13] M. Droettboom. „Understanding json schema — understanding json schema 2020-12 documentation“. (Jan. 2023), [Online]. Available: https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf.

[14] D. Molteni. „Landscape of api traffic“. (Jul. 2023), [Online]. Available: https://blog.cloudflare.com/landscape-of-api-traffic/.

[15] „What is Angular?“ (Feb. 2022), [Online]. Available: https://angular.io/guide/what-is-angular.

[16] „Angular material“. (2023), [Online]. Available: https://material.angular.io/cdk/categories.

[17] „Ngrx docs“. (2023), [Online]. Available: https://ngrx.io/.

[18] „Rxjs“. (2023), [Online]. Available: https://rxjs.dev/.

[19] „Sass: Sass basics“. (2023), [Online]. Available: https://sass-lang.com/guide/.

[20] „Angular styleguide“. (Feb. 2022), [Online]. Available: https://angular.io/guide/styleguide.

[21] „What is python? executive summary“. (2023), [Online]. Available: https://www.python.org/doc/essays/blurb/.

[22] „Flask documentation (2.3.x)“. (2023), [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/.
[23] K. Yang. „Spectree“. (2020), [Online]. Available: https://0b01001001.github.io/spectree/index.html.
[24] „Odmantic“. (2023), [Online]. Available: https://art049.github.io/odmantic/.
[25] „Pydantic documentation“. (2023), [Online]. Available: https://docs.pydantic.dev/latest/.
[26] J. Finkhaeuser. „Documentation of prance“. (2022), [Online]. Available: https://prance.readthedocs.io/en/latest/.
[27] A. Maciag. „Openapi-spec-validator“. (2023), [Online]. Available: https://openapi-spec-validator.readthedocs.io/en/latest/.
[28] J. Hinrichs. „Api — pysftp 0.2.9 documentation“. (2016), [Online]. Available: https://pysftp.readthedocs.io/en/release_0.2.9/pysftp.html.
[29] „Flask: Alles wichtige zum micro-framework“, *IONOS Digital Guide*, Mar. 1, 2023. [Online]. Available: https://www.ionos.de/digitalguide/websites/web-entwicklung/flask-framework-im-ueberblick/.
[30] „What is amazon s3? - amazon simple storage service“. (2023), [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html.
[31] „What is mongodb? — mongodb manual“. (2023), [Online]. Available: https://www.mongodb.com/docs/manual/.
[32] „Github - openapitools/openapi-generator: License information“. (2023), [Online]. Available: https://github.com/OpenAPITools/openapi-generator/#34---license-information-on-generated-code.
[33] H. Nielsen, J. Mogul, L. M. Masinter, *et al.*, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616, Jun. 1999. DOI: 10.17487/RFC2616. [Online]. Available: https://www.rfc-editor.org/info/rfc2616.
[34] L. M. Dusseault and J. M. Snell, *PATCH Method for HTTP*, RFC 5789, Mar. 2010. DOI: 10.17487/RFC5789. [Online]. Available: https://www.rfc-editor.org/info/rfc5789.
[35] L. Richardson. „Justice will take us millions of intricate moves: Act 3“. (2008), [Online]. Available: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.
[36] J. Webber, S. Parastatidis, and I. Robinson, *REST in practice, Hypermedia and Systems Architecture*. "O'Reilly Media, Inc.", Sep. 17, 2010.
[37] „Docker multi-stage builds“. (Aug. 2023), [Online]. Available: https://docs.docker.com/build/building/multi-stage/.
[38] „Angular security“. (2023), [Online]. Available: https://angular.io/guide/security#sanitization-example.
[39] N. Basal, „Diy keyboard shortcuts in your angular application - netanel basal“, Dec. 10, 2021. [Online]. Available: https://netbasal.com/diy-keyboard-shortcuts-in-your-angular-application-4704734547a2.
[40] „Ngrx docs - store“. (2023), [Online]. Available: https://ngrx.io/guide/store.

