CARDIFF
UNIVERSITY

PRIFYSGOL
CAERDYDD

A platform that combines spaced repetition and Code-katas, to teach python more effectively.

By: Thomas Mahony-Kelross

Supervisor: Martin Chorley

Abstract

To improve our programming skills, we need effective methods of practice. In this project, a technique for the spaced-repetition based practice of code-katas is demonstrated. The project focuses on creating an open-source and self-hostable platform to make this method available within the context of undergraduate computer science courses. There is a focus on using kata for developing python programming language skills. Users practice kata through a web application using the Monaco editor and sending their solutions to a solution testing API. This API validates a solution by running it against a set of PyTest unit tests. Once a user completes a kata, the SuperMemo 2 algorithm determines when the kata should be practised next. Work was started to sandbox the solution testing workload using AWS Firecracker. Katas are authored using a GitOps-like methodology that uses a reconciler pattern to sync a set of declarative configuration files within a Git repository with the Store. The Store is a JSON HTTP API that implements the platforms data model, acts as its persistent data store, and makes it available to others. The project successfully demonstrates this platform as a minimal viable product and outlines the steps needed for its wider deployment. The report also discusses the initial architectural decision to implement the solution through an isomorphic typescript core.

Acknowledgements

To Martin Chorley, for his supervision and support throughout the project.

To my parents, for their support, encouragement and diligent proofreading

Table of Contents

Contents

Table of Figures

1 Introduction

People may desire to improve their skills for many reasons, perhaps for professional development, curiosity or in pursuit of mastery. However, they are unlikely to improve significantly without an effective way of practising. Common wisdom says, ‘practice makes perfect’ and famous studies such as those by K. Ericsson show the importance of deliberate practice[1]. This report is about my individual project, which has been concerned with building a platform to orchestrate the effective practice of python programming skills. The project has been focused on when practice should be scheduled, how it can be made deliberate and how it can be automatically assessed.

The focus on when people should practice is centred around the concepts of spaced repetition and the spacing effect. The spacing effect [2], [3] is the phenomena that, over a long period, if people distribute their practice of something over numerous small discrete intervals, they will achieve better results than if they were to cram all their practice close together. Spaced repetition [4], [5] is the process of algorithmically scheduling when, after having just practised something, you should practice it again to make the most use of the spacing effect. It has been used with flashcards in secondary language acquisition and has also seen success in several other domains [6], [7].

The focus on how people practice has centred around the concept of a code kata. The term derives from a karate kata, a physical exercise in which a student moves through a "combination of positions and movements”[8]. Although the individual moves may be simple, there is a significant focus on the repetition of the exercise, allowing the student to perfect the form of each step gradually over time. This term was re-imagined in the context of software development by Dave Thomas[9] as a code kata (further abbreviated to kata). A kata is a programming exercise in which a student practices a specific software development skill. Katas are intended to isolate the practice of a skill from its application. Also, like their Karate counterparts, they are intended to be practised repeatedly to improve the learner's proficiency over time.

Katas generally come in two primary forms, which we will label instructional and practical. Instructional katas are instructions for the learner to carry out in a self-directed and self-assessed way. These tend to focus on general software development skills such as data modelling, object-oriented design, and practising test-driven development. Practical katas consist of a prompt and an accompanying set of unit tests. The prompt directs the learner to solve a problem by producing a piece of code. The unit tests are then used to assess the validity of the learner's solution programmatically. These katas tend to be more focused on skills related to programming, such as the implementation of various algorithms or data structures and the general use of different programming language features. This project focuses on practical katas, seeing them as a helpful way of demarcating deliberate practice of programming skills.

There are several existing platforms where learners may practice practical kata. They typically consist of a web application where a learner receives a prompt and then uploads their solution, which is automatically assessed using a hidden set of unit tests. Several popular open-source projects allow learners to make use of spaced repetition while practising various skills. For example, spaced-repetition flashcard software, which is well suited for practising secondary language vocabulary acquisition and general memorization. This project is concerned with the problem that there is not currently an open-source platform that combines spaced repetition and the deliberate practice of practical katas.

It is believed combing these two evidence-based techniques will enable learners to improve their programming skills more effectively. Therefore, the overarching goal of this project has been to:

build a platform for the authoring and
spaced repetition-based practice of katas.

Although the platform is intended to be general-purpose, the project has focused on how it could be used within the context of undergraduate computer science courses. The project has three key stakeholders: Learners, Authors, and Administrators. Learners are current or recently graduated members of the course whose primary motivation is to practice katas. Authors are teaching staff whose primary motivation is to collaborate on creating katas for Learners to practice. Administrators are people responsible for managing and maintaining an instance of the platform. Due to time and resource constraints, the project has only been concerned with practical katas related to the general use of the python programming language. Also, as the project has been concerned with developing a platform, there was a considerable focus on architecture. The project was developed with a waterfall methodology, where a core set of aims were attempted to be realized before additional functionality was added.

At the end of this project, the following primary outcomes have been achieved. Learners can use a progressive web application to practice kata in a spaced-repetition-based way. A solution testing API has been created to validate the result of a python kata given a set of python unit tests. A workflow demonstrated that allows Authors to create kata using a GitOps-like methodology. The platforms data model has been exposed as an HTTP JSON API. Finally, the platform has been secured using the open-source Keycloak authorization server and OAUTH. Further testing and development are required before the solution can be deployed, but the built solution does achieve all of the four core aim set out within the specification.

2 Background

This part of the report provides some background on existing spaced-repetition software and platforms for practising katas.

2.1 Existing spaced-repetition software

2.1.1 SuperMemo

Version one of Super Memo was created by P. Wozniak in 1987 [10] and has continued to be developed as commercial software through to version 16 in 2013. It is an extensive piece of software, attempting to provide spaced-repetition functionality not only for flashcards but for incremental reading, video, and audio. The latest version of its propriety spaced-repetition algorithm (SM 16) considers a great variety of factors, including even a user's circadian rhythms.

2.1.2 Anki

Anki[11] is arguably the most popular open-source spaced-repetition software package. Accessible on desktop, the web and mobile, it focuses on flashcard-based study. It implements a derivative of SM 2, an openly licensed early version of the SuperMemo algorithm. Because of its open license, derivations of the SM2 algorithm are a popular choice for open source spaced-repetition software.

Anki is an effective learning tool for subjects such as Spanish[12] through to dentistry[13]. There have also been several popular blog posts of people attempting to use it for software development [14] [15]. However, these approaches have focused on memorization of syntax and not the deliberate practice of programming skills.

2.2 Existing platforms to practice kata

As katas are supposed to be carried out in isolation from what a learner needs to do in the professional or academic contexts of their lives, the practice of them needs to occur in a separate dedicated environment. Here we analyze how four online platforms provide such an environment. These platforms are Dave Thomas's original website [16], which demonstrates a provider of instructional kata. The second is the community platform CodeWars[9], which demonstrates the online practice of practical kata. The third is LeetCode[17], a technical interview preparation site, to see how the concept of practical kata generalizes to other online platforms for developing programming skills. Finally, we look at Execute Program[18], a professional training website that implements spaced repetition.

We start with an overview of the platform, focusing on its architecture and how learners access it. Then we look at what katas are available to practice. Then we consider data mobility; what data can you bring in and out of the platform. Finally, we look at how you practice a kata within the platform. To conclude our analysis and extend the karate analogy, we describe each platform as a physical dojo[1].

Dave Thomas's website, CloudKata.com, is a simple static website that anyone can browse. Twenty-one instructional katas are provided as blog posts, and these are the only data you 'take' out of the platform. You practice these instructional katas by carrying out the instructions in the blog post in a self-directed manner. If you are required to write code, you do so by using your local tooling. In a few instances, you are provided with simple unit tests in one programming language, which you can use to self-assess your answers.

[1] This is just an explanatory aid and to help draw comparisons between the platforms and the proposed solution. It should not be confused with a coding dojo[62], a physical community of deliberate practice.

If this website were a dojo, it would be one where you enter, take some instructions from a teacher, and leave to practice them at home.

CodeWars is a freemium community-orientated platform. To register, you must complete a very simple practical kata (to make a function return correctly). There are over 6000 public practical kata available for users to practice for free. Each kata has one too many sets of hidden unit tests, one for each programming language that learners may use to write solutions. Users browse katas through a web interface and create their solution through a web-based IDE. A user then submits solutions until it passes the hidden unit tests. You may choose a 'training style' that emphasizes solving new kata or repeating ones that have been already solved. However, this repetition style emphasizes what kata will appear next in a user's feed and does not involve spaced-repetition scheduling. Solving kata and other tasks grants users of the site reputation points called 'honour'. Once users have enough points, they can author katas on the platform, initially subject to administrator approval. The platforms business model involves subscription-based access to an improved practice experience. For example, premium users' solutions are tested using a more powerful server to receive quicker feedback. In terms of data portability, there is a public API to consume public data about the platform. However, there is no way to export your account data, and if you delete your account, any katas you produced will remain public on the platform. If this website were a dojo, it would be expansive and have a wide variety of kata you could practice. However, it would be the only place you could practice, and if the company behind the platform shut down, you would lose access to all your data.

LeetCode is a freemium technical interview preparation platform. Although it does not use the terminology of katas, it does include 2000 problems that learners solve by producing a solution in code that passes a set of hidden unit tests, just like a practical kata. Like CodeWars, users can upload their own problems. The platforms business model involves subscription-based access to specific sets of problems and video solutions. Again, in terms of data mobility, there is no easy way to export or delete a user's account data. This platform is less like a dojo and more like a vocational training centre. Its purpose is to improve people's skills for a professional aim instead of being concerned with mastery.

Execute Program[18], only discovered by the author part way through the project, is a professional software development training platform. It includes courses on regexes and JavaScript, and learners repeat topics in a spaced-repetition based way. Users do not bring any data in or out of the platform. Again, this platform is less like a dojo and more like a vocational training centre; however, in this case, the training is delivered in a spaced-repetition-based way.

2.3 Discussion of existing solutions

While learners may improve their programming skills through all four of these platforms, only one of them, Execute Program, provides a spaced-repetition based way to practice katas. However, this platform is unsuitable for undergraduate computer science courses for several reasons. Firstly, it is a private, commercial, and costly platform, setting back learners $19 a month to use. Secondly, it provides no way for authors to create kata. The target solution should aim to capture the features of code wars, but with the spaced repetition features of execute program.

3 Specification

3.1 Stakeholders

Three key stakeholders for the project have been devised.

Name:	Learners
Motivation:	To practice kata.

Name:	Authors
Motivation:	To create kata for learners to practice.

Name:	Administrators
Motivation:	To manage an instance of the platform on behalf of Learners and Creators.

3.2 Core Aims

These four aims represent the platforms' core functionality. It was anticipated in the initial plan that all four would be achieved before the end of the project. An aim's objectives represent the actions required to realize the aim and are used later in the report to assess to what extent an aim was achieved. Each aim and objective have an identifier used to refer to it throughout the rest of the report. Some aims have optional objectives, representing opportunities to expand core functionality if time allows.

3.2.1 CA 1 – Kata Creation

CA 1	Kata creation	Enable Creators to author a kata.
Objectives		
O1	Determine a serialization format for a kata, including its prompt and test cases.	
O2	Demonstrate an example workflow for creating katas in this format.	
O2b	Extend this workflow to be more effective and accessible. *(Optional)*	

3.2.2 CA 2 – Kata practice

CA 2	Kata practice	Enable Learners to practice a kata.
Objectives		
O3	Create a mechanism to validate a Learner's solution to a kata.	
O4	Create a client for Learners to practice through.	
O4b	Provide an alternative interface for Learners to practice through. *(Optional)*	

3.2.3 CA 3 – Spaced Repetition

CA 3	Spaced Repetition	Implement spaced repetition for katas.
Objectives		
O5	Implement a well-known spaced repetition algorithm.	
O5b	Enable Learners to specify their own spaced repetition algorithm. (*Optional)*	

3.2.4 CA 4 – Store Data

CA 4	Store Data	Create a data-store for platform data.
Objectives		
O6	Design an appropriate data model.	
O7	Implement this data model using a DBMS technology.	
O8	Provide a service to access this data store.	

3.3 Secondary Aims

These secondary aims represent ways of extending the project past its core functionality. It was anticipated in the initial plan that at least one of these aims would be achieved and that it may only be feasible to implement two or three. Multiple aims were provided to give different paths for the project to continue to be developed once the core functionality was realized.

3.3.1 SA 1 - Security

SA 1	Security	Ensure the platform has a robust security model.
Objectives		
O9	Document the platform’s security boundaries.	
O10	Document the platform’s actors and levels of trust between them.	
O11	Implement appropriate authentication for the platform's infrastructure and services.	
O12	Implement appropriate sandboxing mechanisms to safely test kata solutions.	

3.3.2 SA 2 – Offline

SA 2	Offline	Learners should be able to practice their kata without an internet connection.
Objectives		
O13	Determine a set of constraints under which offline functionality could be delivered.	
O14	Attempt to implement this offline functionality.	
O15	Explore a reasonable synchronization method for when connectivity is restored.	

3.3.3 SA 3 – Alternative Clients

SA 3	Alternative Clients	Should be possible to use alternative clients in the future.
Objectives		
O16	Explore how the platform could work without a progressive web application.	
O17	Explore how alternative clients such as mobile applications could be implemented.	

3.3.4 SA 4 – Data portability

SA 3	Data portability	Data should be portable between different instances of the platform.
Objectives		
O18	Enable katas to be practiced across instances.	
Q19	Enable users to import and export their data from different instances of the platform.	
Q20	Explore potential ways to import and export data to other software systems.	
Q21	Explore potential uses of Open Linked Data and federated technology.	

4 Design

This part of the report maps the core aims and their objectives to the final architecture of the platform by the end of the project. Section 4.1 beings by showing how the platform is decomposed into three separate systems and the core aim(s) each system is concerned with. In sections 4.2, 4.3 and 4.4, each system is broken down into containers, and there is a discussion of what each container should do to meet the aims objectives. In the places that the static architecture covers secondary aims and objectives, this is also indicated and discussed.

The discussion in each section centres around a level one or two C4 Model[19]diagram, found throughout section 13.1. Section 4.1 covers level one with a system context diagram, and sections 4.2, 4.3 and 4.4 cover level two with container diagrams. In the C4 model, a container does not refer to containerization technology but instead means any application or data store. It is highly recommended the reader engages with the C4 Model website[19] if any further clarification on the model is required.

4.1 Overall Platform

For a graphical interpretation of this section, refer to Section 13.1.1.

The platform is decomposed into three systems. The first, the Store System, is responsible for storing persistent data in the platform (CA-4). This includes things such as katas and learners progress on katas. The second, the Practice System, is responsible for allowing learners to practice their kata (CA-2) in a space-repetition based way (CA-3). It achieves this by providing a client for learners to practice through and an API to test learner's solutions during practice. The third, the Authoring System, is responsible for allowing authors to create katas (CA-1). It achieves this by providing a workflow that automatically syncs the store with the state of a Git repository.

A deployment of all three of these systems that have been configured to work together is referred to as an Instance. A user and their data always belong to a particular Instance.

4.2 Store System

For a graphical interpretation of this section, refer to Section 13.1.2

The Store System has two main containers, the Database and the Store API. The Database represents the DBMS where data is stored (O7), and the Store API is an HTTP JSON API that allows various CRUD and other operations on this data (O8).

4.2.1 Data model

The data model of the Store can be represented with the following entity-relationship diagram. Throughout the report, the entities the Store models are referred to as resources.

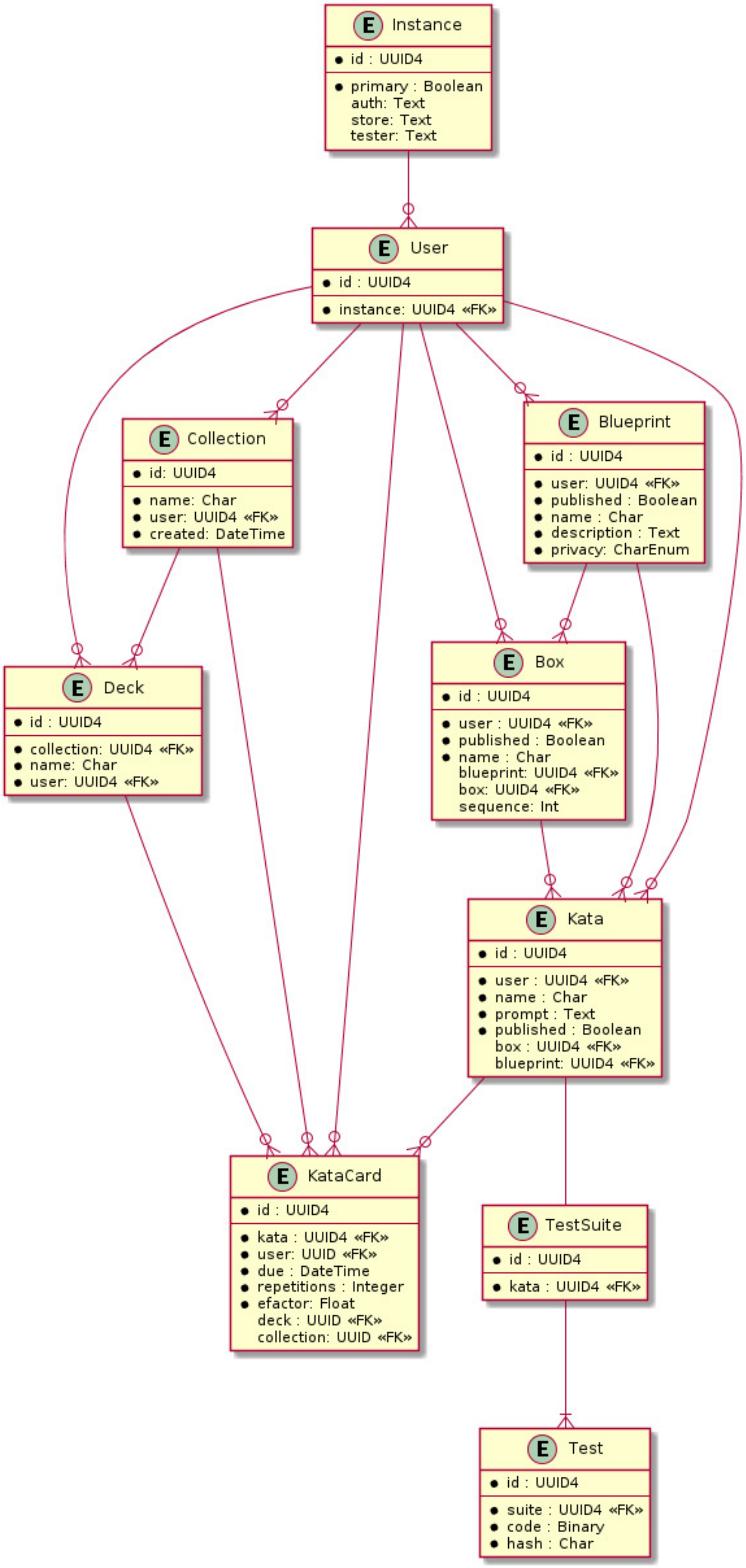

E Instance
● id : UUID4
● primary : Boolean
auth: Text
store: Text
tester: Text
E User
● id : UUID4
● instance: UUID4 «FK»
E Collection
● id: UUID4
● name: Char
● user: UUID4 «FK»
● created: DateTime
E Blueprint
● id : UUID4
● user: UUID4 «FK»
● published : Boolean
● name : Char
● description : Text
● privacy: CharEnum
E Deck
● id : UUID4
● collection: UUID4 «FK»
● name: Char
● user: UUID4 «FK»
E Box
● id : UUID4
● user : UUID4 «FK»
● published : Boolean
● name : Char
blueprint: UUID4 «FK»
box: UUID4 «FK»
sequence: Int
E Kata
● id : UUID4
● user : UUID4 «FK»
● name : Char
● prompt : Text
● published : Boolean
box : UUID4 «FK»
blueprint: UUID4 «FK»
E KataCard
● id : UUID4
● kata : UUID4 «FK»
● user: UUID «FK»
● due : DateTime
● repetitions : Integer
● efactor: Float
deck : UUID «FK»
collection: UUID «FK»
E TestSuite
● id : UUID4
● kata : UUID4 «FK»
E Test
● id : UUID4
● suite : UUID4 «FK»
● code : Binary
● hash : Char

Figure 1

The Instance resource models an instance of the platform and the URI where each of its component services can be located. Each Instance should have one, and only one, Instance resource in its Store with the primary field set to True. This should be considered the source of truth about where the other systems in that instance are located. The User resource models a user of the platform. The users' ID is their persistent identifier from the authentication service, and each user must be associated with a particular Instance resource. Although the platform provides two different systems for the two different use cases of Learners and Authors, there is only one type of user within the data model. Any Learner may behave as an Author and vice versa.

The Kata resource models a kata and has precisely one TestSuite resource. A TestSuite has one too many Tests, and the Tests' code field is where the actual unit tests are stored. A Learner should only be able to access a Katas TestSuite if they authored that Kata.

A Learners' progress is tracked with a KataCard resource. The Kata resource represents the details of a kata exercise. The KataCard resource is like a piece of paper the Learner carries around with their learning data for that exercise. This is where the due date is stored. This date is when a Learner should practice a kata again and is determined by spaced repetition. A KataCard resource is generated for the user when they decide to practice a Kata for the first time.

Authors and Learners may wish to organize their Kata and KataCards for various reasons. To this end, two different tree structures are provided. Authors may create Blueprints. A Blueprint resource represents the root of a tree, with Box resources as branch nodes and Kata resources as leaf nodes. Similarly, Learners may create Collections, with Decks as branch nodes and KataCards as leaf nodes. A Learners KataCards must belong to a Collection, and so any user that attempts to practice will automatically have a default collection created for them if one does not exist.

Authors may wish to hide Katas and the containers they come in (Blueprints and Boxes) until they are ready to be consumed by Learners. For this reason, these resources have a published key that determines their visibility to Learners.

Each of these resources is owned by a particular user.

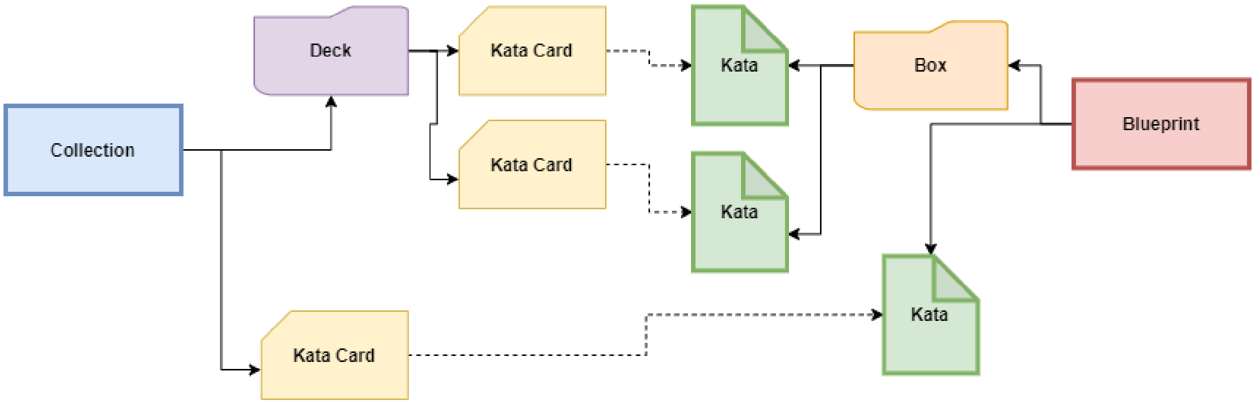


Figure 2

4.2.2 Database Container

This data model needs a database to be implemented, and because it is a relational model, it necessitates a relational DBMS. However, the database schema should not be manually defined through DDL statements. Instead, the data model should be implemented at the application level through an ORM in the Store API, and from this, a set of suitable migration scripts should be derived. Therefore, the DBMS must be one that the ORM supports. It is deemed to be outside the scope of this project to consider scaling or sharding the database.

4.2.3 Store API Container

The Store API is an HTTP JSON API that should implement the data model and provide various API endpoints to interact with the Stores resources. Primarily it should provide CRUD operations for every resource. However, it should also provide other endpoints, such as uploading test files for kata and convenience endpoints that return partial data of various entities.

