Hybrid Architecture for Desktop and Web Applications

Louis Gabriel Jean Landelle

August 2021

Acknowledgments

I would like to express my sincere thanks to Prof. Bugnion for his valuable expertise, ideas and feedback during this semester. I also thank the team at Financial Risk Pilot AG for giving me the supportive environment and freedom to explore all the directions needed to finish this project. Finally I am infinitely grateful towards my parents, for their continuous love and encouragements.

Contents

1 Introduction

The software-as-a-service delivery model offers an increasingly popular mean for software companies to deliver their software system to their customers in a *Web deployment* form. The same software could still need to be released in an alternate installable *Desktop deployment* form, that works offline. For instance, an offline setup provides the option of running in an air-gap system, which provides the data protection guarantee to customers of an on-premise solution which cannot communicate over the web.

Cloud computing advances have made it possible for small teams or lone developer(s) to host web services with minimal infrastructure overhead, through paradigms such as infrastructure-as-a-Service and platform-as-a-Service abstracting more and more concerns away from the "domain developer". In an effort to save such developer resources, we want to create a deployment-agnostic codebase, which can be mutated into the two deployments based on a simple configuration switching. For our needs that means a single codebase that can be transformed into a Windows-installable offline desktop application, as well as a web application accessible over some public static URL, with both having the same functionalities, but also minor differences. With this decision we don't split those two deployments into two projects or teams, but view it as a single one instead. This is motivated by the vast majority of both Deployments sharing the same logic. There is not a lot of documentation that describe how to implement such a project, so the hope is that this report can provide an example on a solution stack, architecture and various implementation designs to achieve this.

First, we will present the solution stack, most notably the historical context and current state of Python-based web development, and present the associated technologies it offers to achieve our needs. A similar background will be given on the web deployment infrastructure side, discussing containers, orchestration and cloud frameworks. Our chosen solution stack works for our needs but it is only an example and parts could be switched for equivalent technologies.

Then, an architecture under this deployment-agnostic constraint will then be shown, achieving this single codebase for hybrid desktop and web deployments and their integration and deployment infrastructure, going form the high-level overview to the specifications of the sub-components of the whole system.

Afterwards, we will present how essential sub-parts of the architecture were implemented to be "hybrid" (i.e, deployment-agnostic), especially those that were found to be pain points in the overall development process. Those sub-parts were specifically needed for our project, but also general and can be re-used for other projects.

Finally, we open a discussion on a micro-service architecture to implement the same functionalities as the monolithic Python web server of the previous parts. Those micro-services are independent servers that communicate in-cluster to implement the same logic as the monolith. With our discussed design, we hope to provide a method to use the same codebase as before, but now to also enable an arbitrary partitioning of our server along split services (hosts). This can offer another layer of flexibility and scalability to the developer team, we will also discuss resulting drawbacks.

2 Background

As this report discusses concepts and technologies along a full web *Solution Stack* (the set of external frameworks and infrastructure tools used), this section provides an overview and the technical context on parts of said solution stack to the reader.

2.1 Web Foundations

2.1.1 Basics

- *Internet*: Global computer network using the Internet protocol suite to communicate.
- *WWW*: World Wide Web, a distributed, application-layer information system built on Internet, which provides access to Resources notably Hypertext, via HTTP
- *Resource*: An abstraction, qualifying anything formally defined as a single entity server-side, accessible at a URL which is available on the WWW. *Examples: a web page, a version number, a data model representation, a remote action with a result, etc.*
- *URL*: Absolute reference pointing to a Resource on the WWW, in the form of `scheme://[user:password@]host[:port]/path[?query]`
- *Hypertext*: A document containing references (Hyperlinks) to other documents. HTML is the standard language to write Hypertext, and Hyperlinks often point to some URL (but could also be relative links for example)
- *Hypermedia*: A disambiguation and superset of Hypertext when the document links to media other than text, such as images and videos.

2.1.2 HTTP

HTTP is a stateless application layer protocol for hypermedia information systems, with HTTP/1.0 introduced in RFC-1945 in 1996, HTTP/1.1 defined in RFC-2068 in 1997 and updated in RFC-2616 in 1999, then again in 2014 with RFCs 7230 to 7236, and HTTP/2 in RFC-7540 in 2015 [1].

Its high-level semantics have not changed since HTTP/1.1, the required summary is presented here.

2.1.2.1 HTTP Requests

HTTP defines a set of *request methods*, also known as *HTTP Verb*s, acting on Resources located at the URLs the request targets. They are defined as *Safe* when they do not alter the server state (read-only) and *Idempotent* when N requests produce the same state change as 1 request. It is important to implement those properties server-side, because requesting agents expect them to hold, for caching mechanisms for example.

Those used in our project are:

- `GET`: requests the representation of the Resource: It is Safe and Idempotent.
- `POST`: submit an entity in the payload to the Resource: It is neither Safe nor Idempotent.
- `PUT`: replaces the Resource with the payload. It is Idempotent, but not Safe.
- `PATCH`: partially alters the Resource based on the payload. It is neither Safe nor Idempotent.
- `DELETE`: deletes the Resource. It is Idempotent, but not Safe.

A basic *HTTP Request*:

```
1 GET / HTTP/1.1
2 Host: fr.wikipedia.org
3 Accept-Language: fr
```

1. The first line specifies `<method> <path> <version>`, here the method is `GET`, the path is stripped of protocol, domain, port and here is the root, and the request indicates using HTTP/1.1.
2. Subsequent lines are *HTTP Headers*, passing additional information with the request, in the `key: value` format. Standard and proposed headers are listed on the IANA registry, but an HTTP-based application is free to define its own.

In this example, the `GET` request does not carry a HTTP Body. We illustrate it in the HTTP Response. HTTP Requests can carry parameters in different ways:

- *Header parameters*: by definitions headers can parametrize the request via their `key: value` syntax.
- *Path parameters*: A server can expose a `users/<user_id>/preferences` parametrized path, and a `GET users/5/preferences` request here carries a `user_id=5` path parameter based on the server specification.
- *Query string parameters*: The query string appears after a `?` character in the URL, commonly used to allow for server-side filtering, sorting and pagination. `GET users?role=admin&page=10&sort=asc` would filter for admin users, sort on some agreed-upon method in the ascending order, and only the page#10 subset.
- *Request body parameters*: Typically a JSON object carrying information to parametrize the request, for instance `PATCH users/5/preferences {"lang":"fr"}` would set the preferred language to french.

2.1.2.2 HTTP Responses

A basic *HTTP Response*:

```
HTTP/1.1 200 OK
Date: Sat, 01 Sep 2021 16:52:15 GMT
Content-Length: 65536
Content-Type: application/json

{"version":"2021.09.01",...}
```

1. The first line specifies `<version> <status_code> <status_message>`. *HTTP Status Code*s are standardized and associated with status messages:
   - Informational responses in range $100 - 199$.
   - Successful responses in range $200 - 299$. Most relevant for us are:
     - `200 OK` indicating the generic success of the HTTP Verb
     - `201 Created` indicating a new Resource was created due to the request
     - `204 No Content` indicating the expected absence of a Resource, typically after a `DELETE`
   - Redirects in range $300 - 399$
   - Client errors in range $400 - 499$. Most relevant for us are:
     - `401 Unauthorized` indicating a wrong Authentication state
     - `403 Forbidden` indicating a wrong Authorization state
     - `404 Not Found` indicating the unexpected absence of a Resource
     - `409 Conflict` indicating the unexpected presence of a Resource conflicting with the request
     - `422 Unprocessable Entity` indicating a payload problem for an otherwise correct request
   - Server errors in range $500 - 599$.

2. The subsequent lines are HTTP Headers. Here `Content-Type` declares the media type of the HTTP Body.

3. A blank line signifies the end of HTTP Headers and the beginning of the HTTP Body.

4. The *HTTP Body*, sometimes called *payload*, contains the representation of the Resource requested.

2.1.3 Cryptography

The cryptography concepts for web development mentioned in this report are:

- *TLS*, Transport Layer Security, is a set of two cryptographic protocols which describes a method to establish a stateful connection and secure encrypted transport-layer channel between two parties. SSL (Secure Sockets Layer) is the name of older versions of the protocol developed by Netscape, but it's still used sometimes to refer to TLS. HTTP is a plaintext application-layer protocol (up to HTTP/1.1, at least). TLS uses two protocols to encrypt the HTTP request/response between the application-layer HTTP and the transport-layer TCP [2]:
  - The TLS Handshake protocol establishes identity via a third-party certificate authority, and then negotiates a common secret key using asymmetric encryption.
  - The TLS Record protocol provides a reliable connection, and can optionally perform symmetric encryption on all HTTP traffic (after the TLS Handshake agreed on the common secret)
- *Secure Hashing*: Transforming an any-length input string into a fixed-length output string using a recognized Cryptographic Hash Function, which is fast, minimizes collisions by having a roughly uniformly-distributed output space, and secure against known hash attacks.
- *Salting*: Concatenating a salt, short random string, to the input string before hashing. This is due to secure hashing outputs being vulnerable to rainbow tables (precomputed reverse hashes for common input strings) when used alone.

2.1.4 Application Programming Interfaces

An API is an interface exposing a set of data and functions between computer programs to exchange information [3].

A *Web API*, in the server-side context, interfaces two agents, the *API Service* and the *API Consumer*. In this sense, Web APIs are a commitment from the API Service to implement some *API Specification*, catalog of data and function signatures, towards the API Consumer. It can be viewed as the outermost layer of the API Service computer program.

APIs can be implemented with various protocols such as the *SOAP* messaging protocol for example, based on XML-encoded messages over HTTP. The specification is provided through the XML-based web service definition language defined by the W3C [4].

SOAP uses XML, which has drawbacks like its high verbosity and slower parsing speed compared to other data formats such as JSON or YAML, made to be human readable. A more fashionable approach is to use the naked HTTP protocol, sometimes alongside an architectural guideline such as REST, which is then free to use any hypermedia type such as JSON to represent Resources.

2.1.5 Representational state transfer

REST, Representational state transfer, was proposed in Roy Fielding's Doctoral thesis, *Architectural Styles and the Design of Network-based Software Architectures*. It is an architectural style for the distribution of Hypermedia on the Web emphasizing *scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.* [5]

REST compliance relies on six constraints, with one optional:

1. *Client-Server Architecture*: The software system is split into two halves: a server to receive, process, and answer or reject requests; and a client to originate such communication with the server and present the results to the user.

2. *Statelessness*: The request between the client and server contains the entire information describing the state of the application. In particular, the server should not hold an application state mutated by a particular client's requests (however, the server of course holds resources upon which the client performs create-read-update-delete operations).

3. *Cacheability*: Fetched Resources should have an implicit or explicit cacheability property. Cacheable resources enable the client to use an internal cache to avoid re-requesting equivalent resources.

4. *Uniform Interface*: Generality of components through four sub-constraints:

   - *Identification of Resources*: Resources should be identified in requests, for example through verbose URLs.
   - *Manipulation of Resources through Representations*: Resource are exposed as explicit representations which contain enough information to modify or delete it on the server.
   - *Self-descriptive Messages*: Messages include information to describe how to process it, for example requests to parse a document includes metadata such as the format to parse from.
   - *HATEOAS*, *Hypermedia as the engine of application state*: Responses contain URLs to other Resources that are accessible, enabling the client to *discover* the application from its single URL entrypoint only.

5. *Layered System*: The client shouldn't be able to know it is connected directly to the end server or to an intermediary such as caches or proxies.

6. (OPTIONAL) *Code-On-Demand*: Servers can also provide executable code to the client, for example self-contained client-side scripts.

API Services satisfying the REST constraints are called REST APIs.

In between ideality and practicality live REST-like services. Some services qualified as REST are misqualified and should be in reality be refered to as REST-like, satisfying only a subset of REST constraints. Often this happens through the partial coverage of the Uniform Interface constraint, such as the departure from HATEOAS [6]. This discussion is continued in relation to our project in § 4.5.1.