Glossary

Angular	Framework for developing frontend applications developed by Google Inc
Cascading Style Sheets	Style sheet language used for describing the presentation of a document written in HTML or other markup languages
Create Read Update Delete	Standard operations of persistent storage, or in user interfaces where data can be manipulated
Data Transfer Object	Objects used to transport data between systems. Represents the payload of API requests.
Docker	A platform to develop and create containerized workloads
Docker Compose	A tool for orchestrating multiple Docker containers. Containers and their dependencies between each other can be defined and then started with a single command
endpoint	An endpoint in a project or sequence represents any HTTP endpoint that was imported from the OAS during project creation. endpoints can be added to sequences and their respective response values can be linked to request values of subsequent endpoints
Flask	Framework to create web applications using Python
Hypertext Transfer Protocol	An application layer protocol used for distributed information systems utilized by most services on the World Wide Web
link	A link represents a connection between a request value (an endpoint parameter or a field in it's request body) and a source for that value. The source can either be a manually created test data field, or a field in a preceding request's response body
link suggestion	A link-suggestion represents the possiblilty of a link existing between two endpoints. Suggestions are created either from the link file from pulse or manually when a user creates a link in the application
MongoDB	MongoDB is a NoSQL Database solution that stores document-based data
NgRx	NgRx Store provides reactive state management for Angular apps inspired by Redux. Unify the events in your application and derive state using RxJS
Node.js	Node.js is a javascript runtime. It can execute javascript outside a webbrowser.
Object–relational mapping	Layer that maps python objects to records of the database. Because MongoDB uses documents it is also called ODM (Object Document Mapper) in this context

OpenAPI Generator	A tool that generates source code for API clients in different programming languages and frameworks based on a OAS document.
OpenAPI Specification	Standard to describes web services
project	A project wraps all data for a specific API. It corresponds with one import of an OAS into Pulse and is based on the OAS file as well as the link file created by pulse during such an import. Multiple sequences can be created in a project
Pulse	Pulse is a cloud service offered by Testifi GmbH that automates API testing using artificial intelligence
Python	Interpreted programming language
Representational State Transfer	Paradigm for creating resource based APIs
RxJS	RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code
Secure File Transfer Protocol	Protocol to interact with a remote file system over a secure network connection
sequence	A sequence describes a list of related endpoints and their relation. It is the main entity of the application as it contains endpoints to test, the order in which they are to be executed, the links between them as well as test data to be used in the requests
Simple Storage Service	A cloud object storage service offered by Amazon Web Services
Swagger UI	User interface to view and interact with APIs based on OASs

Acronyms

AI	Artificial intelligence
API	Application Programming Interface
CDN	content delivery network
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DTO	Data Transfer Object
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	integrated development environment
JSON	JavaScript Object Notation
MVC	Model View Controller
OAS	OpenAPI Specification
ORM	Object–relational mapping
REST	Representational State Transfer
S3	Simple Storage Service
SFTP	Secure File Transfer Protocol
XML	Extensible Markup Language

List of Figures

List of Tables

List of Listings

Declaration of Academic Honesty

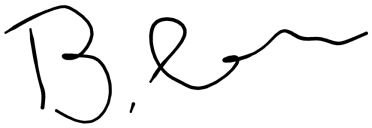
We hereby declare that the present bachelor's thesis has been independently authored solely by us, utilizing only the specified sources. Passages extracted verbatim or in substance from the referenced sources have been duly indicated in the work as either direct quotations or paraphrased content. This bachelor's thesis has not been previously published, nor has it been made available to other parties or submitted to any other examination authority.