Before users can call the Store, the Store must know the user exists. To this end, there should be a 'hello' endpoint, which will create a User resource for the user the first time they make an authenticated call to it.

Also, as discussed in the following section, the Store will sometimes need to forward requests to the solution testing API when a user does not have the authorization to retrieve the unit tests themselves. Further details about API authentication are discussed in section 4.5.

For easy consumption of the API by other systems in the platform and by external consumers, the Store API should also provide an OpenAPI 3 specification.

4.3 Practice system

For a graphical interpretation of this section, refer to Section 13.1.3

The practice system has three primary containers. The web application is the client (O4) through which learners practice their kata. It is also where spaced repetition is implemented (O5). The Web-App calls the Store to retrieve details about kata and save learning data. To test solutions, the Web-App sends requests to the Solution Tester. The Solution Tester provides an API that validates a solution for a kata (O3) by running the katas unit tests over the solution. As the test cases of a kata may need to be hidden from a learner while practising, the web application sends its requests to validate solutions via the Store. The store then retrieves the relevant tests and forwards the request to the solution tester before returning the results to the web application. The Solution Sandbox, discussed in further detail in section 4.5, acts as a proxy for requests headed to the solution tester and redirects them to sandboxed instances of the tester (O12).

4.3.1 Spaced Repetition Algorithm

The web-app should implement the SM-2 spaced repetition algorithm.

4.3.2 Web-App Container

The Web-App should be a PWA which will serve as the client Learners use to practice their Kata. This section describes the main activities a learner should be able to carry out within the Web-App. All these activities will begin after the web-app has been initialized and the user authenticated.

4.3.2.1 Add kata to deck

The Learner should be able to browse a list of published Blueprints. From there, they should be able to see any published Boxes or Katas the Blueprints contains. When they select a kata in a Blueprint or Box, they should be able to create a card for the kata in one of their Decks if they do not already have a card for this kata. This will enable them to start practising.

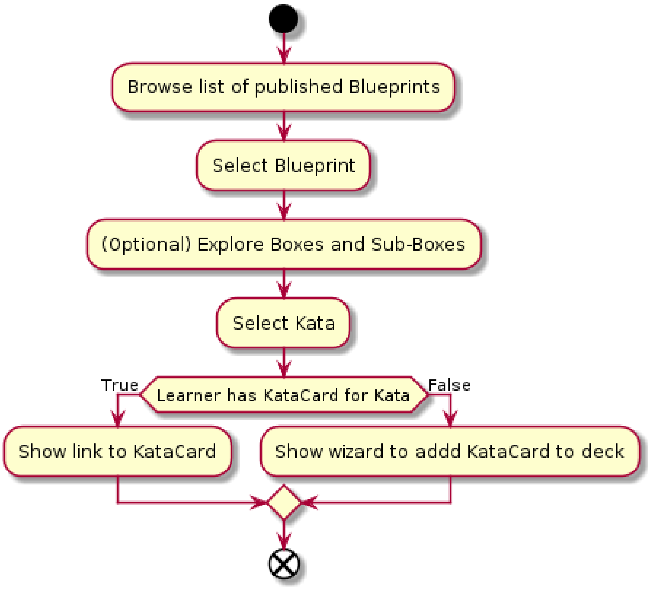

Browse list of published Blueprints
Select Blueprint
(Optional) Explore Boxes and Sub-Boxes
Select Kata
True
Learner has KataCard for Kata
False
Show link to KataCard
Show wizard to addd KataCard to deck

Figure 3

4.3.2.2 Select Katas to Practice

The Learner should be able to practice all katas in a Collection that are due to be practised. Learners should also be able to practice a kata before it is next due manually, without any spaced repetition.

4.3.2.3 Practice

Practising a kata has two stages, solving and scheduling. Users should solve a kata by developing a solution in response to the katas prompt and submitting it to the solution tester until they develop a correct solution. The scheduling phase should determine the next due date for the kata with the SM-2 algorithm.

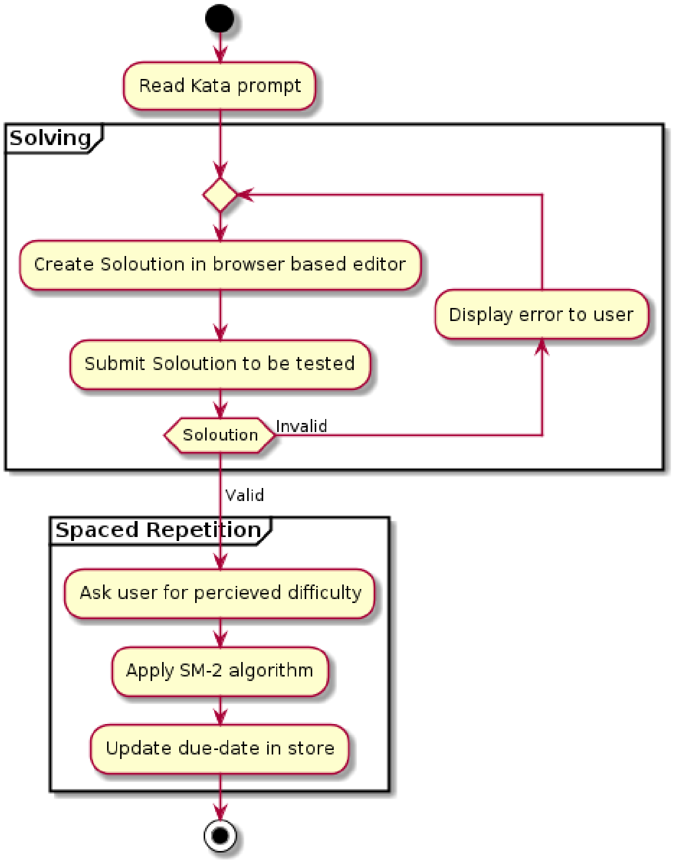


Figure 4

4.3.3 Solution Tester Container

The Solution Tester is an HTTP JSON API that should provide an endpoint that takes a fragment of code and a test file and returns whether the code passes the provided test. As the project is concerned with practical kata covering the python programming language, it is expected the test file will contain a series of python unit tests. The API should implement roughly the following interface.

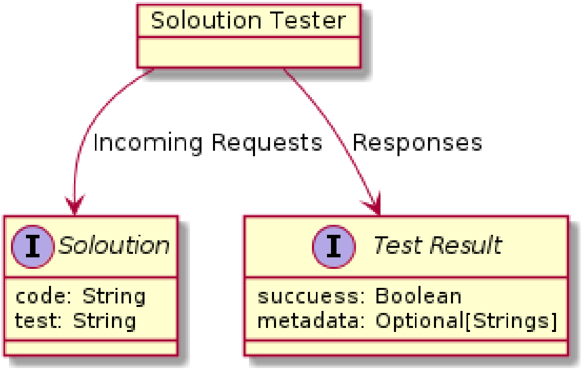


Figure 5

4.4 Authoring system

For a graphical interpretation of this section, refer to Section 13.1.4

The authoring system has two containers, which both work in conjunction to provide a Git-based workflow for Authors to create kata for the platform (O2).

4.4.1 GitOps

GitOps is a term coined by Weaveworks [20]. The key idea is that the state of a system is described by a set of declarative configuration files and that the desired system state is kept at the HEAD of a Git repository. Then when the configuration at the HEAD changes, the system is automatically updated to match the new configuration. This idea has primarily been used to manage Kubernetes and various cloud-native tools. In this project, a GitOps-like methodology is used to provide a workflow for authors to create kata collaboratively.

To recap, the Store supports organizing Katas into tree structures called Blueprints. A Blueprint resource marks the root of a tree while Box resources represent branch nodes and Katas represent Leaf nodes. A set of declarative configuration files to describe this tree structure can be defined within a Git repository. Each resource is represented by a directory in a directory-tree structure. The configuration for each resource is then provided by a configuration file within that directory. When a modified version of the directory tree is pushed to the HEAD of the Git repository, the Blueprint it represents in the Store can be automatically updated.

To denote the type of resource, a directory in a directory-tree represents a configuration file that should be placed within the directory. For example, the following diagram shows how a configuration to define a Blueprint with one child Kata and one child Box containing another Kata could be represented through a directory tree.

As well as marking a directory as a particular type of resource the configuration files should contain all the mutable properties of that resource. The configuration file should be provided in a human-readable format.

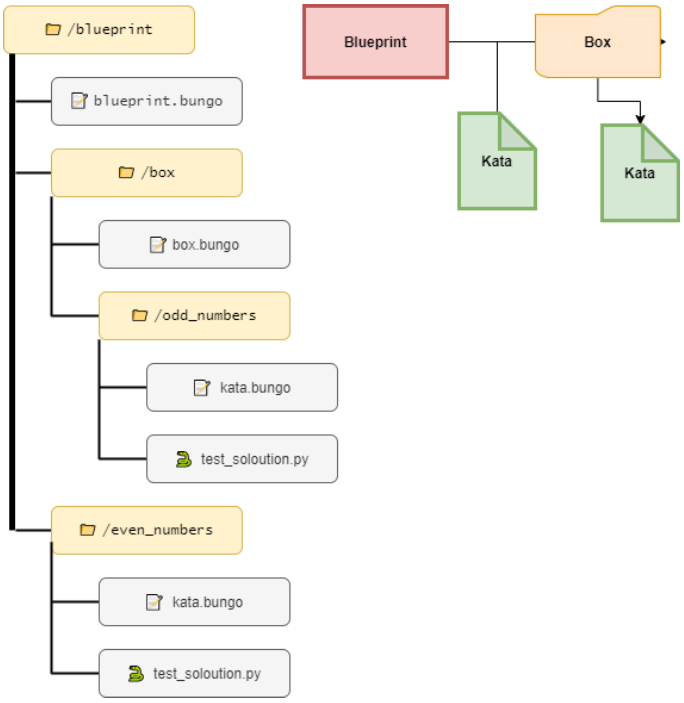


Figure 6

4.4.2 CLI Container

The CLI's primary responsibility is to enable Authors to manage directory-tree configurations representing the desired state of Blueprints on the platform. This involves creating new directory-trees and adding configuration to represent Kata or Boxes resources to them.

To achieve this, the CLI should support four commands: Login, New Blueprint, New Box and New Kata.

The Login command should authenticate the Author with an Instance, preparing the CLI to connect to the Store. The results of authentication should be saved for subsequent commands.

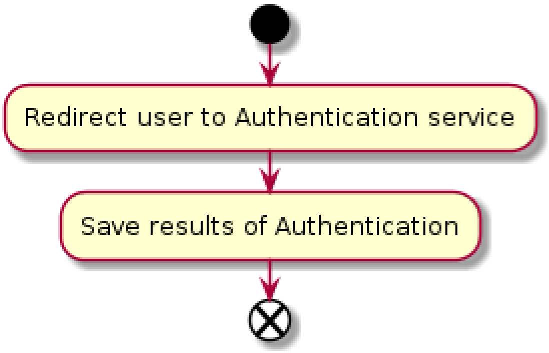

Redirect user to Authentication service
Save results of Authentication

Figure 7

The New Blueprint should check the target directory is not a sub-directory of an existing Blueprint. Then, it should create an unpublished Blueprint in the Store and create a corresponding configuration file in the target directory.

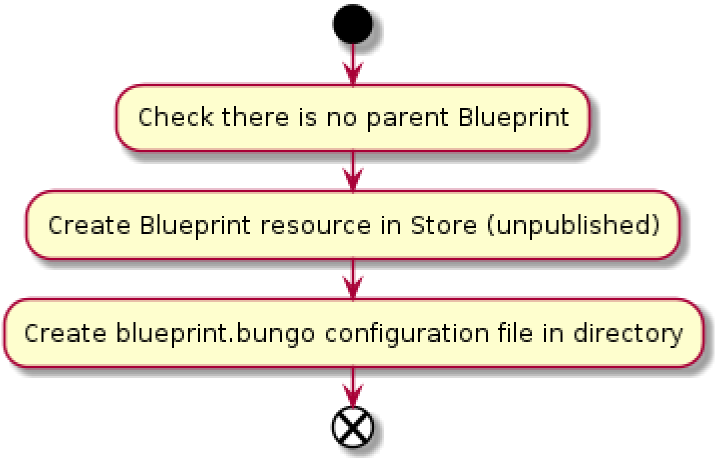

Check there is no parent Blueprint
Create Blueprint resource in Store (unpublished)
Create blueprint.bungo configuration file in directory

Figure 8

The New Box and New Kata commands should check the target directory is a sub-directory of an existing Blueprint. Then, again, it should create an unpublished resource in the store along with the corresponding configuration file.

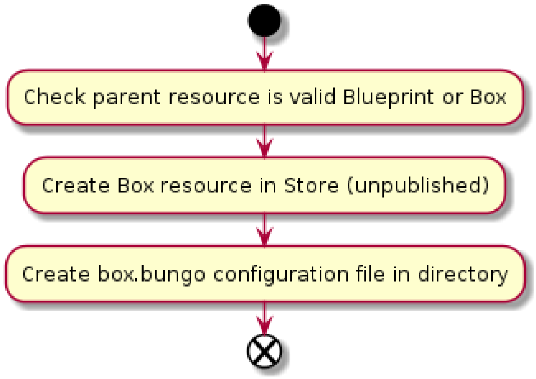

Check parent resource is valid Blueprint or Box
Create Box resource in Store (unpublished)
Create box.bungo configuration file in directory

Figure 10

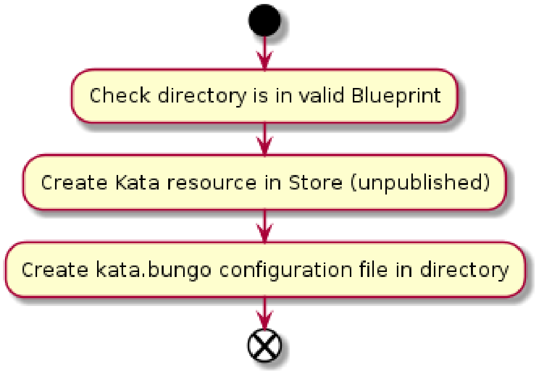

Check directory is in valid Blueprint
Create Kata resource in Store (unpublished)
Create kata.bungo configuration file in directory

Figure 9

4.4.3 Reconciler Container

The reconciler pattern[21] is a way of comparing the current state of a set of resources with the desired state and reconciling the differences. The reconciler container should be an HTTP JSON API that listens for changes to the HEAD of a Git repository and then updates Blueprints in the store to match the state of the directory-tree configuration in the repository. In other words, this service compares the differences between the configuration files for the Blueprint and the state of the Blueprint within the store and updates the Blueprint to match the new configuration.

4.5 Security design

4.5.1 Firecracker

Firecracker[22], created by Amazon Web Services, aims to provide "strong security and minimal overhead" to virtualise serverless workloads on public infrastructure providers. It provides a better security posture than containers but does not have all the performance drawbacks of using a heavier virtualization tool such as QEMU. The solution sandbox should use attempt to use Firecracker to isolate solution testing workloads.

4.5.2 Solution Sandbox Container

The solution sandbox should intercept requests to the solution tester. It should then create a Firecracker microVM, forward the request to the microVM and after returning the response, tear the microVM down. This means that each call to the solution tester will be completely isolated.

4.5.3 Authentication

Authentication in the system should be through a separate third-party identity server or broker. It is anticipated an open-source identity and access management solution such as Keycloak will be used.

5 Implementation

This part of the report outlines and discusses how the design was implemented. For each of the three systems, it focuses on the implementation of each component described within the design section. The discussion of each component's implementation is accompanied by a C4 Model level 3 component diagram, found throughout section 13.2 below. It then discusses additional work that went into realizing the secondary security aim, generating API clients from OpenAPI specifications and the limited extent to which the platform could be deployed.

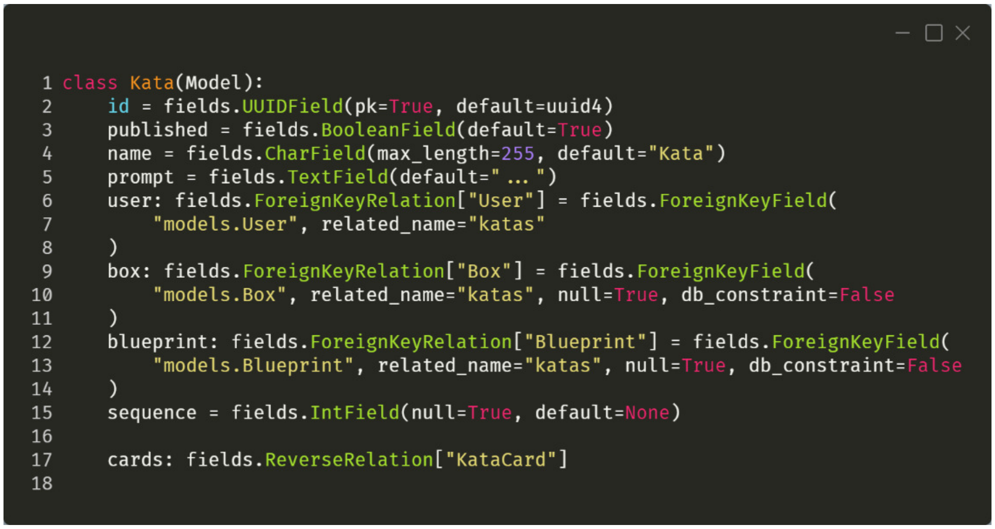
The design section decomposed the platform into systems and components. This was done to outline the platform's architecture and demonstrate which objective each component is concerned with. This implementation section shows how each component was realized at a code level to pursue these objectives. Some of the discussions of components below, where relevant, contain code, snippets of source code, or UML activity diagrams describing how processes were implemented. This part of the report focuses on what was achieved and is not a linear representation of how the work was done.

5.1 Store System

For a graphical interpretation of this section, refer to Section 13.2.1

Tortoise ORM

To implement the data model at an application level, I used Tortoise ORM[23] library. To do so, I defined each entity described in the entity-relationship diagram as a python class which inherited from Tortoise's base-model class. Before the ORM can be used, it must be initialized, which is done when the API starts. This connects Tortoise to the database and handles backwards relations between the model classes. The following code snippet gives an example of how the Kata resource was defined as a Tortoise model. I was now able to manipulate and query the database within python by manipulating instances of these classes and using Tortoise's query API.



```python
class Kata(Model):
    id = fields.UUIDField(pk=True, default=uuid4)
    published = fields.BooleanField(default=True)
    name = fields.CharField(max_length=255, default="Kata")
    prompt = fields.TextField(default=" ... ")
    user: fields.ForeignKeyRelation["User"] = fields.ForeignKeyField(
        "models.User", related_name="katas"
    )
    box: fields.ForeignKeyRelation["Box"] = fields.ForeignKeyField(
        "models.Box", related_name="katas", null=True, db_constraint=False
    )
    blueprint: fields.ForeignKeyRelation["Blueprint"] = fields.ForeignKeyField(
        "models.Blueprint", related_name="katas", null=True, db_constraint=False
    )
    sequence = fields.IntField(null=True, default=None)

    cards: fields.ReverseRelation["KataCard"]

```

Figure 11

FastAPI and Pydantic

With the data model implemented, the next problem was to find a suitable way to expose it as an HTTP JSON API to the Practice and Author systems. I chose the FastAPI framework, which I also used for all the other APIs created throughout the rest of the project.

The next problem was how to serialize an instance of a Tortoise model into JSON. Pydantic[24] is a python serialization library that allows you to use python classes to define and validate schemas. Tortoise provides helpers to automatically generate Pydantic models that match the schema of a Tortoise model class. This allowed me to easily convert instances of Tortoise models for each entity into corresponding Pydantic model instances.

The benefit of this is that FastAPI allows you to define the interface of API endpoints with Pydantic models. This allows API requests and responses to be automatically validated and serialized between JSON and Pydantic model instances. This meant I was able to tightly couple the interface of the API with the implemented data model.

Model-View Pattern

As there were many resources in the Store and several API endpoints to implement for each resource, I organized the API code in the following way to keep things modular and DRY. As well as having a Tortoise model, each resource has a controller and a router. The controller implements all the ORM-specific logic for each resource and generates the appropriate set of Pydantic models to serialize the results of these operations to JSON. The router uses the Pydantic models to define the interface of all the API endpoints for that resource and uses the methods exposed by the controller to implement the functionality of the endpoints.

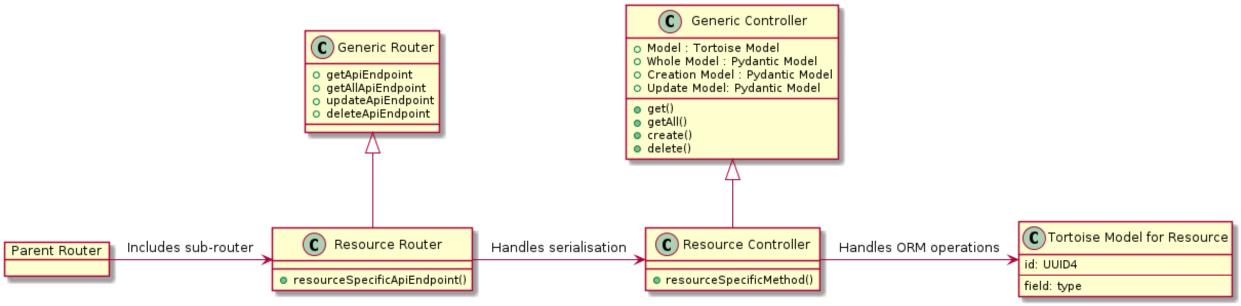


Figure 12

Router nesting

The routers for each resource define API endpoints using relative paths. These routers are then included within parent routers to organize the API into the following tree structure.

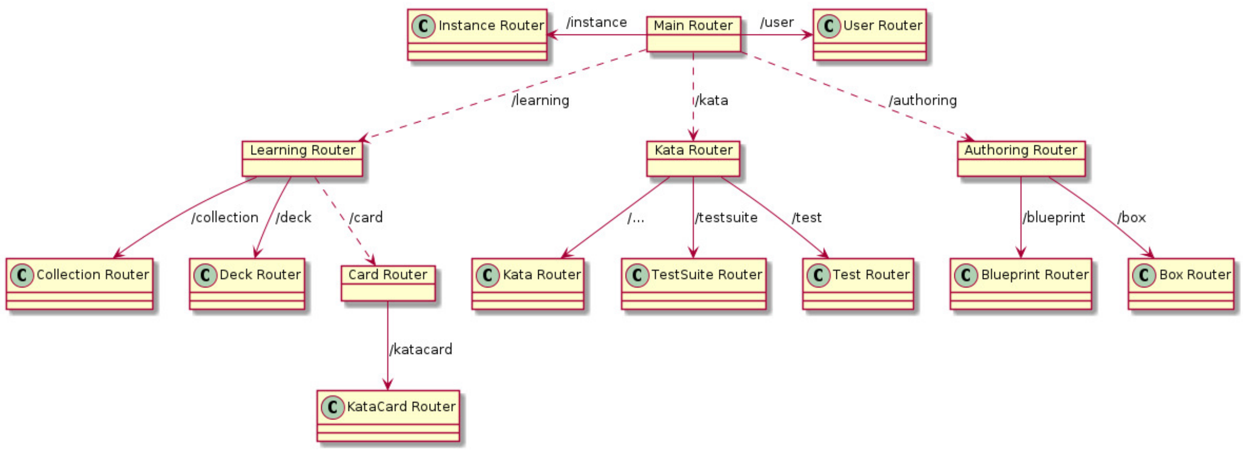


Figure 13

Example API request code-path.

If a consumer of the API requests a particular KataCard by ID, then the request would have the following code path. (This example is a simplification and ignores details such as authentication) Firstly, FastAPI would route the request to the KataCard router by following the route tree described above. The KataCard router, as it implements the GenericRouter, has a `getEndpoint` method registered to the relative path `‘/{id}’,` which matches the request. Now that the request has been matched to an API endpoint, FastAPIc converts the `id` variable in the path to a function argument and call the `getEndpoint` method. The router then asks the controller for the resource with that ID. The controller constructs and executes the Tortoise query, using the query methods defined as class properties on the Tortoise model class. Finally, the router converts the results of the query to Pydantic model instances and returns those. FastAPI then coverts these instances to the JSON body of the response.