2.2 Pythonic Web

In this subsection we will discuss the technologies used to process HTTP traffic and transmit it to and from our domain-specific code. We use Python for multiple reasons, one being that we want to take the trade-off of having dynamic typing, easier introspection and reflection, monkey-patching, etc. in exchange of weaker component-level robustness because we are creating a prototype of an exotic, hybrid architecture. Python also has mature Web technologies which are used in large-scale production systems, and is widely known.

2.2.1 Web Server Gateway Interface

2.2.1.1 Context

HTTP / the WWW is static by default, as web pages are static files on the server that the agent requests. Per HTTP design, there could not be such things as dynamic web page content.

The desire for dynamic content however, lead to making web servers able to invoke external scripts to process requests and produce answers in the 90s. This was done by forking the web server process with whichever binary would run the script, and by inheriting the environment as a child-process, the web server could pass request information to the script through environment variables. The script had to know which environment variables to call upon and which request content they were supposed to contain. With different web servers and the potential convenience of webserver-agnostic scripts, this created a need for a standard.

A standard for calling command line executables to process requests was created at the NCSA and discussed in 1993 through the www-talk mailing list. [7]

Its adoption by developers de-facto standardized it, and an effort starting in 1997 to formalize it resulted in RFC-3875, with a first revision in 1998 and released in 2004.[8]

2.2.1.2 CGI

RFC-3875 describes the Common Gateway Interface (*CGI*) which specifies the responsibility shared between server and CGI-scripts. It describes the responsibilities of the server on how to invoke CGI-scripts, the mapping of environment variables names to purposes such as `REQUEST_METHOD` for holding the HTTP verb and `QUERY_STRING` for holding the URL query suffix, and the format of the response, outputted via `stdout`. [8]

CGI-scripts have a flaw: forking a child process and invoking a compiled script or even worse, the whole python interpreter, is a very slow process. In fact, it is most of the time elapsed to process a request.

To solve this, tools called Application servers (in reality, they are minimal web servers) perform pre-forking. They create "workers", by forking multiple child processes and pre-loading python interpreter instances and dependencies. The first-in-line web-servers such as Nginx would now only forwards HTTP requests to Application servers. Workers process requests one at a time (hence the need for multiple workers).

Our scripts could then be functions processing requests in actual python programs running over the lifetime of the workers, called directly by the Application servers, eliminating the large overhead.

To make Application servers and Python web frameworks organizing those processing functions inter-operate, a standard interface was needed.

This was answered by the Python community in PEP-333 [9], and later PEP-3333 for Python3 support [10], describing a simple and universal interface between web servers and web applications or frameworks, the *WSGI*: *Web Server Gateway Interface*.

2.2.1.3 WSGI Concepts

- *WSGI HTTP Server*: HTTP Server which is WSGI-compliant to interface with a WSGI Web Framework. *Examples: GUnicorn, uWSGI*
- *WSGI Web Framework*: This is a Python Web Framework package supporting the development of a Python Web Application and providing a WSGI Application Callable. *Examples: Flask, Django*
- *WSGI Application Callable*: Any Python callable with the signature `(environ:dict, start_response:Callable) -> body:list[str]`
- *Python Web Application*: A project implemented using a WSGI Web Framework, like our API

The WSGI Web Frameworks needs to expose the WSGI Application Callable to the WSGI HTTP Server. The mechanism of the WSGI HTTP Server for loading this Callable is not specified - GUnicorn for example, uses introspection to import the Callable from Flask. The Callable parameters are:

- `env` a dictionary containing environment variables using the CGI standard describing the request.
- `start_response` any callable with the signature `start_response(http_status, http_headers_list) -> None` with:
  - `http_status:str` like `"200 OK"`
  - `http_headers_list:list[(str, str)]` a list of tuples representing headers keys and values, such as `("Content-Type", "text/plain")`

The WSGI Application Callable needs to returns the HTTP Response body content, as a `list[str]`.

This provides a minimalist and flexible framework interface, which was deliberate in order to improve compliance of existing web frameworks.

An in-depth look into the mechanism of a WSGI HTTP Server (GUnicorn) interacting with a WSGI Web Framework (Flask) is provided in § 7.4.

2.2.2 Werkzeug

Flask depends on the *Werkzeug* toolkit, a WSGI utilities library to build web applications. We can see in § 7.4 how the WSGI HTTP Server workers repeateadly call the WSGI Application Callable. The `environ` and `start_response` arguments need some processing work to be usable in domain-logic. Werkzeug provides concepts and utilities to perform this work:

- *Werkzeug Request*: `Request` wrapping the `environ` for easy access to request data.
- *Werkzeug Response*: `Response`, itself a WSGI Application Callable, to be created at the beginning of the main WSGI Application Callable, interacted with by the domain-logic code, and called with the `(environ, start_response)` at the end, for nicer Response-making.
- *Werkzeug View Function*: Werkzeug's `Request` and `Response` allow the developer to instead work on $ViewFunction := Request \rightarrow Response$ callables to contain domain-logic, instead of directly in the WSGI Application Callable with its awkward parameters and two-step responding protocol. Synonym: *controller*.
- *Werkzeug Rule*: a URL pattern with potential path arguments, associated with its ID/Name, called *Endpoint*.

  $$Rule := (URLpattern, Endpoint)$$

  Patterns should look like `"/report/<int:year>/<int:month>/"`.
- *Werkzeug Map*: object collecting Werkzeug Rules, which can be used to pattern match on the `environ` target URL, to extract the Endpoint and path arguments key-values, with `Map.match`

  $$Map := [Rule_1, Rule_2, ...]$$

  We can see how Werkzeug allows us to define the domain-logic, by creating some mapping:

  $$Views := \{Endpoint \rightarrow ViewFunction\}$$

  And then in the WSGI Application Callable, were we to use Werkzeug only, we can extract the URL and argument key-values with the function `endpoint, args = map.bind_to_environ(environ).match()`, then call our mapping `views[endpoint](request)` and return a call of the resulting Werkzeug Response.

2.2.3 Flask

Flask is a Python Web Framework which builds upon the bare-bone Werkzeug as its core. Another popular choice for Python-based web development is *Django*, which provides a more opinionated, full-stack utility. Flask, instead, relies on community extensions or the developer for implementing more advanced web development features. Flask is the base layer upon which we choose to build the back-end/API of our software system, abstracting the vast majority of WSGI/Werkzeug considerations away from us, answering HTTP traffic to implement our API specification. The rationale for this choice is that our uncertain architecture would benefit from the flexibility of a less opinionated, less rigid framework.

As a WSGI Web Framework, it provides `Flask` class, an instance of which is a valid WSGI Application Callable. It essentially performs the exact logic in our last point. An in-depth look into the mechanism is provided in § 7.1, but its finality is to call the internal `self.view_functions[rule.endpoint](**req.view_args)`.

The *Flask View Function*, sometime called confusingly *"views"*, differs from the Werkzeug View Function, because it takes the actual path arguments and can return different types such as `str`, `dict`, and many more - thus the Flask developer can dedicate his time to write only the domain-logic inside the view functions.

This is due to the Flask package introducing features like wrapping the Werkzeug Request and Werkzeug Response, through the Application and Request Contexts.

We discuss features that were used extensively in this project:

- Application and Request Contexts
- Blueprints, for our own modularity
- Extensions, for Flask's modularity

2.2.3.1 Blueprints

Flask-based Python Web Applications are, for the developer, like a mapping $\{Endpoint \rightarrow FlaskViewFunction\}$. They begin as single python files and eventually need to be split up. Larger Flask applications tends to have partitioned sets of Flask View Functions being related in purpose/role and accessing the same internal states, so they want to live in the same files.

Blueprint provides the out-of-the-box feature of grouping related views, often in separate files, typically with 1 file = 1 `Blueprint`.

A Blueprint can be seen as a $\{Endpoint- > FlaskViewFunction\}$, like a Flask application.

Using Blueprints, we compose with `Flask.register_blueprint(blueprint)`:

$$flask\_app := Blueprint_1 + Blueprint_2 + ...$$

This can be leveraged for microservices, topic discussed in § 6.

NB - Blueprints are so useful that during the course of the project, unaware of their existence, I had to re-implement most of their functionalities, almost with the same wording (except I called them Components), out of the frustration of the classical views registering system when creating microservices.

2.2.3.2 Request Contexts

How does Flask manage to not use the `Werkzeug View Function` signature of $(Request- > Response)$? Instead of passing Request data as input, it provides a global proxy object pointing to the current Werkzeug Request treated. This is possible due to the one-at-a-time processing of request by WSGI Workers.

Requests handled are pushed onto a stack and the `request` proxy points at its top. The stacking of requests is rare, for purposes such as internal redirecting. The developer can use this proxy within each view function to access the Werkzeug Request data.

Flask views, as a result, were made to cleanly take as inputs the path arguments of the rules, i.e., the view for the rule `"/portfolios/<portfolio_id>"` will have a signature `(portfolio_id: str -> Response)`.

2.2.3.3 Application Contexts

Flask is designed to allow more than one `Flask` instance in the same Python process. This is a rare scenario, but it is permitted, a notable example for this being unit testing with tiny instances of the App.

A View function cannot know which `Flask` instance it is attached to, but like the Request Context, Flask provides a proxy pointing to the top of the Application context stack, `current_app`.

The Flask object contains application-specific/not thread-global data such as the config. The `app.config` is important as all Flask extensions are configured through this dictionary.

2.2.3.4 Flask Extensions

Flask is designed as a "microframework". It relies on *Flask Extension*s, community made, to augment it with additional features.

The Flask documentation provides a checklist for Extension behavior expectations:

- Extensions need to support working with concurrent `Flask` instances.
- Extensions are implemented with a main class, with handles to standard methods such as `teardown` for disposing of resources after a request.

- The main Extension class needs to get instantiated. It also needs to get initialized, and provided the `Flask` instance, with two acceptable scenarios. Taking the example of Flask-SQLAlchemy:
    1. During instantiation: via `db = SQLAlchemy(app)`, which is fit for small projects, without the need for concurrent `Flask` instances support. As a result, the extension instance is allowed to keep a reference to the `Flask`.
    2. Delayed: `db = SQLAlchemy()` then later, `db.init_all(app)`, which is fit for projects where concurrent `Flask` instances are needed. Here, the extension cannot keep a reference to `Flask` and **must** rely on the `current_app` proxy of the Application context.

2.2.4 GUnicorn

GUnicorn stands for *"Green Unicorn"*, it is a Python WSGI HTTP Server for production and through WSGI it is compatible with a large number of WSGI Web Frameworks, including Flask.

Flask contains a lightweight debugging HTTP server itself, but for production one should use a proper WSGI HTTP Server.

GUnicorn can be started against our Flask server main module, with a command such as:

```
gunicorn -b 0.0.0.0:5000 server:app
```

This makes GUnicorn start on the `server` module and searches for the `app` WSGI Application Callable. GUnicorn performs reflection on the specified module to find and use the WSGI Application callable, details about this process are discussed in § 7.4.

2.3 Data Background

Data terminology used in the report:

- *Relational*: paradigm for database systems, where a Relational Database is a set of tables, called relations, with the data fields described in columns similar to spreadsheets, using primary/foreign keys and bridge tables, to express the relationships. Fields are typed and typically the size is bounded, so this paradigm is fit for well-defined data specifications, but can be awkward to use for unstructured/semi-structured data, or structured data with a specification changing often over time [11].
- *Non-Relational*: paradigm for database systems, which departs from relation tables. It is broad, with sub-paradigms such as key-value-stores, graph databases, document stores, etc. In our context we will only use the key-value-store non-relational paradigm, so key-value-store and non-relational will be used interchangeably. It can be represented as a JSON-like store. Drawbacks due to the departure from the relational paradigm include the loss of data structure rigidity and the loss of join operations [11].
- *RDBMS*: Relational Database Management System, a software system that enables users to define, create, maintain and control access to a relational database [12].
- *Unit-of-work*: a record accepting successive changes to the RDBMS representing a business transaction, performing the actual alteration at the end - to avoid transactions for each change when the whole transaction is atomic.

