

Finals Report




Github: <https://github.com/Mutestock/snoozebox-py>

Name:	Henning Wiberg
Cph Id:	hw98
Education:	Software Development
Semester:	3.
Report deadline:	11-10-2022 12:00
Supervisor:	Todorka Stoyanova Dimitrova

Abstract	4
Target Audience	4
Terminology	5
Introduction	6
Design	8
State of the Art	8
Templating	8
Code Generation From Schematics	10
IBM CICS	11
Other Templating Tools	12
Requirements Specification	13
Methodology	13
Client Analysis	13
Functional Requirements	14
Non-functional Requirements	16
Technology	18
Service Types	18
Kafka	18
RabbitMQ	18
gRPC	19
REST	19
SOAP	20
Database	20
MongoDB	20
Cassandra	21
Postgres	21
Redis	21
MySQL	22
Language	22
Python	22
Rust	22
Typescript	23
Java	23
DevOps Ecosystem	23
Docker	23
Docker-Compose	24
Kubernetes	24
Github Actions	24
Architecture	25
Architecture of Snoozelib	25

Architecture of Snoozebox-py	26
Architecture of Generated Applications	28
Microservice Architecture	29
Violation of Microservice Principles	30
The Schematics Problem	30
Modular Building Blocks	31
Asynchronicity	34
Business Model	34
Licensing	34
Implementation	35
Language	35
Dependencies	36
Poetry	36
Docker Portability	37
Package Security	38
Docker	39
DBeaver	39
Internal dependency management	40
Testing	41
Snoozebox CLI Testing	42
Snoozelib Testing	43
Generated Project Testing	44
Automated Testing	44
Compatibility	45
Actual User Experience	46
Documentation	49
Versioning	50
Snoozebox Distribution	52
Maintenance	53
Workflow	54
Brainstorming	54
Continued Development Guidelines	56
Possible Extensions	56
CLI	56
Adding Database Support	57
Connection Templates	59
Logic Handler Templates	60
Service Templates	61
Misc Templates	61
Utils Templates	61
Examples Templates	61
Model Templates	62

Template Management	63
Snoozelib	64
Adding Service Support	66
Connection Templates	66
Logic Handler Templates	66
Misc Templates	67
Model Templates	68
Protogen Templates	68
Service Templates	68
Template Management	69
Supporting Additional Programming Languages	69
Rust Support	70
Typescript Support	72
Java Support	72
Continuing Development from Generated Projects	73
Generated Gateways	73
Retrospection	74
Conclusion	74
	

Abstract

Arguably REST APIs and other services have at its core one specific major difference between them: Schematics. This is because the schematics are deeply tied to the purpose of the individual service. I.e. the defining factor which differentiates one API from another is often directly dependent on the context. Developers might be tempted to recreate services from the bottom or from a base where significant amounts of code has to be written.

In actuality, the services we write can largely be reused whenever a new product must be made as a means of saving resources.

Snoozebox is an experimental templating CLI tool which creates services from schematic definitions. Part of its purpose is to be compatible with multiple schematic formats such as table definitions in SQL or simple JSON. Snoozebox assumes that even more of the code's logic is boilerplate code, and generates it for you on execution.

Service types (REST, gRPC, Kafka, etc) and database technologies (Postgres, Redis, MongoDB, etc) are split into building blocks which can be swapped upon initial generation.

Target Audience

The target audience of this report is other software developers. Whilst there is heavy usage of hyperlinks in the report, it still requires knowledge about web development. Basic concepts won't be described.

As for the targeted user base of the product itself: This is also for other developers. Although the testing part of the generated projects is also covered, this is meant to be a convenience for architects, students and developers alike.

Terminology

Service type: Technologies which are capable of transmitting data either between services, or to a client, i.e. frontend. Examples include REST, gRPC, Kafka, RabbitMQ, SOAP, RPC implementations, etc.

Schematic: Files containing custom data-structures. This could be .sql, .cql, .json, etc. Usually contained in clusters with one definition in each file. Each cluster gets loaded during the CLI generation process.

Generation process: The code which gets executed with the tool's append function.

Init root: The directory in which the init CLI command has been executed. Contains the schematics directory, the services directory, docker-compose file, and the snooze file.

Model: Classes which are code equivalents of entities, with entities being records stored in databases. E.g. if a Person entity is in a database, then the model would be the code equivalent of the Person entity in the database. This term is found more often in the MVC architecture, where the acronym is Model-View-Controller. This architecture is non-applicable to this assignment, as they revolve around statically generated visual content, usually HTML. Examples of MVC frameworks are [Ruby On Rails](#) (Ruby), [Django](#) (Python), [Laravel](#) (PHP) and [Phoenix](#) (Elixir).

Introduction

There's a shift away from monolithic architectures and towards microservices. In the year 2020 alone, adoption of microservices in the cloud for newly created applications grew by 22.5%. The prediction was that by 2022, 90% of all newly created applications would follow some variation of a microservice architecture¹. AWS defines microservices as follows:

*“Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams.”*²

Small independent services, means small independent applications. Small applications means several internal architectures. A finished product will, in the microservice architecture, consist of multiple services which fulfil a single purpose, but each component; each service, in the network (e.g. Kubernetes) of services have individual responsibilities. These services can contain a variety of languages and technologies depending on what the purpose is. I.e. an API gateway component won't necessarily use the same language as your internal microservice which uses gRPC to connect and communicate with the gateway and/or other microservices. It depends on the architecture of the individual project.

With specific components, comes specific technologies. With specific technologies, comes specific specialisations. With specific specialisations, comes different developers. Etc. This ends up in a variety of preferences and internal architectures of the microservices themselves - Uniform standards are at risk of getting lost. This means transferring developers between microservices requires temporal expenditure with the reintroduction into a team's interpretation of what the internal architecture of a microservice looks like. It becomes even harder to preserve the same individual internal service architecture standards in a microservice architecture which makes extensive use of its polyglot nature.

Template based services are entire applications which follow a set of standards defined by the author of the templates themselves. This allows a team to define the structure of all microservices as each of them complies with the standards the templates introduce. This means elimination of the rewriting of boilerplate code for each microservice, and at least partially the need to dissect older services for code which satisfies whatever requirements a scenario could call for.

If in this scenario a network of microservices complies with the same internal set of architectures, i.e. each API gateway or service bus has the same internal architecture, another fact becomes apparent - The primary differences between the services become the schematics which the microservice handles, as well as how those schematics are handled. I.e. the difference between internal microservices which interact with the same database paradigm, becomes different primarily in the schematics they handle, and how they handle it before it is transmitted to the other microservices in the network or exposed to potential frontends.

¹ <https://www.charterglobal.com/five-microservices-trends-in-2020/>

² <https://aws.amazon.com/microservices/>

Snoozebox proposes that even more boilerplate code can be generated automatically with templating solely on the basis of providing it with schematics, and whatever purpose and technologies you may desire. The handling of the data-structures are more subject to the individual purpose of the service, however snoozebox will provide the basics like CRUD handlers and routes thereof where applicable.

Another point is, that by generating fully functional, yet barebones, services is useful for educational, and theorycrafting purposes. I.e. getting an idea of what a service with a specific set of technologies might be handled for the specific schematics that you, the individual developer, might be tasked with handling/already handling.

Time is a non-refundable and finite resource. Spending it to rewrite arbitrary boilerplate code seems ill advised. That's why Snoozebox would be a useful tool.

Design

Snoozebox isn't something that I've seen before. There are plenty of templating tools in the industry, but none of those I've seen are using this exact concept.

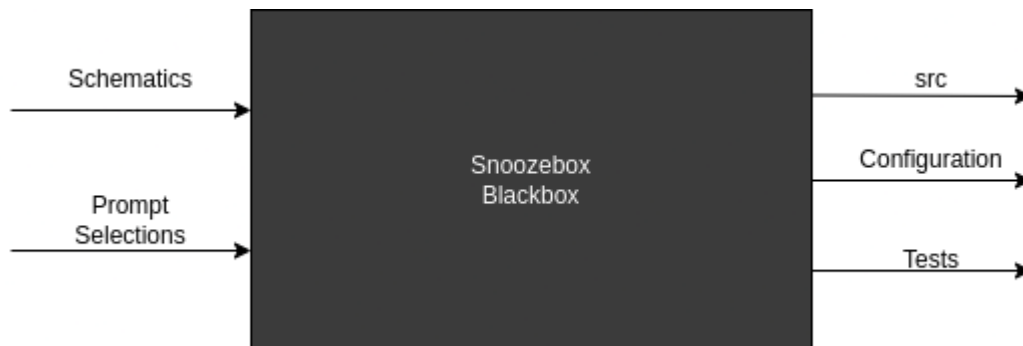


Figure 1: Snoozebox blackbox visualization

A visualization of the concept can be seen in Fig. 1.

There are two inputs:

- Schematics (SQL/CQL/JSON/Cypher/etc)
- Prompt Selections. Name of project, database selection, service selection

There are three outputs:

- Source code. The generated project's code itself.
- Changes in configuration. Snooze file and docker-compose file, which makes the new service compatible with existing ones.
- Integration and performance tests get appended to the test suite. Unit tests are not included in the test suite, and they can instead be found in the source code's individual test directory.

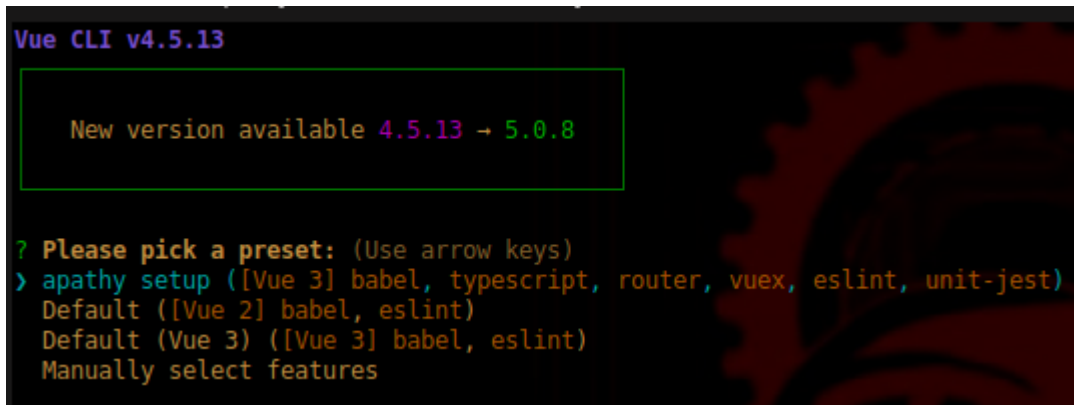
This chapter will go into detail with all of the concepts which were laid out and considered before the project began. Note that it is very optimistic, and not all of the features were implemented in the end.

State of the Art

The concept of this project is slightly different from what already exists. The idea isn't based on any single product that already exists. Instead it was created after observing the similarities in services which were produced by the same programmers repeatedly.

Templating

Templating isn't an uncommon concept. Code generation is done by every CLI tool which creates an initial project for the programmer to work with. Common examples would be the CLI tools which accompany frameworks like [Vue](#) or [React](#), build automation tools like [Maven](#) or package managers like [Cargo](#) or [Poetry](#). Technologies such as Vue's CLI tool is also reactive in the sense that it generates components, changes imports, sets variables, etc. depending on the input parameters submitted during generation.



```
Vue CLI v4.5.13

New version available 4.5.13 -> 5.0.8

? Please pick a preset: (Use arrow keys)
> apathy setup ([Vue 3] babel, typescript, router, vuex, eslint, unit-jest)
  Default ([Vue 2] babel, eslint)
  Default (Vue 3) ([Vue 3] babel, eslint)
  Manually select features
```

Figure 2: Vue CLI showcase

This isn't atypical for frontend frameworks. It's a different story for backend frameworks. But there are some outliers, e.g. [Spring Initializr](#). This online tool supplies the user with a GUI which can be used from a website instead of a GUI tool.

Snoozebox is different from all of these. Adding the vuex or the router library from the Vue CLI tool are relatively minor changes compared to the ones taken by Snoozebox. An equivalent to adding Vuex in a Vue project would be adding [Flask-RESTPlus](#) support in a REST service using a three-tier architecture using Python. But Snoozebox building blocks define the entire foundation of generated projects. For example in Snoozebox the default RDBMS templates use the SQLAlchemy ORM. That logic is discarded entirely if you want to connect with NoSQL databases like [MongoDB](#). This changes the entire foundation of the project. At the same time database technologies and service type technologies are split into completely modular blocks, which can be swapped around freely. With Snoozebox the entire nature of the generated project changes, since the way technologies like gRPC works are vastly different from technologies such as Kafka.

Regardless of the database paradigm chosen, the generated project will always have a typical basic three-tier architecture.

It is also different because it supplies context between generated projects via. docker-compose. This means that the tool to some extent ensures that the generated projects can be executed in tandem with one another.

But the largest difference lies in the base premise of the tool - The way it interacts with schematics. This means the programmer only needs to have the schematics figured out before getting an runnable service where the major technologies can be swapped depending on necessity. Not only does this allow the user to get a fast initial foundation which is constant between each generated service for production purposes, but this also allows rapid theorycrafting on how the data itself should be used. Additionally, this could also make for fast showcasing of how the implemented technologies function in practice, such as if someone wanted to see gRPC equivalent of a service in action, then the user would just need to supply the tool with the supplied schematics (e.g. SQL "create table" queries.)

Code Generation From Schematics

Snoozebox utilizes schematics to create code. We're referring to complex schematics which the different database paradigms can understand.

The primary example is a simple 'Create Table' script.

```
create table example(  
    id serial not null,  
    thing varchar(255)  
);
```

Figure 3: SQL data structure definition

The equivalent Python class looks like this with Sqlalchemy in Python:

```
class Example(Base):  
    __tablename__ = "example"  
  
    def __init__(self, thing: str = None):  
        self.thing = thing  
  
    id = Column(Integer, primary_key=True, nullable=False, autoincrement=True)  
    thing = Column(String(255))
```

Figure 4: Equivalent of SQL data structure definition example

This principle is applicable through all supported languages and technologies.

Of course this isn't actually reverse engineering in the sense that we don't want to be dependent on a database to create our program. I.e. we don't want to have an instance of Postgres running, then execute the sql script and then do reverse engineering with existing software like [sqlcodegen](#). Instead we'd want to generate the code from the SQL script by itself without being dependent on an existing database connection. It's closer to what you can find on [CodVerter](#), but with integrated relationships, sqlalchemy and generated code which uses the aforementioned sqlalchemy functionalities for executing stored procedures, triggers, etc.

To clarify: with software such as MySQL workbench you can reverse engineer existing tables and retrieve SQL code equivalents. That is not what Snoozebox does. You don't need an existing database to use Snoozebox. However, the SQL code the workbench generates *can* be saved in the schematics directory, and Snoozebox will create a service with a service type of your choice which is built around that SQL code.

IBM CICS

The closest existing technology to Snoozebox-py I could find is a specific feature within [IBM CICS](#). On [Wikipedia](#) it is defined as:

“IBM CICS (Customer Information Control System) is a family of mixed-language application servers that provide online transaction management and connectivity for applications on IBM mainframe systems under z/OS and z/VSE.”

It should be noted that programs used with IBM CICS are specifically suitable for being deployed to the CICS server, whereas Snoozebox-py simply uses docker-compose to map some default values, and then Snoozebox's job is over. With Snoozebox it's the user who gets to define how the service is to be deployed - The product is just a normal service; it's a barebones template project which is able to execute with some basic functionalities. Batteries are included up until the point of deployment. CICS projects seem to be generated with deployment with the asterisk that the generated service only works for CICS, and it's proprietary. So it's a different marketing strategy versus Snoozebox-py's open-source strategy as well. Additionally, the generated CICS applications are only functional on z/OS and z/VSE which are operating systems made for IBM mainframes. Needless to say this aspect is the polar opposite of Snoozebox-py, where portability, accessibility, as well as ease of use are valued much higher than proprietary gain. Services generated with Snoozebox need to be usable outside of a Snoozebox context.

Note that CICS is 53 years old.

The part which is most similar to Snoozebox is its ability to generate deployable programs from JSON. There are some notable differences, however.

In Snoozebox:

- The schematics directory in an init root can contain any schematics type.
- All services and databases run in tandem with each other. Services are not connected to each other.
- Deployment is not included. Docker-compose setup is ultimately voluntary. The user can deploy it on whatever platform they want.
- The list of supported methods (CRUD) are defined by the template developers. It's configurable on the snoozefile, but only by exclusion. I.e. the user can't request a PATCH method if PATCH hasn't been implemented in a template. But the user can exclude CREATE in the snoozefile, in which case it will be excluded if it has been implemented in a template.
- Anyone can design templates.*³
- It's open-source. Everything is full-disclosure*⁴
- The process is done through a CLI tool with simple prompts.
- Services created with Snoozebox-py are executable and functional directly after the generation process: Make your init root, define your schematics, run append to start the generation process, run the docker-compose file, and you have a fully functional basic service with CRUD functionalities up and running.
- Project name and other variables are chosen during append prompts.
- Snoozebox is designed to support a microservice architecture, and not an external monolith

³ In the final fully realized version of the project, templates are uploadable and downloadable through a portal and the CLI tool, not unlike Github

⁴ Except if an alternate monetary route is taken with proprietary templates. Will be discussed later.

Other Templating Tools

I've opted to use [Jinja2](#) for my templating. When using files with a .jinja or .jinja2 or .j2 file extensions they refer to the Jinja2 library. For example when a file in the project repository contains [.py.jinja](#) in the file extension, then it means that the jinja template generates a python file. There are also examples of [.sql.jinja](#) files, [.proto.jinja](#) and [.sh.jinja](#) files in this project.

I didn't always use this approach. My first solution was to package the code generation into several "writers". They were essentially just formatting strings, and they had an abstract class as well.

```
indent_writer(  
    lvl=0,  
    text=f"""  
        from sqlalchemy import create_engine  
        from sqlalchemy.ext.declarative import declarative_base  
        from {utils}.config import CONFIG  
        from typing import Optional, Any  
  
        PG_CONFIG = CONFIG["database"]["postgres"]  
  
        conn_string = f'postgresql+psycopg2://{PG_CONFIG["usr"]}::{PG_CONFIG["pwd"]}  
        engine = create_engine(conn_string, pool_size=20, max_overflow=0)  
        Base = declarative_base()  
  
        def db_init() -> None:  
            Base.metadata.create_all(bind=engine, checkfirst=True)  
  
        def db_drop() -> None:  
            Base.metadata.drop_all(bind=engine, checkfirst=True)  
  
        def exec_stmt(stmt) -> Optional[Any]:  
            with engine.connect() as conn:  
                result = conn.execute(stmt)  
                return result  
    """,  
    file_writer=file_writer,  
)  
file_writer.close()
```

Figure 5: Legacy template writer example

It was always my perception that this approach was primitive, yet I couldn't figure out what the correct solution would be - I would even refer to the files as templates.