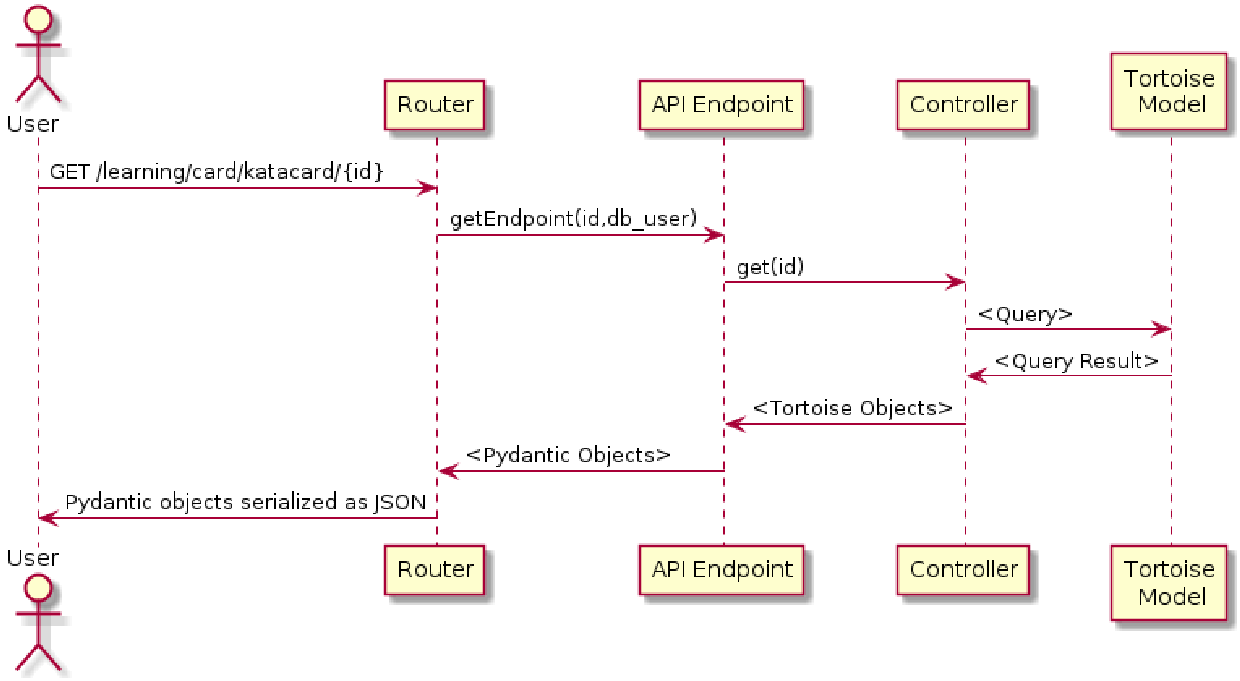
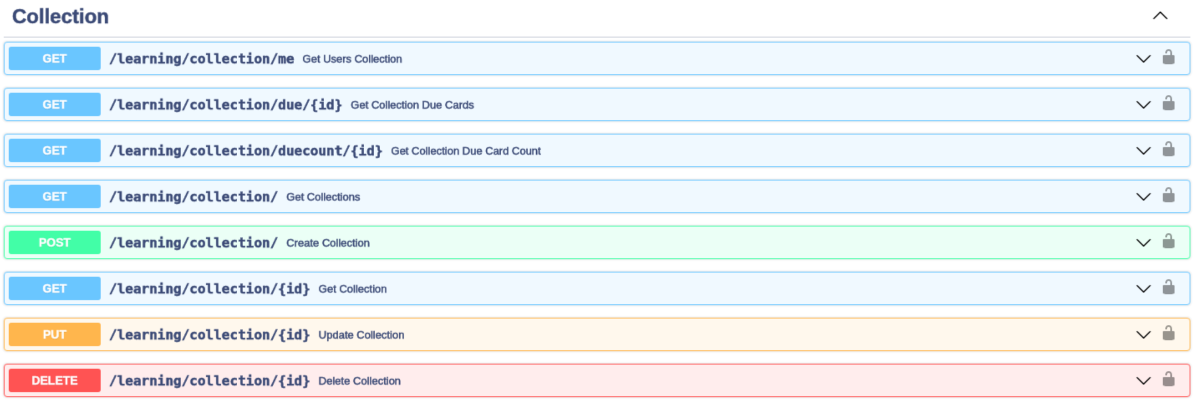


Figure 14

OpenAPI Specification

FastAPI is also able to generate an OpenAPI 3 schema automatically. This allows the Store to be automatically documented and consumed by humans. As discussed later in the report, it is feasible to generate clients to interact with the store automatically. The below is a snippet of the specification viewed through the Swagger Editor, showing the methods available for Collection resources.



Collection

GET **/learning/collection/me** Get Users Collection

GET **/learning/collection/due/{id}** Get Collection Due Cards

GET **/learning/collection/duecount/{id}** Get Collection Due Card Count

GET **/learning/collection/** Get Collections

POST **/learning/collection/** Create Collection

GET **/learning/collection/{id}** Get Collection

PUT **/learning/collection/{id}** Update Collection

DELETE **/learning/collection/{id}** Delete Collection

Figure 15

Database

To manage the database schema, I used aerich[25], a tool that generates database migration files from a set of Tortoise models. I opted to use SQLite during local development and intended to use PostgreSQL for deployments.

5.2 Practice System

For a graphical interpretation of this section, refer to Section 13.2.2

5.2.1 Web-Application

Tooling

I chose to use Svelte[26] as the framework to build the PWA. I used Typescript [27] to write the application and Snowpack[28] to build it. Svelte is a component framework where you compose sets of components to build applications. Typescript brings a type system and types checking to JavaScript. Snowpack is a modern build tool that exploits modern browsers native support for ES modules to provide a fast developer workflow that removes the typical step of bundling a web application.

Frame, routing, and pages.

The web application has two main parts. The first, the core, is a typescript class that encapsulates the application logic and is responsible for interfacing with other systems and (C4) components. The Svelte application handles presentation, providing the application through a series of Page components implemented using a shared set of Layout and UI components.

The web app is bootstrapped by a `Frame` component which displays a loading spinner to the user while the core is initialized. Initialization means creating a singleton instance and ensuring the user is authenticated correctly. Next, the frame renders the main `App` component, passing the core as a prop. The App component uses the Tinro[29] routing library to define the web app's routes and, based on the route, loads the corresponding Page component.

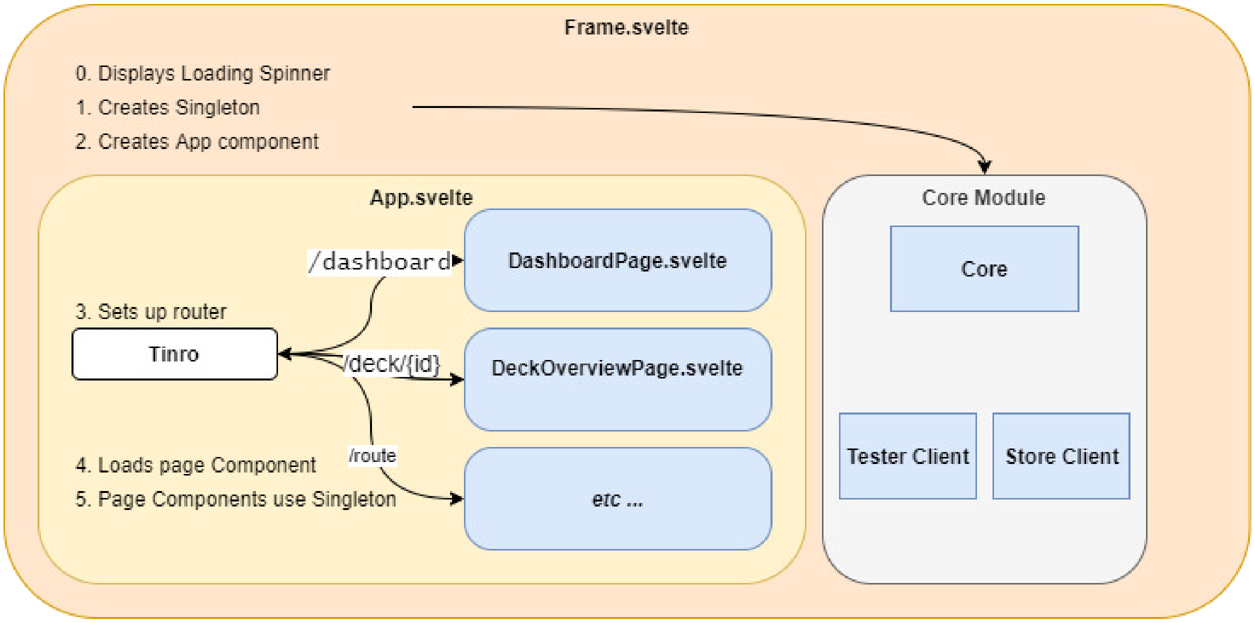


Figure 16

Pages

The web application is laid out across the following set of pages at the following routes.

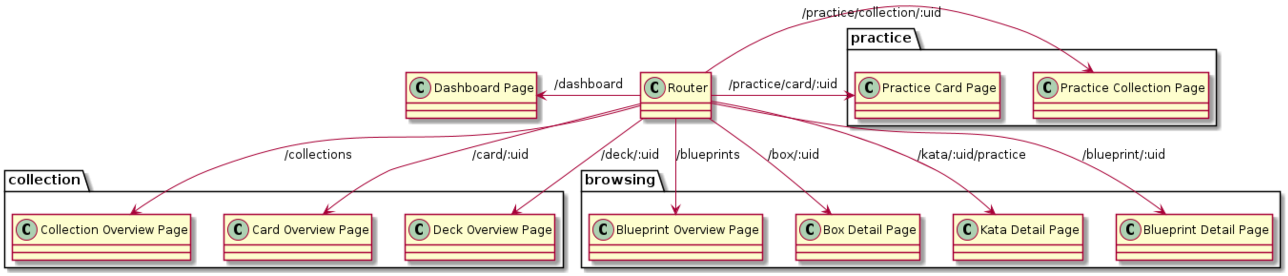


Figure 17

If the web app is opened at a particular URL, it starts serving that page, provided the user is authenticated correctly. This means that users would be able to bookmark pages if they desired.

Shared Components

While each page has its specific components to implement its functionality, a set of shared UI components is used to keep the application consistent between pages.

This includes a `Layout` component for consistent page layouts, a `Transition` component to animate between pages and simpler components such as a `Button` component.

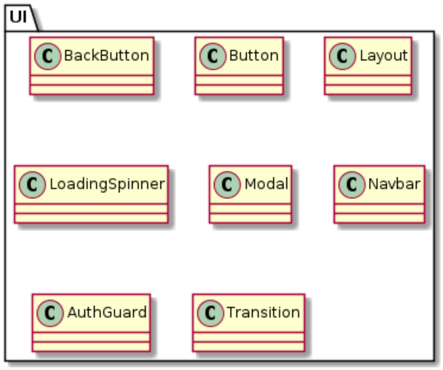
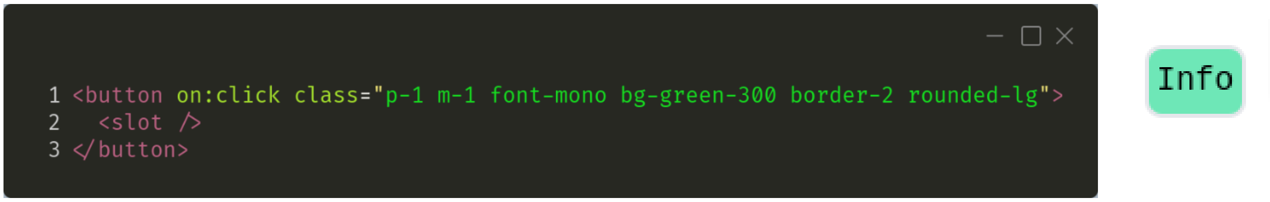


Figure 18

Component styling

Components are styled with Tailwind[30]. This is a utility-first CSS framework that enables components styling to remain part of their HTML. This is done by providing many utility classes that can be used to declaratively describe a component's style, as opposed to having to create separate CSS classes. The following code snippet demonstrates how the Button component is styled using Tailwind. It can be read as saying that the button object should “have one pixel of padding, one pixel of margin, a green background, a two-pixel wider border and a large, rounded edge”.



```
<button on:click class="p-1 m-1 font-mono bg-green-300 border-2 rounded-lg">
  <slot />
</button>
```

Figure 19

Monaco

To provide a code editor for Learners to create their Solutions, I used the Monaco[31] editor. This is the open-source editor produced by Microsoft for the VS Code IDE. Monaco is loaded within a Svelte component and configured to provide syntax highlighting for python. I attempted to connect Monaco to a python language server, which would have provided features such as code suggestions and linting via Typefox's monaco-languageclient[32]. Unfortunately, I could not get this to work and deemed it to be out of the project's scope.

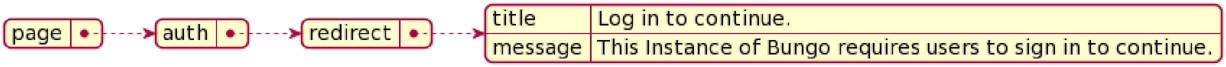
Spaced-Repetition

The SM2 algorithm was implemented using the supermemo NPM package[33], which provides a Typescript implementation of SM2. Once the web application receives confirmation back from the solution testing API that the Learners solution is correct, the Learner is prompted to rate the perceived difficulty of the kata. This number between zero and five, the number of repetitions, the interval in days and the current ease factor is passed to the SM2 algorithm to determine the next due date and new ease value. These values are then updated in the store.

Internationalization

If the platform is to be used in the author's region of study, Wales, it must provide Welsh language support. To this end, the svelte-i18n[34] internationalization library was used. Only an English locale was defined, but the author hopes to add more languages in the future.

A locale file is a JSON object that contains message definitions for a language. For example, the segment of the locale file that defines the messages shown to the user on the authorization redirect page looks like this.

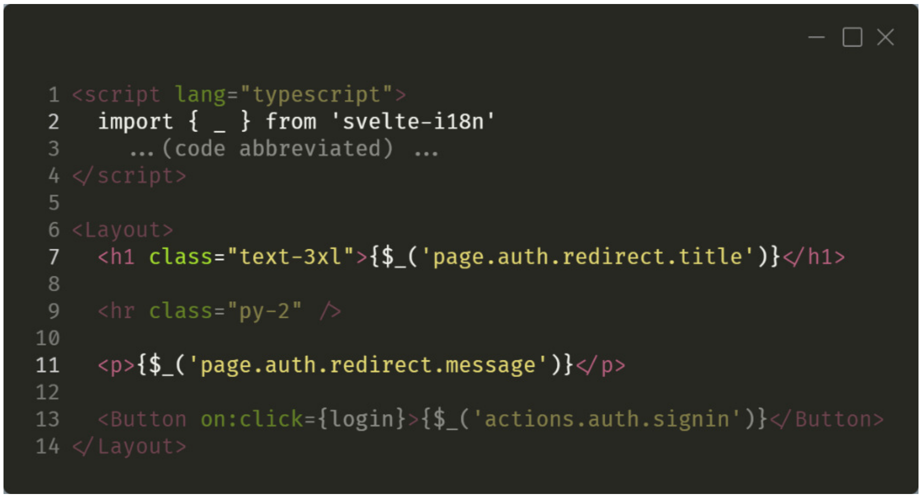


page → auth → redirect →

title	Log in to continue.
message	This Instance of Bungo requires users to sign in to continue.

Figure 20

Then, in the relevant Svelte component, the correct message is selected by passing the message's key to the internationalization function.



```
<script lang="typescript">
  import { _ } from 'svelte-i18n'
    ...(code abbreviated) ...
</script>

<Layout>
  <h1 class="text-3xl">{$_('page.auth.redirect.title')}</h1>

  <hr class="py-2" />

  <p>{$_('page.auth.redirect.message')}</p>

  <Button on:click={login}>{$_('actions.auth.signin')}</Button>
</Layout>
```

Figure 21

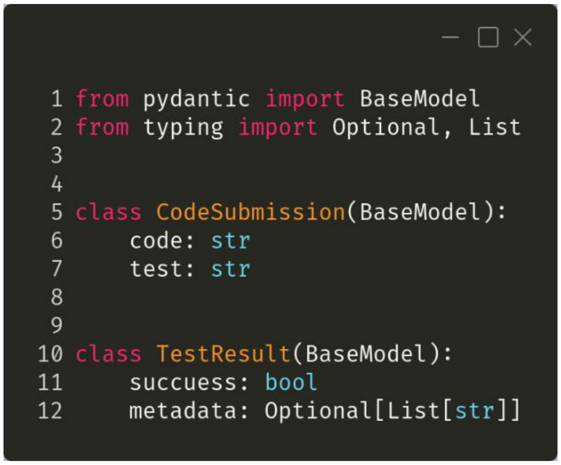
The i18n-ally[35] Vs Code extension made adding internationalization seamless and appears to provide a solid workflow to aggregate translations in future work.

5.2.2 Solution Tester

As the project was focused on practical kata for general python language features, to assess solutions automatically, the test cases had to be provided in the form of python unit tests. While Python has built-in unit testing support, I decided to support unit tests written in the PyTest[36] framework. PyTest was chosen because it allows for more concise testing code, has a rich plugin ecosystem, and can support default python unit tests and nose framework tests.

To implement the Solution Testing API, I again used the FastAPI framework. The following code snippet shows how the interface given in the specification was easily implemented using Pydantic models. The metadata field was used to return details about any exceptions thrown during the test suite's execution.

PyTest begins by running through a collection phase, in which it identifies test files and source code files in the directory from which it is invoked. Therefore, the contents of the Code Submission object had to be written to a temporary directory on disk, which is deleted after PyTest finishes executing. I attempted to find a way to get PyTest to collect a series of file-like objects to avoid this but could not find a way to do so.



```
from pydantic import BaseModel
from typing import Optional, List


class CodeSubmission(BaseModel):
    code: str
    test: str


class TestResult(BaseModel):
    succuess: bool
    metadata: Optional[List[str]]
```

Figure 22

When a user executes PyTest from the command line, they see a rich summary of the test suites execution, such as which tests passed and failed and details of any exceptions for failed tests. However, the method to invoke an in-process PyTest run only returns an integer representing the status code of the PyTest process if PyTest had been run from the command line. This means I could only determine if the test suite could run, not if the test suite passed or failed. PyTest is powered by a plugin framework called Pluggy[37]. Pluggy allows plugin authors to listen to events within PyTest or override functionality. Much of PyTest's core functionality is implemented as a series of plugins. For example, the rich summary users see is provided by the `TerminalWriter` plugin.

I created a custom ReportProblem plugin to solve this problem, which enabled the extraction of exception information from failed tests from the in-process PyTest runs. The plugin is used by registering it with PyTest when an in-process run is invoked.

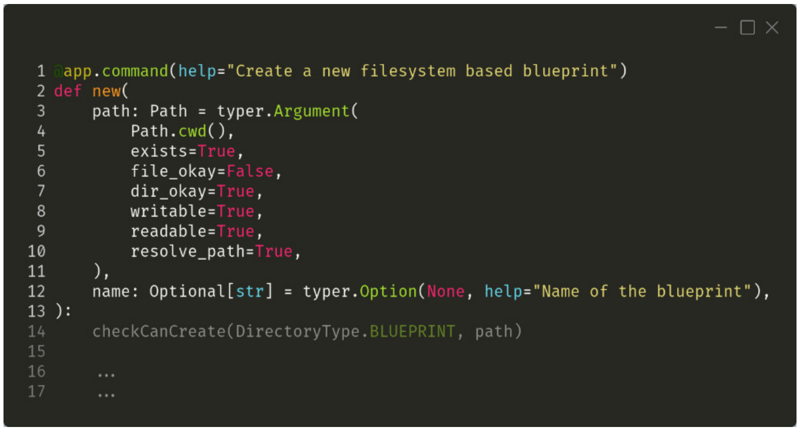
I encountered a bug while building this container. The first request to the service would work, however, subsequent requests would simply return the result of the first request, even though the temporary directory had been deleted and PyTest was executing in a different directory. This appears to have something to do with how PyTest manipulates `sys.modules` to enable tests to import source code correctly, but I could not confirm the exact issue. The solution I found was to execute the plugin-customized PyTest run in a subprocess, using Pythons built-in subprocess library. The subprocess outputs the `TestResult` model to std-out as JSON. The parent process serializes back into a pydantic model used for the response of the API endpoint.

5.3 Authoring System

5.3.1 CLI Container

Tooling

I built the CLI using Typer[38], which allows you to create Click CLIs[39] by defining CLI command as type-hinted python functions. I also used the Rich[40] library to format the CLI output. An example of how type hints were used to define the interface of CLI commands is given below.



```
@app.command(help="Create a new filesystem based blueprint")
def new(
    path: Path = typer.Argument(
        Path.cwd(),
        exists=True,
        file_okay=False,
        dir_okay=True,
        writable=True,
        readable=True,
        resolve_path=True,
    ),
    name: Optional[str] = typer.Option(None, help="Name of the blueprint"),
):
    checkCanCreate(DirectoryType.BLUEPRINT, path)

    ...
    ...
```

Figure 23

In this example, for creating new Blueprints, you can see the command takes two arguments; a positional argument representing the directory in which the Blueprint should be created and an optional name flag which can be used to set the name of the Blueprint as it is initially created.

Configuration

A CLI users' settings, such as their authentication token for a particular instance, are stored in a configuration file that defaults to a `.bungorc` file in their home directory.

Login Process

To log in, Authors use the login command and specify the location of the instance. When the platform is deployed, it is assumed that the instances services are found on specific subdomains. When this is not the case, for example, during local development, additional flags are passed to the CLI to point it towards the store and authentication services. Once the command is launched, users are instructed to navigate to the authentication server, which they can do by visiting a local ephemeral HTTP server that will redirect them there. After a successful login, the user is redirected back to another HTTP server, which captures a code from the query parameter in the redirect. The CLI then exchanges this with the authorization server for an OAuth access token. This access token is then persisted within the user's configuration file, and the user can use other CLI commands. The CLI detects if a user is logged in when they ask for help using the CLI and toggles the visibility of commands accordingly. The CLI also prompts the user to log back in if a subsequent command fails due to their authorization having expired.

CLI Local data model.

The configuration files are represented with Pydantic models, using the pydantic-yaml[41] library. This makes it easy to validate configuration files. It also makes it easy to convert back and forth between YAML files on disk and instances of the Pydantic models. Each configuration file has a metadata section used to store the `UUID` of the resource the configuration file should configure. The other properties in the configuration file represent the mutable properties of the resource the user can alter and expect to be reflected in the store when the local tree is reconciled.

The nodes in the tree structure are also represented with Pydantic models. Pydantic validators are functions that run when an instance of Pydantic models is created. In this case, they are used to ensure that a tree node for a resource can only be created if the path it references towards contains a valid configuration for that type of resource.