We also define:

- *Model Definition*: Type describing a piece of data with fields, their type, constraints and relationships with other models (in this report, we use this term only in a Relational context).
- *Model*: An instance of a Model Definition type.
- *Model Representation*: A representation of a Model instance.
- *Model View Function*: A Flask View Function serving requests on the model Resource, typically accepting and returning a Model Representation
- *Non-Relational Model*: the Non-Relational counterpart to the Relational Model. We choose to use the Relational Model for data which *suit*s defined, typed fields (i.e. most data), therefore non-relational models are reserved for data which needs to handle dynamic fields and types.

2.3.1 DBAPI

Python's popularity resulted in many packages to interact with RDBMS. There was a need for consistency and standardization across those modules. Such a standard was introduced through PEP-248 [13] and then PEP-249 [14], describing an API to encourage similarity between the Python packages used to access databases: the *DBAPI*, *Python Database API Specification*.

The defined interface provides, among others:

- A set of types, notably to split the python string into database specific types that it can represent. Types include `Date`, `Timestamp`, `ROWID`, etc.
- A set of database access related errors sub-classing the python exception (defined as `StandardError` in the PEP written in the time of Python2, now `Exception`)
- A *DBAPI Connection* class interface managing access to the database, which creates DBAPI Cursors, must be opened and closed, and can either `commit` or `rollback` transactions.
- A *DBAPI Cursor* class interface, which interfaces with the cursor concept of the underlying database or implements it virtually if the underlying has no cursor support. As for RDBMS cursors, its purpose is to provide lazy/stream access notably for large transactions, which are sets of queries as one Unit-of-work. Cursors get executed, which yields `1`, `N>=0` or `None` resp. for Data Definition, Data Manipulation (`N` being the count of rows affected), and Data Queries. Queries return `None` to indicate that results need to be fetched from the cursor. The interface then specifies fetching functions, like `fetchone`, `fetchmany`, `fetchall`.

Old modules were incentivized to adapt and become compliant. New modules had an official framework to comply with. As a result, there exists popular modules respecting the Python DBAPI specification creating Python interfaces for most RDBMS, such as:

- sqlite3 for SQLite (in the Python standard library)
- psycopg for PostgreSQL, or pg8000 for a pure python alternative
- mysqlclient for MySQL

DBAPI Drivers are those DBAPI-compliant python packages. They facilitate generalist ORMs such as Django-ORM and SQLAlchemy because they can use the same interface while changing DBAPI Drivers.

2.3.2 Object Relational Mapper

An *ORM* (Object Relational Mapper) is a technology that creates a bridge between the object-oriented world and the relational world, eliminating duplication of data, maintenance cost and susceptibility to error associated with it.[15]

Summarized more informally, for a Python program, an ORM would be a framework that allows the developer to define data models as Python classes. It would then create a mapping from those classes to their representation as relations in a RDBMS, and give the developer Python functions to perform database transactions. In that sense, the developer interacts with a Python abstraction layer on top of the database.

We have seen that the DBAPI provides a single interface for RDBMS-specific packages for connection and cursors, but doesn't cover the actual SQL transactions. This is one of the jobs of SQLAlchemy, notably through its `Dialect` concept.

SQLAlchemy is an ORM that can be used to create data models and expose methods to manipulate database access away from the specifics of the different RDBMS.

SQLAlchemy implements this ORM using structures:

- *SQLAlchemy Dialect*: Since multiple DBAPI Drivers exist for the same RDBMS, such as psycopg2 and pg8000 for Postgres, SQLAlchemy uses `Dialect` and sub-classes to represent:

$$Dialect := (RDBMS, DBAPIdriver)$$

`Dialect` is the parent of `DefaultDialect`, which is sub-classed in RDBMS-specific classes like `SQLiteDialect`, `PGDialect` which prepare SQL statements in the RDBMS's flavor. Then, they are sub-classed again for specific DBAPI Drivers, to get classes such as `PGDialect_psycopg2`, `PGDialect_pg8000`, etc.

- *SQLAlchemy Pool*: The `Pool` is an abstract base class for a pool of active DBAPI Connection in memory, re-used across requests for efficiency.

$$Pool := [DBAPIconnections]$$

  The default implementation of `Pool` is `QueuePool`. `Pool.connect()` returns a `_ConnectionFairy` implementing the DBAPI Connection concept.

- *SQLAlchemy Engine*: The `Engine` is the main instance of SQLAlchemy applications. It joins the `Dialect` for preparing DBAPI-driver-compliant transactions, with `Pool` to network them.

$$Engine := (Dialect, Pool)$$

  `sqlalchemy.create_engine(url)` prepares the `Engine` instance, making the `Dialect` from the URL's scheme, like `"postgres+psycopg2://..."` vs. `"postgres+pg8000://..."` (`"postgres://"` will use the default psycopg2). The `Engine` is created once per URL and typically held globally for the lifetime of the application.

- *SQLAlchemy Session*: `Session` tracks `SessionTransaction` instances, one at a time. `SessionTransaction` is the Unit-of-work, their lifespan limited to the business transaction (implemented as a Python context manager). As such, the `Session` is like a super-`Transaction`. The `Session` instance is bound with the `Engine`: `with Session(engine) as session:...` and serves as a "holding zone" for Models changes.

$$Session := (Engine, SessionTransaction, \Delta State)$$

  On `Session.add` of a Model, it gets *instrumented* which binds their instance state changes into changes event in the `Session`. `Session.commit`, `Session.query`, or `Session.execute` executes the `SessionTransaction` SQL Statements via the `Engine`, or it can be canceled with `Session.rollback`.

2.3.3 Flask-SQLAlchemy

Flask-SQLAlchemy is the Flask Extension wrapping SQLAlchemy.

Flask-SQLAlchemy is officially recommended by SQLAlchemy for Flask projects. It provides notably:

- the `SQLAlchemy` extension object, managing `Engine` and `Session` based on the `app.config["SQLALCHEMY_DATABASE_URI"]`
- managing multiple `Engine` when switching the `SQLALCHEMY_DATABASE_URI`
- a `Model` base class for making data Model classes easy
- the `SQLAlchemy.session` proxy to an internal `scoped_session` registry, letting the developer forget about `Session` - also resulting in the alignment of the lifespan of a SQLAlchemy Session instance with the lifespan of the HTTP Request (automatic `Session` tear-down)

Basically, the only things we have to do is to define the data model classes, and in views, perform session operations, and commit/rollback.

2.3.4 Marshmallow

Marshmallow can be qualified as an *ODM*, *Object Document Mapper*, which bridges object types to and from their document representation. Marshmallow being a Python package, document representations are `dict`, to represent JSON.

With our definitions it can be used to translate Model objects to and from their JSON representation, the Model Representation, to interface directly with the HTTP API.

Marshmallow could also be used as an ORM for a non-relational database such as MongoDB, since it works on JSON-like documents.

In Marshmallow, the developer implements *Marshmallow Schema Definition* types which defines the representation of an object type. It then provides two methods:

1. `Schema().load` which accepts the Model Representation and produces the Model that it represents.
2. `Schema().dump` which accepts a Model and produces its Model Representation

2.3.5 Flask-Marshmallow

Flask-Marshmallow is the Flask Extension wrapping Marshmallow.

It is a thin integration layer for Flask and marshmallow which notably provides support for URLs fields mapping Flask endpoints to the URL to access them.

2.3.6 Marshmallow-SQLAlchemy

Marshmallow-SQLAlchemy is the Python package bridging the SQLAlchemy and Marshmallow packages.

After defining a new or changing an existing Model type, defining or maintaining the Model Representation type explicitly is tedious as the number of models increases.

In a nutshell this package allows us to create Marshmallow Schema Definitions from SQLAlchemy models automatically.

Using it, we implement a Marshmallow Schema Definition on an *difference*-basis where potential fields could be excluded but most would be directly represented. This is much more efficient for the developer. For APIs in fact, it can be desirable to keep model resources as close as possible to the underlying, so most Marshmallow Schema Definitions will be implemented to reflect the direct representation of the database model.

This topic is discussed further in § 5.2.2.

2.4 Cloud

2.4.1 Virtualization

Virtualization in computing refers to the abstraction of some physical component into a logical object. Traditionally server-side software would run on physical server hardware. Virtual machines can provide resources from their physical counterparts such as an OS, network drivers, storage access, etc. This increases flexibility because virtual machines can be cloned, upgraded, moved, without the disruption of a similar operation in physical machines. [16]

Virtualization can be performed via *Hypervisor Virtualization*. The hypervisor abstracts the physical layer and presents this abstraction to VMs becoming an interface between the hardware devices on the physical server and the virtual devices of the VMs. VMs are then booted on the hypervisor, detecting and using the virtual hardware devices. It can be installed directly onto a server (native or bare-metal hypervisors) or on top of a host OS (hosted hypervisors). [17]

Virtualization is used in the context of Cloud Computing by turning data centers into highly scalable and available pool of resources. Virtualization frees the time of system administrators from host hardware and host OS concerns, reducing costs and increasing deployment abilities and automation capabilities. Virtualization enables the emergence of the "virtual data center" which abstracts the physical layer away from administrators, and as such is the cornerstone of the modern Cloud Computing industry. [16]

Container Virtualization is a lightweight alternative to Hypervisor Virtualization. Instead of emulating an OS on top of virtualized hardware as in VMs, containers use isolation of processes at the OS-level and

run on top of the actual host OS kernel, in Linux typically via control groups: `cgroups`. Applications inside the container are not aware that they are running inside a container, and they are not aware of the host OS.

Container-based virtualization typically offer better performance than Hypervisor Virtualization, with the trade-off of less perfect isolation of resources. [17]

2.4.2 Containerization with Docker

Docker is a container-based solution which used to extend LXC (LinuX Containers, a low-level container-based solution) with a kernel- and application-level API and tools for building and packaging applications into portable environments called *Container Image*s (in newer versions Docker replaces LXC with its own Go component, `libcontainer`). Docker is used to start a *Container* from a Container Image.

Docker container images are bootstrapped from base images. Typically, base images include at least a base OS. New images are created by layering/chaining on top of base images, in a process automated using *Dockerfiles*. Dockerfiles are scripts composed of commands executed sequentially to bootstrap the new image from the base image. [18]

Typically, a Dockerfile is structured like this:

```
1  FROM python:3.9
2
3  WORKDIR /code
4
5  COPY ./api /code/api
6  COPY ./requirements.txt requirements.txt
7  COPY ./entrypoint.sh entrypoint.sh
8
9  RUN pip3 install -r requirements.txt
10
11 EXPOSE 5000
12
13 CMD ["/bin/bash", "entrypoint.sh"]
```

- The `FROM <name>:<tags>` instruction specifies the Parent Image from which we build. In this example, `python:3.9` is an official image on the Docker Hub.
- `WORKDIR <path>` specifies a path prefix for subsequent relative paths in the image.
- `COPY <p1> <p2>` copies a file or directory from local filesystem to the image filesystem. Here we copy our API Flask server code, requirements, and a shell script starting GUnicorn for our Flask app on port `5000`.
- `RUN <command>` executes the command in a shell, by default `/bin/sh -c`. Here we use it to install Python packages requirements inside the image.
- `EXPOSE <port>` instructs to listen on `port` at runtime, by default TCP. We expose `5000` because `entrypoint.sh` starts the server against it.
- `CMD [<command>, *<params>]` can appear at most once and provides defaults for an executing container. Here by default we execute the shell script `entrypoint.sh` on execution, starting the server. The content of this shell file in our project is the command given in § 2.2.4.

`docker build -f <path> .` will build an image from this Dockerfile. After build, we know that we have a self-contained server that will work on any compatible Docker engine, removing the "works-on-my-machine" problems of bare metal or misconfigured VMs.

2.4.3 Container Orchestration

A software service can be created to run inside a Docker container, but more complex services could be imagined to run as a set of concurrent containers, communicating over TCP/IP. A very basic example of such a software service would be a server container communicating with a database container for data permanence. This concept is generalized with the microservices architecture, where core functionality is decomposed into small containerized services, communicating via messages or events.