Windisch, 18. August 2023

Name: Jonas Volken

Unterschrift:

Name: Benjamin Leu



Unterschrift:

A Initial Assignment

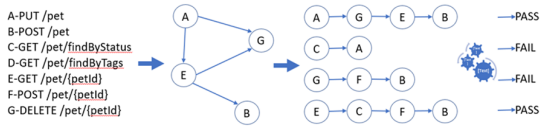
Fachhochschule Nordwestschweiz

App Web | Design UX

23FS_IMVS12: Interactive Use-Case Generation for Functional REST API Testing

			Priority 1	Priority 2
Advisor:	Martin Kropp			
Client:	Testifi Gmbh	**Work scope:**	P6 (360h pro Student)	P5 (180h pro Student)
		Team size:	2er Team	2er Team
Languages:	English			
Study course:	Computer Science			

Initial position



REST Endpoint graph

Regardless of product, enterprise, or domain, software is an inevitable part of the industrial operation. Software development, therefore, is one of the most important aspects of business success in our age. Digitalization is everywhere. Software testing is an essential quality assurance step used in software development. Testing gives feedback to developers about the quality and functionality of their software so they can correct and optimize their product. The later an issue is found the more expensive it is. Testing manually is simply not enough for modern software development that demands speed and continuous development. Therefore, test automation - tests executed automatically via software tools - is a critical success factor for modern software delivery, and therefore, for modern business. Since modern software development methods employ microservices extensively, test automation for REST APIs is the focus of this project.

Objective

This project aims to to develop a GUI that eases test case creation for REST API-based applications. Testifi is already able to analyze the endpoints of APIs, infer the producer-consumer relationship between endpoints, and create a graph. That helps create a sequence of requests to send to the server to check if the logic behind this sequence is implemented. This sequence can be login-purchase-logout, or similar use-cases which the applications might have. With the generation of sequences of requests, we can test if all endpoints are able to work in harmony with other endpoints (which is called integration tests), and if all planned functionalities are implemented (which is called functional tests). At the end of the execution of these tests, the overall picture of what is implemented correctly and what is not can be seen.

Moreover, since graphs might not have all actual connections which exist in real applications, manual generation of sequences should be implemented using UI. The information learned from interactively created connections can also be used to fix the wrong edges on or add the missing edges into the graph.

Problem statement

Planning, generation, and execution of test cases are time-consuming and significantly affected by human errors. In addition, while the project is evolving, APIs are also updated according to current needs. Changes in APIs need to be also reflected to test cases. When all these processes (creating requests, entering correct parameters and payload data, creating test datasets, following all changes and refactoring test cases, executing and reporting tests) are performed manually, testers spend most of their time debugging errors and maintaining the tests. They usually end up having less than necessary amount of tests. This project will minimize the efforts to create, maintain, and execute the tests so that testers can create more tests.

Development of this feature involves usage of graphs and other data structures, algorithm design, optimization of resources, development of GUI on frontend, development of endpoints on backend, database design. AngularJS will be used for GUI development, Flask (python) will be used as a server and DynamoDB will be used as a database. The students will need to learn about API design principles, OpenAPI standards, and software testing methods

Technologies/Technical emphasis/References

AngularJS, Python, Flask, DynamoDB(AWS)

References

· https://swagger.io/specification/

· https://docs.gitlab.com/ee/user/application_security/api_fuzzing/

B Project Agreement

Fachhochschule Nordwestschweiz
Hochschule für Technik

Project Agreement

Interactive Use-Case Generation for
Functional REST API Testing

Benjamin Leu & Jonas Volken
Study Course Computer Science
Spring Semester 2023

Client: Testifi GmbH, Nejdet Dogru
Coach: Martin Kropp, Fabian Affolter

Windisch, 13.03.2023

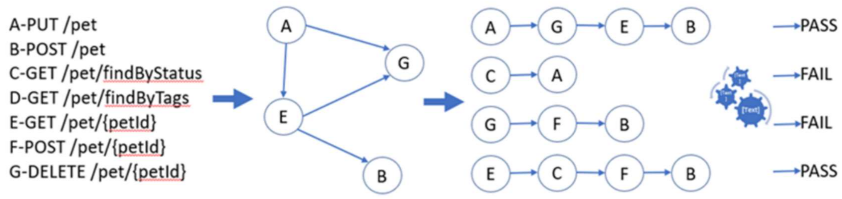
0 Table Of Contents

1 Client

Testifi GmbH is an international company headquartered in Munich, Germany. Their focus is on optimising software delivery processes through DevOps integrations while providing AI-automated quality assurance solutions. They provide a highly advanced software tool for fully integrated automated testing and aim to give non-technical teams the ability to test and create automated tests with no expert knowledge required.