Eventually I remembered how [Flask](#) manages its [templates](#), and figured out that that templating method, although uncommon, wasn't exclusively for .html files.

The old Snoozebox templating structure can be seen in the [github repository's legacy branch](#).

Requirements Specification

Methodology

This project is using a simplified version of SCRUM, in the sense that certain ceremonies of it have been cherry-picked and applied as deemed appropriate. The fact of the matter is, that this is a one-man project and that most of the ceremonies can't be applied. Examples of features which can be implemented would be the SCRUM board, and some parts of the brainstorming process. However, roles like SCRUM masters and product owners can't exist without a group. This also takes away many of the points from ceremonies like dailies, retrospectives and sprints, though they can arguably still be applied with some success albeit arguably strongly reduced, which has been the case for myself.

Client Analysis

Since this project wasn't created in collaboration with a firm, there isn't much to analyse here. This is also because of the fact that the product that a developer can use to create services independently. One of the initial ideas of Snoozebox was that the Snoozebox was able to store a structure which would be continuous through all future generated projects. E.g. if the "models" directory should be named "entities" instead, then this could be configured inside the Snoozebox. This would allow some settings to be specific for entire firms at a time. This feature was down-prioritised and eventually scrapped.

An example of what Snoozebox could be compared to would be Vue, Sphinx, a JSON parser library in Python, etc. Snoozebox is a contribution to open-source and shouldn't be owned by any firm in particular. Much like most libraries for most programming languages, however, Snoozebox is something any firm could use free of charge if they deemed that it'd be beneficial to do so.

A final example if it is necessary. The last internship I attended was at a firm which makes Java services in JaveEE. The equivalent of Snoozebox isn't a service created with JaveEE. Instead it is JaveEE itself. Snoozebox is a tool which anybody can use to create services. It is not itself a service.

The client is therefore other programmers. In extension, since we want the software to be as advanced and feature-rich as possible, this project would also be open-source.

Clients could use Snoozebox for theorycrafting to look at how a service could look like with their schematics, to see a service in a larger connected cluster of services with docker-compose, or they could simply use it as a starting-point for their new services. After some statistics over how people would use the tool (in a scenario where Snoozebox is an actual complete and distributed tool), the nature of the tool itself would likely shift in whichever direction deemed most relevant by the user base. But with the tool being in less than its infancy, and the environment of programming being as volatile as it is, it is difficult to exactly specify the final nature of Snoozebox (in a scenario where development would continue post-examination).

Functional Requirements

The table below is a merge between Agile concepts by equalising the title with the use stories.

Use Case	Priority	Acceptance Criteria	Story points
<p>As a client,</p> <p>I want to be able to use custom schematics,</p> <p>So that I may generate services which are specific to my own interests</p>	Major	<ul style="list-style-type: none"> • Schematics directory must be available post-initiation • There should be support for multiple different sets of schematics. Likely with multiple directories • The tool should be able to differentiate between data-structures, i.e. only directories with SQL should appear if the selected database is postgres. 	5
<p>As a developer,</p> <p>I want to have a simple way to add support for new technologies,</p> <p>So that the tool can cover the needs that are of use to me, or that is of interest to me for any reason.</p>	Major	<ul style="list-style-type: none"> • The implementation system should be modular. • Some universal templating mechanism should be used. • The tools required should be an external library with abstract classes and interfaces. External meaning, that these tools should be independent from the primary tool. These tools must be easy to acquire and implement. 	8
<p>As a developer,</p> <p>I want support for interpretation of SQL,</p> <p>So that I may implement RDBMS technologies such as Postgres, MySQL, MSSQL, etc.</p>	Major	<ul style="list-style-type: none"> • A separate library needs to exist. This library must contain tools for generating Python code from SQL code. • The output should be objects for generating Python code. These objects must be usable in the templating context. • The tools must be able to interpret all relevant data types in the chosen database technology. • The tools should be flexible as newer developers may not choose to use the same Python libraries as those other database technologies use. 	13
<p>As a client,</p> <p>I want to be able to use my services in a docker context,</p> <p>So that I can be sure that my services will</p>		<ul style="list-style-type: none"> • Each service should have their own generated Dockerfile with the necessary instructions for executing the service. These instructions must be added dynamically • Services added should be able to be coexisting, but they must not be codependent. • Ports between services must be unique 	8

work on my machine.		<ul style="list-style-type: none"> Ports used must not be occupied 	
<p>As a developer,</p> <p>I want to be able to add new entries to the prompt sequence</p> <p>So that I may add support for new technologies in the future</p>		<ul style="list-style-type: none"> Dictionaries with prompt options will be present in the files Default settings in a toml format will make a universal method of adding dependencies to new technologies. Dependency concatenation will happen automatically. 	2
<p>As a client,</p> <p>I want to have a method of selecting technologies</p> <p>So that I may generate services with the specific setup that I want to use.</p>	Medium	<ul style="list-style-type: none"> Snoozebox will be a CLI program. There will be input prompts and CLI options to supply Snoozebox with technology choices 	3
<p>As a client,</p> <p>I want to be able to define which methods (CRUD) I want to use,</p> <p>So that the services I generate won't be filled with bloat I don't need.</p>	Small	<ul style="list-style-type: none"> Make exclusion functionality in the Snooze file. Checks for exclusion of methods and subtract them from the default methods of the specific service type 	2

Table 1: Functional Requirements

Non-functional Requirements

- Security
 - All generated projects should provide functionality for hashing and salting whenever a variable named “password”, “pwd”, etc is found during the generation process.
 - The hashing algorithm used should be mature and tested. BCrypt or Argon2 is preferred.
 - All libraries should have well-defined versioning to ensure that the software won't break on a longer timeframe. In extension, Github must be able to capture the package vulnerabilities as they occur.
 - Fields inserted which are recognized as passwords shouldn't be recorded normally. Caching should exclude raw submissions. At no point should the password be inserted into the database in its pure format.
 - Support for API keys should be created for generated services, yet not implemented into any significant route as usage of such a feature is optional. A route for example usage should be inside a set of routes which are separate from the CRUD functionalities.
- Testing
 - Code coverage of all libraries and the CLI tool itself should be as high as possible. Preferably 100%
 - Generated services should have integration testing implemented between services.
 - The testing frameworks used should be the same across all products of the code, as well as generated services. E.g. pytest with Python, native options for both Rust and Typescript.
- Robustness
 - Minor errors from the side of the client should be caught, and a significant and telling response should be returned in all possible scenarios where such a thing might occur.
 - E.g. submitting an empty project name during generation shouldn't break the service, but rather inform the user that they must try again.
 - There should be custom exceptions for all scenarios where it makes sense to have them to rapidly, efficiently and correctly fix errors as they may occur.
 - Requirements of technologies used in a Docker context should be defined clearly. These requirements must be generated onto the readme file of generated projects.
 - Technologies used with Docker should be stretched to their minimum hardware usage to further ensure that the client is able to run the services. E.g. Cassandra memory usage should be stripped to below 4gb RAM in total from 3 nodes.
 - Generated services should, on a machine which upholds the defined requirements, have on average a 95% uptime.
 - CRUD functionality on successfully generated services on localhost should work 100% of the time.
- Compatibility
 - Services generated with Snoozebox should work on Linux and Windows when executed manually.
- Portability
 - Services should be completely executable in a Docker context.
 - Services run in a Docker context must work on all major operating systems.

- Python projects must be set up in a structure which correctly supports and expects users to use virtual environments and encapsulation of packages to prevent internal conflicts in global packages on the client's machine.
- Performance
 - Generation of projects should take no longer than 10 seconds (excluding prompts)
 - Execution of docker-compose command should take no more than 1 minute on an average machine per added service (excluding the fetching of docker images). This is mostly a note to not clutter the docker-compose file with unnecessary images.
 - Basic CRUD functionality should take no longer than 0.5 seconds in a localhost environment
 - Basic CRUD functionality on a cached value should take no longer than 0.1 seconds in a localhost environment
 - A service generated with Snoozebox should by itself use no more than 30 MiB of RAM (excluding docker images such as databases)
- Usability
 - The prompt sequence should take no more than 30 seconds for an average user to use.
 - The prompt sequence should have no more than 10 input prompts.
 - Optional CLI command options should be able to replace prompt sequence input prompts.
 - An additional CLI command should be able to print the output of Snoozebox-py's interpretation of data-structures from the schematics directory.
- Scalability
 - New services must be able work in tandem with each other
 - New services must never use the same ports
 - Adding new models and routes to a generated service must not make the existing logic dysfunctional
 - Templates must adhere to the low coupling, high cohesion principles to allow for simple addition of the template database
 - Templates must be uploadable and downloadable to an external server.
 - The CLI tool must be able to upload and download templates to the server for ease of addition of new templates.

Technology

This chapter defines basic knowledge about not only the actually implemented technologies, but also the planned technologies or technologies which are relevant to the project. Additionally, these subchapters will explain why they were or weren't chosen.

Service Types

Service type as in the technologies used for communication between applications, or exposed to an external front end, e.g. web applications, mobile phone applications, etc.

Kafka

In event-driven architecture, Kafka is one of the main platforms. Specifically it uses event streaming, which is the practice of capturing data in real-time from event sources. These event sources could be databases, cloud services and software applications. It uses a pub/sub strategy, i.e. various actors either publish (write) data onto an event stream, or subscribe to (read) event streams. It's action and reaction. The log of immutable data is stored inside "topics". This way of storing data is not unlike having an internal database within Kafka's core.

Why:

Kafka is widespread, and extremely popular in the world. In the Stackoverflow insights 2022 survey, 11.96% of all professional developers rated Kafka as their favourite technology in the "Other frameworks and libraries" section, which is just above Tensorflow and just below Flutter⁵. This technology is especially relevant in the microservice architecture. Implementation of it in a Snoozebox context is feasible, yet not as simple as REST. Note that in a Snoozebox context, additional Docker containers are connected to the docker-compose file which generated projects with Kafka connect to.

RabbitMQ

RabbitMQ is an open-source distributed message-broker. It supports concepts like Pub/Sub and topics, and is primarily known for its work queues. It supports a variety of protocols, perhaps most notably AMQP (Advanced Message Queuing Protocol) and MQTT (Message Queue Telemetry Transport). As such it is very often used as a lightweight middle-man between services in a microservice architecture.

Why:

Much like Kafka, RabbitMQ is used in event-driven architectures, but as more lightweight and as an implementation of the Message-Oriented Middleware concept, versus Kafka's more heavyweight stream-processing event store. Notably, RabbitMQ is pulled from docker over a billion times, alongside other containers such as Ubuntu, Redis, Python, Node and Postgres - The most downloaded containers in the world. Much like Kafka, in a Snoozebox context the generation process creates a RabbitMQ server which other services in the grid connect to.

⁵ <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-frameworks-and-libraries>

gRPC

gRPC is an implementation of RPC. The acronym means: General-purpose Remote Procedure Calls. Originally it was developed by Google, but was made open-source in 2015. It uses a universal format called protobufs, which is officially supported by all the most popular programming languages, and even more unofficially. It uses this format to connect polyglot services in a microservice architecture, e.g. an application written in Go can communicate seamlessly with an application written in Typescript. Additionally, it was created as a means of connecting devices to backend services with minimal overhead. Paradoxically, one of its primary strengths is also one of its primary weaknesses: It uses bi-directional streaming with http/2 based transportation, since it, as the official gRPC website states:

“It is currently impossible to implement the HTTP/2 gRPC spec in the browser, as there is simply no browser API with fine-grained control over the requests.”

Connecting gRPC to the browser is therefore only possible via. technologies such as an Envoy or nginx reverse proxy. From there it is possible to use the grpc-web Javascript library to connect to Envoy over HTTP/1.1 instead.

Why:

gRPC may not be an optimal choice when creating gateways, but it is arguably superior to REST when it comes to internal communication of polyglot services in a microservice architecture. Since Snoozebox is used for rapidly creating new services from templates, it was always intended to be used in a microservice context, hence gRPC's relevance.

REST

REST or Representational State Transfer is actually an architectural style or a design pattern, before it is a specific technology. It involves sending resources between the server and client in the HTTP standard methods. Consult the table below as a reference to various database technology equivalents to CRUD.

Operation	SQL/CQL	Cypher	HTTP
CREATE	INSERT	CREATE/MERGE	POST
READ	SELECT	MATCH	GET
UPDATE	UPDATE	SET	PUT/PATCH
DELETE	DELETE	DELETE/REMOVE	DELETE

Table 2: CRUD function equivalents

The de facto standard resource format is JSON, but it can technically also be XML, HTML, Toml, or whatever else is required in the given scenario.

Note that with REST comes terminology such as RESTful and HATEOAS. As an architectural style, a REST application is an application which satisfies a variety of constraints before it can be referred to as RESTful. One of such constraints is HATEOAS. This makes use of extra information such as

hyperlinks in the responses. In reality, few services satisfy such constraints as hyperlinks are for users only. Therefore, much to its creator's dismay, most people use REST in a simplified procedure.

Why:

REST is the standard when it comes to serving data from applications. This is especially useful in gateways, but it can also be used as internal communication between services in a microservice architecture context, although with some caveats compared to the alternatives. REST cannot not be covered.

SOAP

Simple Object Access protocol is a way of using remote procedure calls via. HTTP. SOAP messages are written entirely in XML. It is itself a protocol.

Why not:

SOAP is considered the predecessor to REST as the standard. I do not find it relevant enough to develop support for it. Case and point, Rust, a relatively new programming language which is gaining massive traction, has a total of two SOAP libraries. One has been abandoned since before 2015, and the other was last updated in 2020 and has a total of 33 downloads per month⁶. SOAP's relevance remains in existing legacy systems, usually in a monolithic architecture context, which further disqualifies it from implementation in Snoozebox.

Database

As a means of gaining an understanding of what the concept of Snoozebox is capable of, there is an increased focus not in supporting as many database technologies as possible, but to have some implementations inside several database paradigms.

MongoDB

MongoDB is a document-oriented database. Its implementation of documents are called records, which is a data structure composed of field and value pairs. In other words, JSON. These records are stored in collections. Although supported, pre-defined schematics are optional with MongoDB. As a NoSQL database it, like many of the others, focuses more on horizontal scalability, and supports sharding.

Why:

Not only is MongoDB the most used document-oriented database, but it is also the most popular NoSQL database in general⁷. Having the option to use pre-defined schematics alongside document-oriented databases also allows for more comprehensive usage of the Snoozebox concept.

⁶ <https://lib.rs/crates/savon>

⁷ <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-databases>

Cassandra

Cassandra is a wide-column database. Unlike RDBMS technologies such as Postgres, tables consist of a list of nested key-value pairs. Tables get another layer of complexity as it contains not just tables, but also columns hence the name. The data itself is unstructured, yet the schematics remain fixed. As for relationships; joins, foreign keys, composite keys, etc, do not apply here due to how Cassandra handles its collections. Despite this, Cassandra has its own querying language: CQL. Although it strongly resembles SQL, it is far more primitive. Uniquely, Cassandra uses a democratic peer to peer architecture. This means that there are no master/slave relationships. Cassandra achieves horizontal scaling by supplying additional nodes instead.

Usage of Cassandra isn't too widespread. One of the factors in this is by how Cassandra isn't ACID compliant. This limits the use cases it has. For example it would be a terrible choice for banking purposes due to its inconsistency. Cassandra is excellent at writing, worse at reading and subpar at updating and deleting. This makes Cassandra a valid choice with logging, tracking, telemetry, data sets, etc. In other words, the paradigm itself is fairly niche.

Why:

CQL is a decent way of defining a schematic. It can be stored just like SQL and JSON files would. Supporting a wide-column database isn't immediately obvious, but Cassandra is a decent choice if there should be one. With Cassandra, CRUD implementation is still possible, but the U and D aren't recommended. It is important that the user of Snoozebox understand that the database paradigms aren't necessarily immediately interchangeable. Alternatives could be ScyllaDB⁸ or just foregoing the paradigm entirely and include it under a multi-paradigm database technology such as SurrealDB⁹.

Postgres

This is one of the most widespread databases in the world, alongside other RDBMS technologies such as MySQL, Oracle, MSSQL, etc. It is completely ACID compliant, mature, both horizontally and vertically scalable and supports both partitioning and sharding.

Why:

Snoozebox needs to support an RDBMS. Postgres has excellent support on Docker, and is simple to set up and configure, which is not only an advantage for the developer of the templates, but also the user who generates applications.

Redis

As a data structure store or key-value store, Redis is primarily used as a cache. Performance is the main interest with Redis, and all transactions are by default in-memory only. As such, Redis aims to be as lightweight as possible, therefore the default Redis setup doesn't have any security whatsoever. Basic Redis is fairly simple: JSON can be stored as a value, which makes a simple dynamic with serialization of objects. It does get more complex when introducing Pub/Sub concepts.

⁸ <https://www.scylladb.com/>

⁹ <https://surrealdb.com/>

Why:

Redis is a different database choice than the others. Caches are very much something that needs to be supported with Snoozebox, be it with Redis or Memcached. However, caching rarely can stand by itself as a concept. If there is nothing to intercept and optimise, then arguably the point is void all together. Because of this, Redis is included with all of the generated applications, simply because it enhances the performance.

MySQL

MySQL is an RDMBS much like Postgres.

Why not:

There is no reason why MySQL support wouldn't eventually happen. In fact it would be relatively easy to implement since the vast majority of the logic can be extracted from Postgres, with both of them being SQL-using RDBMS technologies. However, as a means of expanding the understanding of what the concept of Snoozebox is capable of, there is no real reason why both Postgres and MySQL should be supported in this experimental edition. Postgres won my interest solely due to its simplicity with settings, most notably in Docker.

Language

Python

As a scripting language, Python is often chosen when velocity in development is favoured over performance in production. It uses duck typing and is often considered easy to learn, but hard to master.

Why:

Given a smaller time frame, Python is almost always on the table with independent applications. With duck typing, it is much simpler to develop new templates.