This paradigm creates a need for container orchestration platform. A *Container Orchestration Platform* manages a *Cluster* of containerized services. It should handle fault-tolerance, availability, and scalability. With such a platform, risk scenarios can be gracefully managed to provide continuous up-time of the service as a whole, for instance:

- A service suffers a critical failure → replaced with a new clone.
- A service is under stress → scaled to handle the growth of traffic (the number of clones is increased and traffic is distributed among copies).
- A service can upgrade → new copies replace old copies seamlessly.

As such, key features of Container Orchestration Platforms include [19]:

1. Cluster State Management and Scheduling, through flexible scheduling of tasks across the cluster, garbage collection, backups, reliable state management and partitioning of data across the cluster, etc.
2. High availability and Fault Tolerance: failure detection, redundancy, reliable crossover, etc.
3. Security: identity and access management policies, network traffic segmentation, secret management, etc.
4. Networking: port allocation, intelligent routing, etc.
5. Service Discovery: highly available and up-to-date service registry, scalable load balancing routing services, etc.
6. Continuous Delivery and Deployment: non-disruptive changes roll-outs, security best practices in the container life-cycle, etc.
7. Monitoring and Governance: on two places, host infrastructure monitoring and container activity monitoring, with definable policies to react.

Docker provides an out-of-the-box solution for this via *Docker Swarm*. Another recent key-player Container Orchestration Platform is *Kubernetes*.

2.4.4 Kubernetes

Kubernetes, often shortened *k8s*, is an open-source container orchestration platform.

Its key components and concepts are [20]:

- *Kubernetes API*: Part of the Control Plane. This is an HTTP API consumed by `kubectl` to query and modify the state of Kubernetes Resources in the cluster. Such resources include Kubernetes Pods, Kubernetes ConfigMaps, etc.
- *Kubectl*: A command line tool with commands to speak to the Kubernetes API, used for convenience by the administrator or CI/CD pipelines instead of direct HTTP requests to the API.
- *Control Plane*: A layer exposing the Kubernetes API and containing global components to define, deploy and manage the life-cycle of Containers. It is the heart of the cluster.
- *Kubernetes Node*: A single host, worker machine in Kubernetes, running Pods.

- *Kubernetes Pod*: The smallest Kubernetes object, a set of running containers on the cluster (technically a set, but, in simple projects, a Pod will host a single Container). Those containers get the same IP address and port space, can communicate using `localhost` and access a shared local storage, etc.
- *Kubernetes ConfigMap*: A resource used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables.
- *Kubernetes Deployment*: A declaration of *desired state* of some Pods/ReplicaSets. The actual adaptation to current from desired state is left to deployment controllers. This is a frequently written resource for a Kubernetes-based project.
- *Kubernetes Service*: A Resource exposing an application running on a set of Pods as a network service, to users or other services. They can be discovered by an internal DNS mechanism. Important implementations include:
  - *ClusterIP*: To expose the service only within the cluster, with an internal IP assigned.
  - *NodePort*: To expose the service on each Node's IP at a static port. A ClusterIP Service, to which the NodePort Service routes, is automatically created. We can access the service from outside the cluster `<NodeIP>:<NodePort>`.
  - *LoadBalancer*: To expose the service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- *Kubernetes Manifest*: A specification of a Kubernetes API resource, in JSON or YAML format.

Kubernetes Resources can be directly created via the `kubectl create` commands, communicating to the Kubernetes API a JSON manifest.

Kubectl also has a `kubectl apply` command that applies a *Kubernetes Manifest* record of intent. We write manifests in YAML files. One can also use a tool such as Helm to get a different kind of control over the cluster, discussed in § 2.4.5.

2.4.5 Helm

Helm is a package manager for Kubernetes. It provides us an environment to split our Kubernetes Manifests in separate files in a *Helm Chart* directory.

Instead of having to manually `kubectl apply`ing all the manifests (if we were to manually split the manifest, the alternative being having a single huge manifest file), Helm provides us with a `helm install` command to install a Helm Chart as a single "package" called the *Helm Release*. It can then be `helm uninstall`ed or `helm upgrade`d a single unit. Helm version controls the releases and provides us with tools such as `helm rollback` to revert to an old release version. Thus Helm becomes the single point of authority over our manifests [21].

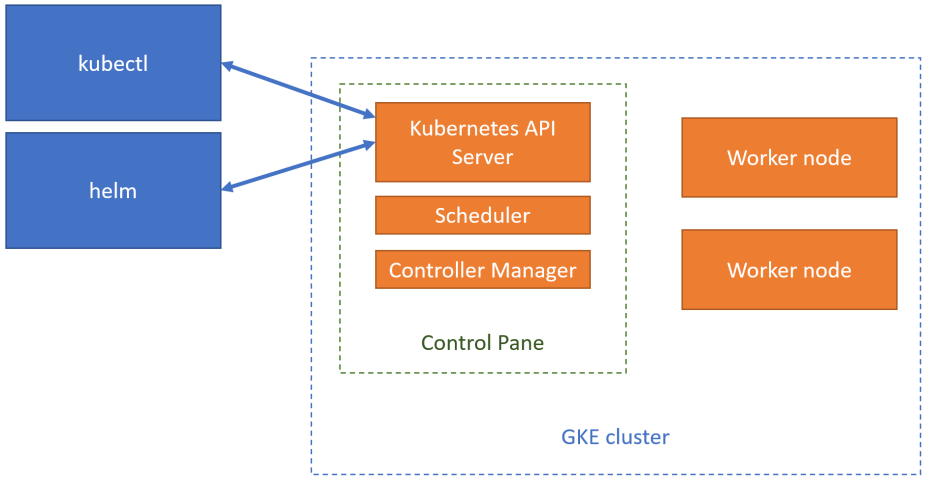

Figure 1: Kubectl/Helm both access the Kubernetes API

Another feature of Helm which makes it essential for this project is its templating engine. Our manifests are actually considered *Helm Templates*. They are placed in the `mychart/templates/` directory. We can create alongside this directory a `mychart/values.yaml` files:

$$Values := \{k \to v\}$$

Through its templating engine, directives such as `{{ .Values.my_value }}` can be used within the Kubernetes Manifest templates, to insert values during `helm install/upgrade`. Helm gives us powerful templating features such as piping and built-in functions such as `quote` to surround a string with quotes.

$$Chart := Values \to Template_1(Values) + ... + Template_N(Values)$$

$$Release := install(Chart(Values))$$

The power of Helm templating is that value files or individual values can also be replaced during Helm command calls, by passing flags such as:

- `helm install ... -f other_values.yaml` to switch value files from the default
- `helm install ... --set x=abc` to parse `{{ .Values.x | quote }}` as `"abc"`

In essence Helm allows us to package, manage and parametrize the Kubernetes Manifests.

2.4.6 Google Cloud Platform

- *Google Cloud Platform*: The Cloud Computing suite offering from Google
- *Google Kubernetes Engine*: The GKE is a managed service for running Kubernetes. It provides connected GCP services interfacing with the cluster, for instance to connect cluster LoadBalancers with GCP load balancers, or to automatically provision GCP persistent disks from cluster persistent volume claims.
- *Google Container Repository*: Docker Container Images may be pushed and pulled from repositories. The Docker Hub can be used to manage repositories, but for organizations this is a paid service. The Google Container Repository provides the same functionalities, one use case is to push images on the GCR within the CI/CD pipeline and reference it in a Kubernetes deployment to pull it from there.

2.4.7 Github Actions

Github Actions is a service provided by Github which allows the repository developers to automate tasks during the software development life cycle. The basics is that the developer can add *Workflows* to the special directory `.github/workflows`, as YAML files.

Those YAML files contain descriptions of event triggers, which specify when the workflow needs to be executed by Github Actions.

For instance:

```yaml
on:
  pull_request:
    branches: [ development ]
```

This triggers the workflow whenever a pull request to merge with `development` is created.

Then, workflows define sequences of jobs which can call upon other workflows from official Github Actions repositories (for example, there are many for interacting with the Google Cloud Platform), or call series of bash commands on a remote Github server which has a copy of our codebase (for example, to perform automated tests).

```yaml
jobs:
  tests:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.9]
    steps:

    - uses: actions/checkout@v2

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install requirements
      run: |
        python -m pip install --upgrade pip
        ...
```

For simple cases, those YAML files are the only infrastructure the developer has to setup. This makes Github Actions particularly fit for CI/CD pipelines for the developers which already source control on Github. It can run workflows before accepting pull requests for Continuous Integration. It can run workflows for specific tags, for published releases, for commits on specific branches, etc. for Continuous Deployment. Both can be done with minimal overhead, compared to what would be needed for other, external CI/CD options.

2.5 Authentication, Authorization

2.5.1 Terminology

- *User*: A Resource representing the unique identity of an agent and containing its credentials, typically in the form of a $(username, password)$ pair.
- *Registering*: Process of appending a User to the server collection.

- *Logging-In*: Process of establishing an authenticated communication channel following a User login request by matching its request credential with its stored credentials (via Authentication).

The difference between Authentication and Authorization is:

- *Authentication*: Process of verifying who a User pretends to be.
- *Authorization*: Process of verifying right of access to some Resource for this specific authenticated User.

This distinction is reflected by the difference between the `401 Unauthorized` and `403 Forbidden` HTTP Status Codes. `401 Unauthorized` is fit for missing or bad authentication (it should actually be called unauthenticated in this sense), where `403 Forbidden` is fit for lack of Authorization under proper Authentication.

2.5.2 Server-Side Sessions

HTTP being a stateless protocol, HTTP Requests have no knowledge about the previous ones. This is a problem if we want an initial request to authenticate a user and a subsequent one checking its authorization for some Resource.

Server-Side Sessions are a method for establishing an authenticated communication channel between a user and the server. A Session stores different kinds of temporary information related to the user across different requests. It is implemented with a server-side store of the session data, plus a cryptographically-signed cookie included in responses from the server identifying the session. The cookie is returned in subsequent requests to the server, which is used to load the session data back.

Note that this approach breaks the REST Statelessness constraint.

Flask-Session is a Flask Extension wrapping utilities for establishing server-side sessions, but we will instead go a different route with JSON Web Tokens.

2.5.3 JSON Web Tokens

JWT, JSON Web Token, is an Internet standard defined as *a compact, URL-safe means of representing claims to be transferred between two parties.* Its definition is proposed through RFC-7519. [22]

A JWT looks like three strings separated by two dots, like: `header.payload.signature`.

The three components to a JWT are:

1. The *JWT Header* specifies at least the token type and signing algorithm such as HMAC SHA256 or RSA: `{"alg": "HS256", "typ": "JWT"}`. This JSON is Base64Url encoded to form `header`.
2. The *JWT Payload* contains all the claims about the user as key-value pairs.
   - Registered claims: mandatory but not recommended, their keys and value expectations are specified in RFC-7519, like `exp` for the JWT expiration timestamp.
   - Public claims: defined at will but to avoid collisions, the community defines them in the IANA JSONN Web Token Claims registry.
   - Private claims: defined at will by the developers, this could include `username` and `role` for example.

   This JSON is Base64Url encoded to form `payload`.
3. The *JWT Signature* prevents man-in-the-middle attacks by tamper-proofing the JWT via a secret known by both parties. It is the result of the `alg` specified in the JWT Header applied to the string `header.payload` and the secret.

Therefore the end result JWT looks like:

```
1  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
2  eyJsb2dnZWRJbkFzIjoiYWRtaW4iLCJpYXQiOjE0MjI3Nzk2Mzh9.
3  gzSraSYS8EXBxLN_oWnFSRgCzcmJmMjLiuyu5CSpyHI
```

It should not be mistaken as encrypted, its content being only `base64` encoded.

The JWT is passed as a *Bearer Token*, included in the HTTP Headers of each request after a successful login, in the form `{"Authorization" : "Bearer " + jwt}`.

The advantage of JWTs over server-side Sessions is that it restores the statelessness of our HTTP traffic. The JWT "session" is thus kept client-side, and the entire "session state" is kept as the minimal set of data base64-encoded inside the JWT, such as the username and token expiration date.