2 Initial Position

Regardless of product, enterprise, or domain, software is an inevitable part of the industrial operation. The use of quality software, therefore, is one of the most important aspects of business success in our age. Digitalization is everywhere. Software testing is an essential quality assurance step used in software development. Testing gives feedback to developers about the quality and functionality of their software so they can correct and optimize their product. Furthermore, testing improves the overall stability of the product and enables the developers to easily add and change features. The later an issue is found the more work is required to fix it. Testing manually is simply not sufficient for modern software development that demands speed and continuous development. Therefore, test automation - tests executed automatically via software tools - is a critical success factor for modern software delivery, and therefore, for modern business. Since modern software development methods employ microservices extensively, test automation for REST APIs is the focus of this project.



3 Problem Statement

Planning, generation, and execution of test cases are time-consuming and significantly affected by human errors. In addition, while the project is evolving, APIs are also updated according to current needs. Changes in APIs need to be also reflected in test cases. When all these processes (creating requests, entering correct parameters and payload data, creating test datasets, following all changes, and refactoring test cases, executing and reporting tests) are performed manually, testers spend most of their time debugging errors and maintaining the tests. They usually end up having less than necessary number of tests. This results in suboptimal test coverage, as well as a lack of transparency regarding the quality and quantity of manually performed tests.

Testifi is already able to analyse the endpoints of APIs, infer the producer-consumer relationship between endpoints, and create a graph. That helps create a sequence of requests to send to the server to check if the logic behind this sequence is implemented. The creation of sequences by AI is very limited, and the manual execution of multiple requests that represent a sequence is time consuming.

The amount of work required to create and maintain tests could be vastly reduced if the testers were supported with automation and could use tools to create the tests instead of writing them. Reducing the effort of writing individual tests would also free up resources to improve the quality and coverage of the executed tests and therefore contribute to the quality and speed of future developments.

4 Goals

The vision of this project is to minimize the effort of testers used to create and maintain testing scenarios, so the testers can focus on software quality instead of writing tests.

This project aims to develop a sequence generator that eases test case creation for REST API-based applications by enabling the tester to easily create and execute sequences of requests. These sequences can be login-purchase-logout, or similar use-cases which the applications might have. The testers can create connections between response values and request values of subsequent requests to reuse returned value and generate a request-response-flow. With the generation of sequences of requests, testers can check if the endpoints are able to work in harmony with each other (integration tests) and if all planned functionalities are implemented (functional tests). At the end of the execution of these tests, the user can see in a report if all requests were executed successfully. As the software is used, it will gain more information about the links between requests and responses. If possible, the information learned from interactively created connections can be used to fix the wrong edges on or add the missing edges into the graph. Subsequently created sequences can provide a higher number of predefined connections in order to further reduce the manual input required by testers.

While providing all the functionality needed, the tool should also meet user experience goals. The software that we provide for the testers should be intuitive and easy to use for users that are not familiar with software development. While state of the art concepts for working with graphs and workflows will be researched and used if they match our requirements, it will also be in accordance with Testifi's design guidelines to match the other products it will later be implemented with.

4.1 Research Questions

Based on the goals, the following research questions are to be answered in this project:

1) Is the test quality increased by using the sequence generator?
    a. Are more edge cases / special scenarios tested compared to manual testing?
    b. Can more complex scenarios be tested by using the tool?
2) Does using the sequence generator reduce the amount of work necessary to create test sequences?
    a. Is there a measurable improvement compared to creating / executing them manually?
    b. Does the increasing number of known links reduce the necessary amount of work further?