Rust

A new contender to the old systems programming languages, i.e. C and C++. It values performance in production higher than velocity in development. It uses neither garbage collector nor manual deallocation of memory, and instead uses borrowing as a means of releasing memory due to how its compiler works. It is considered rather difficult to learn. Rust itself has a very minimal runtime. Instead it is normal to download runtimes when developing web applications. Interestingly enough, the vast majority of these runtimes and the technologies that circle around them are either asynchronous by default or simply don't support synchronicity.

Why:

Distributing Rust applications becomes significantly easier because of the way it handles binaries. Writing templates is more time consuming, but the results should also be better.

Although personal preference is definitely a factor here. As a side note, Rust has been the most loved programming language for 7 years straight on the stackoverflow survey¹⁰.

Typescript

Typescript is a trans-compiled superset of Javascript. It has been gaining traction especially for backend developers who have been drawn to the extensive size of the Javascript ecosystem.

Why:

Because of the role of generated applications, Typescript is valued higher than Javascript. Additionally, there has been an evolving tendency with building new Javascript/Typescript runtimes powered with systems programming languages in an attempt to further optimise the scene. Notably Deno with Rust¹¹, and Bun with Zig¹² which both outperform Node in many aspects. With this trend, it is likely that Typescript becomes even more relevant as a backend language than it already is.

Java

Java is an object-oriented and mature programming language. It has a large scene in legacy systems. Java is also the primary programming language in the Software-Development education.

Why not:

Java comes with a massive amount of boilerplate code. This code is generated alongside new Java projects generated with Maven. There is no good way to circumvent this in an effective manner. Dependency management is also more complicated. Although personal preference is also a factor here.

DevOps Ecosystem

Docker

Containerization is required in a modern world where we want software to function on all major platforms. Docker is the standard when it comes to containerization. It ensures that the application functions the same way every time regardless of the configuration of the machine it is being executed on.

Why:

There simply are no alternatives. Docker does the job and it does it well.

¹⁰

<https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages>

¹¹ <https://github.com/denoland/deno>

¹² <https://github.com/oven-sh/bun>

Docker-Compose

The name is on the nose. Docker-compose is about composing (or orchestrating if you will) docker services.

Why:

It's an alternative to having to type CLI commands for every service you want initiated. Simply appending new services to the list in the docker-compose file as they get generated allows for extra automation for the user.

Kubernetes

Kubernetes is a container orchestration tool. It is a very powerful instrument when managing a larger network of microservices. Unlike docker-compose, Kubernetes has features such as service discovery, load balancing as well as automated and scheduled rollouts/rollbacks. With Kubernetes you don't just execute a service. You start a handful of "pods", a group of containers which share the same storage settings and network connections. Pods, the instances of your service, are ephemeral and disposable. If one instance of the service in question breaks down, another instance takes its place, and what Kubernetes calls "self-healing" begins for the one that got replaced.

Why not:

Obviously Kubernetes would be a powerful tool if implemented. In fact it would take the place docker-compose has right now. However, Kubernetes is a rather advanced tool. Doing metaprogramming around it is too time consuming as of this point. Besides, implementing it would increase the likelihood of confusion in the user. Sticking to docker-compose seems less overwhelming. I consider it to be unfeasible to implement Kubernetes support for this deadline.

Github Actions

Inbuilt into Github repositories is a continuous integration system. Using Github Actions means not having to use an additional 3rd party, like CircleCI, Travis, etc, for continuous integration. Instead Github Actions is free and it's unlikely that the repository of the project in question will be stored on any other platform than Github since Subversion for the most part has faded into obscurity.

Like most other CI software, Github Actions can trigger deployments, handle secrets, use automated testing, automated push of compiled binaries to the releases / packages page, etc.

Why not:

Time constraints. Github Actions is useful not only for automated testing with the CLI tool itself, but there is no reason why it couldn't be implemented for the generated services as well.

Architecture

Like most other parts of this project, there are three major points in the architecture: Snoozelib, Snoozebox-py and the generated projects.

Architecture of Snoozelib

[Snoozelib](#) has been a bit of an experiment for me. This is because we've not been educated in how to properly write and upload libraries to official online repositories. I.e. in Python it'd be <https://pypi.org>, in Java it'd be <https://mvnrepository.com/>, in Rust it'd be <https://crates.io/>, Typescript/Javascript <https://www.npmjs.com/>, etc. I personally perceive it as something you need to learn at some point as a software developer, so one of my own goals was to create a library with the same kind of structure as libraries other people have written. It wasn't difficult finding examples for these since it's possible to look at any given library in existence, and observe similarities between them. One thing I did observe was common usage of the "[setup.py](#)" file, something you gracefully dodge when using [Poetry](#). Since Snoozelib is a library it also doesn't have a main method. Another thing I picked up rather quickly was to place the logic you expose in the `__init__.py` file; a facade if you will. Though, I could've explored this way of structuring projects further. Either way, this is how you make sure that the methods in your API are connected to the root of your module when you import it:

```
from snoozelib import sql_tables_to_classes
```

Although there are no advanced design patterns used in Snoozelib, there is some fairly complicated logic in it. To clarify - Complicated, not complex. To quote "[The Zen of Python](#)":

"Simple is better than complex. Complex is better than complicated."

Not to undermine my own efforts, but although it works it doesn't do so gracefully. I reckon my inexperience in writing libraries shows, and I've therefore opted to not share the library on Pypi, even though that was one of my goals at the start of the project. The output is, at the moment, only usable in the context of Snoozebox, and while I wouldn't say it is spaghetti-code per se, I'm not confident enough in its quality to upload it to the public. This has the extended downside of the users of the Snoozebox having to build Snoozelib themselves (one extra command with poetry), which can be seen in the chapter about the [intended user experience](#) point 3.

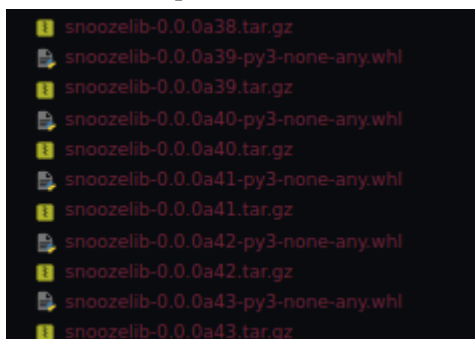


Figure 23: Snoozelib packages created with Poetry

Since we're making packages, there are plenty of older versions of my library which are stored inside my dist directory. As an addendum to my reflection I'd say that I find the dynamic interesting, and I'd like to explore the concepts of proper etiquette and practices in writing decent libraries in the future. Hopefully by then I'll be able to uphold the standards I'm demanding of myself.

Architecture of Snoozebox-py

As a CLI program, the architecture is directed toward providing features for the CLI commands. In this case there are three: “init”, “append” and “describe-sql”. This also means that the [main file](#) does nothing else than initiating the CLI program upon execution.

Apart from some code inside utils which handles things like pathing and internal poetry management inside generated projects, as well as some logic for writing and interpreting the contents of the Snoozefile, the entirety of the project lies inside the gen directory. Gen being short for “generation”. These are files which interact with the templates. The templates themselves are stored within the same directory.

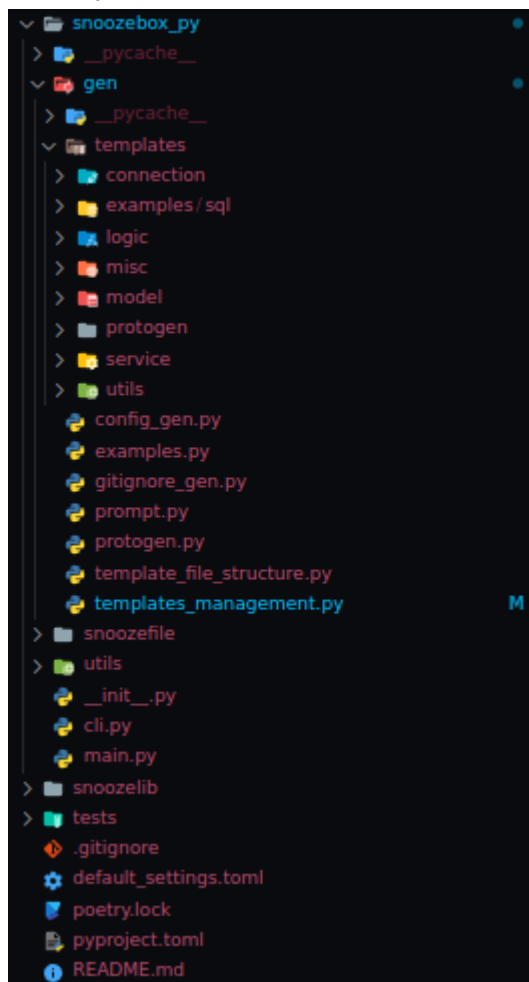


Figure 24: Snoozebox directory structure

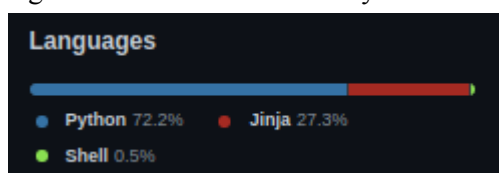


Figure 25: Github language association statistics on repository page

It should be noted that the templates, at this point of writing, contain 27.3% of all the code in the snoozebox project. This remainder of the code includes snoozelib and the other files previously mentioned.

Since the architecture of the project is centred around providing features for the CLI commands, the command “describe-sql” does not interact with the gen directory, but instead communicates with Snoozelib directly and provides the interpretations of it made of the schematics directory. This is to provide the client with insight over whether or not the code inside the schematics directory was written correctly.

The templates directory contains files which are contained within subdirectories which are equivalent to those generated during the generation process. This is primarily for the “append” command, i.e. the command which populates the generated projects for the client, but also includes the examples generated on initiation.

The way jinja files work is that every file inside .jinja extension, preceded with the filetype that the file generates. E.g. [grpc_handler.py.jinja](#) generates a python file, while [proto_file_gen.proto.jinja](#) generates a .proto file ([protobuffers](#)). More about this can be the usage of jinja can be read in the chapter about [other templating tools](#). Descriptions on how to write your own can be seen in the [continued development guidelines](#).

Architecture of Generated Applications

The architecture of the generated projects will in all cases be the same, regardless of programming language and technology choices. They will always use a “three”-tier architecture.

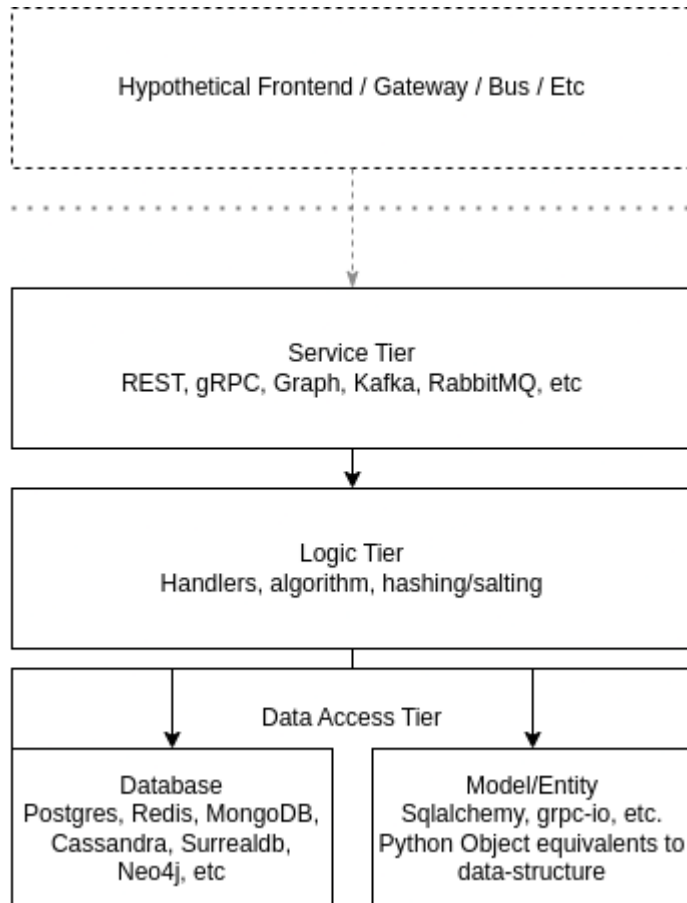


Figure 23: Generated application three-tier architecture

What makes this different from what you would normally argue was a three-tier architecture, is how the database- and model-layer are sharing the bottom tier. I prefer separating these in two when writing services.

The directory structure neatly supports this model as well it should:

There are some additional directories which are dependent on the selected technologies, e.g. the protogen directory when using gRPC.

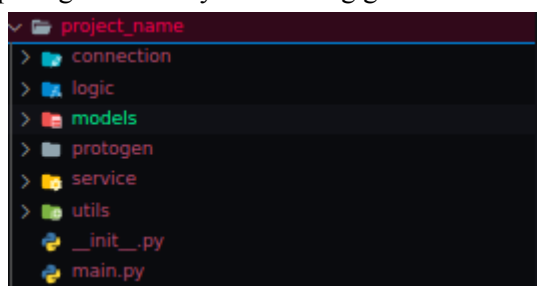


Figure 26: Partial generated project directory structure for protogen showcase

Additionally, in the hypothetical scenario where Snoozebox supports additional languages, the structure of technology specific directories may vary. An example of this is once again with gRPC but with Rust, where the generated code isn't code which you need to export to a directory like it is with grpc-io in Python. Instead, with [Tonic](#), there's a build.rs file in the root of the directory which compiles the protobufs into an importable library. This significantly reduces bloat.

Although currently impossible, some redesign would allow any architecture to be used within generated services.

Microservice Architecture

As a way of showing how the project works in its entirety, when connected to docker-compose see Fig. 27.

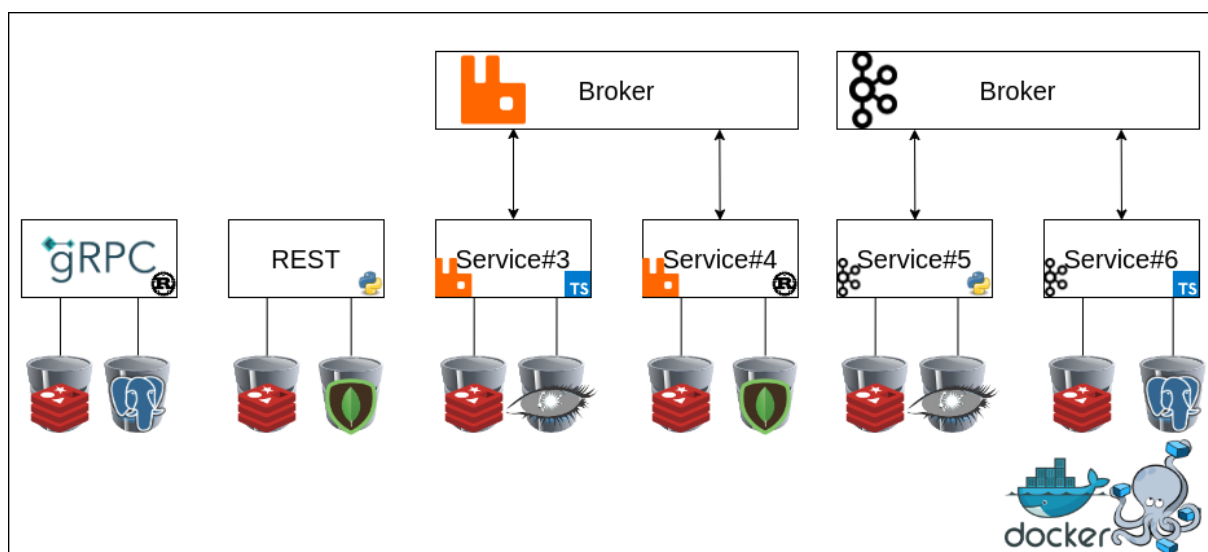


Figure 27: Simplified Diagram Docker-Compose setup

There are some key takeaways. We're making use of several different programming languages to show how microservices support polyglot setups. The service types are built around standards which all state that they can interoperate in tandem with each other regardless of what programming language is used.

Multiple database technologies are used between services. Each language has its own implementation of the database technology in question. Additional database technologies can be implemented through additional templates.

RabbitMQ and Kafka are different, since when a service has been created of their type, they connect to a broker. Regardless of how many services of type RabbitMQ or Kafka are created, they will all connect to the same broker.

Each service gets their own Redis cache, since it significantly increases the performance of all services. This is especially true with RDBMS technologies such as Postgres, MySQL, MSSQL, etc.

Every element seen in the image has its own container.

New services can technically continue being added to the setup until there are no available ports left.

There are no connections between services. If a user wants connections between services, they can create their own gateway or bus between services. That part is not Snoozebox' job.

Readers might ask themselves: Why would you make a setup as the one in Fig. 27? The answer to that question is: You probably wouldn't use that exact one. It's a showcase of how services work in tandem when new services are appended. That said, it ultimately isn't my job to figure out how you would want to use it. Creating the tool is my job. Figuring out when and how the individual wants to use it is not my job.

Violation of Microservice Principles

One of the principles in the microservice architecture is "One or zero databases per service"¹³. This rule has been created as a means of keeping services micro, and because tests show that microservices in general has an approximately doubled chance of failing with two databases connected instead of one. And in fact, there doesn't seem to be much reason why a developer would want multiple databases in the first place.

However, personally I find that to be an odd case with caching in particular. The reason why you want caching is to limit more taxing requests, such as requests against a service with an RDBMS whose response requires a massive amount of joins to conjure. The alternative to the setup above would be to have individual services with Redis as their database. This means that an entire service would be the cache. Services like those would be minimal, and arguably wouldn't make sense in a Snoozebox context. Therefore I reckon that the alternate strategy would be to not use Redis at all.

Another point is the relevance of caching with the various database technologies. There's a good argument to be made that RDBMS should have a cache connected. Cassandra, however, not so much. Entangling and disentangling caching depending on the chosen database technology would be a task which is currently outside of the scope.

The Schematics Problem

One of the biggest problems with the concept is all the other functionalities which can't or will not be supported. It is difficult to ascertain and determine how more complex functionalities should be supported. An example of where problems occur is with Postgres [functions](#) or [the many data types Postgres supports](#). The benefit of using queries of pre-existing querying languages such as SQL, CQL or Cypher is that the user will not have to learn a new format and can use pre-existing queries to generate services with Snoozebox. An issue lies with the conveying of information on how much isn't and can never be supported. E.g. a simple Select query in SQL doesn't make sense conceptually inside Snoozebox. Snoozebox can't use querying languages for querying. Instead it is able to understand specific "create table" queries. Arguably Snoozebox would've been stronger if it had had a single universal and configurable language between all technologies. This could've been something as simple as [toml](#). But this creates a different project entirely, since users will no longer be able to use their older schematics for theorycrafting purposes.