It's also useful for micro-services architectures: instead of having to coordinate sessions between all independent services, the JWTs can just be carried inside the internal HTTP requests, and treated as-is by all micro-services for Authorization.

2.5.4 Flask-JWT-Extended

Flask-JWT-Extended is a Flask Extension providing utilities for working with JWTs in Flask projects, replacing and extending the older Flask-JWT project on Github after abandonment by its core developers.

Server-side, its features can be used for a basic user system:

- The `app.config["JWT_SECRET_KEY"]` value is set to the JWT secret string.
- The extension is initialized with the classical `jwt = JWTManager(app)`.
- A *User Identity Object* type is picked, for example a custom `User` type.
- A function needs to be decorated with `@jwt.user_identity_loader`, accepting some User Identity Object and converting it to a JSON serializable format.
- A function needs to be decorated with `@jwt.user_lookup_loader`, accepting two parameters, the JWT header and data, and returning the User Identity Object if the JWT results in a successful user lookup.
- Sensitive Flask View Functions can be protected with `@jwt_required()`, and inside those functions use the `current_user` proxy to access the current `User` instance which sucessfully authenticated to access it (if unsuccessful, the extension handles `401`s by itself). Authorization can be implemented by the developer, for example by including a `role` private claim in the JWT that is checked against the server-side `User` in role-specific view functions, or simply partitioning the data per-user where only resources created by the user can be seen, via filtering by the `username` in all queries.
- Our Registering Flask View Function would be unprotected and accept registration forms to append new `User` objects to the database, preferably after a human verification, even if outside of the scope of our prototype.
- Our Logging-In Flask View Function would be unprotected and accept a `User` credential form like the `username` and the `password`, then perform a secure password credential check against the password hash of the matching `User` in memory, and finally produce a JWT using Flask-JWT-Extended's `create_access_token(identity=user)`.

Client-side, it is expected to cache the JWT after the login request and include it in subsequent request headers with `{"Authorization" : "Bearer " + jwt}`.

A minimal demonstration of some of those concepts is provided with the code snippet in § 7.3.

2.6 Client-side

A classical way of writing Ajax interactive web pages/clients is to combine HTML, CSS, JavaScript and jQuery to modify the HTML DOM. Overtime, methods to provide reactive UI based on the underlying data bindings were developed to simplify the front-end developer's job. A modern choice for building asynchronous Web applications, and especially single page Applications, is *Vue.js*.

Using Vue over plain HTML/CSS/JS/jQuery provides organization, code conciseness, and clarity. This is due to its mechanism based on writing reactive components. A way to write Vue components is through Vue "single file components" files, which combine `<template>`, `<script>` and `<style>` sections. To summarize their basic roles and frequently used features:

- The `<template>` section is filled with an HTML template where the developer declares how the DOM should be rendered. The templating syntax augments plain HTML by providing a mustache syntax (double curly braces) to insert data from the `<script>` section. It provides flow control with `v-if`, `v-show`, `v-for` directives. It provides reactivity to HTML tag attributes via the `v-bind` directive. The developer can insert sub-components after binding them to custom HTML tags, and pass data to them via *props* with `v-bind`. It provides an event system with `v-on` directives triggered when sub-components fire events. The directives can call upon data, methods, and computed properties defined in the `<script>` section, conveniently in the same file.

- The `<script>` section describes the component data. It needs to export a component object that can hold script methods in `methods`, computed properties in `computed`, an internal state in `data`, etc. It also contains built-in events such as `created` and `mounted`, to run methods respectively before and after the component is rendered. As said previously, this object can be interacted with in the `<template>` section. This section is usually written in JavaScript, there is also support for alternatives like TypeScript.

- The `<style>` section is used to insert CSS styling, and the convenient `<script scoped>` attribute restrict the styling directives to the single file component only.

A Vue client is thus made of a tree-structure of nested, re-usable components, which contains in single files the traditional HTML/CSS/JavaScript separated sections. The developer builds the application, inspecting the result using `npm run serve` to have a local self-hosted preview which is hot-reloaded when the codebase changes. Then, the traditional HTML/CSS/JavaScript files can be generated from the Vue codebase, through a command like `npm run build`, and hosted.

The classical hosted files can be consumed by any modern web browser, and can also be packaged to be consumed as an *Electron* application. Electron uses *Node.js* and the *Chromium Rendering Engine* to serve Desktop GUI applications development through web technologies such as JavaScript+Vue.js, as if it was a tiny web browser with no GUI on its own, only rendering the served static files. This allows offline Desktop deployments of the web client.

3 Problem Statement

Lone developers or tiny teams have limited resources for software development, and developer-time is a very important, scarce resource. Ideally, most of the developer time is focused on domain-logic, i.e, writing the data structures and routines to implement the features directly exposed to the end-user, as this is where the business value is ultimately created. Under this fact, we had to implement a Software service exposed via two different delivery channels, or *Deployment* (A method of transfer or copy of the constituents of a Software System via some Infrastructure from some Producer Site to a Consumer Site [23]):

1. *Desktop Deployment*: A Deployment resulting in our Software System being available as a Windows executable program, offline - referred to as `desktop`.

2. *Web Deployment*: A Deployment resulting in our Software System being available as a Web-hosted *Software-as-a-service* - referred to as `webapp`.

Here, we define the *Software System* as the coherent collection of artifacts, such as executable files, source code, data files needed at the end-user site to offer our service[23].

While the straight-forward process could be to split the time between two different projects (with varying level of re-usable logic that could potentially be consolidated within a third project to serve as a common library), it seems attainable to create a fully generic, hybrid architecture and codebase that has the ability to generate both Desktop and Web deployments. This represents a substantial initial time investment but once such a setup is laid out, this could yield returns by having the developers working with stronger focus on this single codebase. As a by-product this architecture would also force the Desktop and Web Deployments to not deviate too much from one-another.

We can imagine our Software System formed with many sub-components such as a presentation layer, a Relational Database System and a User system for the data access layer, a business logic layer, infrastructure, etc. All of the sub-components of the Software System would have to be written with this hybrid requirement in mind.

We will attempt to converge towards architectural patterns and implementation details for those common sub-components which satisfy this requirement, as well as ways to assemble them to form the software service. This hybrid codebase idea can be defined formally with a term:

- *Deployment-Agnosticism* of the Software Codebase: Allow both deployments from a generalized, unique Software System codebase.

To complement this, we will also have to create the *Infrastructure* (A pipeline of ancillary systems configured to deliver the Software System from the producer to the consumer site). To continue with our goal to provide the maximum focus on the domain-logic, we would also like to automate as much of the infrastructure concerns as possible, which will imply two additional goals:

- *Continuous Integration*: Enable the developer to write tests that are automatically applied against the codebase, triggered by a simple event such as a created pull request to merge into some staging branch.

- *Continuous Deployment*: Enable the developer to focus on domain-logic and tests by automating all DevOps/IT operations triggered by a simple event such as a pull request merged into some staging branch.

Putting this together, our results should allow the typical small development team to focus their efforts on deployment-agnostic domain-logic with minimal efforts expanded on the infrastructure IT operations, and remove the need to split into two sub-teams developing each of the two deployments. This report will attempt to stay holistic and generic while describing enough details about the sub-components, such that the reader is provided with a good overview to start a similar project.

4 Architecture

4.1 Two axis of separation

Because such an application needs a Web Deployment ultimately, it makes a natural case for a Client-Server Architecture, which is one of the REST constraints. There is a separation of concerns between a presentation layer (front-end), the "*Company Client*", and data access layer (back-end), the "*Company API*". Both sides could be seen as entirely separate projects.

The server is tasked to keep the user data secure and partitioned between clients. The client is tasked to communicate with the server based on an agreed-upon protocol. They communicate via HTTP and as such the two projects only need to agree on a single HTTP API, then the client needs only the base URL for targeting the server.

This is the first axis of separation, the *Concern Axis*. The Deployment-Agnosticism paradigm also separates it orthogonally along a *Deployment Axis*:

- The Web Deployment will need to provide the two components:
  1. *Company Web API*, being the *Company API* hosted and accessible over the WWW.
  2. *Company Web Client*, the "official" API Consumer, delivered with a static host, which knows the URL of the Company Web API server.

  This allows "unofficial" clients and direct API access from authorized agents as well, which can find value in unforeseen, innovative consumer applications. A fee for API access would capture a fraction of this value creation, which would happen outside of the resource-constraints of the Company Client developer(s).
- The Desktop Deployment can then be construed as a special case, where the *Company Desktop API* would run on `localhost` and be shipped with the *Company Desktop Client*, running in Electron, bundled via the *Desktop Installer*.

We can visualize the two-dimensional axis of separations in a single figure.

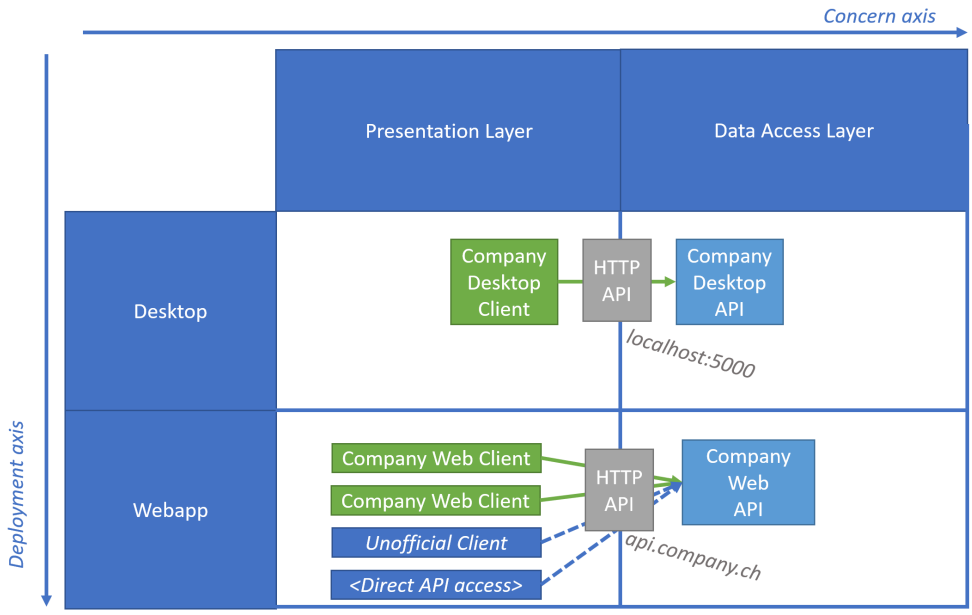

Figure 2: Two axis of separation

4.2 Sites of Deployments

We can formalize the *Site* notion as a partition of a computer network hosting a set of resources, perhaps a single computer[23]. In particular:

- *Consumer Site*: the site where the end-user can utilize the Software
- *Producer Site*: the site ready to service the Software to the consumer site.

To visualize the end product of this hybrid architecture, we define the Sites that would be involved, and the triggers of the Deployment events.

- *Development Site*: the computer of the developers involved in the conception of the codebase.
- *CI/CD Site*: the computer or network of, which host the codebase and provision the infrastructure via continuous integration/deployment.
- *Web Client Site*: the Producer Site hosting the Company Web Client
- *Web API Site*: the Producer Site hosting the Company Web API

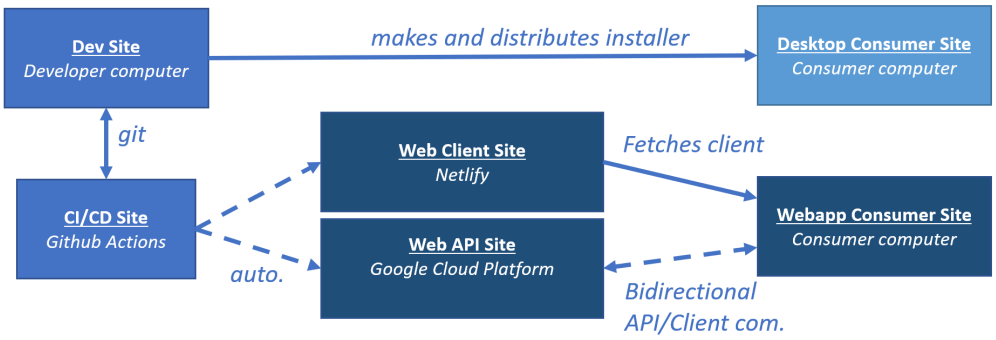

Figure 3: Sites architecture, prototype

- The solid lines are triggered on human action
- The dashed lines are triggered by the code. For the web app, the Provisioning Infrastructure from the CI/CD Site to the Web Client Site and Web API Site is described architecturally in § 4.7 and its implementation is discussed in § 5.6.