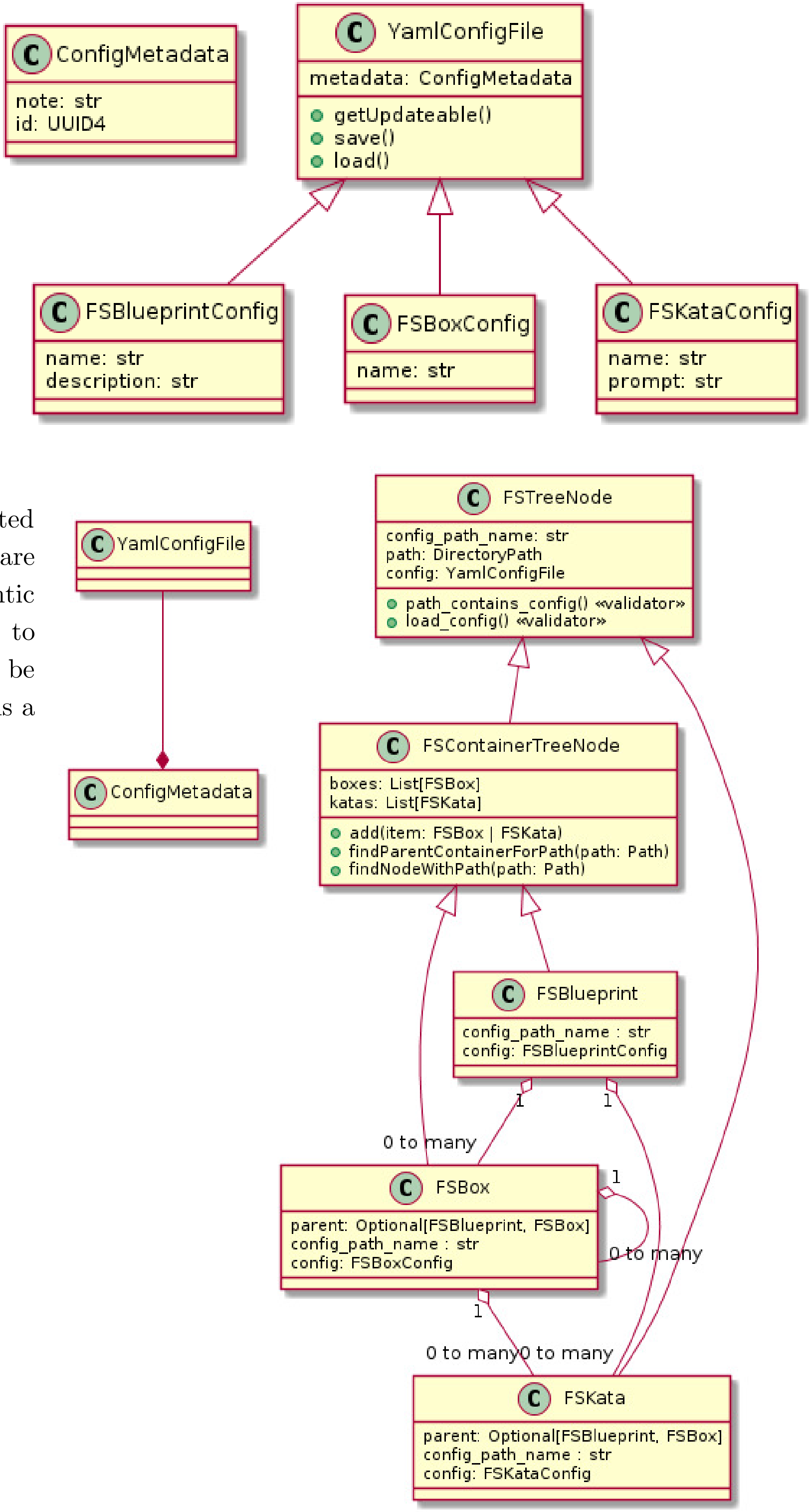


Figure 24

Gathering process

The process of converting a tree of directories and accompanying configuration files to a valid `FSBlueprint` object is described by the diagram on the right.

When we create an `FSBlueprint, FSBox or FSKata object`, because of the Pydantic validators, this is only possible if the directory contains a valid configuration file for that resource. (Note that a valid configuration does not necessarily mean the file represents a resource that still exists in the Store).

Then, we walk through the directory tree using `os.walk`.

The directory type is determined by the presence of the correct type of configuration file in the directory.

The *gather head* refers to the last collected node. When the walk switches to a different branch of the directory tree, this is recalculated by finding the common parent between the last gather head and the new directory.

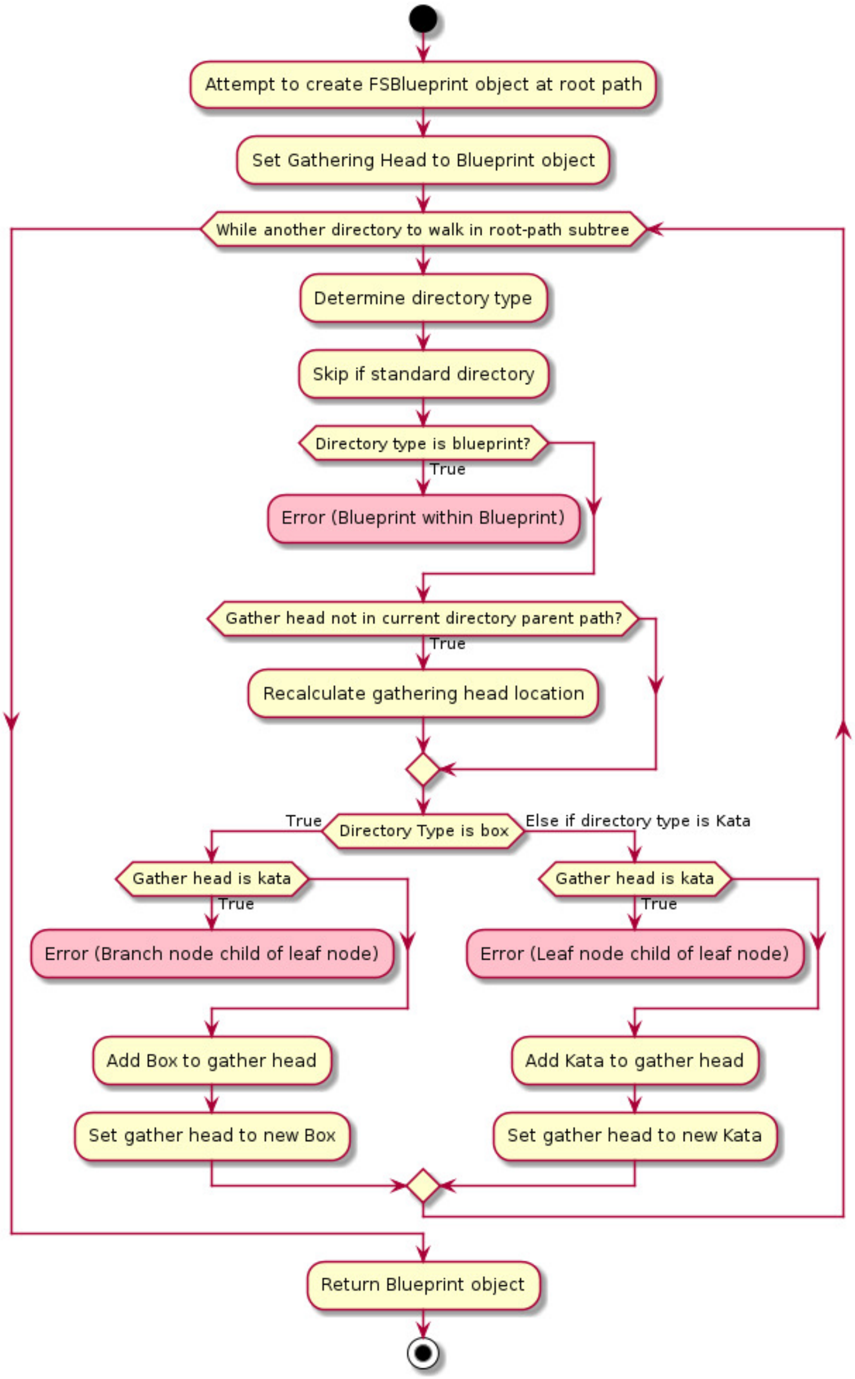

Attempt to create FSBlueprint object at root path
Set Gathering Head to Blueprint object
While another directory to walk in root-path subtree
Determine directory type
Skip if standard directory
Directory type is blueprint?
True
Error (Blueprint within Blueprint)
Gather head not in current directory parent path?
True
Recalculate gathering head location
True
Directory Type is box
Else if directory type is Kata
Gather head is kata
True
Error (Branch node child of leaf node)
Gather head is kata
True
Error (Leaf node child of leaf node)
Add Box to gather head
Set gather head to new Box
Add Kata to gather head
Set gather head to new Kata
Return Blueprint object

Figure 25

5.3.2 Gitlab

I decided to self-host a GitLab instance to further demonstrate that the whole project could be self-hosted independently. I used a docker image of GitLab Community Edition. I attempted to automate the setup as best I could. However, I could not find a way to automate the initial onboarding of the GitLab instance. However, I was able to create and manage repositories using Terraform.

5.3.3 Reconciler Container

The reconciler API begins by registering a web-hook with a GitLab instance. In this case, it registers a webhook with the demo repository that I configured in Terraform. Then, when it receives a call back from GitLab, it clones the repository into a local cache. If it has already cloned the repository, it fetches and checks out the HEAD of the repository.

To clone the repository, when the reconciler service registers its webhook, it creates and registers a deploy key. While I did not have difficulty generating the SSH keys, I did have difficulty getting the version of git within the containerized application to use them. I spent a long time trying to debug the containerized application before I realized what the problem was. Because the keys were new and had not been used, the host they were being used to connect to (GitLab) was not trusted, and so the Git clone command silently failed. The temporary solution was to override the ssh command the git library used to set `StrictHostKeyChecking=no`, but this is not something that would be appropriate once the service is properly deployed.

However, I was really running out of time at this point in the project, so I opted to finish implementing the reconciliation functionality within the CLI instead. This did, unfortunately, mean that the Authoring service did not behave in a GitOps-like way. There was no guarantee that the HEAD of the repository and the store would stay in sync. However, the reconciliation pattern was still successfully implemented, and the Authors can update a Blueprint in the store to match its local representation on disk through the CLI.

Structured logging

In between successfully cloning the library and deciding to finish the reconciliation functionality within the CLI, I decided to implement structured logging into the reconciliation service. As this service would automatically affect the store's contents and have access to various Git repositories, I decided it would be essential to have good logging for the service in place to deploy it with confidence. I used the Structlog[42] python library for structured logging. I implemented an environment variable switch that would cause the service to produce human-readable logs during development and JSON logs once it was deployed. These latter logs would be suitable for being ingested into an ELK stack deployment or other log aggregate tool. I hope to add structured logging to all the FastAPI services in the future.

5.3.4 Reconciliation Process

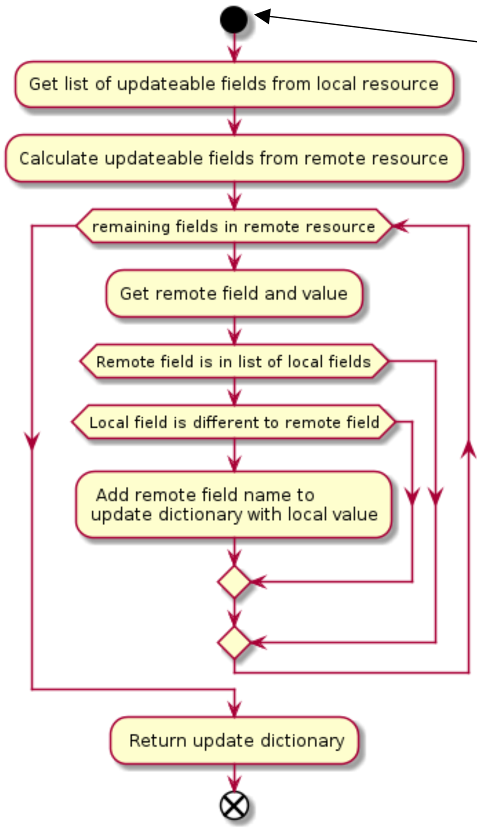

Get list of updateable fields from local resource
Calculate updateable fields from remote resource
remaining fields in remote resource
Get remote field and value
Remote field is in list of local fields
Local field is different to remote field
Add remote field name to
update dictionary with local value
Return update dictionary

Figure 26

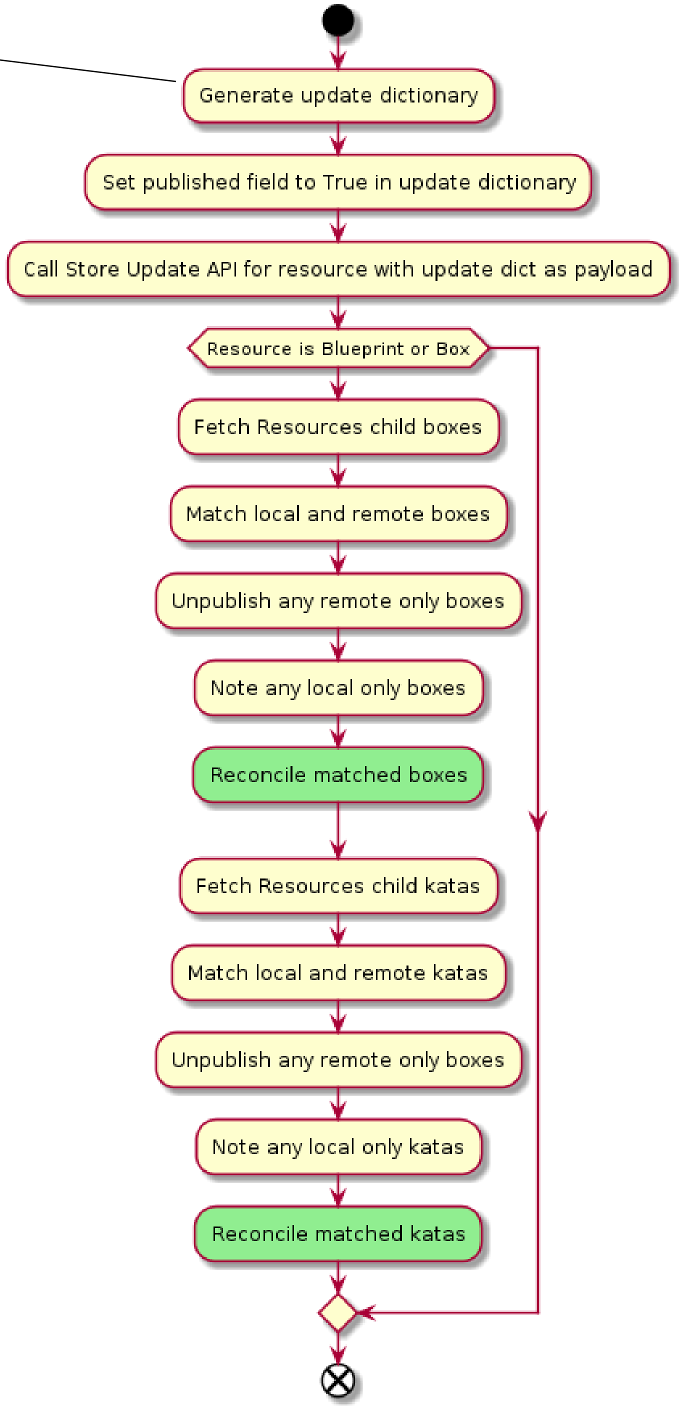

Generate update dictionary
Set published field to True in update dictionary
Call Store Update API for resource with update dict as payload
Resource is Blueprint or Box
Fetch Resources child boxes
Match local and remote boxes
Unpublish any remote only boxes
Note any local only boxes
Reconcile matched boxes
Fetch Resources child katas
Match local and remote katas
Unpublish any remote only boxes
Note any local only katas
Reconcile matched katas

Figure 27

A resource in the tree is reconciled by fetching the local and remote state of the resource. Then, an updated dictionary is determined by comparing the fields from the local and remote resource that can be reconciled (mutable fields). Then the resource is updated within the store.

If the resource is a Blueprint or Box, it may contain zero to many boxes or katas. So, for each type of resource, the following process happens. All the containers remote resources are fetched from the store. They are then matched to the local resources and reconciled.

If an unmatched resource is remote only, it does not exist in the HEAD of the Git repository anymore and therefore should be unpublished. It should not be deleted because it may be in development in another Git branch and may have other resources linking to the resource. Local only configuration files should only occur if the resource no longer exists in the store. Further work is needed to address these two edge cases more effectively.

5.4 Security secondary aim

5.4.1 Keycloak

The platform uses Keycloak[43] as its authorization server. To deploy Keycloak, I used a community provided docker image. I created a realm and two Open ID Connect clients, one for the Web-App and one for the Store. I automated this process with the Terraform and wrote a shell script to provision the Terraform client automatically. This enables the entire authorization server to be automatically provisioned when the platform is deployed.

To implement Keycloak into Svelte, I followed the work of Matěj Bucek.[44]. To implement Keycloak into the store API, I used the python-keycloak[45] library alongside FastAPIs built-in Oauth2 handling.

5.4.2 Solution Sandbox

My initial plan was to follow Julia Evans's work [46] and create an API that intercepts requests headed for the solution sandbox and redirects those requests to individual microVMs. However, to use the firecracker SDK, I would have had to learn how to use Go. After my experience with trying new programming languages and technology earlier in the project, which is discussed in section 6, I was very reticent about trying to learn another new programming language.

Therefore, I decided to use Ignite[47] from Weaveworks. Ignite can run in a daemon mode where it watches a folder for configuration files. When a new configuration file is added or changed, it automatically creates a microVM to match the configuration. This allowed me to create the solution sandbox as another FastAPI API which would behave according to the below activity diagram.

I was able to successfully implement a service that was able to use Ignite to create new MicroVMs. However, in the remaining time I worked on the solution sandbox, I could not correctly forward network requests to the MicroVM. This was because Ignite provides two networking options for MicroVMs. The first is a docker bridge plugin, which would have made it trivial to connect to the MicroVM from the solution tester, which would be running as a container on that bridge network. However, this was an unacceptable security risk as it would give the sandboxed solution full network access to the platform and the wider internet. The second solution was to provide the microVM with a CNI[48], which I attempted, but I did not know enough about Linux networking to bridge the CNI network and the docker bridge network properly, nor how to properly clear up the network interfaces and Iptables rules after everything was done.

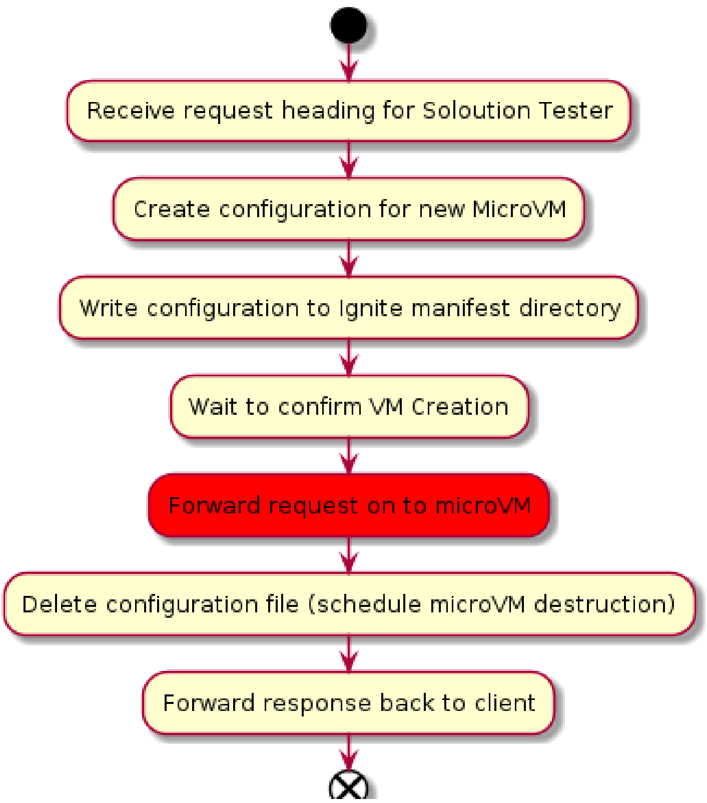

Receive request heading for Soloution Tester
Create configuration for new MicroVM
Write configuration to Ignite manifest directory
Wait to confirm VM Creation
Forward request on to microVM
Delete configuration file (schedule microVM destruction)
Forward response back to client

Figure 28

5.5 Store API Clients Work

As discussed later in the report, the evolution of the Stores data model over time caused me many difficulties. This section describes how I overcame some of these problems in the latter part of the project.

Because of the way I used FastAPI in combination with Tortoise while implementing the Store API, FastAPI would automatically generate an OpenAPI 3 specification that described the APIs endpoints and the overall data model of the platform. This specification allows clients to be automatically generated. The benefit of this is that the data model is effectively and accurately represented within the client. This, combined with type checking, allowed me to be confident that the Web-App and Authoring systems code bases representations of the data model were tightly coupled with its actual implementation in the Store API.

To generate the Store API client for the web application, I used OpenAPITools typescript-fetch generator[49]. To generate the Store API client for the authoring CLI, I used David Montagues fastapi_client[50]. This had the added benefit of generating pydantic models for the resources used during the reconciliation step.

5.6 Platform deployment.

The systems in the platform are currently deployable as a docker-compose stack. Unfortunately, due to time constraints, I could not deploy the platform off my local machine. This should not be too difficult as each service is containerized. The only real thorn is that AWS Firecracker requires KVM access to function. This restricts the deployment of the Solution Sandbox to bare metal providers or providers that support nested virtualization, such as Digital Ocean. This might have implications if the platform was intended to be deployed on an OpenShift instance.

6 Initial Design and Implementation

This section of the report contains discussion and reflection about the projects' initial design, the work that went in over the first third of the project to implement it before it had to be abandoned, and the implications this had on the project overall.

6.1 Initial design motivation

Before explaining the design and how I attempted to implement it, I would like to explain the background reading that inspired it. Over the Christmas recess, I read the article "Local-first software: You own your Data, in spite of the cloud"[51], which made me consider the impact of software architecture on user experience and the importance of users having ownership of their data. I then read through the unhosted.org[52] website, which advocates for unhosted web applications. An unhosted web application does not send a user's data to a backend server. Instead, it stores it within the browser or the users' own first-party data store. This is not to say an unhosted web application cannot communicate with a backend server. It may use many networked services; however, the focus is on how the users data stays local and within their control.

Throughout the project, I have thought about practising kata through the analogy of practising karate in a dojo. If you quit going to a dojo in the real world, you would leave with the implicit knowledge of performing the skills you had been practising there. Let us assume you were also particularly studious, and you took regular notes about how you had practised these skills and when you expected you to practice them again. You would be free to leave with this explicit knowledge and be able to walk into another dojo and continue your practice uninterrupted. With existing code kata platforms, you are free to take your improved skills out of the platform (the dojo) and apply them. But generally, you are not free to take the data concerned with how you have deliberately practised those skills with you. Spaced repetition aims to maximize the effect of practice over a long time, but it requires long-term access to this learning data. Undergraduate students using this platform to practice might want to practice throughout their course and continue practising throughout their future academic or professional careers. So, the idea of building an unhosted application appealed to me as it seemed to be a way of architecturally baking in this assumption that a learner's data should be kept with them and not within a particular dojo (on a remote backend server out of their direct control).

However, I understood that the platform I would build during this project would require backend services. Firstly for tasks that could not be carried out in a browser, such as validating solutions to python katas. Secondly, to provide a mechanism for authors to create kata and a mechanism for learners to practice them without direct access to the unit tests. While I intended to create a web-application as the main client for Learners, I also appreciated that future work might support other clients such as mobile apps. These would benefit from having the application logic already implemented in a backend service that could then be accessed through an API.

I wanted a way to write the bulk of the application in one code base and have it work entirely on the browser, entirely on the server or partly between the two. After I started reading about Deno, I began to get mischievous ideas. Deno[53] is a runtime for Typescript built by Ryan Dahl to address some of his regrets while building Node.js. While the permission-based sandbox, URL based imports and Go inspired standard library were interesting, the thing that caught my attention was the mirroring of web-platform APIs such as the fetch API for making HTTP requests.

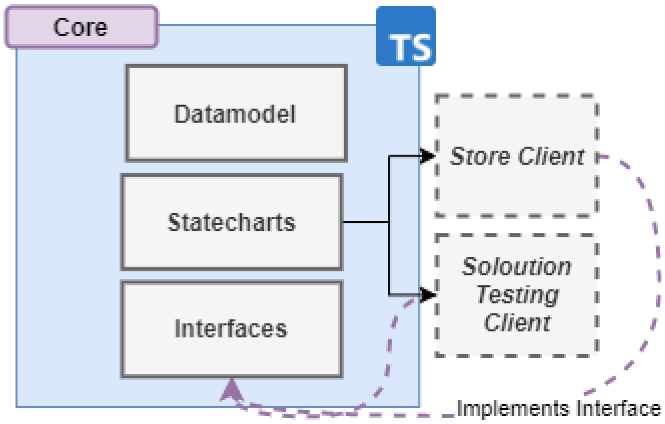


Figure 30

I decided to implement as much of the solution as possible within an Isomorphic[54] Typescript core. A set of Typescript type definitions would represent the platforms data model. Then the platforms 'business' logic would be encapsulated within a set of state machines. These state machines would then use various clients to perform their operations. For example, the state machine for practising kata would utilize a 'Store' client to retrieve data and a 'Test' client responsible for validating solutions. The clients' interface would expose to the state machines would be defined within the core, and the implementation of the clients would then vary depending on the environment in which the core was running. This would allow 'what' the application did to be consistently defined within the core, while 'how' it was done would vary depending on the run time environment and how the application had been configured.