¹³ <https://akfpartners.com/growth-blog/microservice-architecture-principles>

Modular Building Blocks

There are two different building blocks in Snoozebox. These building blocks must work in tandem with each other. Adding support for one database, means that said database support must work with all implemented services. To do this the building blocks need to be truly modular. This is the reason why whatever changes a service type might require on the model layer needs to be injected - That is a chunk of code which by itself is out of context, but it is instead injected into the code in the model code from the Database block.

Database blocks must have:

- A Model base. The database has the final say in how models are defined. For example in the case of Postgres, the contents of models are primarily defined in a way that is relevant to it. This could for example be an ORM being rather invasive.
- Database connectors. These have default settings which are defined inside the `default_settings.toml` file. The same settings are duplicated inside the `docker-compose` file.
- Handler component implementations. The handlers in this project are themselves modular in the sense that we're making use of composition instead of inheritance. These components make use of duck typing to assume that they actually just exist in the same place in the generated directories. Alternatively, we can make use of traits or interfaces to define this constraints.
- Snoozelib implementation. Snoozelib is that the library which generates some objects which Snoozebox can insert into the templates. These are database technology specific, i.e. Snoozelib needs to understand SQL when using Postgres. If there are changes in MySQL from Postgres, then they need to be addressed with Snoozelib. Adding support for entirely different schematics like Neo4j's Cypher would require entirely separate code.
- Technology Specific Logic. This is basically straggler code. These are for the most part to be restricted inside the utilities directory if possible.
- Tests which assures that connection can be established and whatever else that could be useful. Most tests will be unit tests which won't be in the final test suite, but the generated applications themselves.

Database	
Model Base	Database Connector
Handler Component Implementation	Snoozelib implementation
Technology Specific Logic	
Tests	

Figure 7: Database building block visualisation

Service blocks must have:

- Service Specific Model Code Injection. The database base model takes preference in how the model classes are defined. Additional code which the service type in question may or may not require must be injected to ensure modularity, and a code base whose size does not increase exponentially as new technologies get supported.
- Main Method. It is ultimately the service's job to ensure that the correct settings are used. These vary greatly between all service types, e.g. Kafka is run vastly differently than REST.
- Handlers. These use the components defined inside the database blocks, so much of the logic has already been defined. But if there are service type specific instructions about how to handle those database instructions, they go here.
- Routes. How should the information passed from the handlers in the logic tier be served? The preference is that these are as universal as possible, however that becomes near impossible with some technology choices. Again, e.g. Kafka and Rest.
- Technology Specific Logic. A good example is gRPC which requires a lot of logic for protobufs to be used correctly. Meanwhile REST was easy to implement, and didn't require much work at all.
- Tests which test all of the above. Many of these tests get appended to the integration tests in the init root's test suite.

Service	
Service Specific Model Code injection	Main Method
Handlers	Routes
Technology Specific Logic	
Tests	

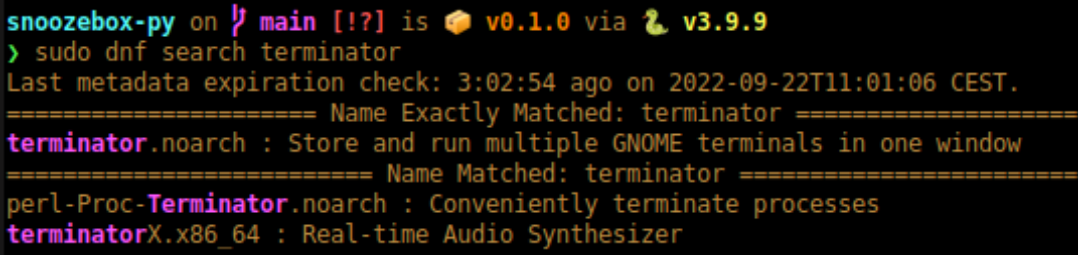
Figure 8: Service building block visualisation

One of the initial concepts of Snoozebox is to have the templates hosted on an external server. This was for multiple reasons. I didn't want the main repository to be bloated with templates. Additionally, if people were to create templates, then I wouldn't want them to be dependent on push/pull requests to the project repository. I'd rather have a setup which would allow them to add their templates, without me having the responsibility to ensure that said templates were functional - That would take too many resources for me to do. Instead the templates should be hosted on an external server, in which people could both download and rate the templates.

To ensure that these templates had the contents required to make a successful service, I'd have each template package implement a specific Python abstract class. I called packages which fulfil the functions "building blocks". Building blocks cover both database and service technologies. The abstract class would be a separate library which would be downloadable via. [Cargo](#), which is Rust's package manager since this was intended to be a Rust project when it started. But the Python alternative would've been [PyPi](#).

The server would have its own portal with a sufficient GUI in which the user can download template packages. This GUI would likely have been created with [Vue](#) or [Nuxt](#) since I have experience with Vue. This server would be hosted on a droplet on [Digital Ocean](#), which I also have experience in using and setting up. I also considered the idea of hosting it on my Raspberry PI, but there are some IP complications.

Additionally, the CLI tool itself would've had the option to search through installable template packages and install them as well. This is not unlike what one would see in Linux package managers such as [Aptitude](#), [DNF](#) or [Pacman](#).

A terminal window with a dark background and colorful text. The prompt is 'snoozebox-py on main [!?] is v0.1.0 via v3.9.9'. The command 'sudo dnf search terminator' has been executed. The output shows the last metadata expiration check and then lists search results for 'terminator'. The results are: 'terminator.noarch : Store and run multiple GNOME terminals in one window', 'perl-Proc-Terminator.noarch : Conveniently terminate processes', and 'terminatorX.x86_64 : Real-time Audio Synthesizer'.

```
snoozebox-py on main [!?] is v0.1.0 via v3.9.9
> sudo dnf search terminator
Last metadata expiration check: 3:02:54 ago on 2022-09-22T11:01:06 CEST.
===== Name Exactly Matched: terminator =====
terminator.noarch : Store and run multiple GNOME terminals in one window
===== Name Matched: terminator =====
perl-Proc-Terminator.noarch : Conveniently terminate processes
terminatorX.x86_64 : Real-time Audio Synthesizer
```

Figure 9: DNF search functionality referencing possible future Snoozebox module management

The idea was to keep everything modular. Snoozebox itself had to be as small as possible, and as independent as possible. I wanted more extensive use of libraries. Building blocks themselves should be user submitted. These building blocks opened up to the idea of potential monetization as well (if that is to be perceived as a necessity), since these building blocks could be made proprietary. Think [Github Enterprise edition](#), but for templates with enterprise specific logic inside them to ensure rapid development of proprietary code.

Obviously all of this was too much work in its entirety for just one person. It was all cut and considered out of scope, but the concept remains.

Asynchronicity

Wanting asynchronous services is a common trend. In fact many of the more advanced architectures recommend using asynchronous services in favour of synchronous ones¹⁴. That said, in order to keep the project simple, asynchronicity hasn't been included inside any of the current services. That said, many of the libraries and frameworks are specifically dependent on asynchronous runtimes such as [Tokio](#). Of course, Rust ended up not being included in the current version of Snoozebox-py, but it is a case of the developers and clients actually not having much of a choice in the matter. For example, if Kafka was one of the services which had ended up getting support, then the [Python library](#) for it uses asynchronicity with its KafkaProducer.

Business Model

This product is not made in cooperation with any firm. Additionally, this project is supposed to be open-source. Therefore, the economy is purely hypothetical to how you could approach the situation in theory.

Licensing

Snoozebox-py is intended to be an open-source project, as has been made apparent [multiple times](#). This would mean that in a real life scenario this would likely fall under the MIT licence. It should be mentioned, however, that the true benefits in using Snoozebox-py in an open-source context, is to expand the amount of features it covers, and not to make additional clones of the library. This does, of course, not mean that we shouldn't give the users the option to fork the project and go off-branch if they deem that they've got the superior concept.

One of the initial ideas was to make the templating modular. Think that Snoozebox would have an [additional library](#) containing little more than a few abstract classes and tools to write their own modules, which could then be synced and downloaded with the user's Snoozebox-py CLI tool. It is certainly something I'd consider in the future. Either way, this structure in further development of Snoozebox, would simplify the process for newer developers, and decrease the size of the Snoozebox-py CLI tool itself. These modules themselves could then be made proprietary, much like repositories in github can be made proprietary and private through additional licensing, which could eventually lead to potential profit with Snoozebox. All of this was, once again, scrapped due to time constraints.

¹⁴ <https://www.torryharris.com/downloads/Web-Services-Synchronous-or-Asynchronous.pdf>

Implementation

The initial idea of Snoozebox always had many technologies which would be unfeasible to be implementable due to the time frame and scope of the examination project. Reversedly, it was never my intention to implement all of them before the deadline. Note, that with the concept of Snoozebox, database and service technologies are modular. In other words, adding support for MongoDB shouldn't require additional work before it would work with gRPC. Likewise, adding support for Kafka shouldn't require additional work before it would work with Postgres.

Database \ Service	Postgres	MongoDB	Cassandra
gRPC			
REST			
Kafka			
RabbitMQ			

Table 3: Implemented Technologies

Adding support for MongoDB, should automatically make gRPC and REST supported with it without additional work required.

Adding support for Kafka, should automatically make Postgres supported.

So on and so forth as we add technologies to the repertoire.

Language

The initial plan was for the project to be written with Rust. Additionally, the project should be able to generate Rust, Python and Typescript(Deno) projects. The primary reason is that Rust is my go to programming language. But since Rust is a compiled language without a run time, it also provides us with a relatively small binary with the memory safety Rust guarantees. The finalised product would be small, fast and secure. The problem with Rust is that it's a complex language. It takes time to produce something with it. Due to the [changes in scope](#) I decided to change the language as well despite already having started out with Rust.

I opted to go for Python because it in many ways is the polar opposite of Rust. It's notoriously inefficient when it comes to execution speeds, but the velocity in development means that it provides results faster which is required when the amount of development time is as limited as it is.

It's important to select the right tool for the job. This time velocity in development was prioritised higher than performance, so I chose Python.

Another thing is the language support. Even though it was always very theoretical, adding support for three different languages with Snoozebox is unfeasible within the timeline. Due to these changes, the title of the project changed to Snoozebox-py, since it is now by Python for Python.

Dependencies

Because of the scale of the project, it's required for me to use certain external software to allow the project to flourish into a product of quality. This chapter takes these into consideration, but also explains the usage of python libraries used in the project and the tactics involved with separating and installing them.

Poetry



[Poetry](#) is a package manager for Python. It's using the newer pyproject standard. It also has an extensive cli tool which allows you to not only generate new projects, make a virtual environment and add/remove/export dependencies; it also allows you to package your project as a wheel/tar archive format which can be uploaded and published on pypi or similar websites. It's a clear advantage over having to manage these interactions manually.

In this project it serves me in several ways:

1. It allows me to make sure that all projects generated has the same structure
2. It gives me access to the cli tool inside the project via. subprocesses, and more convenient addition of dependencies with the pyproject standard.
3. It enforces usage of virtual environments which allows me to further guarantee stability in the generated product.
4. It allows me to effectively and conveniently upload Snoozelib to the Pypi repository.
5. It allows me to set a specifically required Python version.
6. It's simply easier and more convenient to use than having to use venv, virtualenv, pyenv, pipenv, etc and with much more extensive tooling. It ensures that other developers are using the same kind of virtual environment, which decreases the odds of "it-works-on-my-machine" being an issue, i.e. it's a portability solution as well.
7. Pyproject offers effective version control of created libraries
8. Poetry has its own build tool
9. Packages installed with Poetry are subject to extensive obligatory security checks with hash codes - Something that simple requirements.txt files won't offer.

Because of this I've decided that it's a dependency for this project that the developer must have Poetry installed on their machine.

Docker Portability

Poetry's benefits in portability is, among the previously mentioned points, noticeable in the context of Docker whose purpose was already portability. This is because Docker can use the same poetry virtual environment as the developer is using:

```
1 FROM python:3.9-slim
2 COPY . ./app
3 WORKDIR /app
4 RUN apt-get update
5 RUN apt-get install -y
6 RUN python3 -m pip install poetry
7 RUN poetry config virtualenvs.in-project true --local
8 RUN poetry install --no-dev
9 RUN poetry add gunicorn
10 WORKDIR /app/rest_testing
11 EXPOSE 15373
12 CMD ["poetry", "run", "gunicorn", "-b", "0.0.0.0:15373", "main:app"]
```

Figure 10: REST application dockerfile

Therefore the portability isn't just between the developers, but also includes the typical Debian OS Docker uses for its containers. There are very few niche cases where you'd might want to use anything else than Debian, e.g. OS specific operations like C# interactions with [Windows](#), compatibility/security checks between software and OS', or situations where bleeding edge is strictly necessary with OS' such as [Arch](#). Therefore it is unlikely that there's going to be any reason why anything other than apt and python dependencies will be required inside [default_settings.toml](#).

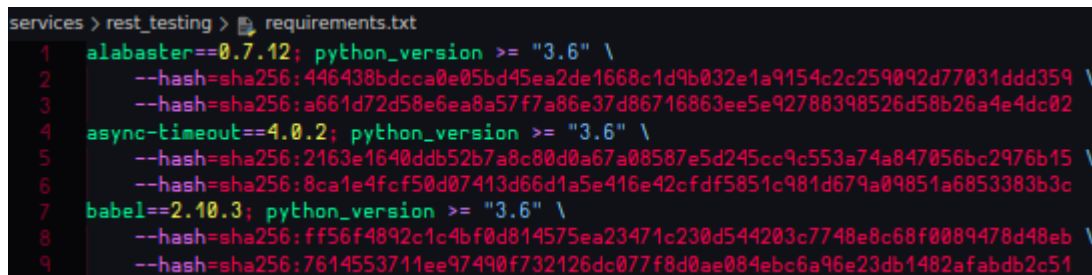
Package Security

Poetry performs many security checks on the packages which it installs. It is fairly easy to see this. One may just use the following command:

```
poetry export -f requirements.txt > requirements.txt
```

This will export the package definitions and requirements defined inside `pyproject.toml` into the typical `requirements.txt` format - `requirements.txt` being the typical format used for `venv` which is the defacto standard in Python package management (excluding using global packages, i.e. the default way of running Python, which more often than not results in dependency collisions and incompatibility between different software and projects).

The output of a typical generated project with REST/Postgres looks like this:

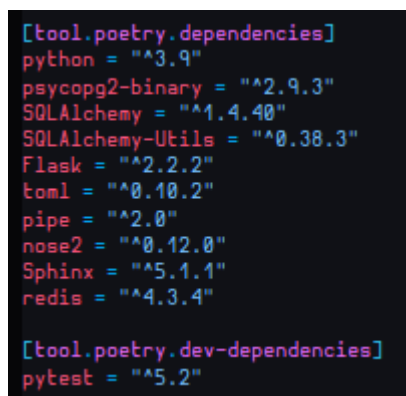


```
services > rest_testing > requirements.txt
1  alabaster==0.7.12; python_version >= "3.6" \
2      --hash=sha256:446438bdcca0e05bd45ea2de1668c1d9b032e1a9154c2c259092d77031ddd359 \
3      --hash=sha256:a661d72d58e6ea8a57f7a86e37d86716863ee5e92788398526d58b26a4e4dc02
4  async-timeout==4.0.2; python_version >= "3.6" \
5      --hash=sha256:2163e1640ddb52b7a8c80d0a67a08587e5d245cc9c553a74a847056bc2976b15 \
6      --hash=sha256:8ca1e4fcf50d07413d66d1a5e416e42cfd5851c981d679a09851a6853383b3c
7  babel==2.10.3; python_version >= "3.6" \
8      --hash=sha256:ff56f4892c1c4bf0d814575ea23471c230d544203c7748e8c68f0089478d48eb \
9      --hash=sha256:7614553711ee97490f732126dc077f8d0ae084ebc6a96e23db1482afabdb2c51
```

Figure 11: Requirements.txt exported dependencies with Poetry

These all validate package versions, hashes and their compatibility with the Python version in question. This is despite the fact there are only a handful of packages used in the entire project, which can be viewed quite clearly in the generated project's `pyproject.toml`.

All of this ensures that the basic services Snoozebox generates aren't using invalid and insecure packages.



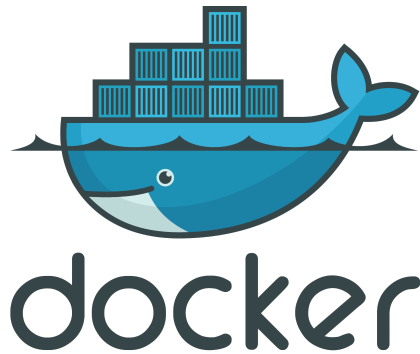
```
[tool.poetry.dependencies]
python = "^3.9"
psycpg2-binary = "^2.9.3"
SQLAlchemy = "^1.4.40"
SQLAlchemy-Utils = "^0.38.3"
Flask = "^2.2.2"
toml = "^0.10.2"
pipe = "^2.0"
nose2 = "^0.12.0"
Sphinx = "^5.1.1"
redis = "^4.3.4"

[tool.poetry.dev-dependencies]
pytest = "^5.2"
```

Figure 12: Poetry Pyproject.toml file

Note that all of this comes with the generated applications as well.

Docker



Docker is the recognized standard in containerization, i.e. the isolation of operating system specific execution of programs while still having non-virtualized access to machine hardware. The advantage is that we're able to add several processes of databases, python instances, etc (including Kafka). By using docker we're able to further ensure that the generated product is stable and consistent across all platforms and machines when executed inside a containerized environment regardless of the configurations of the developer's/server's machine.

In this specific scenario I'm using docker-compose, which is a recipe-like structure which we're easily able to organize in a yaml format. The advantage here is, that as we add new services with snoozebox-py, then we're easily able to append additional services to the docker-compose file. By reading the snoozebox.toml file, we're able to ensure that the format in the docker-compose file only contains the contents that it should and that none of the ports dedicated to each services contradict each other. Read more about the snoozebox [here](#).

Aside from the docker-compose.yml file in the root of the orchestration directory, there's also a dockerfile in the root of each of the generated projects' directories. This generated Dockerfile comes with instructions on how the python program itself should be run. The docker-compose context points to each of the dockerfiles.

```
generated_grpc_service:
  build:
    context: services/generated
```

Orchestrating the projects like this allows for collection of all services for execution, distributing port and connecting all services to the same network.