5 Project Organisation

The total amount of work assigned to the project is 720 hours. Each student works 360 hours on the project.
A project roadmap with the most important milestones is attached.

Scrum is used as a project method. Each scrum sprint has a duration of two weeks. Before each sprint the project team conducts a sprint planning where tasks for the next sprint are prioritised. At the end of each sprint a sprint review will be held where the progress is presented to the customer and the project advisor.
To track the scrum sprints Jira is used, where all tasks will be managed.

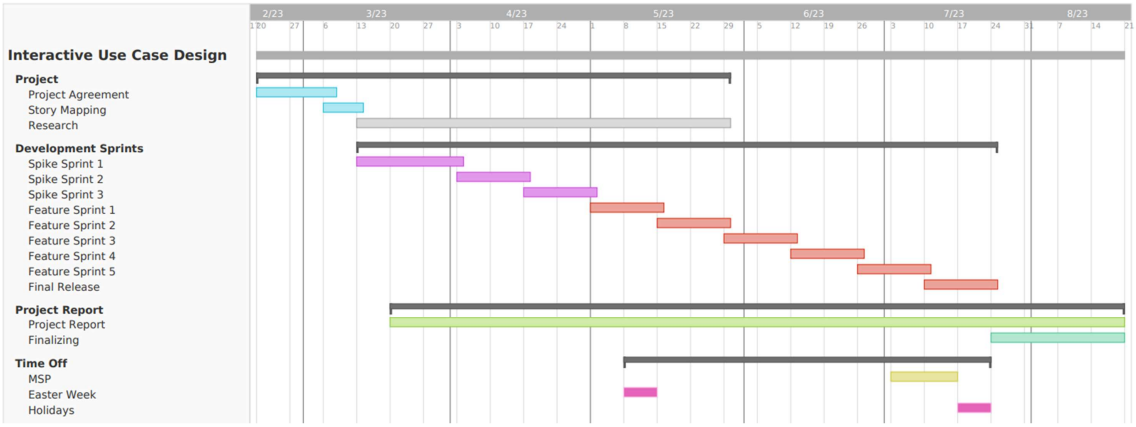
Slack is used as a platform for quick and informal exchanges during the sprints.
For creation of screen designs and mock-ups Figma is used.
All meetings will be held online via Google Meet, whenever possible the project team will join the meetings from the FHNW campus.

6 Technologies

Testifi uses the following tech-stack for their “pulse” product. Therefore, our project is bound to these technologies. The code produced in the scope of this project will be separate from any existing source code. The integration into the existing solution is out of scope for this project.

- Angular
- Python / Flask
- DynamoDB (AWS)

7 Planned Sprints



8 Signatures

Students

Benjamin Leu	_____	Location, Date:	_____
Jonas Volken	_____	Location, Date:	_____

Client Testifi GmbH

Nejdet Dogru	_____	Location, Date:	_____

Advisor

Martin Kropp	_____	Location, Date:	_____

C REST API Documentation

Generated from openapi defintion using https://www.swdoc.org/

Pulse Sequence Editor

Overview

Tags

project

== Paths === *GET* `/api/v1/projects` get_projects <GET> .Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/import` create_project_from_file <GET>

Parameters

Type	Name	Description	Schema
query	**schema_file** *required*		string
query	**link_file** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
409	Conflict	No Links

Code	Description	Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}` get_project <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/endpoints` get_endpoints <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links

Code	Description	Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/link_suggestions`
get_link_suggestions <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/sequences`
get_sequences <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
query	**text** *optional*		string
query	**endpoint_ids** *optional*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

POST `/api/v1/projects/{project_id}/sequences`
create_sequence <POST>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

DELETE `/api/v1/projects/{project_id}/sequences/{sequence_id}`
delete_sequence <DELETE>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string