The application could be deployed as an unhosted web application by implementing the core in the following way. The store client would use the browsers IndexDB to handle data, and the solution testing client would make HTTP requests to the solution testing API (which would be written in python). Finally, a presentation layer would be created to create a user interface through which the Learner could view and control the state charts.

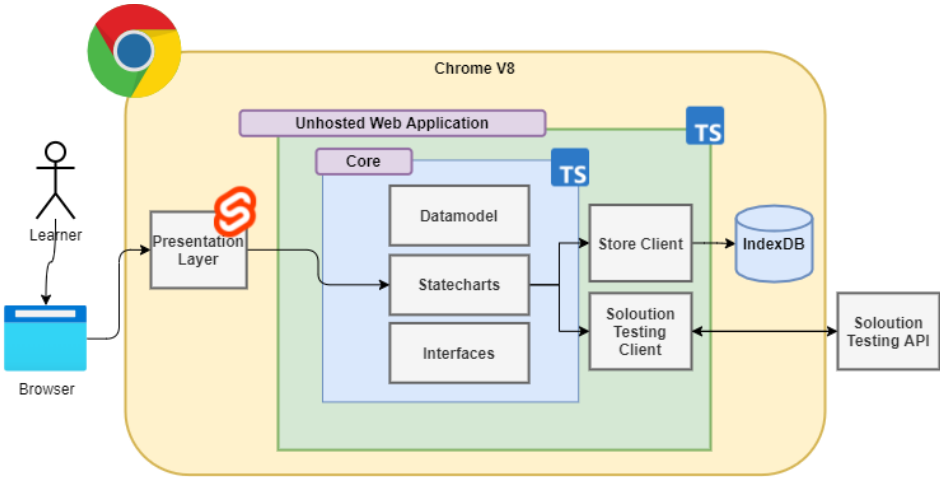


Figure 29

Alternatively, the application could be deployed entirely on the server within Deno. The application would run 'headless' and expose an API that would allow alternative clients such as mobile apps to be integrated. Here a different store client would be implemented to instead communicate with a backend PostgreSQL database. Because the fetch API is the same between both environments, the Solution Testing client would be fungible between the browser and backend.

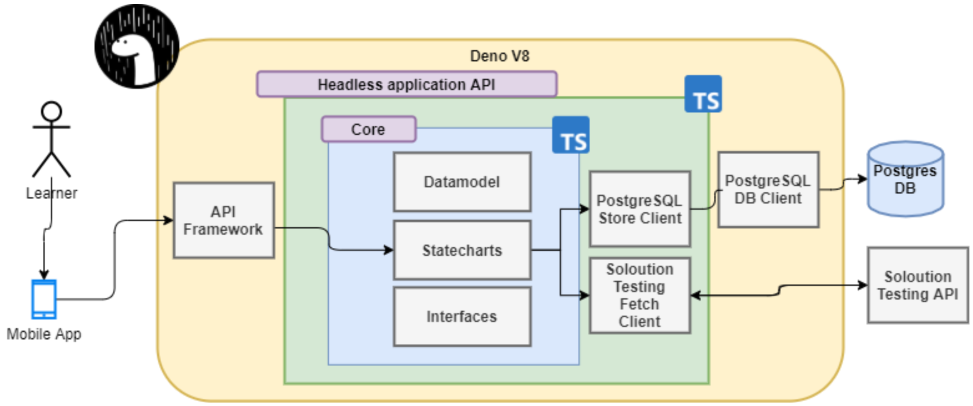


Figure 31

As well as making it possible to quickly deploy the application as a browser-first unhosted web application or as a backend-first headless API, I saw this core pattern as having value for what I needed to create first: an application with a more traditional client-server model.

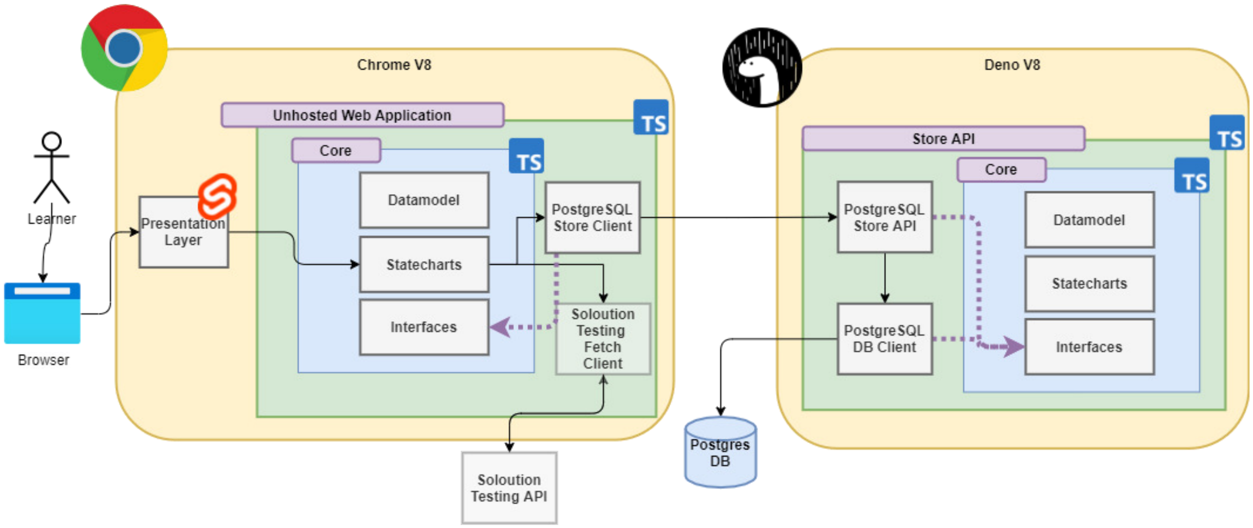


Figure 32

If the clients' data were stored on a remote database, the Store API that made this data accessible would also implement the core. As the same Interface and Data model definitions would be used across each codebase, it would be possible to type check across the network boundary. It would also be possible to perform runtime schema validation on requests and responses using schemas derived from the type definitions.

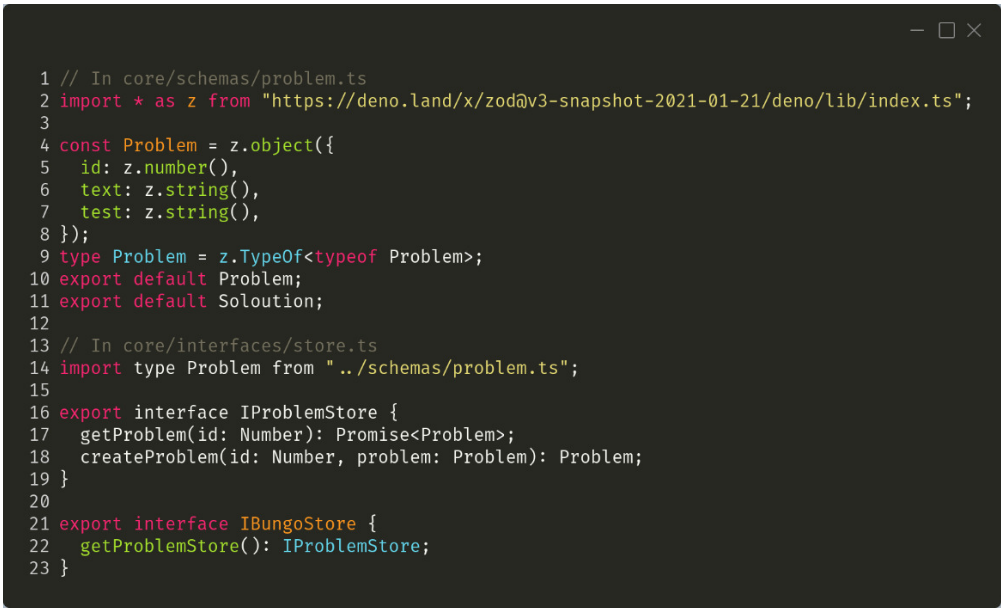
So, not only would this approach make it much easier to deploy the application to alternative clients and platforms (which was one of my secondary aims for later in the project), it appeared it would also allow me to create a more resilient code base earlier on in the project. I also saw this approach as providing a good base for the project to be extended as required by end-users in the future. For example, if the project was deployed, but an alternative database was required, then they would only have to create a client that conformed to the Store interface to use the database.

I saw this approach as building a jigsaw, the centre piece would remain the same, and the connecting pieces would change as required.

6.2 Attempted implementation

I spent the first four weeks of the project attempting to implement this idea of an isomorphic Typescript core.

I began by creating the core as a module in Deno. Using Zod[55], a Typescript schema validation library like Pydantic, I created the first iteration of the data model and defined a schema for a kata object (which at the time I called a problem). Then within the core, I defined an interface for how a Store should be implemented.



```
// In core/schemas/problem.ts
import * as z from "https://deno.land/x/zod@v3-snapshot-2021-01-21/deno/lib/index.ts";

const Problem = z.object({
  id: z.number(),
  text: z.string(),
  test: z.string(),
});
type Problem = z.TypeOf<typeof Problem>;
export default Problem;
export default Soloution;

// In core/interfaces/store.ts
import type Problem from "../schemas/problem.ts";

export interface IProblemStore {
  getProblem(id: Number): Promise<Problem>;
  createProblem(id: Number, problem: Problem): Problem;
}

export interface IBungoStore {
  getProblemStore(): IProblemStore;
}
```

Figure 33

The `IBungoStore` interface that the state machines would expect the store client to implement when they run.

I then began creating the first iteration of the Store API in Deno as two modules. The first module, the ‘Postgres DB Client’, contained all the code that queried the Postgres database, using a popular Postgres database client library. The second module, the ‘Postgres Store API’, used the oak[56] framework to expose the client through an HTTP API endpoint. My intention was for the modules to use the Problem schema in their function headers.

My next goal was to create the ‘Postgres Store Client’ within the Web-App that would implement the `IBungoStore` interface. This would use the fetch API to make a request to the API module, which would use the DB Client module to interact with the database. Because all three would be sharing type definitions from the core, I could be sure that everything would remain type-safe even though data was crossing network boundaries. The Zod library would throw an exception if the data passing through did not match the type definitions.

6.3 Abandoning design

. At the end of the fourth week, disaster struck as I attempted to bring the core into the browser. When first reading about Deno, I learned that it provides a bundle function that converts a Deno module and all its dependencies into a standalone ES Module. This was the mechanism I intended to use to bring the core into the web application. However, I discovered there was no way to bundle Deno code to include type definitions in a form that the browser could consume.

Without a way to export the type definitions, there is no way to export the interfaces. I would be able to import the core into the web application, but instead of it being a well-defined puzzle piece that the rest of the application would clip onto, it would just be a big blob.

Until the ability to generate types for bundles is implemented and resolves issue #3385[57], then the idea of having an Isomorphic Typescript core shared between Deno and the browser is sadly untenable.

6.4 Continuing work

At this point, I was at the end of the fourth week, and I had made significantly less progress than I had intended to towards the project's core aims. I decided to abandon using Deno entirely, opting instead to implement all backend services using FastAPI. I opted to keep the use of Typescript within the web application. The design then changed to meet what is described in the final design section.

I also changed my architectural aims. I previously wanted to create an unhosted application, using the core to relegate only the most crucial functionality to a backend server. However, my approach now changed, and I intended to create a self-hosted platform instead. If backend servers were going to be involved, as they would be with the decision to use FastAPI more extensively, I wanted administrators to deploy an instance of the platform without any dependence on third-party SaaS providers.

7 Results.

This part of the report considers the extent to which the solution developed during the project meets the project's aims. It considers the extent to which the aims from the specifications worked on during the project were achieved. For each aim's objectives, a decision is made about whether the objective was met. Secondly, it considers the extent to which the overall solution addressed the overall project aim.

Due to time constraints, the project did not go through a process of acceptance or user testing. Therefore, the evaluation of whether an objective has been met relies on the author's subjective interpretation.

7.1 CA 1 – Kata creation

CA 1	Kata creation	Enable Creators to author a kata.
Objectives		
O1	Determine a serialization format for a kata, including its prompt and test cases.	
Met: Yes	• Kata and their test cases are modelled as resources within the Store API • It is possible to perform CRUD operations on Kata and their test cases • Therefore, I consider this objective to have been met.	
O2	Demonstrate an example workflow for creating katas in this format.	
Met: Yes	• It is possible to define Kata, their test cases, and the Blueprint structures in which they are organized as set of local configuration files. • It is possible to reconcile the state of a Blueprint in the store with the contents of these local configuration files. • The Store API means that another workflow could easily be implemented to create katas. • Therefore, I consider this objective to have been met.	
O2b	Extend this workflow to be more effective and accessible. *(Optional)*	
Met: No	• I was unable to implement the reconciliation service fully and so I do not think I can argue it has been extended to be more effective. • I did not find time to properly document the CLI and its usage and therefore I do not think I can argue it has been made accessible. • Therefore, I do not consider this optional objective to have been met.	

7.2 CA 2 – Kata practice

CA 2	Kata practice	Enable Learners to practice a kata.
Objectives		
O3	Create a mechanism to validate a Learner's solution to a kata.	
Met: Yes	• Kata were scoped to be concerned with general python programming skills. • The PyTest-rest service is successfully able to validate a Learners solution to these kata provided a set of a file containing unit tests written in the PyTest testing framework. • Therefore, I consider this objective to have been met.	
O4	Create a client for Learners to practice through.	
Met: Yes	• The web-application provides a client through which Learners can ○ Practice Kata ○ Browse and add new Kata to their collections	

	• Therefore, I consider this objective to have been met.
O4b	Provide an alternative interface for Learners to practice through.
Met: No	• Although initial design work was done to anticipate alternative clients and interfaces, none were implemented during the project. • Therefore, I do not consider this objective to have been met.

7.3 CA 3 – Spaced repetition

CA 3	Spaced Repetition	Implement spaced repetition for katas.
Objectives		
O5	Implement a well-known spaced repetition algorithm.	
Met: Yes	• The SuperMemo-2 Algorithm is used within the Web-App to determine the next time a kata should be practiced. • This means that the Web-App provides spaced-repetition based practice. • Therefore, I consider this objective to have been met.	
O5b	Enable Learners to specify their own spaced repetition algorithm. (*Optional)*	
Met: No	• There is no way for Learners to specify their own spaced repetition algorithm. • There is no way for Learners to customize the already implemented algorithm. • Therefore, I do not consider this objective to have been met	

7.4 CA 4 – Store Data

CA 4	Store Data	Create a data-store for platform data.
Objectives		
O6	Design an appropriate data model.	
Met: Yes	• The data model implemented in the Store API allowed the other systems to perform their functionality adequately and was therefore appropriate for the scope of the project. • Therefore, I consider this objective to have been met.	
O7	Implement this data model using a DBMS technology.	
Met: Yes	• Although the data model is modelled at the application level within the Store API, it still uses a DBMS to implement and store the data. • Therefore, I consider this objective to have been met.	
O8	Provide a service to access this data store.	
Met: Yes	• The Store API provides a HTTP JSON API with an accompanying OpenAPI 3 specification • The Store API provides API endpoints to perform CRUD operations on all the resources in the data model. • Therefore, I consider this objective to have been met.	

7.5 SA 1 – Security

SA 1	Security	Ensure the platform has a robust security model.
Objectives		
O9	Document the platforms security boundaries.	
O10	Document the platform's actors and levels of trust between them.	

Met: No	• Neither of these tasks were completed during the allotted time during the development stage of the project • While the Author considered rushing through them for the sake of a better-looking report, they are important pieces of work if the platform is to be deployed in a context where it will handle real user's data. • So, this work will be scheduled as a piece of future work. • Therefore, I do not consider these objectives to have been met.
O11	Implement appropriate authentication for the platform's infrastructure and services.
Met: Yes	• All three systems successfully implemented with the Keycloak authorization service. • The Store API restricts operations on resources owned by other users • The Web-Application and CLI allow users to log in. • Therefore, I consider this objective to have been met.
O12	Implement appropriate sandboxing mechanisms to safely test kata solutions.
Met: No	• A considerable effort was made to use AWS Firecracker to isolate solution testing workloads • Although I was able to create microVMs through Weaveworks Ignite, I was not able to successfully add networking to the microVM. • Further work is required to complete the solution sandbox. • Therefore, I do not consider this objective to have been met.

8 Evaluation of solution

This part of the report is concerned with evaluating each of the three systems and the overall platform. Evaluation is carried out using De Bonos six thinking hats.

8.1 De Bonos six thinking hats explained

De Bonos six thinking hats [58], [59] is a technique to encourage lateral thinking throughout the evaluation process. There are Six metaphorical hats, which, when 'worn', encourage different approaches to look at a situation. In this case, the six hats are used to provide various viewpoints from the author about the parts of the solution developed through the project. The hats are used in each evaluation section in the following way:

- Red hat – The "Emotions" hat.
  Used to explore the author's emotions and intuitions.

- White hat – The "Information" hat.
  Used to explore what was achieved and what aims were met.

- Yellow hat – The "Positives" hat.
  Used to explore the things that went well and lessons learned

- Black hat – The "negatives" hat.
  Used to explore the things that did not go so well.

- Green hat – The "Creativity" hat.
  Used to discuss future work and any tangents.

- Blue hat – The "Overview" hat.
  Used to conclude the evaluation.

8.2 Store System Six Thinking Hats Evaluation

Red Hat. I am happy with how the Store API turned out. After the initial architecture did not work out, I told myself I would restrict myself to tooling I was familiar with, and as such, I should have used Django for the Store API. But I took a risk on using Tortoise, and it paid off. It was not without some difficulty, there were some bits of Tortoises documentation that were very poor, and sometimes the library did not behave how it should have. However, I believe it was worth it for how the API turned out. The OpenAPI 3 specification alone added so much value later in the project, and I feel optimistic about how easy it will be to add more features in the future.

White Hat. The system successfully met all three of the objectives for core aim four.

Yellow hat. As the API provides an OpenAPI 3 specification, it made it easier to interact with it from the other systems in the platform. I am also happy with the combination of Tortoise and FastAPI, which allows the specification to be tightly coupled to the implementation of the data model, giving further assurances to other systems that the data model in the database matches the API specification. Thirdly, the use of the swagger editor made debugging and manually testing the API significantly easier. Finally, the use of the router-controller pattern and the nesting of routers into a route tree has enabled the code to remain modular and DRY.

Black hat. The runtime generation of the pydantic models causes a significant delay in the startup time of the API, of about 10/15 seconds. The API has inconsistent permission behaviour for different resources. There is also inconsistent implementation in API endpoints about how resources use other resources controllers.

Green hat. The most important piece of future work will be adding the ability for users to export and import their learning data. The next step after that will probably be to add improved querying capacity. Currently, a user can request an individual resource or every single instance of a resource they own with no gap in between.

Blue hat. Overall, the Store system was a success. It meets all the objectives required of it, and the way it was built helped expedite my progress on other parts of the solution. However, some work is needed to ensure the API behaves consistently, and the API must be expanded to allow Learners to import and export their data.

8.3 Practice system six thinking hats evaluation

Red hat. A part of me is scornful that the user interface is, in my mind, ‘fundamental’, and I wish that I had the time to develop a nicer looking UI or had used a popular component library instead of trying to do my design. Another part of me is frustrated that I only implemented spaced-repetition in a quite limited way near the end because of how chaotic everything else was. Besides these two frustrations, I am proud of this part of the solution as I learnt an awful lot completing it, and these lessons will put me in a good place to develop web applications in the future.

White hat. The practice system successfully met all three of its objectives. It did not meet the optional objective of providing an alternative practice client.

Yellow hat. I believe several small features add a lot of value to the web application. For example, using the Monaco editor to edit solutions is so much more ergonomic than a regular form input. Secondly, small features like internationalization will be necessary for this solution to be able to be deployed. It is exciting that spaced repetition is implemented and works. I have learnt a lot about frontend development creating the Web-App. I believe supporting PyTest was a good practical choice that will encourage people to create kata.

Black hat. The solution tester is very delicate, and I am not sure I would trust it not to keel over or to be able to extract exception information from each test run meaningfully. Also, while the web application implements spaced repetition, it does not do so in a way that has been tailored to the activity of practising kata. Therefore, it may not currently be utilized most effectively.

Green hat. I think the most important future piece of work is to make the solution tester more robust. Then after that, it should be to adapt the spaced repetition to how quickly or effectively users solve the kata instead of relying on a 0-5 scale like flashcard solutions.

Blue hat. The practice system successfully demoed practising kata in a spaced-repetition based way and was a source of learning for the author. However, significant work is still required to make it something that Learners would want to use.

8.4 Authoring system six thinking hats evaluation

Red hat. I have three powerful feelings about this piece of work. First, shock that I managed to go so far through the development cycle before realizing that I had not adequately planned how to carry it out. Second, pride, with how much I was able to implement within the two weeks sprint I worked on it. Third, frustration that I had to stop due to other academic commitments when I was so close to getting the Git-ops like workflow to work correctly.

White hat. The system successfully met both of its objectives. It did not meet the optional objective of making the workflow more effective and accessible.

Yellow hat. I believe that the provided workflow will, with a suitable set of unit and acceptance tests to validate its behaviour and remove edge cases that could render it frustrating to use, be an effective way of creating katas within the platform. Considering katas are mostly just instructions and unit tests, this workflow makes so much more sense than using a web application to create kata.

Black hat. But did it have to be this complicated? What if Authors do just want to upload a file and be done with it. I guess this work pre-supposed that Authors want to use git to collaborate on building kata. Unlike Learners, of which the author is one, I could have done some more work to find out what Authors needed in a workflow instead of what I thought they would want.

Green hat. The next step is to finish the reconciliation service. Also, to add the ability to manually recreate the file-system configuration files from the state of a Blueprint in the Store as part of adding data export.

Blue hat. Overall, the authoring system successfully demonstrates what I believe will become an effective workflow for authoring kata. It is just a shame that this work was not appropriately prioritised throughout the project and, as such, has ended in an unfinished state.

8.5 Security work six thinking hats evaluation

Red hat. I have mixed feelings about the security work. I am happy that I was able to implement Keycloak. But I cannot help but wondering if I had not made such a fuss about having everything self-hosted if I would have had more time to get the Solution Sandbox to work. I still actually feel like having the solution be self-hosted is important. I just wish I had been able to get the Sandbox to work.

White hat. The security work completed one of the three objectives for the secondary security aim. It did not meet the objective to document security boundaries, document platform actors and trust, and finally did not implement the objective to sandbox solution testing.

Yellow hat. Having Keycloak setup should make it trivial to enrol users in bulk, as it can connect to performing user federation against active directory. This would make the platform much easier to deploy in corporate networks. Also, although the solution sandbox was not wholly realized, good progress was made, and Firecracker will provide a better security posture than simply running the workloads as containers.

Black hat. I spent an entire day trying to work out how to use Keycloak's REST API to provision the Terraform client automatically. It was not a bad idea, but it's just one of several examples of how implementing the authorization service slowed everything down. Setting Keycloak up was slow, integrating it into the other systems was slow, and once all the API endpoints required me to constantly recreate new access keys, testing slowed down.

Green hat. The next step is to finish implementing the solution sandbox and make sure that not only do services like the Store API require requests to be authenticated but that users have adequate OAUTH scopes for operations they wish to carry out throughout the platform.

Blue hat. Overall, the security aim was not finished successfully and constitutes essential work before the platform can be deployed in a wider manner.

8.6 Suitability for Learners

While the solution is usable for Learners, I do not yet believe it is entirely suited to what they would need to use it regularly. Firstly, the web application is not properly responsive and so may not display appropriately at different viewports. Secondly, there is no documentation or tutorials available to show Learners how to make use of the solution. These are issues that may make using the web app unappealing as a learning client. However, I think the larger problem is the implementation of spaced repetition. As spaced-repetition was implemented very late into the project, I do not believe the SM-2 algorithm has been suitably adapted to deal with the fact is now concerned with the practice of kata as opposed to the memorization of flashcards. It does not feel appropriate to ask users for their perceived difficulty on a scale of 0 to 5 after they have just successfully demonstrated they could solve the kata. I believe an important piece of future work in this regard will be to adapt the SM-2 algorithm to be more appropriate for the practising of Katas. With this criticism in mind however, I do believe that the solution satisfies the Learners core motivation to be able to practice practical kata in their browser in a spaced-repetition based way. Therefore, I think that the solution satisfies Learners needs but requires some quality of life and usability improvements before it would be suitable to offer to Learners.