DBeaver

This isn't as much of a dependency as it is a recommendation. Snoozebox uses multiple database paradigms and technologies. As such, it is likely that the user would have to use several different database client software applications. DBeaver is capable of administrating all of the database technologies Snoozebox supports and many others, NoSQL and RDBMS alike. Using it would most likely decrease the amount of clutter.

Internal dependency management

Each of the building blocks, service types and database types alike, have some connected set of dependencies associated with them which are required to run the program. These dependencies are defined inside the [default_settings.toml](#) file. Apart from that they're separated in debian dependencies and module dependencies:

```
[settings.database.postgres]
host="localhost"
port=15342
usr="snoozebox-pg-user"
pwd="snoozebox-pg-pwd"
db="snoozebox-pg-db"
dependencies=["psycpg2", "sqlalchemy", "sqlalchemy-utils"]
debian_dependencies=["gcc", "libpq-dev"]
```

Figure 13: default_settings.toml file

These dependencies are collected later on in the process. Aptitude dependencies are appended to Dockerfiles, Python dependencies are installed with Poetry during the generation process.

```
def get_apt_dependencies(config: Dict) -> str:
    """Collects apt dependencies to be used in the generated Dockerfile. An example would be gcc with postgres

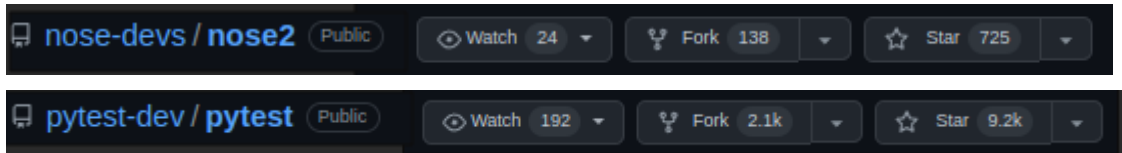
    :param config: Configuration dictionary which gets passed around and modified during the generation process
    :type config: Dict
    :return: Concatenation of all apt dependencies as a string.
    :rtype: str
    """
    database_settings: Dict = config["settings"]["database"]
    server_settings: Dict = config["settings"]["server"]

    return (
        database_settings.get(config["database"].lower())["debian_dependencies"]
        + server_settings.get(config["service"].lower())["debian_dependencies"]
    )
```

Figure 14: Dependency Concatenation in templates_management.py

Testing

Testing happens over three spectra due to the nature of snoozebox. This includes the generated projects, snoozelib and the snoozebox cli tool itself. Do note that while snoozelib uses [pytest](#), the generated projects use [nose2](#). There is no particular reason for this aside from the fact that I've transitioned towards using pytest in newer projects, and some of the logic I've used for snoozebox has been taken from some of my older projects. In a real life scenario you'd want to remain consistent. Note that pytest is more popular than nose2, so in this case of relative indifference I'd go with what's standardised.



Both testing frameworks have CLI tools which search for patterns in file names, functions, classes and directories to execute the testing on.

```
snoozebox-py/snoozelib on main [!] is v0.0.0a43 via v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> pytest -v
===== test session starts =====
platform linux -- Python 3.9.9, pytest-5.4.3, py-1.11.0, pluggy-0.13.1 -- /home/apathy/.cache/pyp
oetry/virtualenvs/snoozebox-py-8koxef_T-py3.9/bin/python
cachedir: .pytest_cache
rootdir: /home/apathy/projects/python/snoozebox-py/snoozelib
collected 9 items

tests/test_snoozelib.py::test_translation PASSED [ 11%]
tests/test_snoozelib.py::test_table_name_contains_keywords PASSED [ 22%]
tests/test_snoozelib.py::test_unique_gets_added PASSED [ 33%]
tests/test_snoozelib.py::test_grpc_types PASSED [ 44%]
tests/test_snoozelib.py::test_conversion_sorted_instructions_has_multiple_values PASSED [ 55%]
tests/test_snoozelib.py::test_no_new_lines PASSED [ 66%]
tests/test_snoozelib.py::test_relations PASSED [ 77%]
tests/test_snoozelib.py::test_examples PASSED [ 88%]
tests/test_snoozelib.py::test_duplicate_table_name PASSED [100%]

===== 9 passed in 0.03s =====
```

Figure 15: pytest test results

Snoozebox CLI Testing

Currently Snoozebox-CLI testing is done manually. This is partially due to time constraints and partially due to inexperience. It is however possible to speculate upon how such a testing system could look like. Most likely it'd be a separate testing suite using subprocesses and testing the output of said subprocesses.

```
1 from deleteme import __version__
2 import subprocess
3 from pathlib import Path
4
5 subprocess.run(["snoozebox", "init"])
6
7 def test_rest_postgres():
8     subprocess.run(["snoozebox", "append", "name=rest_postgres_test", "service=rest", "database=postgres"])
9     assert Path("services/rest_postgres_test/rest_postgres_test/connection/postgres_connection.py").exists()
10    Path("services/rest_postgres_test").rmdir()
```

Figure 16: Hypothetical CLI testing

This could look something like the above. Do note that this code does not actually exist inside the project, and is purely speculation. You'd probably want to pack line 8 and 10 into a decorator as well. You'd also want to isolate all of the created directories and files into a separate directory and finally erase all of them once complete. The fact of the matter is, that this would take a long time to make, and it'd be unlikely to provide many "points" in the finalized rendition of this examination project. Therefore it was promptly skipped. In a real life scenario, you'd want to create testing even for the CLI tooling. For this particular non-real-life scenario it was considered out of scope. But these unit and integration testing strategies are worth taking into consideration in future projects.

Snoozelib Testing

The snoozelib testing is done via. a few unit-tests inside the [test directory of the snoozelib directory](#). Arguably it is lacking in terms of how many things it covers. Falling into the manual-testing trap is definitely a flaw.

```
def test_examples() -> None:
    """
    This tests the examples generated with Jinja as per 25/08/22
    """
    o2m01: str = """
CREATE TABLE IF NOT EXISTS person(
    id SERIAL NOT NULL,
    name VARCHAR(255) UNIQUE NOT NULL,
    age INT NOT NULL,
    occupied BOOL NOT NULL
);
    """
    o2m02: str = """
CREATE TABLE IF NOT EXISTS company(
    id SERIAL NOT NULL,
    title VARCHAR(255) UNIQUE NOT NULL
);
    """
    m2m01: str = """
CREATE TABLE IF NOT EXISTS tag(
    id SERIAL NOT NULL,
    name VARCHAR(100) UNIQUE NOT NULL
);
    """
    m2m02: str = """
CREATE TABLE IF NOT EXISTS msg(
    id SERIAL NOT NULL,
    contents VARCHAR(5000) NOT NULL
);
    """
    m2mAssoc: str = """
CREATE TABLE IF NOT EXISTS tag_msg(
    FOREIGN KEY tag_id REFERENCES tag(id),
    FOREIGN KEY msg_id REFERENCES msg(id)
);
    """
    (conversions, associations) = sql_tables_to_classes(
        [o2m01, o2m02, m2m01, m2m02, m2mAssoc]
    )
    assert conversions != None
    assert associations != None
```

Figure 17: Snoozelib testing examples

This is an example of a test function inside snoozelib. This way of defining tables and then inspecting the results is the typical way these tests are written. As is apparent by what is asserted, these tests are currently rather primitive compared to what you might have wanted inside a finalised product. Arguably even in a product which is still in development.

Future development of interpretations and translations of schematics would mean an increase in separate test files as new database paradigms get supported.

Generated Project Testing

The tests are just as dynamically generated in the testing as the code they test itself. At least in theory. The fact of the matter is, that the testing templates have been severely down-prioritised due to time-constraints. Despite the fact that testing drastically increases the quality and reliability of the product, reality is that they don't apply more of the content which would be valued highly on examination. That is my interpretation and perspective, anyway. But the setup is there at least partially. This is because unit-testing is there but lacking. Integration testing is absent all-together and would need a different strategy entirely.

To create integration testing inside this context would be a trial in and of itself. It'd likely involve a separate dynamically created testing environment inside the init root alongside the services and schematics directory.

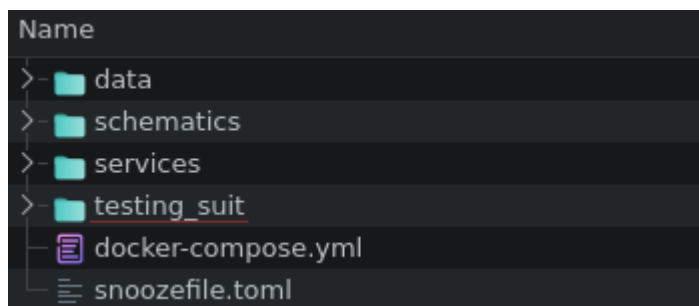


Figure 18: Snoozebox base directory or "init root" with hypothetical testing_suite

This test-kit would need to involve all of the services unless specifically defined that it should only target separate services. This specific feature would possibly involve having to create a separate generated CLI in the init step, which can focus on specific services instead of all of them at once. Alternatively, it might be possible to configure pytest to do so for us with its configuration files. Most likely it'd involve having to include the test automation library [tox](#). That's a considerable amount of work that wouldn't necessarily be valued very highly in an exam context. In a real life scenario I would consider it unskippable before being fully able to consider a product finished.

Automated Testing

Automated testing was always considered out of scope. It's unfeasible within such a timeline. However, in an ideal situation, you would set up automated testing with continuous integration tools like [Github Actions](#), [CircleCI](#) or [Jenkins](#).

Compatibility

While Snoozebox-py should work for all major operating systems, the support for Mac and Windows is currently untested. It should be noted that due to the usage of shell scripts, it is important that the user uses a platform which is compatible with it. Because of this, a Windows user is in theory required to use a terminal emulator which is compatible with it, e.g. [Git BASH](#).

Specifically it's this code:

```
subprocess.run(
    ["chmod", "+x", protogen_sh], check=True, text=True, cwd=relative_project_path
)
subprocess.run(["./protogen.sh"], check=True, text=True, cwd=relative_project_path)
```

Figure 19: Subprocess shell script execution

And the file it references:

```
snoozebox_py > gen > templates > protogen >  protogen.sh.jinja
1  #!/bin/bash
2  echo "generating gRPC..."
3  proto_dir=proto
4  for entry in "$proto_dir"/*
5  do
6      proto_name="$(basename $entry)"
7      proto_name="${proto_name%.*}"
8      echo "found ${proto_name}..."
9      python -m grpc_tools.protoc -I./proto --python
10     sed -i "s/import ${proto_name}_pb2 as ${proto
11 done
12 echo "done"
```

Figure 20: protogen.sh

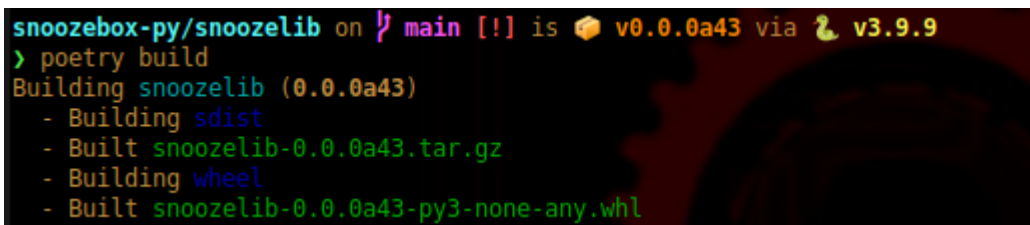
There currently aren't any plans to make any powershell or batch files which support the same process.

Execution of the projects are covered with Docker in terms of portability. As for running them manually outside the project - It hasn't been tested on anything except Linux, but it will most likely work. Poetry provides some portability in the sense that it'll work independently from the packages the client has installed on their machine, provided that Poetry has been installed and used correctly.

Actual User Experience

I'll go through what a user should do whenever they want to use Snoozebox to create services. Please note that Snoozebox-py remains untested on any non-Linux operating system.

1. Ensure that the correct dependencies are installed. This includes [Poetry](#), [Docker](#), [Docker-Compose](#)
 - a. If you're going to use postgres and Linux: [libpq](#) and potentially [libpq-devel](#) or your distribution equivalents thereof. Snoozebox-py is currently untested on any other operative system.
 - b. If Windows (untested) use [Git Bash](#). This project runs shell scripts internally. CMD won't work. Powershell might work with some adjustments, but it remains untested.
 - c. I can't help you if you're using MacOS
2. Clone the project <https://github.com/Mutestock/snoozebox-py>
3. Build Snoozelib:
 - a. Change directory to snoozebox-py/snoozelib
 - b. `> poetry build`









```
snoozebox-py/snoozelib on  main [!] is  v0.0.0a43 via  v3.9.9
> poetry build
Building snoozelib (0.0.0a43)
- Building sdist
- Built snoozelib-0.0.0a43.tar.gz
- Building wheel
- Built snoozelib-0.0.0a43-py3-none-any.whl
```

Figure 21: Poetry build command

4. Open a poetry shell in Snoozebox-py and update



```
snoozebox-py/snoozelib on  main [!] is  v0.0.0a43 via  v3.9.9
> cd ..

snoozebox-py on  main [!] is  v0.1.0 via  v3.9.9
> poetry shell
Spawning shell within /home/apathy/.cache/pypoetry/virtualenvs/snoozebox-py-8koxef_T-py3.9
. /home/apathy/.cache/pypoetry/virtualenvs/snoozebox-py-8koxef_T-py3.9/bin/activate


snoozebox-py on  main [!] is  v0.1.0 via  v3.9.9
> . /home/apathy/.cache/pypoetry/virtualenvs/snoozebox-py-8koxef_T-py3.9/bin/activate

snoozebox-py on  main [!] is  v0.1.0 via  v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> poetry update
Updating dependencies
Resolving dependencies... (1.4s)

Writing lock file

Package operations: 0 installs, 7 updates, 0 removals

• Updating certifi (2022.6.15.1 -> 2022.9.14)
• Updating idna (3.3 -> 3.4)
• Updating pathspec (0.9.0 -> 0.10.1)
• Updating pluggy (0.13.1 -> 1.0.0)
• Updating black (22.6.0 -> 22.8.0)
• Updating pipe (2.0 -> 1.6.0)
• Updating pytest (5.4.3 -> 7.1.3)

snoozebox-py on  main [!] is  v0.1.0 via  v3.9.9 (snoozebox-py-8koxef_T-py3.9) took 4s
```

5. Change directory to a directory where you want your snoozebox projects

```
~/Documents/junk via 🐡 v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> mkdir snoozebox_testing03

~/Documents/junk via 🐡 v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> cd snoozebox_testing03/
```

6. While inside the poetry shell, point to the main file of snoozebox-py and run “init”
 - a. E.g. while inside a directory called:
/home/apathy/Documents/junk/snoozebox_testing/
and the Snoozebox project is at:
/home/apathy/projects/python/snoozebox-py/
use:
> python /home/apathy/projects/python/snoozebox-py/snoozebox-py/main.py init
 - b. Personally I’ve created an alias called “snoozebox” in my .bashrc file:

```
alias snoozebox="python ~/projects/python/snoozebox-py/snoozebox_py/main.py"
```

```
Documents/junk/snoozebox_testing03 via 🐡 v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> snoozebox init
```

Verify that a variety of files and directories have been generated

```
Documents/junk/snoozebox_testing03 via 🐡 v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> ls
docker-compose.yml  schematics  services  snoozebox.toml
```


7. Run “append” > type your project name, e.g. “project_name” > database: postgres > service: gRPC > schematics/examples

```
Documents/junk/snoozebox_testing03 via v3.9.9 (snoozebox-py-8koxef_T-py3.9)
> snoozebox append
Welcome to snoozebox

Note that poetry is required: https://python-poetry.org/

Please type the project's name
project_name
Please choose a database paradigm

1. Postgres
2. MongoDB
3. Cassandra
Type 1-3

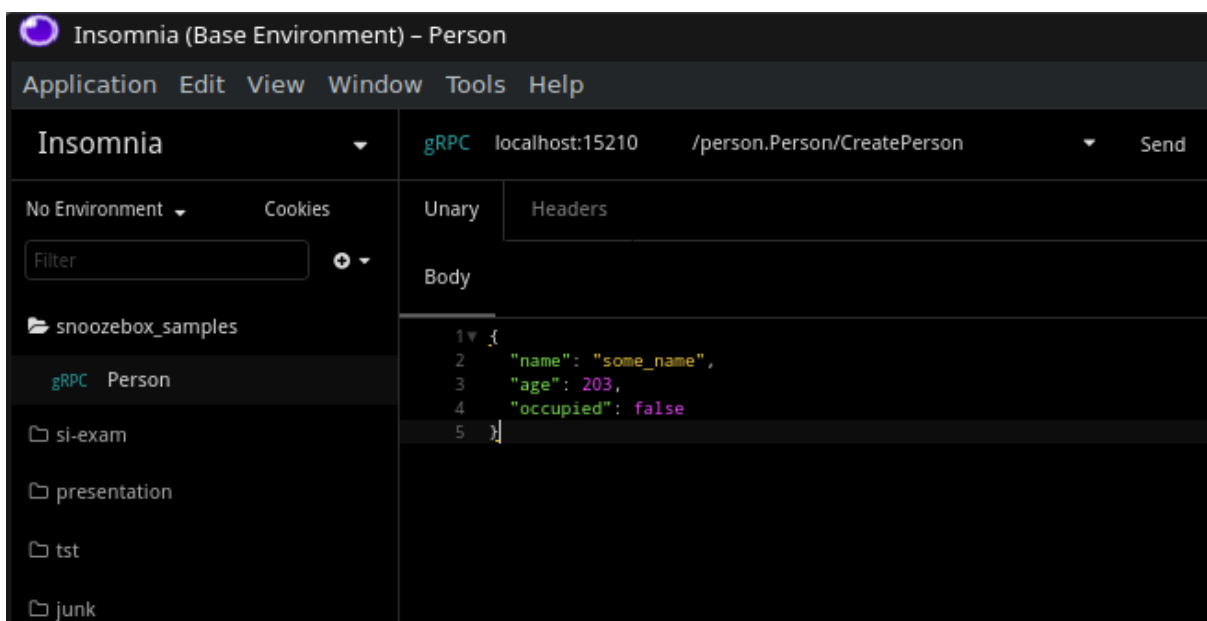
1
Please select a service type:

1. Rest
2. gRPC
3. Kafka
4. RabbitMQ
Type 1-4

2
Grabbing schematics...
These are the directories in the schematics directory which contain sql files
Select one

1. schematics/examples
```

8. > docker-compose up --build
9. Test with [BloomRPC](#), [Insomnia](#), [Postman](#), etc. The port should be 15210 if this is your first time.