An ideal set of sites for production grade deployments, with staging areas to validate the results of continuous deployment in a two-step phase to production, is described in § 7.5. In our project however, we only prototype this architecture and keep it simple by bypassing the staging areas, and manually creating and delivering the installer in the `desktop` deployment.

4.3 Tiers and Configuration Environments

In addition to having two Deployment end-user results, we have *Deployment Tiers* which represent system Sites where our software will be executed on, for different purposes along the deployment stages. Classical tiers are:

- `local` or local development tier, on the developer sites, for manual testing
- `test` or testing tier, on the continuous integration sites, for running automated tests
- `stage` or staging tier, on the staging sites, for having an image of the deployed solution to validate before deploying to production

- `prod` or production tier, on the production sites, for hosting the services that the consumers are able to access

Since our prototype foregoes staging, we keep `local`, `test` and `prod`. Therefore the real variations for deploying our software are a Cartesian product between Deployments, and Deployment Tiers. We store parameters describing variations between deployments in *Configuration File*s.

Our configuration files take the naming scheme: `<deployment>.<tier>.yaml`.

- For the Desktop Deployment, tiers would be `desktop.local.yaml`, `desktop.test.yaml`, `desktop.prod.yaml`, etc.
- For the Web Deployment, tiers `webapp.local.yaml`, `webapp.test.yaml`, `webapp.prod.yaml`, etc.

The variations of configuration are two-fold:

1. The `env` for the Company API. It holds values such as databases target URI hosts, ports, etc. and other non-sensitive key-values.
2. Metadata specific to the Deployment for the Infrastructure, such as the Docker image name to pull for `webapp`, whether to use managed certificates for TLS (`prod`) or not (`local`), etc.

In software systems, usually there are sensitive configuration key-values that cannot be version controlled for security reasons, or "secrets". The tier configuration file thus would only hold the non-sensitive part of configuration key-values. Sensitive secrets have to be passed in a different pipeline, notably for the Web deployment. This process is discussed in detail in § 5.6.3.

4.4 Anatomy of a Deployment

4.4.1 Definitions

There are two components for a deployment, the Software System itself (which has variations based on the deployment), and the Infrastructure to place it to the consumer site. The starting point is the *Codebase*, the set of code files that can be built/interpreted into the artifacts of the Software System or to generate the Infrastructure to deliver it.

Infrastructure can be further qualified to illustrate an important difference:

1. *Delivering Infrastructure*, which would refer to:
   - for Desktop Deployment, the Desktop Installer
   - for Web Deployment, the Web Client Site and Web API Site serving the Company Web Client and the Company Web API respectively.

   This kind of Infrastructure is in the form of provisioned systems, not code.
2. *Provisioning Infrastructure* that will prepare the Delivering Infrastructure:
   - for Desktop Deployment, the make script(s) for the Desktop Installer.
   - for Web Deployment, the workflows executed at the CI/CD Site.

   This kind of Infrastructure is *Infrastructure as Code*, an approach to Infrastructure that uses repeatable routines for provisioning and changing infrastructure systems and their configuration [24]. So it is infrastructure itself, and their execution provisions the Delivering Infrastructure. They are dependent on the Deployment D because the two infrastructures are different files for the Desktop Deployment and Web Deployment.

$$Infrastructure_D = Infrastructure_{delivering,D} + Code_{provisioning,D}$$

As a result, the codebase $Code$ is split:

1. The Software System $Code_{software}$, Deployment Axis-agnostic but Concern Axis-specific:
   - The Company Client codebase, capable of mutating into either the Company Web Client or the Company Desktop Client.
   - The Company API codebase, capable of mutating into either the Company Web API or the Company Desktop API.

   The code exhibits Deployment-Agnosticism, by being a function of a Configuration File $conf$, which provides deployment-specific data:

   $$Code_{software}(conf) = Code_{API}(conf) + Code_{client}$$

   It is shown why $Code_{client}$ is not a function of the configuration file, in § 4.4.2, and why and how $Code_{API}$ is, in § 4.4.3.

2. The Provisioning Infrastructure $Code_{provisioning,D}$, Deployment Axis-specific.

$$Code_{provisioning,desktop} = makeinstaller$$
$$Code_{provisioning,webapp} = Code_{CI} + Code_{CD}$$

Here the take-away is that the two codebase parts are completely independent, based on their specificity regarding the deployment axis:

$$Code_{software} \perp\!\!\!\perp Code_{provisioning,D}$$

As such when we implement Deployment-Agnosticism, we are really talking about the $Code_{software}$

4.4.2 Company Client Deployment-Agnosticism

The Company Client is implemented in such a way that *Deployment-Agnosticism* is achieved through:

1. A deployment-specific `host[:port]` $host_{target}$, `localhost:5000` for the Desktop Deployment, and `api.company.ch` for the Web Deployment.
2. A deployment-agnostic API Specification $Spec_{API}$ describing the resources

Therefore URLs will be in the form of: `http[s]://<host_target>/<resources>`.
The Company Client knows its deployment and thus $host_{target}$ at run-time through a method discussed in § 5.1, without relying on $conf$. It knows $Spec_{API}$ implicitly, as it is constant.
Here the take-away is that the client code does not depend on the configuration file:

$$CompanyClient = Code_{client}(Spec_{API}, host_{target}) \perp\!\!\!\perp conf$$

4.4.3 Company API Deployment-Agnosticism

The Company API is implemented in such a way that Deployment-Agnosticism is achieved through the codebase being a function of the environment variables only. The Company API server code can take an optional flag `--env=<config file>` setting the environment variables before execution of the rest of the code. This is a flexible design:

- For the Desktop Deployment, the Provisioning Infrastructure (installer make script(s)) passes the Configuration File and thus the `env` by bundling it, as an embarked file. When starting the Company Desktop Client, the Electron bootstrapping code forks a child process running the Company Desktop API binaries. It includes the flag `--env=desktop.prod.yaml` to provide the configuration file, the server reading it and parsing it into its own environment variables.

- For the Web Deployment, the Provisioning Infrastructure (CD workflows) passes the Configuration File to the Delivering Infrastructure (GCP/GKE cluster) using helm charts, the implementation of which is discussed in § 5.6.3. The end result here is that our Container Image's environment variables are set to `webapp.prod.yaml`

This can be summarized as:

$$CompanyAPI = Code_{API}(env) = Code_{API}(conf)$$

4.5 API Specification

4.5.1 REST-like APIs

The constraints of a REST architecture are enumerated in § 2.1.5. Our API specification can be viewed through this prism, Table 1 discusses if and why they are followed in our API specification.

Pure REST constraints are partially covered. This can be referred to as *REST-like*. Were all constraints followed, the robustness of the application consumer would be greater - for a tab-based application for instance, tabs suffering service interruption could just not be included in the list of available follow-up actions after fetching the entrypoint `api.company.ch`. Then, the adapted client, would only show tabs where the follow-up action is available.

Having a prior knowledge of the static API specification allows one to consume the API without having to enter it through `api.company.ch` first and then branching to the desired resource. This can be useful for external programmers who want to make a quick call to a few API resources (from a small script or Jupyter notebook, for instance). It also helps the client development, since the client developer can make the static GUI first and then link everything with matching API endpoints. It does imply that the API provider has to keep old endpoints alive and provide ample notice of depreciation, to avoid breaking the consumers dependent systems.

REST Constraints		
Name	Valid	Discussion
(1) Client-Server Architecture	Yes	As discussed in § 4.1, our Deployment-agnosticism leverages the client-server architecture.
(2) Statelessness	Yes	We differentiate between application state and resource state. Previous HTTP requests to CRUD resources on the server change the resource state (persistent after the application stops). Application state is temporary. As discussed in § 2.5, server-side sessions break this constraint. We use JWTs to keep it, instead. Authentication and Authorization will be checked from the JWT included in each HTTP request to ensure the User has the rights to read or write on the Resource it requests. Therefore routes can return `401 Unauthorized` or `403 Forbidden` on top of their normal behavior.
(3) Cacheability	Yes	The server codebase is split in Blueprints, as described in § 5.3. Blueprints expose views to communicate with each other, and the API consumer. When needed, cache mechanisms can be added per-Blueprint to answer directly when we know the resource requests will yield the same result as a prior one. We invalidate the relevant cached data when resource requests change the resource state.
(4a) Identification of Resources	Yes	Our resources are exposed through verbose URLs which represent the hierarchical structure and contain the resource IDs. The rest of this section discusses this identification scheme.
(4b) Manipulation of Resources through Representations	Mixed	Most of our internal resources are exposed as JSON representations, which is explicit by the `Content-Type` of the HTTP response. It also contains data such as a Model's `id` to re-create the URL of the Resource upon which the explicit HTTP verbs are applied. However, we could go so far as to include the available verbs and signature of API actions upon a returned resource inside its representation, but this is left implicit in our implementation.
(4c) Self-descriptive Messages	Yes	By default the HTTP Verb + endpoint + `Content-Type` is enough information for describing the entire request. For some resources like `<prefix>/importer`, the posted file is joined with metadata to describe how to process it (this should be done for all endpoints where the request is not self-explicit).
(4d) HATEOAS	No	HATEOAS implies making the entire API discoverable from `api.company.ch`. The client UI would have **no prior knowledge** of the API, and construct the GUI based on API's available actions. Many public APIs forego this constraint for convenience [6].
(5) Layered System	Yes	In both the Desktop deployment and Web deployment cases, the API entrypoint and specification stays the same. However, in the Desktop case the monolithic server answers directly, in the other, the system is within a cluster, potentially a micro-services architecture as described in § 6 and requests pass through an ingress to be routed to different services. In both cases the representations are the same.
(6) Code-On-Demand	No	This constraint is optional, we don't have a use-case for code-on-demand in our specific implementation.

Table 1: REST Constraints validity and discussion for our API

We introduce a classification on the typical Resources used for a REST-like API.

4.5.2 Models

Model and Non-Relational Model Resources are made accessible on a collection-basis via the URL `<prefix>/.../<models>`, accepting:

- `GET`: retrieves the Model Representation JSONs of the collection of models inside an array. Returns `200 OK`.
- `POST`: appends a new Model with an auto-generated ID to the collection. The requests carries the Model Representation JSON payload to describe it. Returns `201 Created` or `422 Unprocessable Entity`.

Model and Non-Relational Model Resources possess a unique ID which is public. They are also interactable on a lone-basis at URL `<prefix>/.../<models>/<id>`, accepting:

- `GET`: retrieves the Model Representation JSON of the singular model. Returns `200 OK` or `404 Not Found`.
- `PUT`: places a new Model at this ID. The requests carries the Model Representation JSON payload to describe it. Returns `201 Created`, `409 Conflict`, or `422 Unprocessable Entity`.
- `PATCH`: edits an existing Model at this ID. Returns `200 OK`, `422 Unprocessable Entity` or `404 Not Found`.
- `DELETE`: deletes an existing Model at this ID. Returns `204 No Content` or `404 Not Found`.

4.5.3 Actions

*Action*s are functions changing the state of the application without returning data. An action example would be the import of a file describing multiple models, parsing it and appending resulting models to the database.