8.7 Suitability for Authors

I believe that the solution is well on its way to being suitable for Authors. I believe the Git-based authoring workflow will allow staff to effectively collaborate on the creation of Kata. However, I believe that a significant amount of manual and automated testing will need to be carried out to ensure the software is reliable enough to be used by people other than the Author and his supervisor who are familiar with the project. Therefore, I think that the solution can satisfy Author's needs but will require further testing to provide the necessary assurances.

8.8 Suitability for Administrators

I believe the solution is suitable for Administrators. It has been an implicit aim of the project to build a solution that can be self-hosted, and that aim has been achieved. The entire project should be able to be deployed in any environment (with a caveat for the solution tester requiring KVM access). The use of Keycloak for the authorization server should make it easy to automatically enrol users onto the platform.

8.9 Overall suitability.

Overall, I believe that the solution created in the project solves all the three key stakeholders' main needs; to practice kata, to create kata and to provide a platform for those two activities to occur.

However, while I believe the project has been successful demonstrated a minimal viable platform for how these activities can be carried out and how the spaced-repetition based practice of katas can be delivered, I believe that there will need to be a period of significant testing before the platform can be released.

Bearing in mind this solution was developed by one person over a twelve-week period, four weeks of which were spent exploring a novel but currently unviable architecture, the author is satisfied that substantial progress has been made toward developing the solution.

9 Evaluation of project

This part of the report is concerned with providing evaluation of the project overall.

The following Gannt charts shows the timeline I set for myself in the initial plan and the timeline that I ended up following during the project.

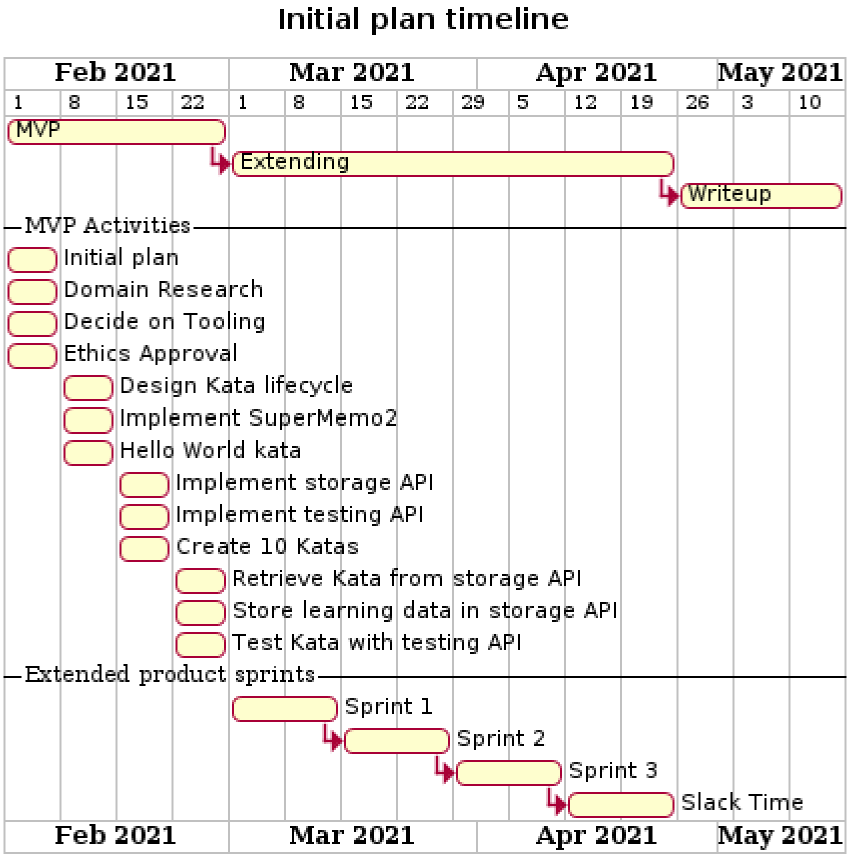


Figure 34

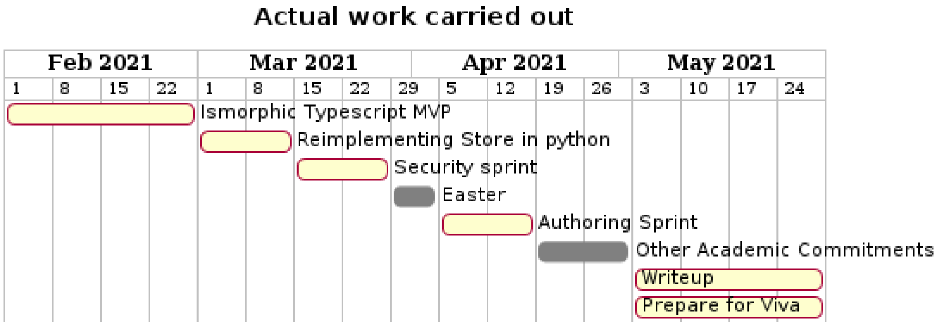


Figure 35

9.1 Problems with the initial plan.

In retrospect, the timeline I set for myself in the initial plan was incredibly ambitious, especially with regards to the minimal viable project phase of work. I gave myself four weeks to complete the MVP and expected that within two of those weeks, I would implement both the Store and Practice systems. I do not think it was reasonable to expect myself to complete 75% of the projects core aims within roughly 16% of the allotted development time. Secondly, I believe that some of the tasks I set myself in the first half of the MVP did not make sense to complete when they were set—for example, attempting to implement the SM-2 algorithm in week two, before the web application in which it would be used was created.

In defence of these choices, I knew that these were ambitious deadlines to set myself. But before I started the work, I believed that applying pressure on myself to quickly implement the core functionality was a worthwhile trade-off because it would allow me more time to work on the project's secondary aims. This was because I thought these were where more of the value-added aspects of the project would be achieved.

The consequence of this choice was that, especially in combination with developing a novel architecture, I perpetually felt behind for most of the project. The consequence of feeling behind was that I spent most of the project in 'crunch' time, believing that I just had to keep working and developing features at risk of falling even further behind. This attitude limited my capacity to properly reflect on the development process as it went along.

It was only after I took a serious break in week 9 that I realized on reflection even attempting to live up to this original ambitious timeline would not be sufficient. This was because the initial timeline did not cover the work required for the authoring system (CA-1) at all! I had been so focused on trying to build an ambitious platform that answered all the secondary aims I did not begin working on the authoring system until the last part of the development cycle.

In the future, If I were to repeat a project like this, I would provide myself with a much generous amount of time to complete the core functionality. I would also be more careful not to let my ambition blind me and ensure that I had sufficiently planned to deliver any core functionality before moving on to additional work.

9.2 Prioritizing infrastructure over data structure

On reflection, another thing that significantly slowed down my development velocity was prioritizing decisions about infrastructure and architecture before having a clearly defined data structure.

As I began to re-implement the Store API in Python at the start of week five, I did not have a clearly defined data model. I was in a rush to demonstrate a 'happy path' to my supervisor. This would include the web app fetching from the store and sending a solution to be tested. So, I defined the data model in enough detail to allow me to reach this objective. However, when I then started work on the authoring system, I had to make significant changes to the data model to accommodate the idea of Blueprints and Boxes as well as introducing the concepts of Collections and Decks. This meant that I had to change all the models in the store, and then I had to change how the web app implemented those models.

This back and forth, having to update how the web app handled its data model because of changes in the authoring system, slowed me down. Fortunately, the ability to generate clients from the stores OpenAPI specification saved me a lot of difficulties. Because the generated clients would represent the data model as either Typescript types, or Pydantic models, I was then able to use relevant language tooling (tsc and pylance) to ensure that I was handling the API results in a type-safe way.

9.3 The decision to attempt initial architecture

I still genuinely believe there may be a lot of value in attempting to build an isomorphic Typescript application, especially once there is a suitable way to share Typescript definitions between Deno and the Browser. However, on reflection, a major consequence of attempting to implement this novel architecture was that I created significantly more work for myself during the MVP period.

Firstly, before the project started, I had never written any Typescript or used Deno. So, as well as trying to complete all the requisite work I was also teaching myself a new programming language and how to use all the associated tooling. This obviously slowed down my development velocity significantly. Secondly, the architecture was complicated. As discussed in the reflection section, I think being blinded by ambition. I had come to see this complexity as a necessary part of the project. However, without any scheduled development time to sit down and carefully design the different interfaces and schemas, it was more of a detriment to making efficient progress.

Perhaps, in the end, it was a blessing that this architecture is not quite feasible yet. This forced me to switch back to using tools and frameworks that I was more familiar and effective with. In the future, I would not discount attempting to using a novel architecture if I saw value in attempting it. However, I would put a sunset clause into the plan to retreat to familiar tooling after a certain period if results were not yet forthcoming.

10 Future Work

This part of the report is concerned with describing the future work that will be required to turn the developed solution from a minimal viable product into a platform that could be deployed within the Authors university.

10.1 Testing for assurance of results

Although the author is confident in their assessment of the results, this section describes the test suite that will be developed to provide a programmatic assurance of this assessment. The Store API will be unit tested using PyTest, specifically focusing on unit tests that assure that all the code paths between API endpoints and controllers work properly. Secondly, it will be integration tested to ensure that resources are appropriately secured. The Solution tester will be unit tested with a variety of Kata and exception cases to ensure it can handle a robust set of test cases. The Web-App will be end to end tested using Cypress. The CLI and reconciliation service will be united testing using PyTest. Finally, a set of end-to-end tests will be developed to demonstrate the creation of a Kata in the authoring system through to its practice in the web app.

10.2 Essential work.

The essential work to make the Store system suitable for deployment is the ability for users to import and export their learning data. The essential work required to make the Practice system suitable for deployment is the completion of the Solution Sandbox. The essential work required to make the Authoring system suitable for deployment is the completion of the reconciliation service so that the Git-ops methodology can be fully realized.

11 Conclusion

At the conclusion of this project, a platform has been developed that achieves all four of the core aims set out in the specification and demonstrates a minimal viable product suitable for future expansion.

Learners can practise kata in a spaced repetition-based way through a web application, making use of the SM-2 algorithm. Authors can collaboratively create Kata using a GitOps-like methodology. The platforms data model is exposed as an HTTP JSON API, and the platform is secured using OAuth. Future work is required to make the platform suitable for being deployed.

12 Reflection

This final section of the report contains reflection about my personal development and double-loop learning from carrying out this project. The reflection follows in the spirit of the agile prime directive[60].

Imagine you need to build two robots to help people practice karate. One of the robots will automatically assess and provide feedback on a Learners' form when they carry out a karate kata. The other robot will algorithmically schedule when the Learner should practice the kata again. You need to build these robots because you are part of a project trying to research whether peoples practice of karate is more effective when they are being helped by robots performing both of those functions at once. However, you realize there is not an existing karate dojo that would be capable of accommodating both robots and the Learner. So, as well as developing the robots, you also need to design and build a suitable dojo in which the Learner can interact with them.

So, you bring on an architect to help you come up with a suitable architecture. Now, almost immediately, the architect begins planning. All is good at first, and you have a more than sufficient idea of what will need to be built to satisfy the problem of co-locating the robots and solving the needs of the other stakeholders involved. Those people are the Learners, who will want to use the Robots to practice, the Authors, who will teach the Robots how to correctly assess different karate kata, and finally, the Administrators who will be responsible for managing one of these high-tech dojos.

However, now the Architect goes on, making ambitious plans for how this high-tech dojo could be generalized to a system that would allow Learners to practice not just in one dojo, but to also practice from home or visit the dojo via telepresence on their phones, or be able to seamlessly practice between many different dojos at once. Now, depending on your project management skills, you might have raised a discouraging eyebrow or tried to provide polite but firm feedback about scope creep before this went much further.

The problem in my case was that I was simultaneously the person who needed to build the robots, the architect who needed to design the dojo, and the person responsible for managing the project. As the project began, I had become disconnected from the problem I set out in my initial proposal, providing a platform to provide access to the method of spaced-repetition based practice of kata. Instead, as I began working on the project, I had become, without realizing it, fixated on creating an ideal architecture for this platform.

The rest of this section reflects on the personal dynamics that lead to this mindset, its effect on how the project was carried out, and the lessons I have learnt for managing myself more effectively in further projects.

12.1 Personal dynamics

12.1.1 Personal roles

I only had one official role during this project, to be the student carrying the project out. However, to aid in describing the double-loop learning I have engaged in, I have decomposed this role into four personal roles. Each of these roles represents a different responsibility I had throughout the project. Each role also stereotypes a set of the personal beliefs, assumptions, opinions, and motivations I held, or hold, with regards to that part of the project.

The first role I had was that of the project proposer. My main responsibility in this role was to define a problem, then propose a project to solve it and then find a supervisor who would supervise me to carry the project out.

The second role I had was that of a project manager. My main responsibility in this role was managing myself to successfully complete the project once it had started. I was responsible for making sure I stayed on track during development, for producing other artefacts such as this report and making sure I engaged with my supervisor.

The third role I had was that of a developer. My main responsibility in this role was to create the method for the spaced-repetition based practice of kata. This work would be how the problem that such an open-source method did not currently exist would be solved.

The fourth role I had was that of a software architect. My main responsibility in this role was to create a platform that would provide a way for this new method of practice to be used. This work would be how the solution to the problem would then be applied to address the needs of the project's stakeholders.

12.1.2 Misaligned mindset

The largest extrinsic motivation I had to complete the project was its academic significance. I knew that this would motivate me in my role as my own personal project manager to ensure I was spending the right amount of time and focus on the project. However, from my experiences during my placement year and from having had to complete this final year remotely through a pandemic, I had already learnt that academic obligations alone would not be enough to get me to produce my best work. I knew that to be my most creative and engaged while performing the roles concerned with delivering the project (the developer and the architect), I would have to tap into some form of intrinsic motivation instead.

However, it was not hard to find a source of intrinsic motivation to tap into. I had been intrinsically motivated to propose the project because I am personally fascinated by the topics of spaced-repetition, software architecture and the effective practice of programming. In fact, I had proposed the project with myself in mind as the target audience because I had found there was a lack of adequate solutions for spaced-repetition based practice of programming while trying to develop my own forms of practice.

Therefore, I reasoned that because the purpose of the project was to fulfil the proposal and that the proposal had been derived from my own problem, that if I followed my own intrinsic motivation and focused on solving my problem, I would naturally complete the project. Said another way, I considered working on the problem as I understood it to be the same as working on the project because the projects proposal had been inspired by my own personal problem.

I believe approaching the work in this way did allow me to work more creatively and effectively. I felt a greater sense of agency working on the project because I was solving a problem that was personal to me. I am very confident that this mindset encouraged me to work with more passion than I would have been able to summon if I were just completing the project because I was academically obligated to solve the problem.

On reflection, however, there was a problem with this decision. Because I did not work explicitly to solve the problem that was defined within the project proposal or project plan but instead worked to solve my own personal interpretation of this problem, other intrinsic motivations I held influenced my perception of what the scope and focus of the problem was.

There were two major instances of this. The first was the reading I did over the Christmas recess. The consequence of that reading was that I was viewing the problem, and as such the project, with a much heavier focus on the architecture and building a platform that would respect users learning data than I believe I made clear in my plan or proposal.

Secondly, I realize now, at the project's conclusion, that I had an intrinsic motivation to try and gain social capital from completing this project. To be honest, I think I felt like this project was an opportunity to demonstrate to myself and my peers the competence I had developed throughout the course by producing something of significant scale or complexity. The consequence of this was that the project suffered significant scope creep as I kept expanding my horizons with regards to what I thought I should and could achieve by the end of the project.

In summary, I motivated myself through this project by focusing on the problem as it related to my own personal experience and needs. I made this decision on the assumption that my perceptions of what was important to address and the scope of functionality I should expect to implement would remain in sync with what I had promised in the proposal and plan. The issue was that due to various internal factors, my perceptions and motivations changed, and at least in the capacity of being my own personal project manager, I was not aware of this change. As such, my mental model of the required work, as well as other underlying assumptions and priorities, became disconnected from what I had promised I would do in my plan and proposal.

The misaligned mindset was a combination of this drift in mental model from the promised work and an assumption that working on my understanding of the problem was the same as working on the project, which stopped me from effectively course correcting.

12.2 Consequences of a misaligned mindset

I have only realized I had this mindset at the conclusion of the project, and it has helped me reflect on several decisions I made that perplexed me when I first looked back through how I had worked.

Firstly, I now understand why I made the decision to attempt the novel isomorphic typescript application in the first place. At the time, I really believed that if I did not attempt such an architecture, then I would be missing something important out of the project. This was because I had implicitly decided that it was an important part of the problem that the solution should have the capacity to be later turned into an unhosted application.

Secondly, I now understand how it took me until the ninth week to realize that I had not started any work towards one of my cores aims, the ability to author kata. My mental model of what I needed to do had drifted so far from what I had promised it was not until I took a week out to properly rest that I became aware of the gap, as I had to look at my initial plan to work out what to do next. Before that, I had been incessantly driven forward by what I felt like I should be doing next.

Thirdly, it helped me understand why, even though I created a system that ultimately met all my core aims, I had been left with a listless feeling that I had not done something right. This was because, in my mind I was, initially, disappointed I had not achieved something much grander.

12.3 Future learnings

Reflecting on how I managed myself during this project has taught me some valuable lessons that I will use to manage myself more effectively on projects in the future.

I now understand that the problem I was personally motivated with solving throughout this project was not the restricted and generalized problem I laid out in the proposal and brief. Instead, I was motivated with solving my personal understanding of this problem.

I now realize that my personal understanding of the problem was not something static. In fact, my understanding of what exactly the problem was changed throughout the duration of the project, and in some cases, I did not realize it had changed.

I now see the consequences of having my mental model of the required work become misaligned with the work that I had promised. Being nerd sniped[61] by my Christmas reading caused me to treat the entire project as an architecture project as opposed to a combination of development and deployment. This ultimately led me to doing a lot of work, but not work that was directly involved with achieving the project's objectives.

Before this project, I had scorn for the idea of personal project management. I believed that the best way for me to work was to find a motivating reason and use that drive me. However, I now understand why people say motivation is fickle and appreciate the need for more careful self-management. As while I worked consistently on my understanding of the problem, my understanding of the problem was not consistent with the problem I had said I would solve.

I now believe that contrary to my previous beliefs, carefully constraining myself would have led me to be significantly more productive. Instead of seeing these constraints and an explicitly defined scope as an obstacle to being creative, I now see them as useful tools for making sure that I aligned with delivering the promised work.

I believe this lesson will be important for the future of this project going forwards, as I intend to publish the platform as an open-source project on GitHub. I realize that if I wish to meaningfully solicit contributions, I will need to make the projects scope, desired functionality, and purpose as explicitly clear as possible so that all contributors can work in an aligned way.

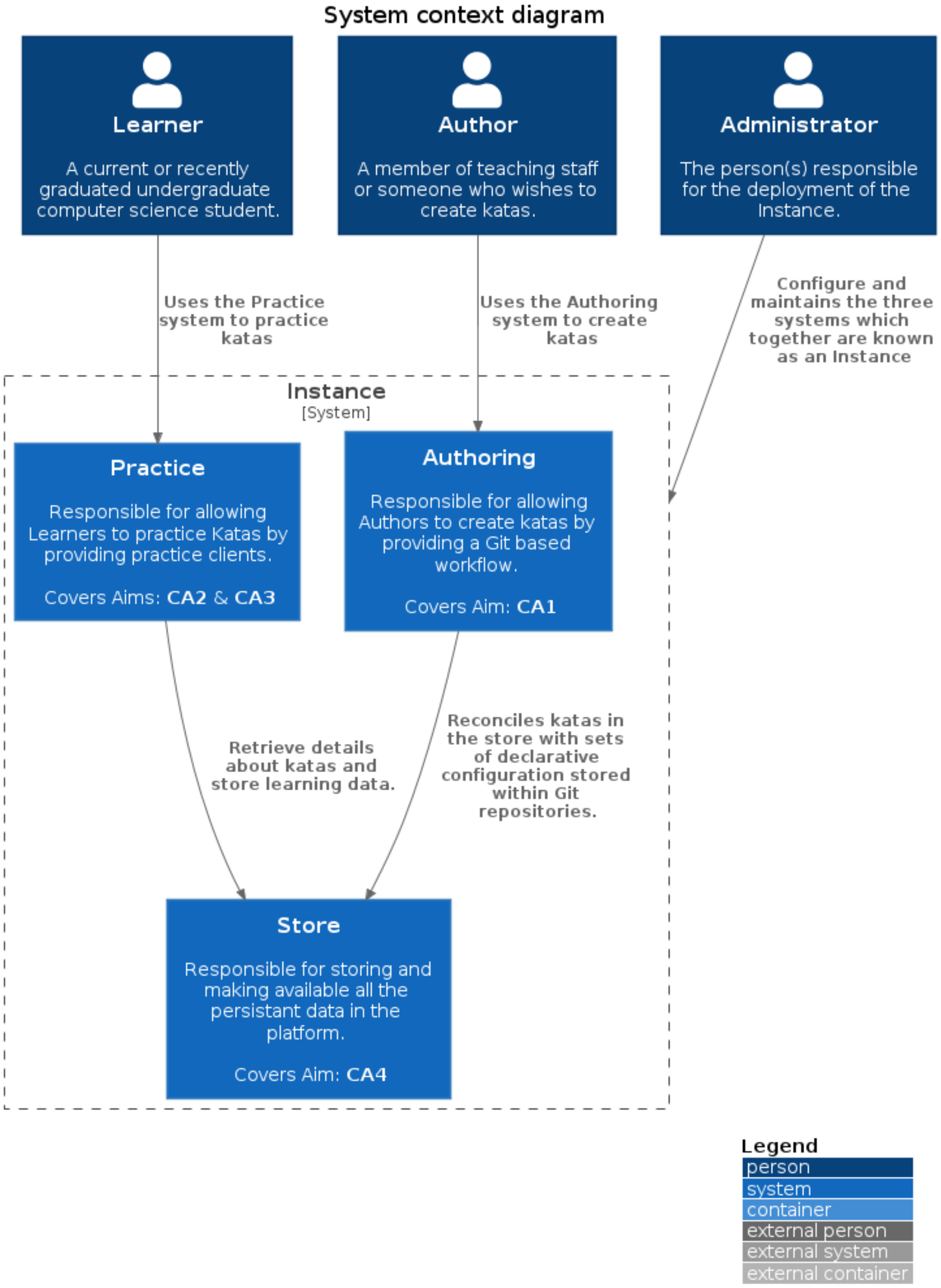
In conclusion, I am grateful for learning this lesson that following my sense of creative inspiration is not an effective project management strategy. It was, however, an effective method of keeping my personal interest in this problem and project at a very high level, and I look forward to continuing to work on this solution after the end of the project.