This was just an example run. If you want to use your own schematics, go to the schematics directory, create a new directory and make some sql files. The contents should have the same format as those in the examples directory. Then follow from step 7. but select your new schematics instead of examples.

If you want to run the projects by themselves:

1. Follow the section above but stop after step 7.
2. Change directory to the generated project (where the pyproject.toml file is)
3. Make sure you're not in a poetry environment already
4. > poetry shell
5. > poetry update
6. python {name_of_your_project}/main.py
7. Test with [BloomRPC](#), [Insomnia](#), [Postman](#), etc. The port should be 15210 if this is your first time.

Assuming everything worked, you can continue appending services. New services will be appended to docker-compose.yml each time you do.

Documentation

There are many different documentation formats in Python. The one I usually go with is [Sphinx](#). It requires a specific documentation style, which the Sphinx CLI interprets and prints in HTML, LaTeX, ePub and more. While this documentation style has been implemented inside Snoozebox, it hasn't been implemented inside the templates for the generated projects. This was one of the stretch goals, however, but much like the more extensive integration testing it was skipped in favour of adding more features. The documentation format of Sphinx can be seen in the code below.

```
def write_examples(path: Path, jinja_env: Environment = None) -> None:
    """Writes a set of example data to append with. It's written during the init phase unless deactivated.

    :param path: The relative path to the schematics directory.
    :type path: Path
    :param jinja_env: The Jinja Environment in which the templates should be written, defaults to None
    :type jinja_env: Environment, optional
    """
    if not jinja_env:
        jinja_env = setup_templating()
    template_file_structure: List[TemplateFileStructure] = [
```

Implementation of documentation generation with Sphinx are, however, fairly low priority; documentation writing doesn't equal documentation generation. Testing, for example, is of a much higher priority, and is in fact one of the project's more bleeding problems. If documentation was to be prioritized, it would likely start with the generated projects themselves, as the client might want some clarification over how exactly the project works. The generation of Sphinx in HTML format could be made automated alongside the code itself during the generation process, which would eliminate the factor of needing the client to do any work on their end to make it work. Question is whether they'd ever use it in practice, which is hard to say. Perhaps there is a reason why much of the generated code in other software which implements generation of code through software, e.g. Vue, React, Spring, etc, don't provide extensive documentation.

The next priority would be Snoozelib since it's a library. This would be beneficial to all who'd want to do further development with databases. With how convoluted Snoozelib is currently, it would, granted people would actually use Snoozelib outside this scenario, likely be of great use in an open-source context. Reversely, the current quality of Snoozelib is too low for me to say that there wouldn't be extensive refactoring of code, which would void the efforts of documenting it effectively.

Snoozebox-py itself would be of the lower priority for me, although not much less than Snoozelib. This is because Snoozebox-py should be fairly easy to understand for an experienced programmer.

Versioning

Versioning of this project comes in Snoozelib and Snoozebox alike with the pyproject.toml standards. A thing should be clarified, however: Snoozebox's version in the pyproject file is rather arbitrary at the moment. The same can't be said about Snoozelib, however, since it is a dependency of Snoozebox.

```
[tool.poetry.dependencies]
python = "^3.9"
click = "^8.1.2"
pipe = "^1.6.0"
snoozelib = {path = "snoozelib/dist/snoozelib-0.0.0a43-py3-none-any.whl"}
rtoml = "^0.8.0"
requests = "^2.27.1"
Jinja2 = "^3.1.2"
```

Snoozelib was always intended to be a library which would eventually be uploaded to [pypi](https://pypi.org/), the official Python package registry. Therefore extensive versioning has been made during the project's development. Note that these packages are made with [Poetry's build command](#).

```
snoozelib-0.0.0a38-py3-none-any.whl
snoozelib-0.0.0a38.tar.gz
snoozelib-0.0.0a39-py3-none-any.whl
snoozelib-0.0.0a39.tar.gz
snoozelib-0.0.0a40-py3-none-any.whl
snoozelib-0.0.0a40.tar.gz
snoozelib-0.0.0a41-py3-none-any.whl
snoozelib-0.0.0a41.tar.gz
snoozelib-0.0.0a42-py3-none-any.whl
snoozelib-0.0.0a42.tar.gz
snoozelib-0.0.0a43-py3-none-any.whl
snoozelib-0.0.0a43.tar.gz
```

The snoozelib versioning follows the following logic:

x.0.0 - Major release version. Whenever this point is reached, I'd call the project finalized. All future development would be bug fixes and maintenance, unless the project takes a new direction (like adding additional services/database support).

0.x.0 - A new service or database has been successfully added and tested on all possible platforms. All thinkable tests have been written, everything about the new technologies have been documented.

0.x.x - Some significant changes are made. This could be new functions being available in the API, refactoring or optimizations in old code, etc. New tests have already been written and no breaking changes are made between this version and the last one. It is also documented.

0.0.0rc - 'rc' being a "release candidate". This version is tested and documented, but there isn't enough confidence to call it a release. Some manual user and feedback would be needed to notch it up over the ledge of being released.

0.0.0b - 'b' being a "beta". The code is written and tested. It is possible that errors will occur when being used - errors that will be sorted out whenever discovered. Documentation is a work in progress and hasn't been completed yet.

0.0.0a - 'a' being an "alpha". A new part of the code has been produced. Most of the code of the change has been made and tested. It is *perceived* to be complete, but expected to break. Documentation has begun in the significant methods and functions. Code which might fail is skipped in documentation and will be added later on whenever it has stabilized.

0.0.0ax - An iteration of changes through the alpha phase. Purely a development phase. This is not something which should at any point be used in production.

As of writing, Snoozelib is at version 0.0.0a43. In other words, this is purely at a development phase. Although the foundation of the project has been made, it is not at a state where I'd publish it to the public as a product. This is why it hasn't been uploaded to Pypi yet. My first upload to Pypi would be at 0.1.0 which would be after I'm completely confident in Snoozelib's competence in managing SQL for use in Snoozebox with Postgres. So there's a long way to go.

Snoozebox Distribution

The initial distribution strategy was different because of the programming language in question. The most reliable solution and the one I was planning to use was with [Cargo](#). Many CLI tools can already be built and installed with Cargo. Examples would be the CLI tool that comes with [Deno](#), a Typescript runtime, and [sqlx](#). This would also be an operating system-independent solution, with the asterisk that the user would need Rust, which wouldn't be an obscene requirement due to the initial scope of the project and in extension who the target audience of the tool is.

It could also be possible to distribute the binary by itself. This would simply be by uploading it to the [releases tab](#) in the github repository.

Since the programming language has swapped to Python, the possibilities of distribution changes from cargo to pip. [Poetry](#) has some [installation options](#) that we can use as examples. They offer an option for using pip, but also have options more native to the operating system which the system is going to be installed on such as bash for Linux and powershell for Windows.

In all cases the CLI tool would have to be bound to the user's machine e.g. with environment variables and aliases, or in Linux' case put it in the bin directory where it should reside anyway.

As for the theoretical modular system which was part of the initial concept of the project, it'd use an external server where people would upload and download the templates. See the [modular building block subchapter](#).

Ultimately, a good solution was never developed. The user must clone the repository, and build Snoozelib by themselves. Afterwards the user can, if desired, bind the CLI tool with an alias.

Maintenance


All package versioning in Python is done with set versions with the pyproject format, and therefore shouldn't be a major hassle to deal with in terms of maintenance. Potential issues could rise later on in the process if security issues arise in the used packages. Rust example with the Tokio asynchronous runtime in case the initial plan was continued:



The screenshot shows a GitHub Security Advisory titled "Race Condition in tokio". It has a "High severity" label, is "GitHub Reviewed", and was published on 6 Jan, updated on 14 Jun. Under "Vulnerability details", there is a table showing affected and patched versions for the 'tokio' package (Rust).

Package	Affected versions	Patched versions
 tokio (Rust)	< 1.8.4 >= 1.9.0, < 1.13.1	1.8.4 1.13.1

Docker versions are set to versions which are known to be stable with the projects, which should severely decrease the likelihood of incompatibilities between technology versions, e.g. Python:

```
snoozebox_py > gen > templates > misc >  grpc_dockerfile.jinja
1 FROM python:3.9-slim
```

Granted testing has been done correctly, and to an acceptable degree, continued support and maintenance should involve making said tests pass, as well as writing new tests whenever errors have been discovered where the current tests may not cover the newly discovered errors.

Potential errors which are difficult for me to circumvent are the dependencies used inside the [default_settings.toml](#) file

```
dependencies=["psycpg2-binary", "sqlalchemy", "sqlalchemy-utils"]
debian_dependencies=["gcc", "libpq-dev"]
```

While the versioning of the debian packages are fairly difficult for me to manage, they're also very static in nature, i.e. they're unlikely to make changes which are breaking due to the policies of Linux packages in general. As Linus himself says it:

"There is one rule in the kernel: We don't break userspace" - Linus Torvald¹⁵

And

"If it's a bug everybody relies on, it's a feature" - Linus Torvald¹⁶

With packages like mature packages like gcc and libpq being mature on an already very mature subdistribution, it is unlikely that they're going to break. If errors do occur here, as unlikely as it may be, they'll likely be more difficult to pinpoint. It is unknown to me when and how the internal docker images update the internal operating system distribution's package manager's keyring.

Note: Yes, I do understand that Linus is talking about the Linux Kernel and not Debian Packages, but the mindset is at least encouraged to be universal across the Linux' sphere of influence. In the same Q&A as referenced earlier, it is in a context of whenever packages made in various distributions (especially Debian) do sidestep and ignore this work ethic.

¹⁵ <https://youtu.be/7SofmXIYvGM?t=516>

¹⁶ <https://youtu.be/7SofmXIYvGM?t=608>

Workflow

As a development strategy, this project has utilized Scrum methods during the development process. This doesn't include certain ceremonies which can't effectively be implemented since this is a one-man project. An example of this would be

This project uses [github projects](#) as a scrum board.

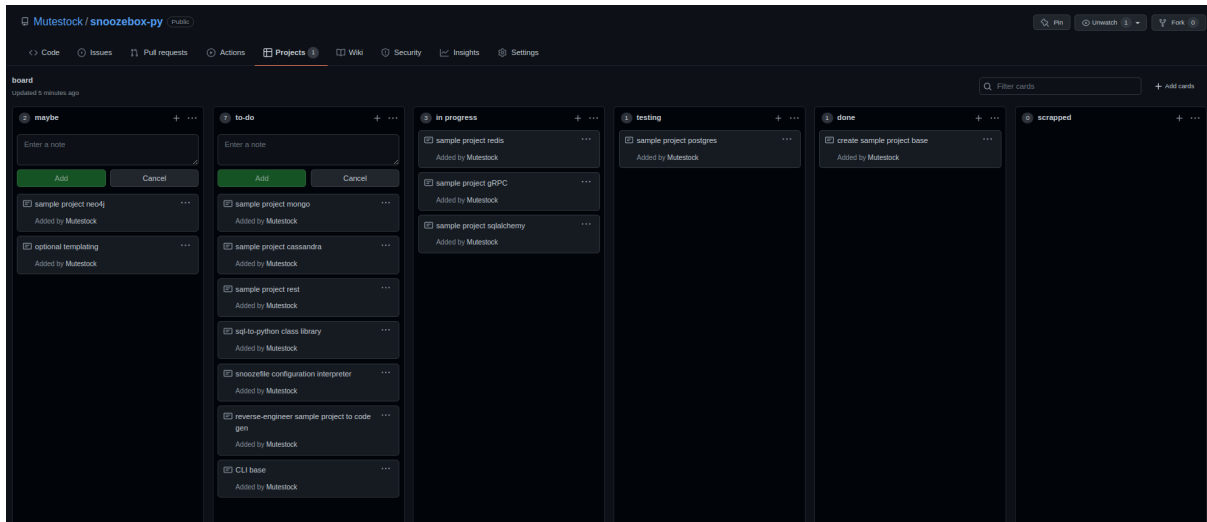


Figure 22: Github Projects scrum board

Like many other Github features like Github Actions, Github Projects provides an easy to use, but less advanced, solution to common issues. Here it's as a Scrum board which you'd have to get from websites like JetBrains's [YouTrack](#) or Atlassian's [Jira](#). It's a massive convenience to not have to create new accounts and subscription plans for those services, and virtually all developers have a github profile, so Github Projects is the chosen method for keeping track of the development.

Please note, that the project is set up so that most technology requirements are actually stretch goals.

Brainstorming

The idea of creating this software came from multiple factors. During the Software Development education, I've found myself simply creating the same solution again and again but with new data inserted. It was the same regardless of language. Be it [Typescript](#), [Rust](#) or in this case Python. The solution somehow always looks like it's following the same structure. And it is. A three-layer architecture between data access, logic and service. It somehow doesn't seem too far off to just, at least at its most basic format, define the primary difference between different services to be the schematics which they need to accommodate and distribute. So in the pursuit of the "Work smarter, not harder" I decided that a tool like this should exist.

Another factor is that it can easily cover most of the subjects in this education. The simulated corporate part of this is that even the corporate world may have an interest in increasing the production speed in smaller projects in a world where the microservice architecture becomes increasingly popular.

Yet another, yet smaller, factor is that I spoke about the idea at my internship and one of the seniors said it's impossible, which made me feel like proving him wrong. I'm not insinuating spite as much as it's me underlining the fact that you shouldn't be too pessimistic in a world of invention and development. Ideas and theories must be entertained before they can provide results. If a company stuck with stale technologies, language versions and architectures will be a hindrance to this potential innovation, then I'll gladly cultivate this idea by myself since it's free anyway. This was also one of the reasons I decided to make this by myself without any company on the side to guide me despite the recommendations.

And finally there's the word "Software". When I started this education I thought we'd entertain software on a broader perspective than just creating backends. Creating a fully functional CLI tool with actual installation options on my machine is something I very much think the average programmer need to learn. I wanted to write a library correctly and upload it to a registry like any of the other libraries we use all the time. I wanted to write a CLI tool working like any other binary in my bin directory. These things just haven't been covered and I felt like a project like this would hit all the birds with one stone.

Like most other software in the agile software development world, the scope and cost(time) changed as the project progressed.

As for the naming of the project itself: 'Snooze' is a synonym of rest, i.e. RESTful, whereas box was meant as an alternative to the word 'kit'. In other words it is a 'rest kit' or a kit (toolbox) for making REST services. That part of the name somewhat lost some of its validity as the scope expanded beyond REST services, but it seemed more memorable than just calling it something arbitrary so I stuck with it. As for the '-py': part of the concept of this project was to showcase how the concept of making services via. data-structures can be applied universally across languages. Due to time constraints, a showcase of this part of the concept never came into fruition, and it instead became a program made in Python for making Python services. To underline the differences between these two variations of the Snoozebox concept, I changed the name of this variation to 'Snoozebox-py' instead.

Continued Development Guidelines

Like all other matters in this assignment, this chapter is about three different sections: The CLI as whole (Snoozebox-py), the library (Snoozelib) and the generated projects (the templates used inside Snoozebox-py). Generally speaking there's a lot of work involved with creating support for new technologies, though the complexity shifts quite a bit between those technologies, as will be apparent in this chapter when gRPC and REST are compared.

Possible Extensions

Many technologies would be able to get added to the setup. Specifically my supervisor has shown interest in [Neo4j](#) and the graph database paradigm in general. For Neo4j we'd use [Cypher](#) as the schematics, which we'd then have to translate to our Python class just like we would SQL with an RDBMS like PostgreSQL. Refer to the [building block subchapter](#) for more info.

You can also implement modern but different technologies like [GraphQL](#). This would require its own set of handlers, but would occupy a service building block. Database interactions would stay the same, which would mean that GraphQL would be relatively non-intrusive in that regard. At least in theory. GraphQL takes a service block, because it's not a typical schematic. There is logic tied to the data structures you pass during method calls itself. Therefore, to gain GraphQL support, you'd need to create service templates around it, because while it isn't too invasive for Database building blocks, it is too invasive for Service blocks which aren't built around GraphQL.

CLI

The CLI itself uses the [click library](#). Adding new commands is fairly straight forward. Just go to the [cli.py file](#) and insert the command you want to use. Here's an example of the "describe-sql" command in the CLI.

```
@manager.command()
@click.option("--path", "-p")
@click.option("--translate", "-t", is_flag=True, help="Gives the snoozelib output")
def describe_sql(path, translate):
    """
    Returns a dictionary with basic information about the found sql files in the directory structure
    Point to the schematics folder with path variable or execute this command inside the schematics folder
    """
    if not path:
        path = os.getcwd()
    if not Path(path).is_dir():
        sys.exit("Path is not a directory. Please point to the schematics directory.")
    directories = get_directories_with_sql_files(path)
    print(directories)
    if translate:
        for file_lists in directories.values():
            for file_ in file_lists:
                conversions: List[Conversion] = []
                with open(file_, "r") as file_reader:
                    conversions = sql_tables_to_classes(file_reader.read())
                print(conversions)
                for conversion in conversions:
                    print(conversion.contents)
```

Figure 29: Click CLI command definition

Options are optional flags and params that you can pass in. "is_flag" means that it's essentially a boolean and doesn't take params. Refer to the [click documentation](#) for more info.

Note that there'd have to be an equivalent "describe" for other schematic types or have one command which describes all of the supported schematics. For now there's only `describe_sql`.

Adding Database Support

Snoozebox is rather modular. It's relatively easy adding support for new databases inside the snoozebox templates. However, snoozelib integration is another story entirely. **It can't be stressed enough how important it is to have a working sample project before beginning.**

First add your option in [prompt.py](#). This enables the user to be able to select it with the CLI tool.

```
DATABASE_OPTIONS: dict = {"1": "Postgres", "2": "MongoDB", "3": "Cassandra"}
SERVICE_OPTIONS: dict = {"1": "Rest", "2": "gRPC", "3": "Kafka", "4": "RabbitMQ"}
```

Figure 30: Option definition. Prompt.py

Next go to [default_settings.toml](#). Add the settings required to connect to your database. The port is the default port. It'll automatically sort through occupied ports during generation. The most important thing is that it doesn't use the same port as the other default settings.