Type	Name	Description	Schema
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

PATCH /api/v1/projects/{project_id}/sequences/{sequence_id} update_sequence <PATCH>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/sequences/{sequence_id}/endpoints` get_sequence_endpoints <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

POST `/api/v1/projects/{project_id}/sequences/{sequence_id}/endpoints` create_sequence_endpoint <POST>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links

Code	Description	Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

DELETE
/api/v1/projects/{project_id}/sequences/{sequence_id}/endpoints/{sequence_endpoint_id}
delete_sequence_endpoint <DELETE>

Parameters

Type	Name	Description	Schema
path	**sequence_endpoint_id** *required*		string
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

PATCH
/api/v1/projects/{project_id}/sequences/{sequence_id}/endpoints/{sequence_endpoint_id}
update_sequence_endpoint <PATCH>

Parameters

Type	Name	Description	Schema
path	**sequence_endpoint_id** *required*		string
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/sequences/{sequence_id}/links` get_links <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links

Code	Description	Links
422	Unprocessable Entity *Content* **application/json**	No Links

POST /api/v1/projects/{project_id}/sequences/{sequence_id}/links create_link <POST>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

DELETE /api/v1/projects/{project_id}/sequences/{sequence_id}/links/{link_id} delete_link <DELETE>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Type	Name	Description	Schema
path	**link_id** *required*		string

Responses

Code	Description	Links
200	OK	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

GET `/api/v1/projects/{project_id}/sequences/{sequence_id}/test_data` get_test_data <GET>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

POST `/api/v1/projects/{project_id}/sequences/{sequence_id}/test_data` create_test_data <POST>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

PUT `/api/v1/projects/{project_id}/sequences/{sequence_id}/test_data/{test_data_id}` update_test_data <PUT>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string
path	**test_data_id** *required*		string

Responses

Code	Description	Links
200	OK *Content* **application/json**	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

DELETE `/api/v1/projects/{project_id}/sequences/{sequence_id}/test_data/{test_data_id}` delete_test_data <DELETE>

Parameters

Type	Name	Description	Schema
path	**project_id** *required*		string
path	**sequence_id** *required*		string
path	**test_data_id** *required*		string

Responses

Code	Description	Links
200	OK	No Links
403	Forbidden	No Links
422	Unprocessable Entity *Content* **application/json**	No Links

Components

Schemas

EndpointResponseList

Title: EndpointResponseList

EndpointResponseList.AnyOfSchema

Title: AnyOfSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: anyOf	enum (anyOf)

EndpointResponseList.ArraySchema

Title: ArraySchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
items *optional*	**Title**: Items	
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: array	enum (array)

EndpointResponseList.BooleanSchema

Title: BooleanSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: boolean	enum (boolean)

EndpointResponseList.ContentType

An enumeration.

Title: ContentType

EndpointResponseList.EndpointResponse

Title: EndpointResponse

Properties

Name	Description	Schema
description *required*	**Title**: Description	string
id *required*	**Title**: Id	string
method *required*		EndpointResponseList.HttpMethod
operation_id *required*	**Title**: Operation Id	string
parameters *required*	**Title**: Parameters	< EndpointResponseList.Parameter > array
path *required*	**Title**: Path	string
request_body *required*	**Title**: Request Body	< EndpointResponseList.RequestBody > array
responses *required*	**Title**: Responses	< EndpointResponseList.Response > array

EndpointResponseList.HttpMethod

An enumeration.