13 Architecture Diagrams

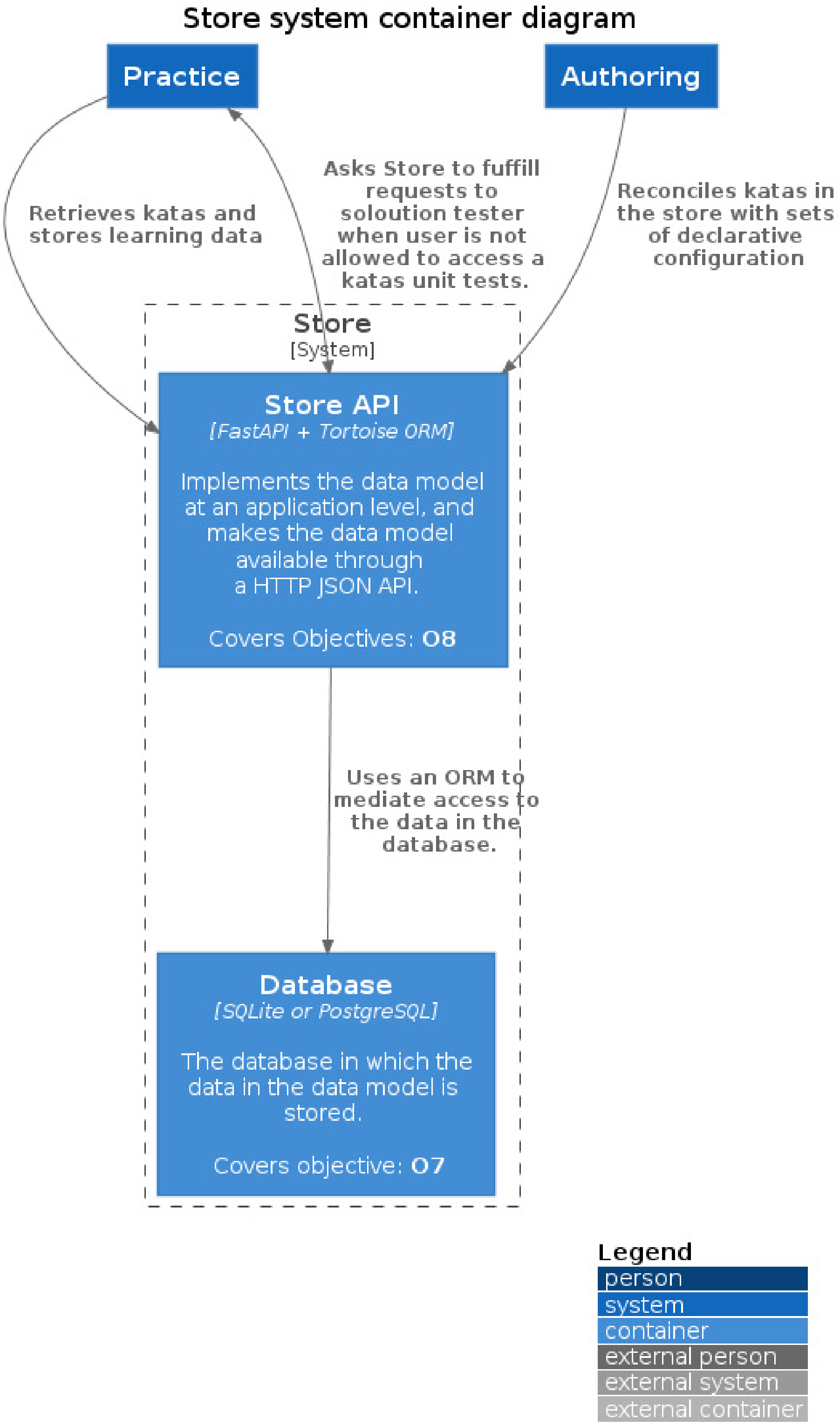
13.1 Design Diagrams

13.1.1 Platform System Context Diagram

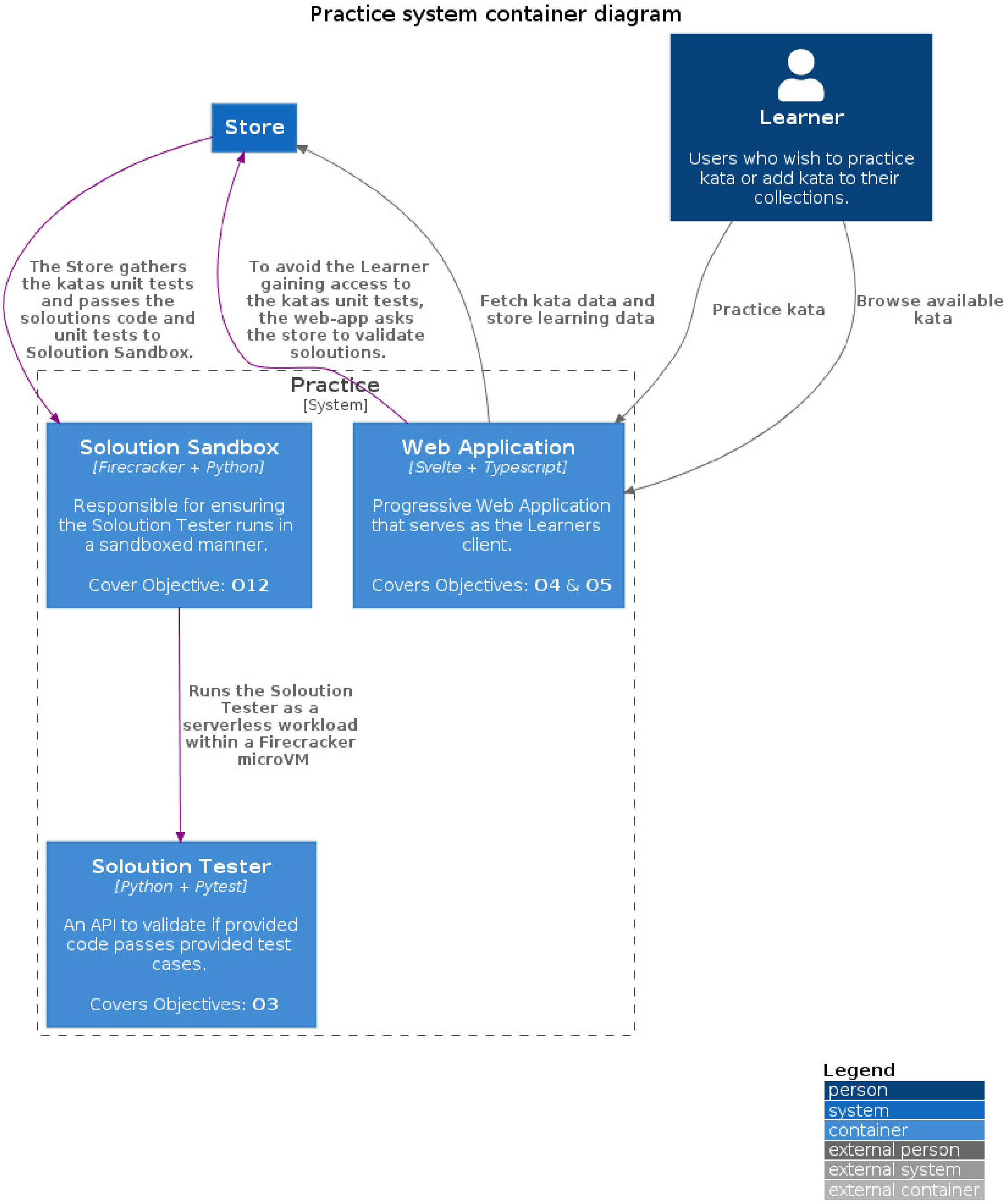
Note The three systems have been placed in the same diagram for ease of reading, however this deviates from the typical implementation of a system context diagram as covering one software system at a time.


System context diagram
Learner
A current or recently graduated undergraduate computer science student.
Author
A member of teaching staff or someone who wishes to create katas.
Administrator
The person(s) responsible for the deployment of the Instance.
Uses the Practice system to practice katas
Uses the Authoring system to create katas
Configure and maintains the three systems which together are known as an Instance
Instance
[System]
Practice
Responsible for allowing Learners to practice Katas by providing practice clients.
Covers Aims: CA2 & CA3
Authoring
Responsible for allowing Authors to create katas by providing a Git based workflow.
Covers Aim: CA1
Retrieve details about katas and store learning data.
Reconciles katas in the store with sets of declarative configuration stored within Git repositories.
Store
Responsible for storing and making available all the persistant data in the platform.
Covers Aim: CA4
Legend
person
system
container
external person
external system
external container

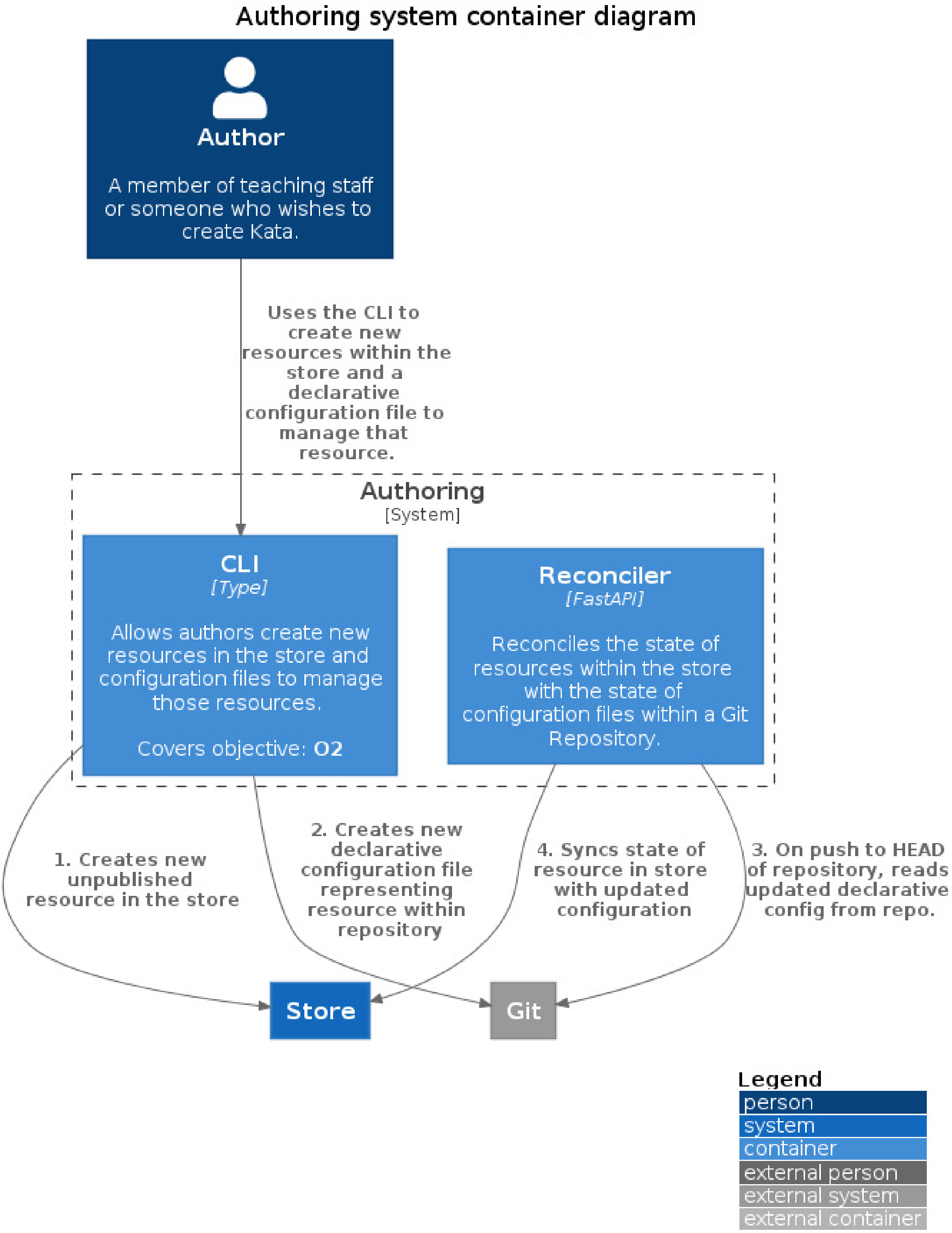
13.1.2 Store System Context Diagram


Store system container diagram
Practice
Authoring
Retrieves katas and stores learning data
Asks Store to fuffill requests to soloution tester when user is not allowed to access a katas unit tests.
Reconciles katas in the store with sets of declarative configuration
Store
[System]
Store API
[FastAPI + Tortoise ORM]
Implements the data model at an application level, and makes the data model available through a HTTP JSON API.
Covers Objectives: O8
Uses an ORM to mediate access to the data in the database.
Database
[SQLite or PostgreSQL]
The database in which the data in the data model is stored.
Covers objective: O7
Legend
person
system
container
external person
external system
external container

13.1.3 Practice System Container Diagram


Practice system container diagram
Store
Learner
Users who wish to practice kata or add kata to their collections.
The Store gathers the katas unit tests and passes the soloutions code and unit tests to Soloution Sandbox.
To avoid the Learner gaining access to the katas unit tests, the web-app asks the store to validate soloutions.
Fetch kata data and store learning data
Practice kata
Browse available kata
Practice
[System]
Soloution Sandbox
[Firecracker + Python]
Responsible for ensuring the Soloution Tester runs in a sandboxed manner.
Cover Objective: O12
Web Application
[Svelte + Typescript]
Progressive Web Application that serves as the Learners client.
Covers Objectives: O4 & O5
Runs the Soloution Tester as a serverless workload within a Firecracker microVM
Soloution Tester
[Python + Pytest]
An API to validate if provided code passes provided test cases.
Covers Objectives: O3
Legend
person
system
container
external person
external system
external container

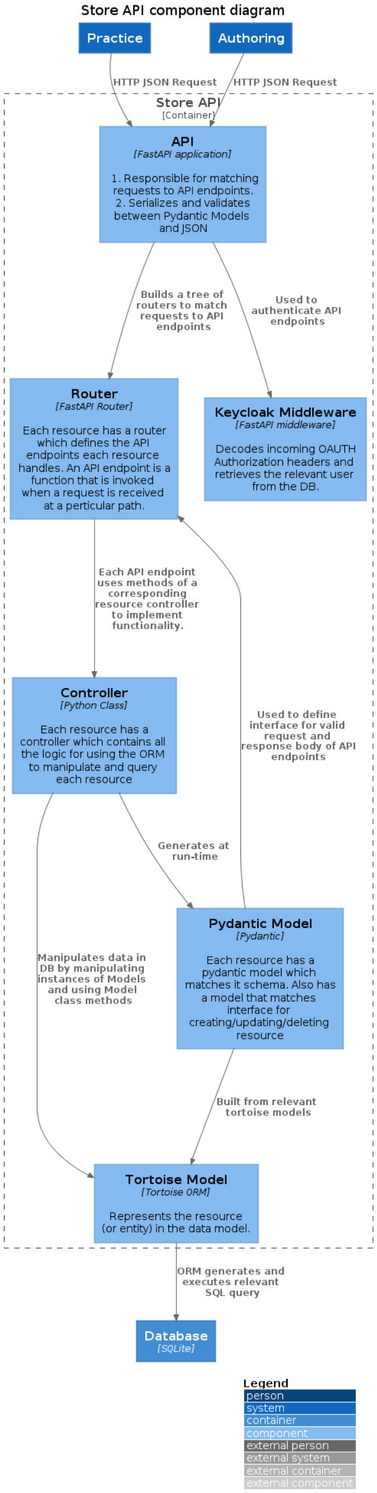
13.1.4 Authoring System Context Diagram


Authoring system container diagram
Author
A member of teaching staff or someone who wishes to create Kata.
Uses the CLI to create new resources within the store and a declarative configuration file to manage that resource.
Authoring
[System]
CLI
[Type]
Allows authors create new resources in the store and configuration files to manage those resources.
Covers objective: O2
Reconciler
[FastAPI]
Reconciles the state of resources within the store with the state of configuration files within a Git Repository.
1. Creates new unpublished resource in the store
2. Creates new declarative configuration file representing resource within repository
4. Syncs state of resource in store with updated configuration
3. On push to HEAD of repository, reads updated declarative config from repo.
Store
Git
Legend
person
system
container
external person
external system
external container

13.2 Implementation Diagrams

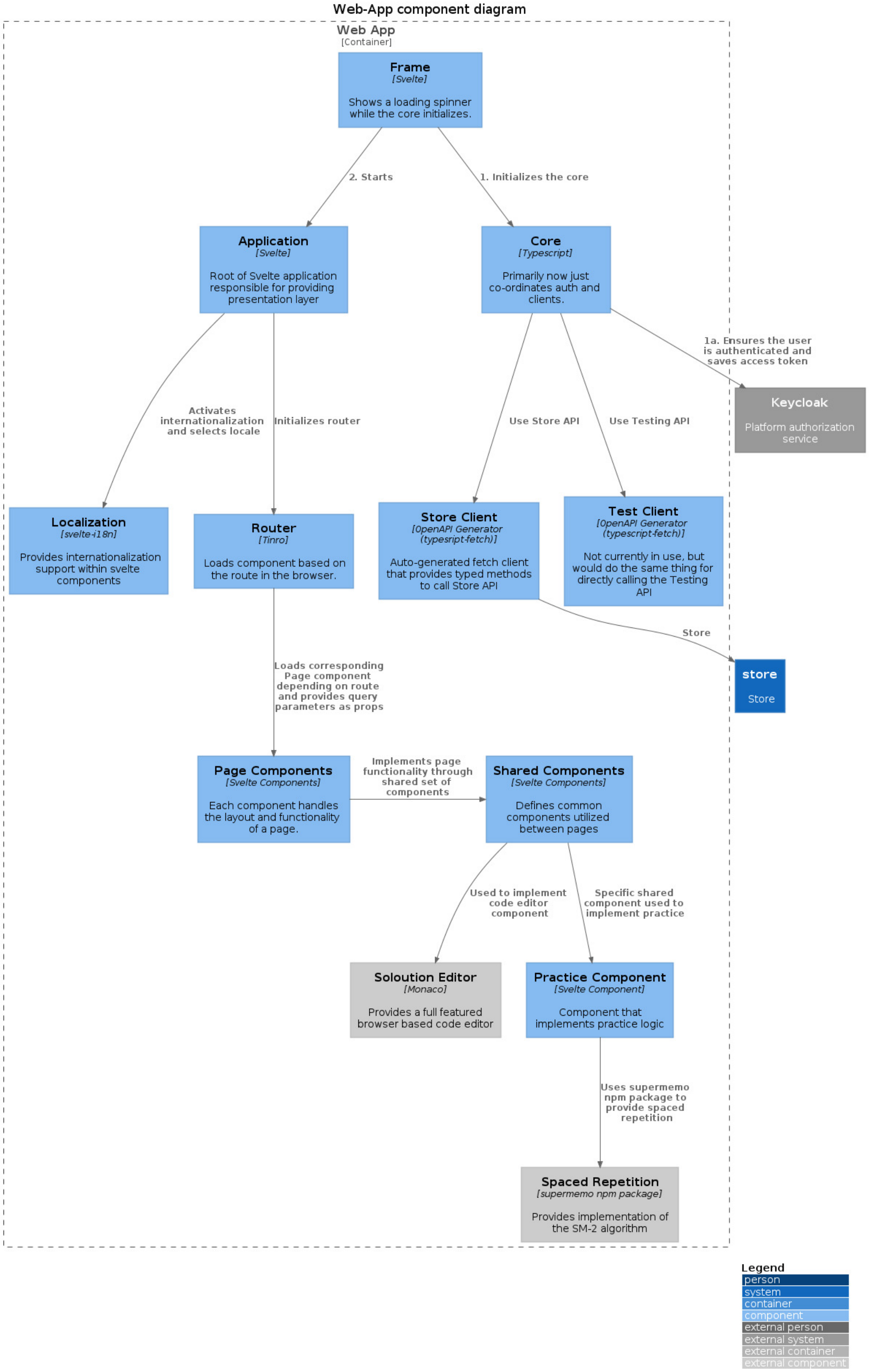
13.2.1 Store System

13.2.1.1 Store API Component Diagram

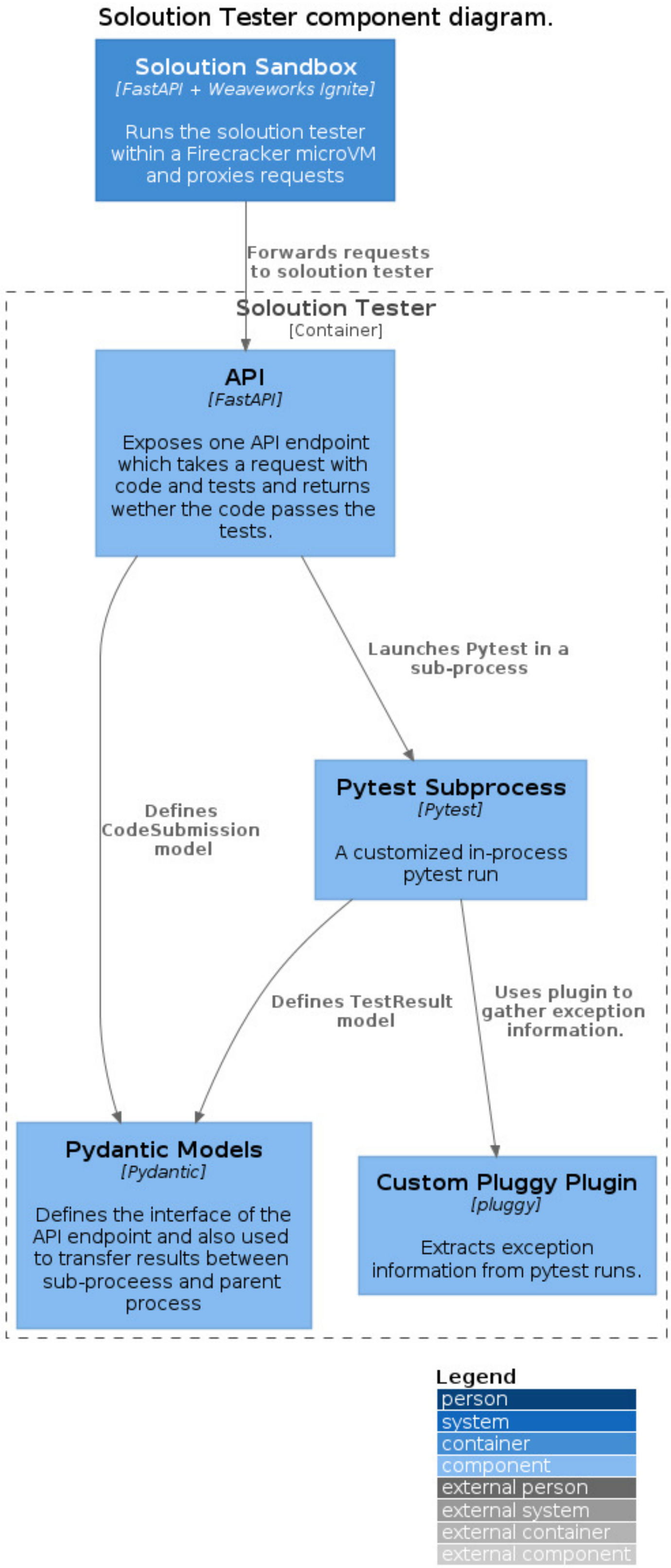

Store API component diagram
Practice
Authoring
HTTP JSON Request
HTTP JSON Request
Store API
[Container]
API
[FastAPI application]
1. Responsible for matching requests to API endpoints. 2. Serializes and validates between Pydantic Models and JSON
Builds a tree of routers to match requests to API endpoints
Used to authenticate API endpoints
Router
[FastAPI Router]
Each resource has a router which defines the API endpoints each resource handles. An API endpoint is a function that is invoked when a request is received at a perticular path.
Keycloak Middleware
[FastAPI middleware]
Decodes incoming OAUTH Authorization headers and retrieves the relevant user from the DB.
Each API endpoint uses methods of a corresponding resource controller to implement functionality.
Used to define interface for valid request and response body of API endpoints
Controller
[Python Class]
Each resource has a controller which contains all the logic for using the ORM to manipulate and query each resource
Generates at run-time
Manipulates data in DB by manipulating instances of Models and using Model class methods
Pydantic Model
[Pydantic]
Each resource has a pydantic model which matches it schema. Also has a model that matches interface for creating/updating/deleting resource
Built from relevant tortoise models
Tortoise Model
[Tortoise ORM]
Represents the resource (or entity) in the data model.
ORM generates and executes relevant SQL query
Database
[SQLite]
Legend
person
system
container
component
external person
external system
external container
external component