The dependencies and `debian_dependencies` should always be filled. `dependencies` is for libraries which will be installed via. poetry during generation. `debian_dependencies` is for the docker container which is running on a debian system. I.e. in postgres' case it is required to have its respective library and `psycpg2-binary` requires a compiler, which is why `gcc` has been added to the list.

```
[settings.database.postgres]
host="localhost"
port=15312
usr="snoozebox-pg-user"
pwd="snoozebox-pg-pwd"
db="snoozebox-pg-db"
dependencies=["psycpg2-binary", "sqlalchemy", "sqlalchemy-utils"]
debian_dependencies=["gcc", "libpq-dev"]
```

Figure 31: `default_settings.toml`. Includes database settings and dependencies

In the `collect_dependencies` function inside [templates_management](#), add your database to the if statements:

```
def collect_dependencies(config: Dict) -> None:
    """Collects all Python dependencies to be used with poetry later on.
    These dependencies are stored within the config file, and will be appended according to the service and database types
    as well as some base dependencies and redis dependencies.

    :param config: Configuration dictionary which gets passed around and modified during the generation process
    :type config: Dict
    """
    database: str = config.get("database")
    service: str = config.get("service")
    dependencies: list = []

    # Match hasn't been added in earlier Python versions
    if database == "Postgres":
        dependencies = (
            dependencies + config["settings"]["database"]["postgres"]["dependencies"]
        )
        print(
            "Note: This project uses psycopg2 as its Postgres driver. Secure that the required postgres library is installed"
        )
        print("Check https://www.psycopg.org/install/")
    elif database == "MongoDB":
```

Figure 32: Dependency collection. `templates_management.py`

You will also need to define how many CRUD instructions your database paradigm supports inside [templates_management](#):

```
def _set_crud_instructions(config: Dict) -> None:
    """Sets the crud instructions to be applied. This is

    :param config: Configuration dictionary which gets passed around and modified during the generation process
    :type config: Dict
    """
    database: str = config["database"]
    config["crud_instructions"] = []

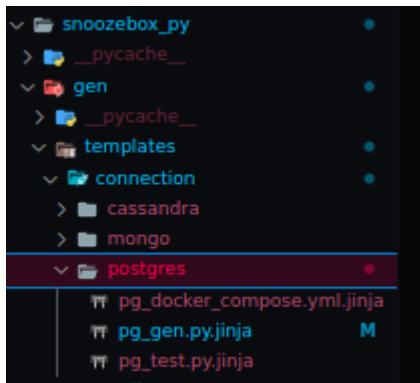
    # Match hasn't been added in earlier Python versions
    if database == "Postgres":
        config["crud_instructions"] = [
            "create",
            "read",
            "update",
            "delete",
            "read_list",
        ]
```

Figure 33: Determination of which instructions are available for the available database technologies

Connection Templates

The connection templates are the template equivalents of what will be generated inside the connection directory of the generated project. The exception is the `docker_compose.yml.jinja` file, which is a snippet which will be appended to the init root `docker-compose.yml` file.

Go to the templates directory. There are three files you need to create here.



First there's the docker-compose entry which contents will get appended (not overwritten) during generation. There are some checks for whether or not the docker-compose file is broken, but not substantial enough for production quality. Either way, fill out the equivalents of your database. We're using the settings you defined earlier. Please mind the indentations. It's important to remain consistent. The finalized docker-compose file will not function unless it is done correctly.

```
1
2 {{config['project_name']}}_postgres:
3   container_name: {{config['project_name']}}_postgres
4   image: postgres:latest
5   restart: always
6   environment:
7     - POSTGRES_USER={{config['settings']['database']['postgres']['usr']}}
8     - POSTGRES_PASSWORD={{config['settings']['database']['postgres']['pwd']}}
9     - POSTGRES_DB={{config['settings']['database']['postgres']['db']}}
10    - PGDATA=/var/lib/postgresql/data
11   ports:
12     - {{config['settings']['database']['postgres']['port']}}:5432
13   volumes:
14     - ./data/postgres:/var/lib/postgresql/data
15   networks:
16     - {{config["settings"]["docker_compose_network"]}}
```

Now create the actual generation template file. Remember that the connection options will point to the generated config file and not to the default_settings of the tool. As such we're not referencing any data via Jinja here.

```
snoozebox_py > gen > templates > connection > postgres > pg_gen.py.jinja
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from utils.config import CONFIG
4 from typing import Optional, Any
5
6 PG_CONFIG = CONFIG["database"]["postgres"]
7
8 conn_string: str = f'postgresql+psycopg2://{PG_CONFIG["usr"]}:{PG_CONFIG["pwd"]}@{PG_CONFIG["host"]}:{PG_CONFIG["port"]}/{PG_CONFIG["db"]}'
```

The last file is the test equivalent. The bare minimum would be to check whether or not a connection can be established at all.

```
snoozebox_py > gen > templates > connection > postgres > pg_test.py.jinja
1  from connection.postgres_connection import engine
2  from sqlalchemy_utils import database_exists
3
4
5  class TestPostgres(unittest.testcase):
6      def test_connection(self):
7          self.assertTrue(database_exists(engine.url))
```

Logic Handler Templates

Inside the logic layer there are in this architecture handlers. These handlers are dynamically added as more schematics get appended during generation. I.e. two schematics equals two handlers. The handlers are primarily for service blocks, but there are some implementations which will have to be made.

Notice how each of the service implementations expect the following imports:

```
snoozebox_py > gen > templates > logic > handlers > rest > rest_handler.py.jinja
1  from models.{{schematic.name.lower()}} import {{schematic.name.capitalize()}}
2  from logic.handlers.handler_utils.crud_handler_component import CrudHandlerComponent
3  from logic.handlers.handler_utils.utils_handler_component import UtilsHandlerComponent
```

This is because they assume that you're going to add those two components for your database as well. Those two components give access to CRUD functionality and whatever else you'd want to give them. The handlers instantiate via composition ("has a" vs inheritance "is a"). This means you can append other components to the handlers with a Jinja check on whether or not the selected database is yours. Keep in mind that the complexity rises significantly as you make those changes.

Add the two components and whatever else you need inside a new directory inside [gen/templates/logic/handlers](#).

```
└─ logic / handlers
   └─ grpc
      └─ grpc_handler.py.jinja
   └─ relational
      └─ generic_tools.py.jinja
      └─ relational_crud_component.py.jinja
      └─ relational_utils_component.py.jinja
```

These components might take a while to create. Essentially they need to interact only by the name of the table and assumes that there's always an id inside the table (which is a requirement for snoozebox to work).

```
class CrudHandlerComponent:
    def __init__(self, object_instance) -> None:
        self.table = object_instance.__table__
```

if there are other things the user could use you can place it inside the utils component which is also required.

Service Templates

You shouldn't have to make any changes to the service templates since we're adding support for a database and not a service type.

Misc Templates

The misc templates are primarily for dockerfiles. These dockerfiles are dependent on the service type and not the database paradigm by default. Theoretically you can, but the amount of work you have to do increases exponentially as we add more databases and services, so it's not recommended to do so when it can be avoided.

The database specific apt dependencies have already been collected since we defined them later, and it's not something we need to think about during this step if it was done correctly earlier.

Utils Templates

If there's any database dependent logic which needs to be added it can theoretically be placed here. Do so sparingly, though.

Examples Templates

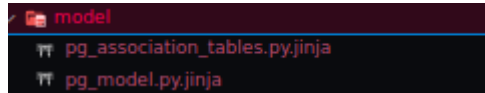
You need to create examples after you've created all the other templates. These examples need to cover a significant amount of the features of the database you've added. The examples themselves are data-structures, i.e. what the CLI will point at during generation. E.g. postgres uses .sql files in the examples directory.

```
snoozebox_py > gen > templates > examples > sql > ¶¶ one_to_many01.sql.jinja
1 CREATE TABLE IF NOT EXISTS person(
2     id SERIAL NOT NULL,
3     name VARCHAR(255) UNIQUE NOT NULL,
4     age INT NOT NULL,
5     occupied BOOL NOT NULL
6 );
```

If the database you're covering would be something like MongoDB then the schematics would be json text.

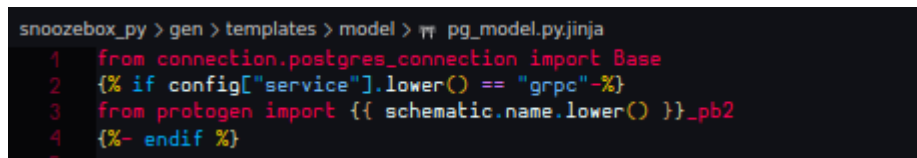
Model Templates

This part is the most difficult one. This is because this is where we're interacting with the output of snoozelib. The contents can vary drastically by how the Snoozelib implementation (that you will need to make) will look like. But you can take inspiration in how the postgres one looks like, then you'll understand why it's complex.

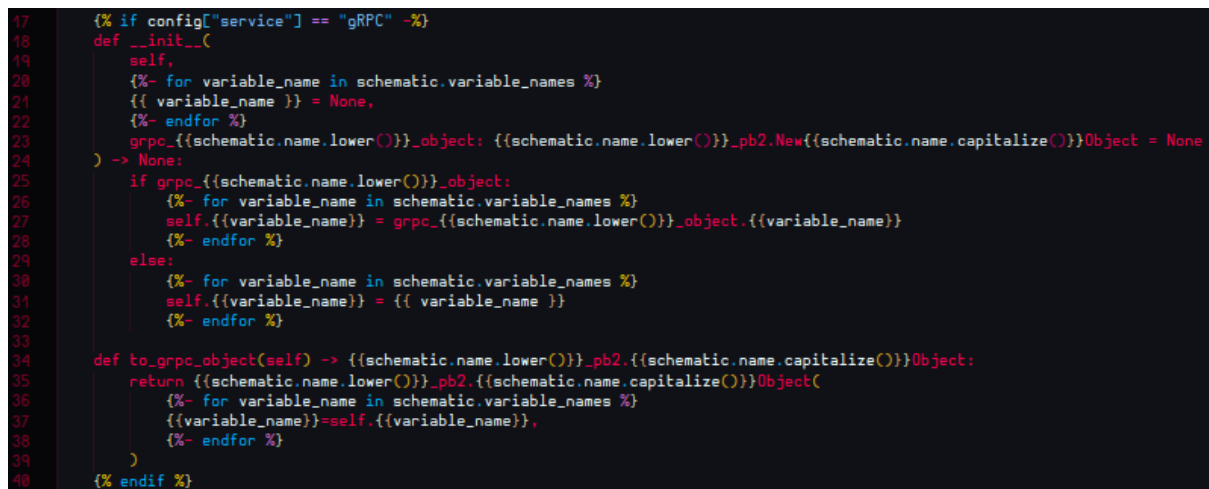


Create your database_model template file. Postgres has an additional one for association tables. You can add other files if you deem it necessary to do so. Just keep the complexity in mind.

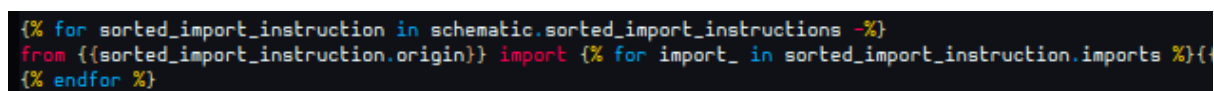
Immediately at the start of the [pg_model.py.jinja](#) file you will notice one of the weaknesses of Snoozebox and why modularity certainly can be improved, although it is currently unknown to me how to proceed.



I'm referring to line 2-4. You'll have to make these three lines as well. As more service types get added, the complexity of all other models can and will increase. This is an exponential increase in complexity in theory. A better example is line 17-40:



This is non-optional code. You'll have to paste similar code into your model and make sure it works. Furthermore, the testing system is currently [lacking quite a bit in integration testing](#) and [CLI testing](#) which is a pretty massive flaw when it comes to future development as complexity dynamically increases.



pg_model has its own way of collecting imports. You can take inspiration from this, but you'll have to implement it yourself within Snoozelib.

All or most other database specific logic to your model will also need to go here. As stated earlier: It can't be stressed enough how important it is to have a working sample project before trying to add support for services and especially databases.

Template Management

Finally it's time to connect your Jinja templates to the generation process. Create a new function called : `_run_{your_db}_templates`. It'll look like the others.

Point to the generated directories like this:

```
src: Path = PathingManager().src
docker_compose: Path = PathingManager().docker_compose
test_dir: Path = PathingManager().tests
```

The PathingManager is a singleton which has been instantiated earlier in the generation process. You don't need to think about it.

Gather your templates like this:

```
template_file_structure: List[TemplateFileStructure] = [
    TemplateFileStructure(
        template_path="connection/postgres/pg_gen.py.jinja",
        generated_file_path=src / "connection/postgres_connection.py",
        jinja_env=jinja_env,
        render_args={"config": config},
    ),
    TemplateFileStructure(
        template_path="connection/postgres/pg_gen.py.jinja",
        generated_file_path=test_dir
        / "test_connection/test_postgres_connection.py",
        jinja_env=jinja_env,
        render_args={"config": config},
    ),
    TemplateFileStructure(
        template_path="logic/handlers/relational/generic_tools.py.jinja",
        generated_file_path=src / "logic/handlers/handler_utils/generic_tools.py",
        jinja_env=jinja_env,
        render_args={},
    ),
    TemplateFileStructure(
        template_path="logic/handlers/relational/relational_crud_component.py.jinja",
        generated_file_path=src
        / "logic/handlers/handler_utils/crud_handler_component.py",
        jinja_env=jinja_env,
        render_args={},
    ),
    TemplateFileStructure(
        template_path="logic/handlers/relational/relational_utils_component.py.jinja",
        generated_file_path=src
        / "logic/handlers/handler_utils/utils_handler_component.py",
        jinja_env=jinja_env,
        render_args={},
    ),
]
```

Where template path is the path to the templates you've been writing, generated_file_path is the path relative path to the generated output, jinja_env is the jinja environment which carries settings for Jinja and render args are what we're referring to inside the jinja files when you see code like this:

```
EXPOSE {{config["settings"]["server"]["rest"]["port"]}}
```


The code below writes the docker-compose specific segment you wrote earlier. The important part is, that we're checking whether or not the segment exists inside the docker-compose file before starting. This is because we're appending and not overwriting.

```
if not f"{config['project_name']}_postgres:" in open(docker_compose, "r").read():
    template_file_structure.append(
        TemplateFileStructure(
            template_path="connection/postgres/pg_docker_compose.yml.jinja",
            generated_file_path=docker_compose,
            jinja_env=jinja_env,
            render_args={"config": config},
        )
    )
```

Notice how

we're iterating through schematics. This is how we're reading through multiple schematics (sql/json/cql/cypher/etc)

```
for conversion in config["schematics"]:
    template_file_structure.append(
        TemplateFileStructure(
            template_path="model/pg_model.py.jinja",
            generated_file_path=src / f"models/{conversion.name.lower()}.py",
            jinja_env=jinja_env,
            render_args={"config": config, "schematic": conversion},
        )
    )
```

Now add this line. As the name suggests it will write contents to the file destinations we've submitted.

```
write_templates(template_file_structure)
```

Finally add your `_run_{your_db}_templates` function to the list here:

```
def _determine_and_run_database_templates(config: Dict, jinja_env: Environment) -> None:
    """Checks for the used database types and initiates templates depending on the result

    :param config: Configuration dictionary which gets passed around and modified during
    :type config: Dict
    :param jinja_env: The environment in which the Jinja should be executed
    :type jinja_env: Environment
    """
    {"postgres": _run_pg_templates, "mongodb": None, "cassandra": None}.get(
        config["database"].lower()
    )(config, jinja_env)
```

Please note that we're passing a reference to the function. We're not executing it. We're taking advantage of ducktyping by assuming that the function we're calling uses the config and jinja_env variables (which all of these kinds of functions do).

And that's it for the templates. Now comes the hardest part.

Snoozelib

Snoozelib is more complicated than complex. This is a bi-product of inexperience in writing libraries. Aside from that I only achieved creating support for Postgres. Unlike snoozebox, snoozelib isn't very modular either. The good news is that SQL is likely to be more complex than if you've chosen a database which uses something like JSON. The bad news is, that it's unlikely you'll be able to reuse much of the contents inside Snoozelib. While you can take inspiration from what's already there, it won't be of much use to you, since the contents are specific to sql.

The `sql_tables_to_classes` in the `__init__.py` file is the entry point for sql files:

```
15 def sql_tables_to_classes(  
16     sql_sequences: List[str],  
17 ) -> Tuple[List[Conversion], List[M2MAssociationTableInfo]]:  
18     collected_statements = []  
19     collected_association_tables = []  
20     for sql in sql_sequences:  
21         sql_lower: str = sql.lower()  
22         if not "create table" in sql_lower or not ";" in sql:  
23             return  
24         statements = sql_lower.split(";")  
25         statements = [  
26             filter_unnecessary_keywords(statement)  
27             for statement in statements  
28             if "create table" in statement  
29         ]  
30         statements = [_make_class_def(statement) for statement in statements]  
31         collected_statements += statements  
32  
33     retrofit_relations(  
34         sql_sequences=sql_sequences,  
35         statements=collected_statements,  
36         association_tables=collected_association_tables,  
37     )  
38     [conversion.finalize_sorted_instructions() for conversion in collected_statements]  
39     return (collected_statements, collected_association_tables)
```

You will need to make an equivalent for whatever you've chosen. This means extracting variables, defining class names, collecting python imports from the libraries you're using. Additionally, the structure of however you've decided to go ahead with this will likely have to have some of the same variables as the [Conversion](#) class. Note that this is where [ducktyping](#) in Python shines, and it's one of the reasons why I chose the language. Check the [Language](#) chapter for more info.

Either way, there are many unknowns and it's unavoidable that there'd have to be refactoring in Snoozelib and some additional checks inside Snoozebox which ensures that it's the correct data being used during the generation process.

Adding Service Support

Adding support for new services is much easier than adding support for new database technologies due to it not being necessary to edit Snoozelib. Many of the same changes that need to be made is a lot like [adding support for another database technology](#). Still it should be underlined once more, that **it can't be stressed enough how important it is to have a working sample project before beginning**.

Like the database section, add your option in [prompt.py](#), insert the settings inside [default_settings.toml](#) and collect the dependencies inside [templates_management.py](#)

Connection Templates

Just like the service templates for database templates, there is no reason why you'd put new code into the connection templates on the basis of adding new services. In fact it isn't allowed since we're using a three-layer architecture to keep things organized.