An action is not directly identified by its URL. Instead, a Resource object implements the action and the client requests it by a body argument. For example, a `POST` at URL `<prefix>/importer` with payload:

```
{ "action":  "import", "format":  "csv", "file":  0xE7B9...  }
```

4.5.4 Transformations

*Transformation*s are functions that do not change the state of the application, but instead only retrieve a representation of a transformed subset of the data, for example aggregations of model collections. They accept `GET` only and return a JSON describing the result. Typical aggregations can benefit from being performed client-side, to spare computation server-side. Transformations are thus fit more for computations we want to also expose to agents accessing the API directly, or leveraging specific server-side information, such as third-party data.

For example, a `GET` at URL `<prefix>/.../financings/cashflows` performs the computation of cashflows based on the fields of the financings in some portfolio.

Details about the implementation of view functions of our resources are discussed in § 6.1.

4.6 Testing Layers

In the context of testing automation, it is usual to split the testing phases in 3 layers:

1. *Unit Tests*, which is the smallest form of testing, to assess the correct behavior of a separable unit of software such as classes or functions.
2. *Integration Tests*, which are testing the interaction among components. Typically for a Web API, those assess the correct response to requests that involve different part of the server.
3. *UI Tests*, which are defined at the UI level to assess the correct resulting UI values and placement

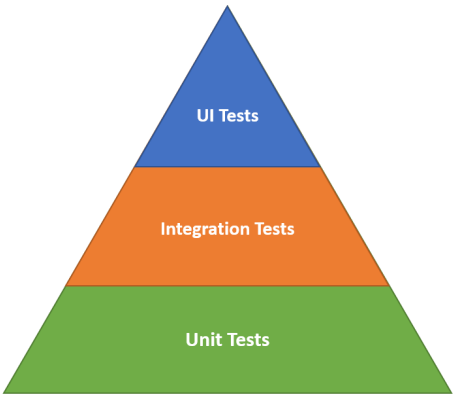

Figure 4: The Pyramidal Testing Layers

UI Tests have been described as *Brittle* (small changes in UI can break many tests) and *Time consuming* (extensive elapsed testing time) [25]. We found it convenient to assess the correctness of the UI manually, and implement testing automation only at the Unit Tests and Integration Tests layers, described in § 5.6.1.

4.7 CI/CD Pipeline

Our CI/CD pipeline has the following properties:

- For Continuous Integration, the execution of all tests during a pull request to the remote server merging into our stable branch. The test implementation is discussed in § 5.6.1 and the CI workflow is discussed in § 5.6.2.
- For Continuous Deployment, when a pull request is merged to our stable branch, it triggers a deploy to the cluster. This is discussed in § 5.6.3. This can be extended to trigger a build of the Desktop Installer.
- Secrets are managed with a RBAC protocol formalized in § 4.7.1 and implemented in § 5.7. They are used to securely deploy to the Google Kubernetes Engine during Continuous Deployment and provision the cluster resources.

To visualize this process across our Sites, we can represent it like this:

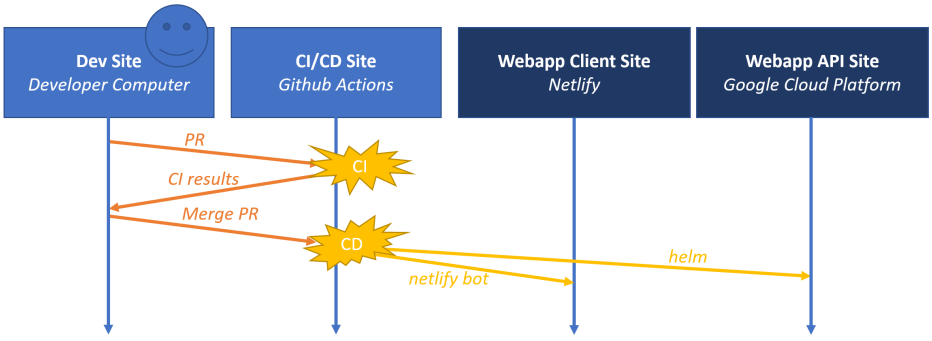

Figure 5: The CI/CD Process

The implementation of both Github Actions workflows is described in § 5.6.

4.7.1 Secret Management through Role-Based Access Control

The following terms are defined:

- *RBAC*: *Role-Based Access Control*, an approach to restricting system access based on a person's role within an organization
- *Secret*: A piece of sensitive information

The Web Deployment orchestrates two databases, and they both need to be provisioned with master credentials. This is a typical case of a Secret. The sensitive nature of those Secret makes it important to restrict their access via RBAC.

A RBAC can be formalized through the resources [26]:

- $Users$, describing people in our company.
- *Roles*, associated many-to-many with $Permissions$ and $Users$.
- $Permissions$ or *Operations*, describing abilities that we want to restrict.

We define the $Permissions$:

1. $WriteSecret$, ability to write Secrets.
2. *ReadSecret*, ability to write Secrets.
3. *AssignRole*, ability to map some user to some permission with a *Role*.

We define the *Roles*:

1. $ProjectOwner$: reserved to the minimum trusted persons in the organization, having all $Permissions$.
2. *ClusterAdministrator*: Having full access to the Kubernetes Cluster, they have *ReadSecret*, by their ability to get a shell into a container for example.
3. *Developer*: They have none of the $Permissions$.

The implementation of this policy is discussed in § 5.7, after laying out the context for Secrets in the CI/CD Pipeline implementation details.

5 Implementation

5.1 Hybrid Client

The Company Client is a JavaScript user interface made with Vue.js, with a *Single Page Application* design in mind. Its internal state is changed based on user inputs, which can generate HTTP Requests to the Company API, and change based on those HTTP Responses as well. The *axios* HTTP client library is used for this. We leave the presentation-layer implementation details outside the scope of this project.

- In the Desktop Deployment, the whole client is shipped in an Electron wrapper.
- In the Web Deployment, the client is built into *Static Files*, the HTML/CSS/JS files that can be rendered by a modern browser, and hosted with *Netlify* for anyone to visit and fetch them.

We implement our Company Client such that it only cares about the API Specification and some `target_url`, which is the base host for all API calls.

To detect whether the software is in a Desktop Deployment or Webapp Deployment, a simple predicate detects if it's running in Electron: `let deployment = "process" in window ? "Desktop" : "Webapp"`.

Based on `deployment` we can then select which URL base/prefix to target, to connect the front-end to the needed back-end:

- In the Desktop Deployment, `"http://localhost:5000/"` to target the locally running API server
- In the Web Deployment, `"https://api.company.ch/"` to target ultimately the very similar API server hosted on the Google Cloud Platform, running in a Docker container, on a Kubernetes Pod, within the GKE Kubernetes cluster (topic detailed in § 5.3).

Using this predicate we can also perform some other adjustments like enabling/disabling Desktop/Webapp specific functionalities, but all the core logic is hybrid by virtue of following the same API Specification.

5.2 Hybrid Data Access Layer

5.2.1 Database Systems

Our choices were:

- *PostgreSQL*: for Relational Model storage in the Web Deployment. There are many database engine options. Planning to host it within a Docker container within our cluster, it was decided to use one with a top-popularity official Docker image, as well as a free and open-source option. This is leaving two main options: *MySQL* or *PostgreSQL*. Ultimately, PostgreSQL was chosen based on online developer reviews (stronger community, reliability, flexibility, between others), more straight-forward open-source licensing, and the adoption rate growth against MySQL which seems to validate the previous points.
- *SQLite*: for Relational Model storage in the Desktop Deployment. Here we want an SQLAlchemy-compatible RDBMS to work on local files, and there is zero concurrency in this context. In the end, SQLite provides this, with zero separate installation, no maintenance, no process forking needs, no overhead, and it also provides the very useful `sqlite:///` path to cache data in memory with SQLAlchemy (for testing, notably).
- *MongoDB*: for Non-Relational User storage in the Web Deployment. For our type of non-relational database need (key-value store), with our wish for a popular, stable, official Docker image, there is no real competition here.
- *JSON* files: for Non-Relational User storage in the Desktop Deployment. Due to our key-value store size, without the need of concurrent access, and with Python's `json` standard library package, this is a straight-forward choice as well.

Those choices can be visualized along a Relational/Non-Relational database management model axis:

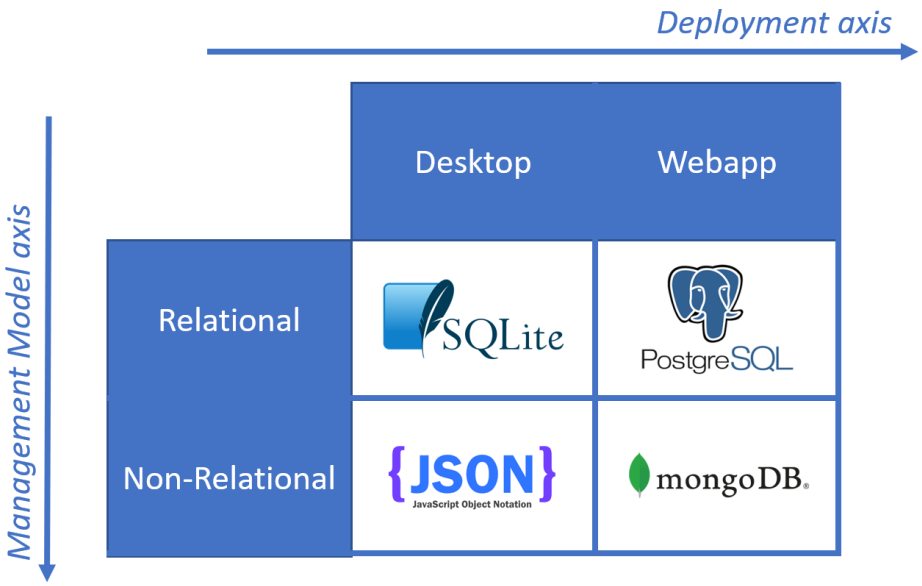

Figure 6: Database System Choices

A valid question is then, *are JSON files suitable for a database?* The Desktop Deployment has a single User entry, as described in § 5.4. Therefore the question could be more relevant as: *Is a database system needed to store Desktop User data? TinyDB*, a JSON-based database Python package was tried, but ultimately discarded for the unnecessary overhead. Ultimately, the Desktop Deployment "database" can be viewed as a simple User preferences file and we therefore answer this need by using a JSON file and the Python standard library `json` package for serialization.

5.2.2 Relational Data Implementation

Using the solution stack overviewed in § 2, we implement the data concepts defined in § 2.3, in the following sequential process:

1. Domain-requirements are discussed, to agree on Model Definitions and Enumerated Types.
2. *Enumerated Type*s are implemented: The Enumerated Types are implemented as `enum.Enum` Python enumeration child classes, introduced in PEP-435 as *a set of symbolic names bound to unique, constant values*, similar to the same concept in many other programming languages [27]. They are provided with enumeration members, with names mapped to values. Here we choose to map them to `dict` values containing all metadata about the enumeration member. For example `Currency` gets enumeration members like `Currency.CHF = { "id": "CHF", "name": "Swiss Franc", "symbol": "Fr." }` . This is a flexible design, as we can have a lot of different metadatas for different Enumerated Types. A decorator is written to be applied on Enumerated Types, to provide them with a `db.Column` factory for Model Definitions using them. For example, the decorator provides `Currency` with the `Currency.column(...)` factory function, returning a `db.Column` instance of type `db.String` constrained to the set of `id`s of currencies, for Model Definitions that contain a currency field.
3. Model Definitions are implemented, as child classes of the Flask-SQLAlchemy instance's `db.Model`. It contains:
   - Native type fields, for example `notional = db.Column(db.Integer)`
   - Categorical fields, for example `currency = enums.Currency.column(db)`

- Relationships to other Model Definitions, for example `financings = db.relationship("Financing", cascade="all,delete", backref="portfolio")` in the `Portfolio` Model Definition. This creates a `portfolio` field in the `Financing` model definition, referencing the parent portfolio. Also, we request a cascading delete behavior. SQLAlchemy handles those mechanisms implicitly.

Model Definitions are defined in a `models.py` class.

4. From this point, Models can be instantiated from the Model Definition classes, and thus serialized to the underlying database.