13.2.2 Practice System

13.2.2.1 Web-Application Component Diagram

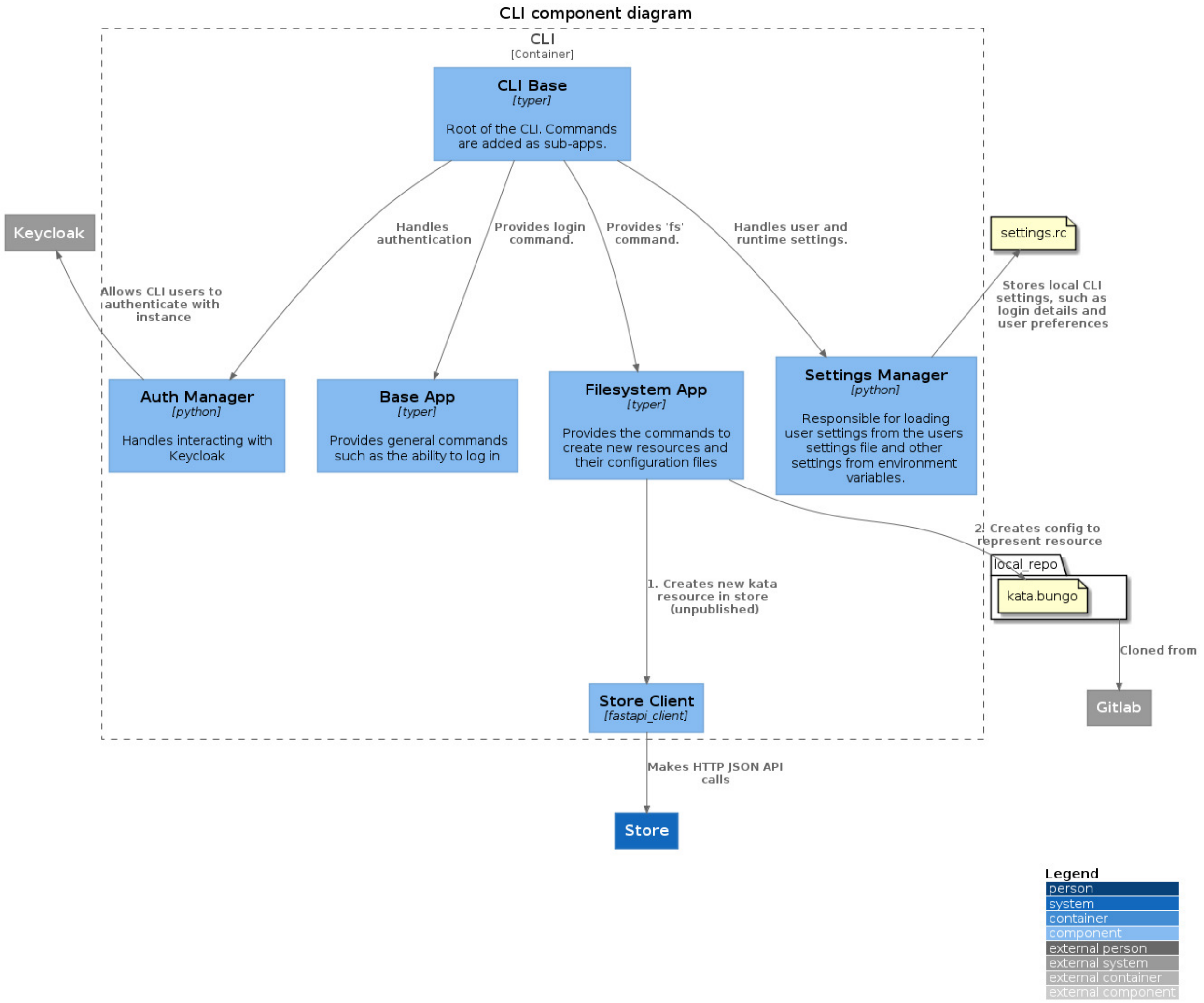

Web-App component diagram
Web App
[Container]
Frame
[Svelte]
Shows a loading spinner while the core initializes.
2. Starts
1. Initializes the core
Application
[Svelte]
Root of Svelte application responsible for providing presentation layer
Core
[Typescript]
Primarily now just co-ordinates auth and clients.
1a. Ensures the user is authenticated and saves access token
Keycloak
Platform authorization service
Activates internationalization and selects locale
Initializes router
Use Store API
Use Testing API
Localization
[svelte-i18n]
Provides internationalization support within svelte components
Router
[Tinro]
Loads component based on the route in the browser.
Store Client
[OpenAPI Generator (typesript-fetch)]
Auto-generated fetch client that provides typed methods to call Store API
Test Client
[OpenAPI Generator (typescript-fetch)]
Not currently in use, but would do the same thing for directly calling the Testing API
Store
store
Store
Loads corresponding Page component depending on route and provides query parameters as props
Page Components
[Svelte Components]
Each component handles the layout and functionality of a page.
Implements page functionality through shared set of components
Shared Components
[Svelte Components]
Defines common components utilized between pages
Used to implement code editor component
Specific shared component used to implement practice
Soloution Editor
[Monaco]
Provides a full featured browser based code editor
Practice Component
[Svelte Component]
Component that implements practice logic
Uses supermemo npm package to provide spaced repetition
Spaced Repetition
[supermemo npm package]
Provides implementation of the SM-2 algorithm
Legend
person
system
container
component
external person
external system
external container
external component

13.2.2.2 Soloution Tester Component Diagram

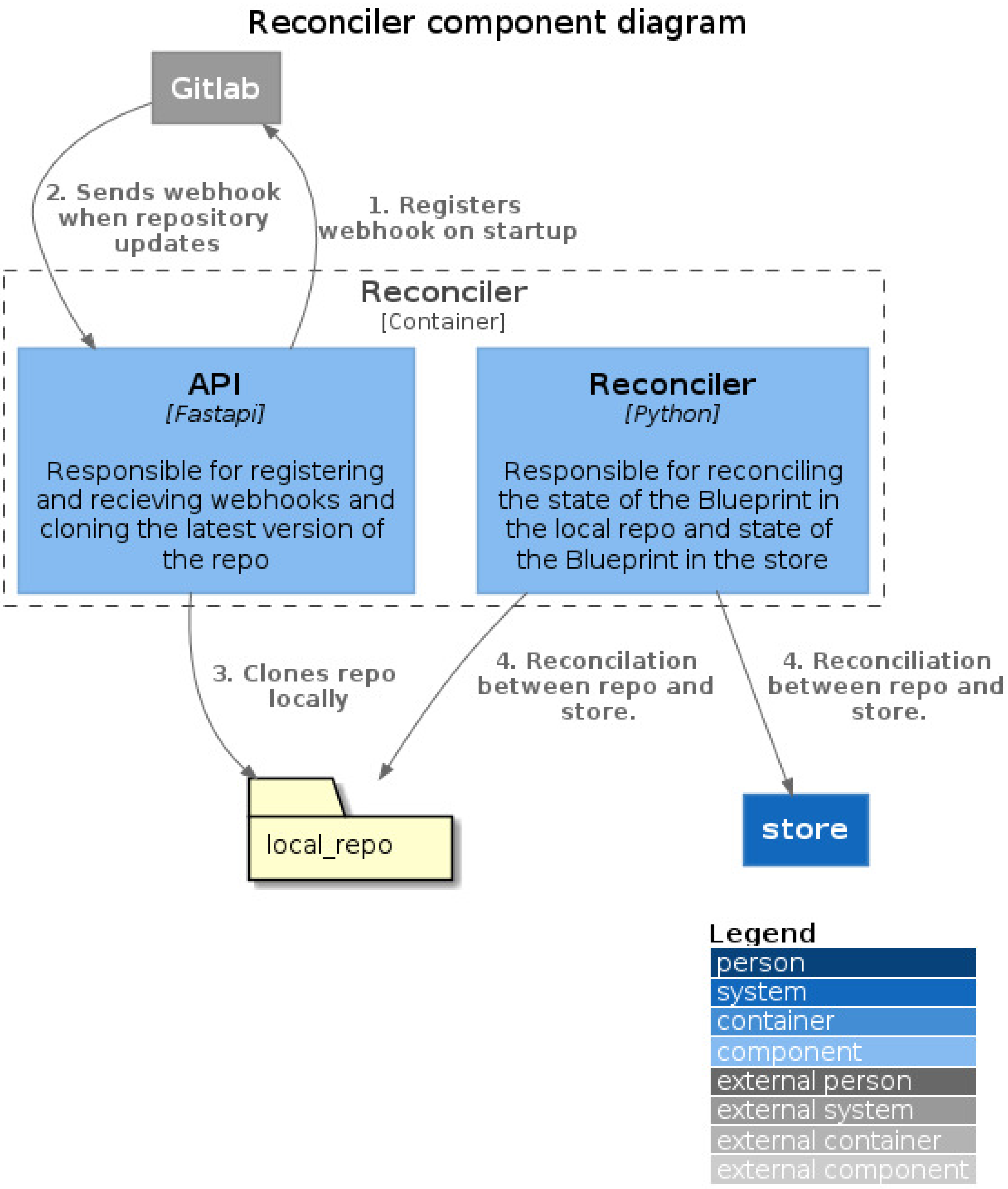

Soloution Tester component diagram.
Soloution Sandbox
[FastAPI + Weaveworks Ignite]
Runs the soloution tester within a Firecracker microVM and proxies requests
Forwards requests to soloution tester
Soloution Tester
[Container]
API
[FastAPI]
Exposes one API endpoint which takes a request with code and tests and returns wether the code passes the tests.
Launches Pytest in a sub-process
Defines CodeSubmission model
Pytest Subprocess
[Pytest]
A customized in-process pytest run
Defines TestResult model
Uses plugin to gather exception information.
Pydantic Models
[Pydantic]
Defines the interface of the API endpoint and also used to transfer results between sub-proceess and parent process
Custom Pluggy Plugin
[pluggy]
Extracts exception information from pytest runs.
Legend
person
system
container
component
external person
external system
external container
external component

13.2.3 Authoring System

13.2.3.1 CLI Component Diagram

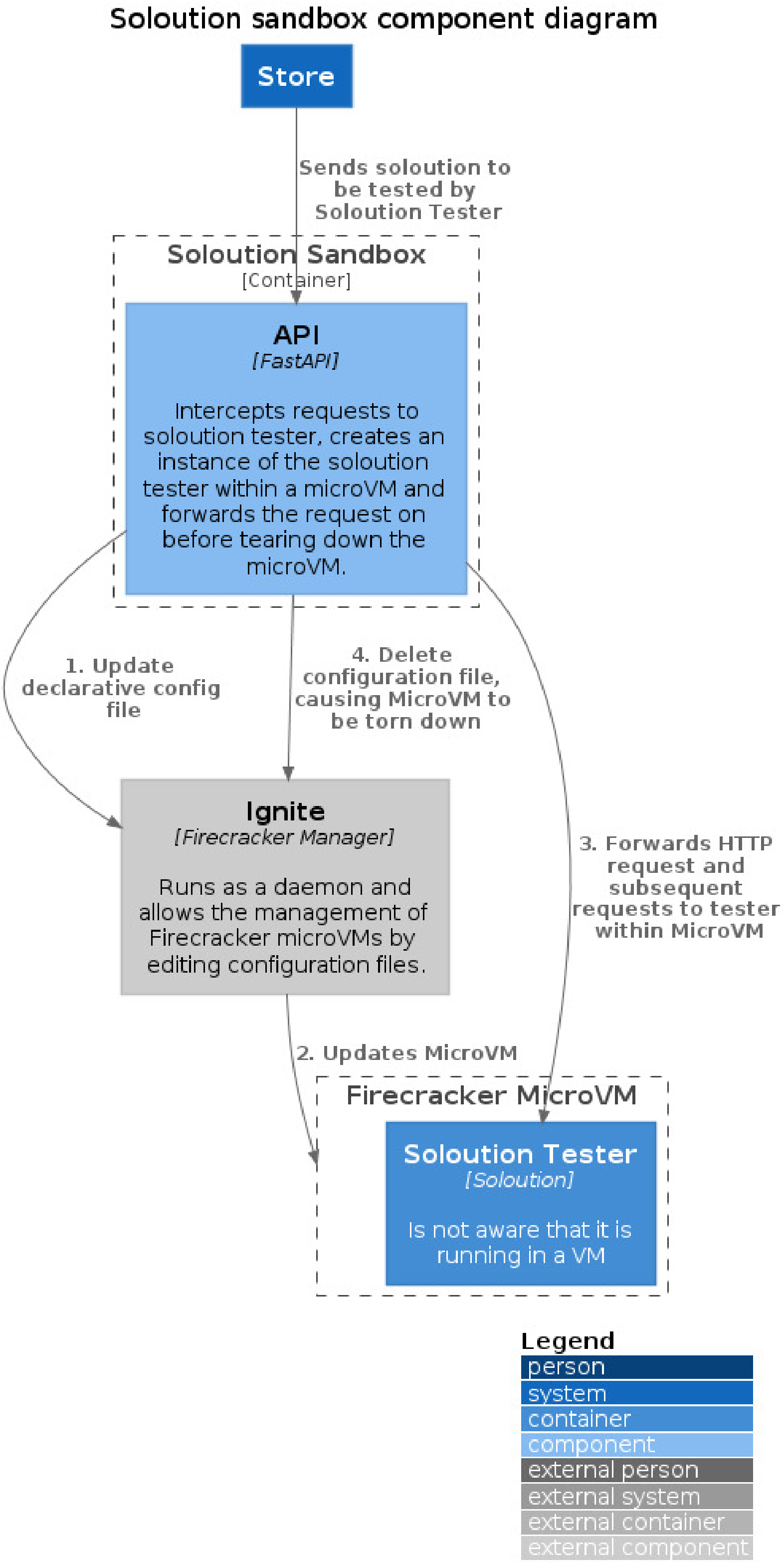

CLI component diagram
CLI
[Container]
CLI Base
[typer]
Root of the CLI. Commands are added as sub-apps.
Handles authentication
Provides login command.
Provides 'fs' command.
Handles user and runtime settings.
Keycloak
Allows CLI users to authenticate with instance
settings.rc
Stores local CLI settings, such as login details and user preferences
Auth Manager
[python]
Handles interacting with Keycloak
Base App
[typer]
Provides general commands such as the ability to log in
Filesystem App
[typer]
Provides the commands to create new resources and their configuration files
Settings Manager
[python]
Responsible for loading user settings from the users settings file and other settings from environment variables.
2. Creates config to represent resource
local_repo
kata.bungo
1. Creates new kata resource in store (unpublished)
Cloned from
Gitlab
Store Client
[fastapi_client]
Makes HTTP JSON API calls
Store
Legend
person
system
container
component
external person
external system
external container
external component

13.2.3.2 Reconciler Component Diagram


Reconciler component diagram
Gitlab
2. Sends webhook when repository updates
1. Registers webhook on startup
Reconciler
[Container]
API
[Fastapi]
Responsible for registering and recieving webhooks and cloning the latest version of the repo
Reconciler
[Python]
Responsible for reconciling the state of the Blueprint in the local repo and state of the Blueprint in the store
3. Clones repo locally
4. Reconcilation between repo and store.
4. Reconciliation between repo and store.
local_repo
store
Legend
person
system
container
component
external person
external system
external container
external component

13.2.4Security work

13.2.4.1 Soloution Sandbox Component Diagram


Soloution sandbox component diagram
Store
Sends soloution to be tested by Soloution Tester
Soloution Sandbox
[Container]
API
[FastAPI]
Intercepts requests to soloution tester, creates an instance of the soloution tester within a microVM and forwards the request on before tearing down the microVM.
1. Update declarative config file
4. Delete configuration file, causing MicroVM to be torn down
3. Forwards HTTP request and subsequent requests to tester within MicroVM
Ignite
[Firecracker Manager]
Runs as a daemon and allows the management of Firecracker microVMs by editing configuration files.
2. Updates MicroVM
Firecracker MicroVM
Soloution Tester
[Soloution]
Is not aware that it is running in a VM
Legend
person
system
container
component
external person
external system
external container
external component

14 References

[1] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, “The Role of Deliberate Practice in the Acquisition of Expert Performance,” *Psychological Review*, vol. 100, no. 3, pp. 363–406, 1993, doi: 10.1037/0033-295x.100.3.363.

[2] “Memory; a contribution to experimental psychology : Ebbinghaus, Hermann, 1850-1909 : Free Download, Borrow, and Streaming : Internet Archive.” https://archive.org/details/memorycontributi00ebbiuoft/page/18/mode/2up (accessed May 19, 2021).

[3] J. M. J. Murre and J. Dros, “Replication and analysis of Ebbinghaus’ forgetting curve,” *PLoS ONE*, vol. 10, no. 7, p. e0120644, Jul. 2015, doi: 10.1371/journal.pone.0120644.

[4] “Spaced repetition - Wikipedia.” https://en.wikipedia.org/wiki/Spaced_repetition (accessed May 19, 2021).

[5] P. A. Woiniak and E. J. Gorzelanczyk, “Optimization of repetition spacing in the practice of learning.”

[6] P. Kelley and T. Whatson, “Making long-term memories in minutes: A spaced learning pattern from memory research in education,” *Frontiers in Human Neuroscience*, vol. 7, no. SEP, p. 589, Sep. 2013, doi: 10.3389/fnhum.2013.00589.

[7] B. Price Kerfoot and E. Brotschi, “Association for Surgical Education Online spaced education to teach urology to medical students: a multi-institutional randomized trial,” 2009, doi: 10.1016/j.amjsurg.2007.10.026.

[8] “Kata | Definition of Kata by Merriam-Webster.” https://web.archive.org/web/20191108092845/https://www.merriam-webster.com/dictionary/kata (accessed May 21, 2021).

[9] “Achieve mastery through challenge | Codewars.” https://www.codewars.com/ (accessed May 19, 2021).

[10] “SuperMemo.com.” https://www.supermemo.com/en/archives1990-2015/english/history (accessed May 19, 2021).

[11] “Anki - powerful, intelligent flashcards.” https://apps.ankiweb.net/ (accessed May 19, 2021).

[12] A. E. Seibert Hanson and C. M. Brown, “Enhancing L2 learning through a mobile assisted spaced-repetition tool: an effective but bitter pill?,” *Computer Assisted Language Learning*, vol. 33, no. 1–2, pp. 133–155, Jan. 2020, doi: 10.1080/09588221.2018.1552975.

[13] W. Al-Rawi, L. Easterling, and P. C. Edwards, “Development of a Mobile Device Optimized Cross Platform-Compatible Oral Pathology and Radiology Spaced Repetition System for Dental Education,” *Journal of Dental Education*, vol. 79, no. 4, pp. 439–447, Apr. 2015, doi: 10.1002/j.0022-0337.2015.79.4.tb05902.x.

[14] “Janki Method Refined | Jack Kinsella.” https://www.jackkinsella.ie/articles/janki-method-refined (accessed May 19, 2021).

[15] “Memorizing a programming language using spaced repetition software | Derek Sivers.” https://sive.rs/srs (accessed May 19, 2021).

[16] “CodeKata.” http://codekata.com/ (accessed May 19, 2021).

[17] “LeetCode - The World’s Leading Online Programming Learning Platform.” https://leetcode.com/ (accessed May 19, 2021).

[18] “Execute Program.” https://www.executeprogram.com/ (accessed May 19, 2021).

[19] “The C4 model for visualising software architecture.” https://c4model.com/ (accessed May 20, 2021).

[20] Weaveworks, “Guide To GitOps.” https://www.weave.works/technologies/gitops/ (accessed May 19, 2021).

[21] “4. Designing Infrastructure Applications - Cloud Native Infrastructure [Book].” https://www.oreilly.com/library/view/cloud-native-infrastructure/9781491984291/ch04.html (accessed May 20, 2021).

[22] A. Agache *et al.*, “Firecracker: Lightweight Virtualization for Serverless Applications,” 2020, Accessed: May 20, 2021. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[23] “Tortoise ORM Documentation.” https://tortoise-orm.readthedocs.io/en/latest/examples.html (accessed May 21, 2021).

[24] “pydantic.” https://pydantic-docs.helpmanual.io/ (accessed May 21, 2021).

[25] “tortoise/aerich: A database migrations tool for TortoiseORM, ready to production.” https://github.com/tortoise/aerich (accessed May 20, 2021).

[26] “Svelte • Cybernetically enhanced web apps.” https://svelte.dev/ (accessed May 22, 2021).

[27] “TypeScript: Typed JavaScript at Any Scale.” https://www.typescriptlang.org/ (accessed May 22, 2021).

[28] “Snowpack.” https://www.snowpack.dev/ (accessed May 22, 2021).

[29] “AlexxNB/tinro: Highly declarative, tiny, dependency free router for Svelte’s web applications.” https://github.com/AlexxNB/tinro (accessed May 22, 2021).

[30] “Tailwind CSS.” https://tailwindcss.com/ (accessed May 22, 2021).

[31] “Monaco Editor.” https://microsoft.github.io/monaco-editor/ (accessed May 22, 2021).

[32] “TypeFox/monaco-languageclient: NPM module to connect Monaco editor with language servers.” https://github.com/TypeFox/monaco-languageclient (accessed May 22, 2021).

[33] “supermemo - npm.” https://www.npmjs.com/package/supermemo (accessed May 22, 2021).

[34] “kaisermann/svelte-i18n: Internationalization library for Svelte.” https://github.com/kaisermann/svelte-i18n (accessed May 24, 2021).

[35] “lokalise/i18n-ally: 🌍 All in one i18n extension for VS Code.” https://github.com/lokalise/i18n-ally (accessed May 24, 2021).

[36] “pytest: helps you write better programs — pytest documentation.” https://docs.pytest.org/en/6.2.x/ (accessed May 22, 2021).

[37] “pytest-dev/pluggy: A minimalist production ready plugin system.” https://github.com/pytest-dev/pluggy (accessed May 22, 2021).

[38] “Typer.” https://typer.tiangolo.com/ (accessed May 22, 2021).

[39] “Welcome to Click — Click Documentation (8.0.x).” https://click.palletsprojects.com/en/8.0.x/ (accessed May 22, 2021).

[40] “willmcgugan/rich: beautiful formatting in the terminal.” https://github.com/willmcgugan/rich/ (accessed May 22, 2021).

[41] “NowanIlfideme/pydantic-yaml: YAML support for Pydantic models.” https://github.com/NowanIlfideme/pydantic-yaml (accessed May 22, 2021).

[42] “Home - structlog documentation.” https://www.structlog.org/en/stable/index.html (accessed May 24, 2021).

[43] “Keycloak.” https://www.keycloak.org/ (accessed May 23, 2021).

[44] M. Bucek, “Secure Svelte with Keycloak - DEV Community.” https://dev.to/matejbucek/secure-svelte-with-keycloak-42g3 (accessed May 17, 2021).

[45] “python-keycloak · PyPI.” https://pypi.org/project/python-keycloak/ (accessed May 23, 2021).

[46] “Firecracker: start a VM in less than a second.” https://jvns.ca/blog/2021/01/23/firecracker--start-a-vm-in-less-than-a-second/ (accessed May 23, 2021).

[47] “weaveworks/ignite: Ignite a Firecracker microVM.” https://github.com/weaveworks/ignite (accessed May 23, 2021).

[48] “CNI.” https://www.cni.dev/ (accessed May 23, 2021).

[49] “openapi-generator/typescript-fetch.md at master · OpenAPITools/openapi-generator.” https://github.com/OpenAPITools/openapi-generator/blob/master/docs/generators/typescript-fetch.md (accessed May 23, 2021).

[50] “dmontagu/fastapi_client: FastAPI client generator.” https://github.com/dmontagu/fastapi_client (accessed May 23, 2021).

[51] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, “Local-first software: You Own Your Data, in spite of the Cloud,” in *Onward! 2019 - Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2019*, Oct. 2019, pp. 154–178. doi: 10.1145/3359591.3359737.

[52] “unhosted web apps.” https://unhosted.org/ (accessed May 23, 2021).

[53] “Deno - A secure runtime for JavaScript and TypeScript.” https://deno.land/ (accessed May 23, 2021).

[54] “Isomorphic JavaScript - Wikipedia.” https://en.wikipedia.org/wiki/Isomorphic_JavaScript (accessed May 23, 2021).

[55] “colinhacks/zod: TypeScript-first schema validation with static type inference.” https://github.com/colinhacks/zod#comparison (accessed May 24, 2021).

[56] “oakserver/oak: A middleware framework for Deno’s native and std/http server, and Deploy 🐿️ 🦕.” https://github.com/oakserver/oak (accessed May 25, 2021).

[57] “Generate types for bundles · Issue #3385 · denoland/deno.” https://github.com/denoland/deno/issues/3385 (accessed May 24, 2021).

[58] E. de Bono, *Six thinking hats*, Rev. and Updated. London: Penguin, 2000.

[59] “Six Hats Thinking | Better Evaluation.” https://www.betterevaluation.org/en/evaluation-options/six_hats (accessed May 25, 2021).

[60] “The Prime Directive - Agile Retrospective Resource Wiki.” https://retrospectivewiki.org/index.php?title=The_Prime_Directive (accessed May 27, 2021).

[61] “xkcd: Nerd Sniping.” https://xkcd.com/356/ (accessed May 26, 2021).

[62] “WhatIsCodingDojo - Coding Dojo.” https://codingdojo.org/WhatIsCodingDojo/ (accessed May 19, 2021).