Logic Handler Templates

The complexity of the handlers can vary greatly depending on which technology is getting added. A great example (and currently the only example) would be between gRPC and REST.

```
snoozebox_py > gen > templates > logic > handlers > rest > rest_handler.py.jinja
1  from models.{{schematic.name.lower()}} import {{schematic.name.capitalize()}}
2  from logic.handlers.handler_utils.crud_handler_component import CrudHandlerComponent
3  from logic.handlers.handler_utils.utils_handler_component import UtilsHandlerComponent
4  from dataclasses import dataclass
5
6
7  @dataclass
8  class {{schematic.name.capitalize()}}Handler():
9      object_instance: {{schematic.name.capitalize()}}
10     crud_handler_component: CrudHandlerComponent
11     utils_handler_component: UtilsHandlerComponent
12
13     def __init__(self):
14         self.object_instance = {{schematic.name.capitalize()}}()
15         self.crud_handler_component = CrudHandlerComponent(object_instance=self.object_instance)
16         self.utils_component = UtilsHandlerComponent(object_instance=self.object_instance)
17
```

The rest handlers don't really need anything else than the components. This is because the components are complex enough to do everything a handler would need to do by themselves. If, however, more logic is to be added in the future, i.e. in some scenario where a programmer would want to use the tool, then these generated handlers would be the place to do it - Like any other handler.

```

17 class {{schematic.name.capitalize()}}Handler:
18     def __init__(self):
19         self.object_instance = {{schematic.name.capitalize()}}()
20         self.crud_component = CrudHandlerComponent(object_instance=self.object_instance)
21         self.utils_component = UtilsHandlerComponent(object_instance=self.object_instance)
22
23     {% if "create" in config["crud_instructions"] %}
24     def create(
25         self, request: {{schematic.name.lower()}}_pb2.Create{{schematic.name.capitalize()}}Request
26     ) -> {{schematic.name.lower()}}_pb2.Create{{schematic.name.capitalize()}}Response:
27         try:
28             self.crud_component.create({{schematic.name.capitalize()}}(grpc_{{schematic.name.lower()}}_object=request))
29             return {{schematic.name.lower()}}_pb2.Create{{schematic.name.capitalize()}}Response(msg=SUCCESSFUL_TRANSACTION)
30         except Exception as ex:
31             logging.error(
32                 f"{{schematic.name.capitalize()}} create failed. Error: {ex}, type = {type(ex)}"
33             )
34             return {{schematic.name.lower()}}_pb2.Create{{schematic.name.capitalize()}}Response(
35                 msg=make_error_message(ex) + " " + str(request)
36             )
37     {% endif %}
38     {% if "read" in config["crud_instructions"] %}

```

[gRPC](#) is a different story entirely. Not only was it necessary to add additional code for each of the options, it was also required to create [dynamic protobuf generation templates](#) as well as a [shell file template](#) (bash) which is able to iterate through the protofiles and generate python code from them. And as was shown earlier in the [model templates subchapter](#) of the [adding database support chapter](#), there's gRPC logic which has spilled into the models since it wasn't obvious how it could be avoided.

The point is that the complexity of the handlers will vary greatly, though I find it unlikely that it'll get much more complex than gRPC granted that the technology in question isn't ancient. E.g. both RabbitMQ and Kafka are service types I was going to support in this project, and my estimate is that they would both be simple to implement relative to gRPC, though not to the extent that REST was.

Misc Templates

These templates are for a variety of things which will not directly be inside the generated project, in the sense that there won't be any generated directory called 'misc', whereas there will be one called 'utils'.

```

v misc
├── docker-compose_base.yml.jinja
├── grpc_dockerfile.jinja
└── rest_dockerfile.jinja

```

It's in this directory where you'll create the dockerfile for your service type. This will depend greatly on what technology you want to use. E.g. REST uses [gunicorn](#) for hosting the flask application. This isn't something we have to do with gRPC. Though it should be mentioned that all dockerfiles will interact with either poetry itself or with the requirements.txt file it generates.

```

snoozebox.py > gen > templates > misc > rest_dockerfile.jinja
1 FROM python:3.9-slim
2 COPY . ./app
3 WORKDIR /app
4 RUN apt-get update
5 RUN apt-get install {{apt_dependencies}} -y
6 RUN python3 -m pip install poetry
7 RUN poetry config virtualenvs.in-project true --local
8 RUN poetry install --no-dev
9 RUN poetry add gunicorn
10 WORKDIR /app/{{config["project_name"]}}
11 EXPOSE {{config["settings"]["server"]["rest"]["port"]}}
12 CMD ["poetry", "run", "gunicorn", "-b", "0.0.0.0:{{config["settings"]["server"]["rest"]["port"]}}", "main:app"]

```

Either way your dockerfile is unlikely to deviate much from the one above, although there isn't any immediate issue if it does. Just make sure the application gets hosted, and both the apt and python dependencies get installed.

Model Templates

The only thing to note here is, that it is possible to make service specific content here if necessary. It is best to avoid it, however, due to exponentially increasing complexity as databases and services get added.

```
{% if config["service"].lower() == "grpc"%}
from protogen import {{ schematic.name.lower() }}_pb2
{% endif %}
```

This is an example of service specific logic inside the [pg_model.py](#). The important thing is, that whatever you decide to add to one model has to be added to all of them. E.g. right now there's only Postgres, but if you wanted to add a new service type with service specific code inside the models, and if there were more database technologies added, e.g. [Cassandra](#), then you'd have to add this service-type specific code into both the Cassandra and the Postgres model variations. This is why the complexity is bound to rise exponentially with the current format. It might be possible to somehow inject this service-type specific model code into templates instead of writing them hard like this inside every jinja template, but for now it looks like this.

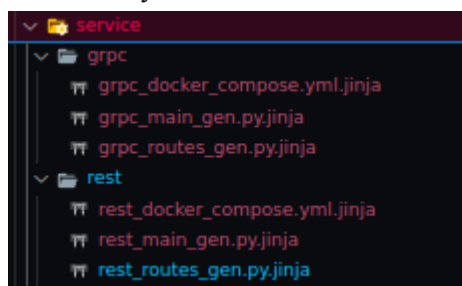
Protogen Templates

There won't be any code to write here, however, it'd be useful to clarify, that there is leeway to add code where it is necessary to do so. The gRPC protobuffer files and the shell files that execute them are gRPC specific code which are housed inside the protogen directory. These are jinja files which generate .proto and .sh files. If required, it is possible to add code as different as those two files.

Service Templates

Finally there are the service templates.

These are split into the docker-compose service entry, the routes which will be generated for each schematic, as well as the entire generated project's main file. This main file should always be small, neat and tidy.



The docker-compose entries so far have been fairly simple, and there haven't been much to add here. Do however keep in mind that the complexity of this specific file will drastically rise with technologies such as Kafka or RabbitMQ.

```

snoozebox_py > gen > templates > service > rest > rest_docker_compose.yml.jinja
1
2 {{config["project_name"]}}_rest_service:
3   build:
4     context: {{config["settings"]["file_structure"]["root_services"]}}/{{config["project_name"]}}
5   container_name: {{config["project_name"]}}_rest_service
6   ports:
7     - {{config['settings']['server']['rest']['port']}}:{{config['settings']['server']['rest']['port']}}
8   restart: always
9   networks:
10    - {{config['settings']['docker_compose_network']}}
11

```

Next there is the routes gen. The complexity of this specific file once again varies depending on which service-type you're using. The idea here is, that for every schematic of the generated project, there will be printed code from the routes templates. See for example the [grpc routes gen template](#).

Finally, the main file. These are to be kept as small as possible. An example would be the [rest main gen template](#), which is just 21 lines long. I am underlining this, because it is necessary to understand that this file is only for the execution process. It would be a common pitfall to add routes inside the main file for REST and Flask for example. It is important to separate this logic from one another as to keep things organised and compliant with the three-layer architecture Snoozebox is subscribed to.

Template Management

Since there won't be any tinkering with Snoozelib when we're just adding services, we'll go straight into the `template_management.py` file and add some entries to make sure the new service is available. This code, however, is exactly the same as the one in the database section, so [I'll refer to that instead](#).

And that concludes adding support for new services.

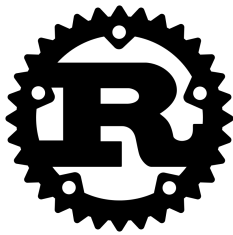
Supporting Additional Programming Languages

Before the project was called 'Snoozebox-Py' the project was called 'Snoozebox'. This change was made because the initial idea for the project was to cover multiple languages, and the tool itself was going to be written in Rust. Explanations on why that change was made can be seen in the [Language subchapter](#). While time-restrictions quickly made it apparent that multiple language support was unrealistic, the concepts of the three-layer architecture and the concepts used in the generated projects in the Python versions being universal aspects between the vast majority of all programming languages still holds true. This subchapter will be about what it'd look like if I had continued with multiple languages.

Because as was stated multiple times in the about chapter about [Continued Development Guidelines](#): **It can't be stressed enough how important it is to have a working sample project before beginning**

The fact of the matter is, that I've written services in multiple languages already with a variety of languages. The concept of Snoozebox was conceived over the similarities between the languages and technologies. While Snoozebox-Py generates Python with Python, there's no reason why it wouldn't be able to generate code for multiple languages while still using Python.

Rust Support



With gRPC, Rust uses a framework called [Tonic](#). This framework is part of [Hyperium](#)'s [tech-stack](#) built around their asynchronous [Tokio](#) runtime. This is because Rust's inbuilt runtime is on purpose [as small as that of C](#), to the extent that some people say it has none. The only implication of this is, that gRPC used with Rust will always be asynchronous, whereas it's [optional with Python](#)'s `grpcio`.

The concept of protobuffers is a universal one much like Rest. The .proto file generated with both frameworks is the same one. This means that the [protogen directory](#) in templates will be untouched.

Generally speaking the Rust result will in some aspects be cleaner. A comparison in one of the most complex parts should show the difference. It is still very much like the [Python solution](#).

This is part of what a Rust Jinja solution for the model could look like:

```
#[derive(%if config["database"] == "postgres"%]sqlx::FromRow, {%endif%}Serialize, Deserialize)]
pub struct {{schematic.name.capitalize()}}Model {
    {% for (variable_name, variable_type) %} in schematic.variables -{%}
    pub {{variable_name}}: {{variable_type}}
    {% endfor -{%}
}

impl {{schematic.name.capitalize()}}Model{
    {% if config["service"] == "gRPC" -{%}
    pub fn to_read_response(&self) -> Read{{schematic.name.capitalize()}}Response {
        Read{{schematic.name.capitalize()}} {
            {%- for (variable_name, variable_type) in schematic.variables %}
            [% if "time" in variable_type or "date" in variable_type %]
            {{variable_name}}: self.{{variable_name}}.unwrap().to_string(),
            {% else %}
            {{variable_name}}: self.{{variable_name}}.clone()
            {% endif %}
            {%- endfor %}
        }
    },
}
```

After execution of Jinja the result would look something like this:

```

tonic::include_proto!("person");

use serde_derive::{Deserialize, Serialize};
use chrono::NaiveDateTime;

#[derive(sqlx::FromRow, Serialize, Deserialize)]
pub struct PersonModel {
    pub id: i32,
    pub name: String,
    pub age: i32,
    pub occupied: bool,
    pub birthday: Option<NaiveDateTime>,
}

impl PersonModel {
    pub fn to_read_response(&self) -> ReadPersonResponse {
        ReadPersonResponse {
            id: self.id.clone(),
            name: self.name.clone(),
            age: self.age.clone(),
            occupied: self.occupied.clone(),
            birthday: self.birthday.unwrap().to_string()
        }
    }
}

```

Note that this is only part of the code. The last of the code hasn't been pasted here for readability reasons. Another thing is, that the code above is the result of running the [one_to_many01.sql.jinja](#) file with an extra datetime variable to showcase how optional datetimes are handled.

Versus the Python result which looks like this:

```

class Person(Base):
    __tablename__ = "person"

    id = Column(Integer, primary_key=True, autoincrement=True, nullable=False)
    name = Column(String(255), nullable=False, unique=True)
    age = Column(Integer, nullable=False)
    occupied = Column(Boolean, nullable=False)

    def __init__(
        self,
        id = None,
        name = None,
        age = None,
        occupied = None,
        grpc_person_object: person_pb2.NewPersonObject = None
    ) -> None:
        if grpc_person_object:
            self.id = grpc_person_object.id
            self.name = grpc_person_object.name
            self.age = grpc_person_object.age
            self.occupied = grpc_person_object.occupied
        else:
            self.id = id
            self.name = name
            self.age = age
            self.occupied = occupied

```

Note that the variables here are using SQLAlchemy data types. This isn't necessary with Rust because of the [sqlx](#) library it uses **instead** of an [ORM](#). It needs to be solidified that it by no means is a requirement that an ORM is used. ORM being Object-Relational-Mapping, i.e. a framework which translates the native code of a language to SQL behind the scenes.

Of course all this is speculation. In reality, Rust is a complex language, and there will absolutely be times where more drastic changes will have to be made. An example would be implementation of better type support for generated projects with different languages.

Typescript Support



One of the languages I wanted to cover during the brainstorming phase was Typescript with the [Deno](#) runtime. I drew some interest to the runtime as the languages which have monopoly over the frontend sector started entering the backend as well with Node. As this trend continues, [Typescript's popularity has increased greatly](#), as it in many aspects tames Javascript's flawed sides, even though Typescript is "just" a transpiled language; a superset of Javascript.

Typescript with Node.js is fairly janky, however, and with Javascript infamous npm modules and tooling, Deno was spawned with promises of making the Typescript experience more streamlined.

Arguably Deno's vision has failed as a result of its incompatibility with Node's rich amount of packages as Deno doesn't have anything close to the level of maturity that that of the [world's most used language](#) - An issue that is shared with newer programming languages that struggle when failing to meet the same amount of libraries and packages as older languages.

As such it is arguably a better choice to go with something like [Bun](#) which is both faster and has compatibility with older Node libraries.

Regardless, if Typescript is still one of the languages I was going to support with Deno, then while [I've already created REST services with Deno](#), those samples are lacking in the CRUD components for composition, and would therefore take more work than Rust would. That said, while there still are many unknowns, the concept of Snoozebox absolutely still applies to Typescript as well.

Java Support



I'd like to briefly comment on Java in the Snoozebox context as well since it is the primary language of the Software Development and Computer Science educations at Cphbusiness - I wouldn't suggest implementing Java for Snoozebox usage. The reason why isn't the language itself, but the extensive amount of additional boilerplate code spawned by Maven, handling of dependencies from the pom.xml, as well as the general inconveniences of using Java inside Docker containers. This is doubly true from a compatibility context from within docker-compose. That said, it is arguable that this can - at least partially - be circumvented by utilising [Gradle](#) instead of [Maven](#).

Either way compatibility difficulties between libraries, a [frustrating dependency format](#), packages, plugins and libraries with serious security issues, [Java specific settings with gRPC inside protobufs](#) (which sometimes makes the .proto files incompatible between languages which voids their purpose), massive amounts of boilerplate code, ad nauseum, has discouraged me from attempted to implement

Snoozebox bindings for Java. I do recognize that I may be biased in this consideration, but those are the reasons. In some cases these issues are transferred into other JVM using languages, but it'd be more content with using [Kotlin](#) or [Clojure](#) instead.

I don't perceive it as being out of subjective preference that [Android's primary programming language for applications has become Kotlin instead of Java](#) in recent times.

Continuing Development from Generated Projects

Which path the client decides to take post-generation isn't set in stone. I'd like to clarify, however, it'd be beneficial to install a gateway which connects all or some of the services generated. It should also be noted, that gRPC as a backend does in fact not work when communicating to a browser-based frontend. [Most browsers do not support http/3](#) to a high degree as of the writing of this report. A workaround could be connecting gRPC to a proxy, e.g. [Envoy](#) or [Nginx](#), and from there using [grpc-web](#) in the selected frontend framework. Another workaround is implementing a gateway, which sends and receives requests from the gRPC service, but exposes routes with Rest. The latter option was a feature which was briefly pondered upon, yet quickly considered too ambitious and out-of-scope.

Generated Gateways

It is not unfeasible to create templates which generate a Gateway which is capable of communicating with the other services. It was one of the ideas I had during development, but it was eventually cut out because of time-constraints. It would most likely be a service which dynamically exposes REST endpoints from the endpoints of whatever the other services are exposing. The advantage here is that the gateway would have the capabilities of sending requests to the other services. By doing this, the gateway is capable of exposing data which would otherwise be limited because of non-REST technologies such as Kafka, SOAP, RabbitMQ, gRPC, etc.

Retrospection

It has been a bumpy ride. The first version of the project ended up in the drink due to some unforeseen issues. Those same issues led to me having to scrap some important parts of the project, such as the programming language. It's not that I don't like Python, but personally I find it most useful as a scripting language. In larger projects its dynamic nature can make it seem unreliable, though I don't feel like that was the case in this project.

I also wanted to support more technologies and languages just to show just how universal the concept is. Even the three-tier architecture is applicable in other programming languages. I did not feel like I could solidify that point without having supported a vast amount of technologies. This just means that even the smallest proof of concept of this project is way off the rails in terms of setting a reasonable deadline. Supposedly the first version was a case of Icaros flying too close to the sun, and got burned out, or perhaps reversedly he never even got close because he had a broken wing. It failed either way.

During the second run I do believe I've managed to turn things around to some extent, but the lack of features still bugs me. Know that my mentality is that I want what I create to be actually useful. I've never liked the premise of spending an extensive amount of hours on a project which doesn't actually have any value whatsoever. Additionally, I wanted to create something closer to actual software.

If there is such a thing as next time, then I'll try to be more realistic. Proper planning and evaluation of what capacities of focus I'm capable of providing during the development timeline, would likely have resolved in opting for a smaller project which would still be useful somehow.

Conclusion

Conceptually, a CLI tool can generate a fully functional project from schematics. The main differences between different services are in its essence as a baseline, the schematics that they manage. There is no reason to reinvent the wheel in every single project. The smart solution is to copy older solutions and modify them to the new solution. The smarter solution is to create a tool which can do this for you in a project which is calculated and guaranteed to work. Snoozebox-py has succeeded in showing this despite its considerably smaller-than-expected size.

Further development would have to be made before this would have any corporate implications in the real world. The structure of the CLI tool exists and it should be relatively straightforward to insert newer services. However an overhaul to said structure could be useful to allow the initially envisioned package structure with the templates being downloadable and uploadable to the cloud. With even more languages and technologies supported, perhaps it would gain wind with the outside world eventually. For now the proof of concept exists.