Title: HttpMethod

EndpointResponseList.IntegerSchema

Title: IntegerSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string

Name	Description	Schema
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: integer	enum (integer)

EndpointResponseList.NumberSchema

Title: NumberSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: number	enum (number)

EndpointResponseList.ObjectProperty

Title: ObjectProperty

Properties

Name	Description	Schema
model *required*	**Title**: Model	
name *required*	**Title**: Name	string

EndpointResponseList.ObjectSchema

Title: ObjectSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
properties *required*	**Title**: Properties	< EndpointResponseList.ObjectProperty > array
schema_type *optional*	**Title**: Schema Type	string

Name	Description	Schema
type *optional*	**Title**: Type **Default**: object	enum (object)

EndpointResponseList.OneOfSchema

Title: OneOfSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: oneOf	enum (oneOf)

EndpointResponseList.Parameter

Title: Parameter

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
location *required*		EndpointResponseList.ParameterLocation
name *required*	**Title**: Name	string
required *optional*	**Title**: Required	boolean
type *required*	**Title**: Type	string

EndpointResponseList.ParameterLocation

An enumeration.

Title: ParameterLocation

EndpointResponseList.RequestBody

Title: RequestBody

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string
model *required*	**Title**: Model	
type *required*		EndpointResponseList.ContentType

EndpointResponseList.Response

Title: Response

Properties

Name	Description	Schema
code *optional*	**Title**: Code	string
content *required*	**Title**: Content	< EndpointResponseList.ResponseContent > array
description *optional*	**Title**: Description	string

EndpointResponseList.ResponseContent

Title: ResponseContent

Properties

Name	Description	Schema
model *required*	**Title**: Model	
type *required*		EndpointResponseList.ContentType

EndpointResponseList.StringSchema

Title: StringSchema

Properties

Name	Description	Schema
description *optional*	**Title**: Description	string

Name	Description	Schema
schema_type *optional*	**Title**: Schema Type	string
type *optional*	**Title**: Type **Default**: string	enum (string)

FilterSequenceQuery

Title: FilterSequenceQuery

Properties

Name	Description	Schema
endpoint_ids *optional*	**Title**: Endpoint Ids	string
text *optional*	**Title**: Text	string

ImportProjectQuery

Title: ImportProjectQuery

Properties

Name	Description	Schema
link_file *required*	**Title**: Link File	string
schema_file *required*	**Title**: Schema File	string

LinkRequest

Title: LinkRequest

Properties

Name	Description	Schema
sourceId *required*	**Title**: Sourceid	string
sourcePath *required*	**Title**: Sourcepath	string
sourceType *required*		LinkRequest.SourceType
targetId *required*	**Title**: Targetid	string

Name	Description	Schema
targetPath *required*	**Title**: Targetpath	string

LinkRequest.SourceType

An enumeration.

Title: SourceType

LinkResponse

Title: LinkResponse

Properties

Name	Description	Schema
id *required*	**Title**: Id	string
sourceId *required*	**Title**: Sourceid	string
sourcePath *required*	**Title**: Sourcepath	string
sourceType *required*		LinkResponse.SourceType
targetId *required*	**Title**: Targetid	string
targetPath *required*	**Title**: Targetpath	string

LinkResponse.SourceType

An enumeration.

Title: SourceType

LinkResponseList

Title: LinkResponseList

LinkResponseList.LinkResponse

Title: LinkResponse

Properties

Name	Description	Schema
id *required*	**Title**: Id	string

Name	Description	Schema
sourceId *required*	**Title**: Sourceid	string
sourcePath *required*	**Title**: Sourcepath	string
sourceType *required*		LinkResponseList.SourceType
targetId *required*	**Title**: Targetid	string
targetPath *required*	**Title**: Targetpath	string

LinkResponseList.SourceType

An enumeration.

Title: SourceType

LinkSuggestionResponseList

Title: LinkSuggestionResponseList

LinkSuggestionResponseList.LinkSuggestionResponse

Title: LinkSuggestionResponse

Properties

Name	Description	Schema
id *required*	**Title**: Id	string
source_id *required*	**Title**: Source Id	string
source_path *required*	**Title**: Source Path	string
source_schema *required*	**Title**: Source Schema	string
target_id *required*	**Title**: Target Id	string
target_path *required*	**Title**: Target Path	string
target_schema *required*	**Title**: Target Schema	string

Name	Description	Schema
target_type *required*		LinkSuggestionResponseList.TargetType

LinkSuggestionResponseList.TargetType

An enumeration.

Title: TargetType

ProjectResponse

Title: ProjectResponse

Properties

Name	Description	Schema
base_path *required*	**Title**: Base Path	string
id *required*	**Title**: Id	string
link_file *required*	**Title**: Link File	string
name *required*	**Title**: Name	string
schema_file *required*	**Title**: Schema File	string

ProjectResponseList

Title: ProjectResponseList

ProjectResponseList.ProjectResponse

Title: ProjectResponse

Properties

Name	Description	Schema
base_path *required*	**Title**: Base Path	string
id *required*	**Title**: Id	string
link_file *required*	**Title**: Link File	string
name *required*	**Title**: Name	string

Name	Description	Schema
schema_file *required*	**Title**: Schema File	string

SequenceEndpointRequest

Title: SequenceEndpointRequest

Properties

Name	Description	Schema
endpointId *required*	**Title**: Endpointid	string
position *required*	**Title**: Position	integer

SequenceEndpointResponse

Title: SequenceEndpointResponse

Properties

Name	Description	Schema
endpointId *required*	**Title**: Endpointid	string
id *required*	**Title**: Id	string
position *required*	**Title**: Position	integer

SequenceEndpointResponseList

Title: SequenceEndpointResponseList

SequenceEndpointResponseList.SequenceEndpointResponse

Title: SequenceEndpointResponse

Properties

Name	Description	Schema
endpointId *required*	**Title**: Endpointid	string
id *required*	**Title**: Id	string
position *required*	**Title**: Position	integer

SequenceEndpointUpdateRequest

Title: SequenceEndpointUpdateRequest

Properties

Name	Description	Schema
position *required*	**Title**: Position	integer

SequenceRequest

Title: SequenceRequest

Properties

Name	Description	Schema
description *required*	**Title**: Description	string
name *required*	**Title**: Name	string

SequenceResponse

Title: SequenceResponse

Properties

Name	Description	Schema
description *required*	**Title**: Description	string
endpoints *required*	**Title**: Endpoints	< string > array
id *required*	**Title**: Id	string
name *required*	**Title**: Name	string

SequenceResponseList

Title: SequenceResponseList

SequenceResponseList.SequenceResponse

Title: SequenceResponse

Properties

Name	Description	Schema
description *required*	**Title**: Description	string

Name	Description	Schema
endpoints *required*	**Title**: Endpoints	< string > array
id *required*	**Title**: Id	string
name *required*	**Title**: Name	string

TestDataRequest

Title: TestDataRequest

Properties

Name	Description	Schema
name *required*	**Title**: Name	string
type *required*		TestDataRequest.Type
value *required*	**Title**: Value	string

TestDataRequest.Type

An enumeration.

Title: Type

TestDataResponse

Title: TestDataResponse

Properties

Name	Description	Schema
id *required*	**Title**: Id	string
name *required*	**Title**: Name	string
type *required*		TestDataResponse.Type
value *required*	**Title**: Value	string

TestDataResponse.Type

An enumeration.

Title: Type

TestDataResponseList

Title: TestDataResponseList

TestDataResponseList.TestDataResponse

Title: TestDataResponse

Properties

Name	Description	Schema
id *required*	**Title**: Id	string
name *required*	**Title**: Name	string
type *required*		TestDataResponseList.Type
value *required*	**Title**: Value	string

TestDataResponseList.Type

An enumeration.

Title: Type

ValidationError

Model of a validation error response.

Title: ValidationError

ValidationError.ValidationErrorElement

Model of a validation error response element.

Title: ValidationErrorElement

Properties

Name	Description	Schema
ctx *optional*	**Title**: Error context	object
loc *required*	**Title**: Missing field name	< string > array
msg *required*	**Title**: Error message	string
type *required*	**Title**: Error type	string