5. Marshmallow Schema Definitions are declared, using Marshmallow-SQLAlchemy's `SQLAlchemyAutoSchema` parent class. It gets an internal `class Meta` member with a `Meta.model` member that references the Model Definition to auto-generate the schema from. For example `FinancingSchema` gets `class Meta: model = models.Financing`. We also set `load_instance = True` to make the output of `FinancingSchema().load(...)` generate a `Financing` instance.

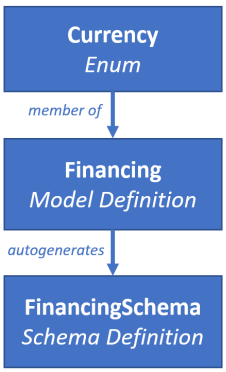

Figure 7: Data Implementation Hierarchy

6. From this point, a `Schema` instance can be created from Marshmallow Schema Definitions. They provide `load` and `dump` functions to resp. instantiate Models from a valid Model Representation `dict`s, and to parse Models into valid Model Representation `dict`s.

7. This is used in the Model View Functions to parse requests with JSON payloads and make responses with JSON representations.

Let's illustrate the `dump`ing of a SQLAlchemy Model instance by a Marshmallow Schema instance to generate the Model Representation during a `GET` request for a clearer picture:

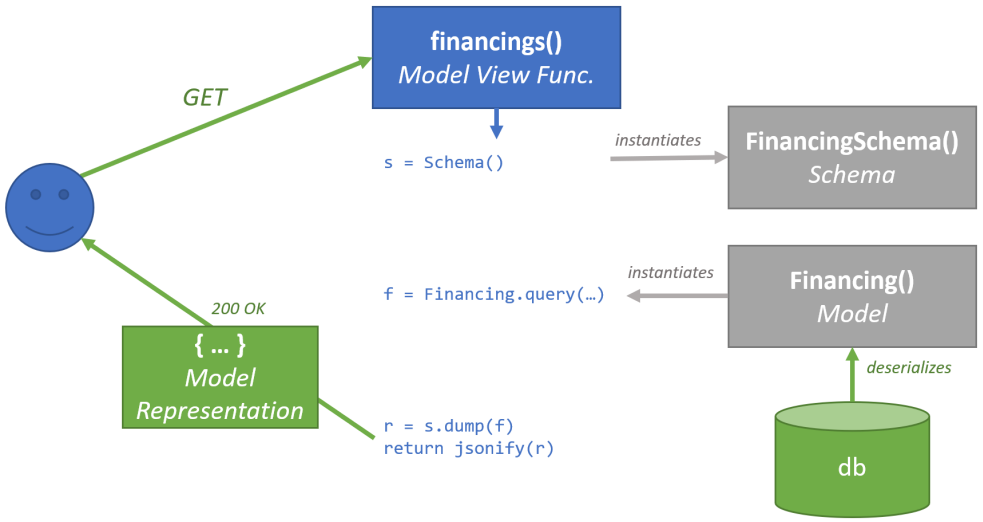

Figure 8: Usage during GET request

Its counterpart of `loading` a Model Representation with a Marshmallow Schema instance to generate the SQLAlchemy Model instance during a `POST` request is similar, but "reversed":

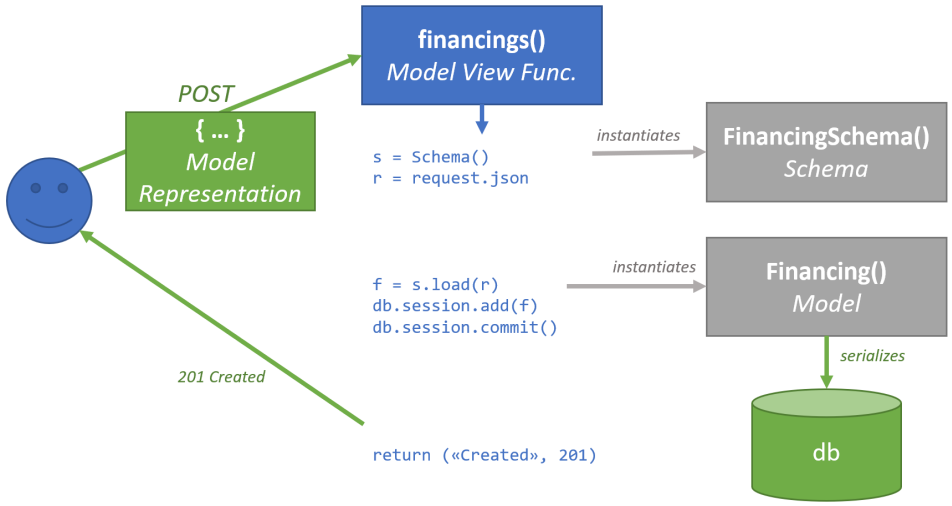

Figure 9: Usage during POST request

As SQLAlchemy binds the model instances with database transactions automatically, a `PATCH <financings>/<id> {"field1":"newvalue"}` operation is implemented by loading the Model matched with `<id>` from the database, setting `setattr(model, "field1", "newvalue")`, and calling `db.session.commit()`. The `PUT` operation follows the same logic as the `POST` operation, and the `DELETE` operation uses a built-in SQLAlchemy function to delete the model in a similar straight-forward fashion.

A minimal demonstration of some of those concepts is provided with the code snippet in § 7.2. Additionally, more implementation details about related view functions are discussed in § 6.

5.2.3 Non-Relational Data Implementation

User data includes username, e-mail, hashed and salted passwords, user preferences, etc. We would like to be able to store arbitrary sized arrays and matrices in user preferences (for domain logic reasons). This can be implemented in Relational or Non-Relational paradigms. A JSON-like key-value mapping gives us flexibility out-of-the-box, notably with user preferences as a nested JSON-like key-value mapping. But loosened constraints compared to the Relational paradigm have the trade-off of potential uncaught errors down the line.

We take this trade-off, however this implies a different implementation of Deployment-Agnosticism, the Non-Relational paradigm preventing us from using SQLAlchemy. This implementation will rely on JSON-like data:

- For Desktop Deployments, JSON-like data can be stored in JSON files in the local filesystem.
- For Web Deployments, JSON-like data can be stored in a MongoDB database. The *Flask-PyMongo* package provides a Flask Extension wrapping the `pymongo` package for interacting with a remote mongo database.

Those two distincts behaviors should be wrapped under an agnostic interface with the User store, providing a generic type wrapping both the `json` standard library and the Flask-PyMongo package, provisioning either of them depeneding on the deployment, and providing a limited subset of their functionalities with a single interface.

We implement this using a *Bridge* pattern, defined as a *Decoupling of interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time* [28].

Two main bridge methods are defined in the *GenericUserStore*, through which additional convenience methods can be created:

1. `GenericUserStore.set(user, key, value)`
2. `GenericUserStore.get(user, key) -> value`

The `GenericUserStore` gets a `mode` flag taking values in `("mongo", "json", "mem")`.

- In the `"mongo"` mode, the bridge store serializes/deserializes the key-values with Flask-PyMongo using the MongoDB URL stored in `app.config["MONGO_URI"]`.
- In the `"mem"` mode, the bridge store uses a plain Python `dict` to store those key-values. No data permanence occurs.
- In the `"json"` mode, the bridge store behaves exactly like the `"mem"` mode but wraps `set(user, key, value)` with a `json.dump` to a `users.json` file in `AppData`. Conversely, on start-up, `get(user, key)` loads the `users.json` into the memory `dict`.

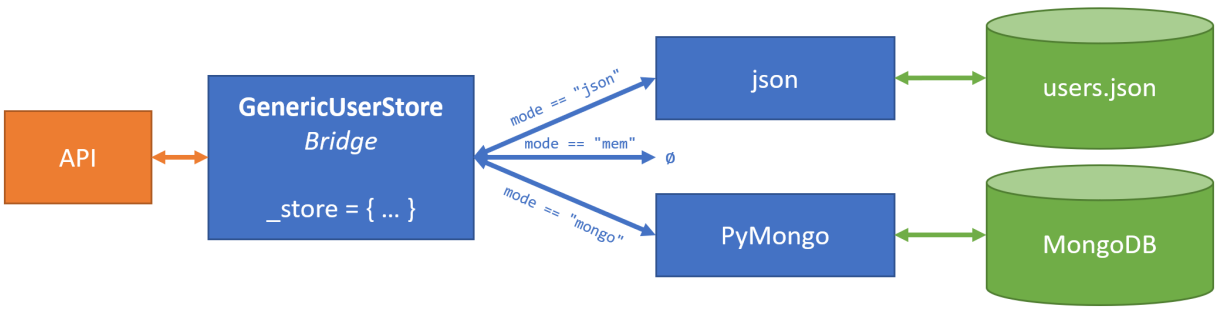

Figure 10: User Store Deployment-Agnosticism Implementation

Therefore, the Desktop Deployment will be starting the bridge store in `"json"` mode, and the Web Deployment in `"mongo"` mode. The `"mem"` mode is used in unit and integration testing.

5.3 Hybrid API Server

Now that we have implementations of a hybrid version of both Relational and Non-Relational databases, the next step is to consider a Flask server implementing all the Resources view functions that we need. It only interfaces with data permanence through SQLAlchemy and GenericUserStore, and thus can be written with Deployment-Agnosticism in mind.

The actual view functions are implemented as parts of Flask Blueprints, grouping related functionalities for our API that share common logic/helper functions, packages, data access, potential caches. A Blueprint can be declared and its view functions linked with endpoints and implemented with:

```
blueprint_project_aggregations = Blueprint("project_aggregations", __name__)

@blueprint_cashflows.route("<username>/<project_name>/aggregation_one", methods=["GET"])
def view_aggregation_one(username: str, project_name: str) -> dict:
    if request.method == "GET":
        ...
```

Details about how to implement view functions code is shown in the examples of § 7.2 and § 7.3, and cross-blueprint communication as well as more context regarding Blueprints is discussed in § 6.

Then we compose our Flask Application from blueprints in our main server file:

```
def register_blueprints(app: Flask):
    prefix = "/" + os.environ.get("url_prefix")
    app.register_blueprint(blueprint_project_aggregations, url_prefix=prefix)
    app.register_blueprint(blueprint_project_import_actions, url_prefix=prefix)
    ...

def create_app(config_yaml_filepath: str | None) -> Flask:
    app = Flask(__name__)
    app.url_map.strict_slashes = False
    if config_yaml_filepath is not None:
        set_os_env_from_yaml(config_yaml_filepath)
    insert_os_env_in_config(app)
    initialize_extensions(app)
    register_blueprints(app)
    return app

if __name__=="__main__":
    config_yaml_filepath = ... # parse from cmd line args
    app = create_app(config_yaml_filepath)
    app.run()
```

For the Desktop deployment this file is compiled an ran as a child process of the client. To have a hybrid API, in the context of the Web Deployment, it needs to be served by some cloud provider. We chose the Google Cloud Platform for this task. It was chosen for mainly two reasons:

1. Its Switzerland physical server location choice, which is not available with the largest Cloud provider Amazon Web Services. The Switzerland server location is a common requirement for sensitive data of Swiss customers.

2. The seamless integration between Google Kubernetes Engine and Kubernetes, which is originally a Google project. We wanted to experiment with both monolithic and micro-services architectures as is discussed in § 6, the convenient option was to host the back-end within a container orchestration platform in a cloud cluster. The choice of Kubernetes+GKE was the natural progression.

In this subsection we discuss the monolithic architecture only, meaning a single execution unit within the cluster, excluding database systems. The Web Deployment uses Docker to containerize the Flask server, as described in § 2.4.2. Then, the image is pushed to the Google Container Repository and tagged with a versioned ID. Then, our Helm Chart contains:

- A Kubernetes ConfigMap which parses the `env` value from the Configuration File using Helm templating, to be passed to the API Docker image.
- Kubernetes Services:
  1. An Ingress as the main gateway to our cluster, which is only used in this monolithic architecture to define annotations detected by the Google Cloud Platform, `ingress.global-static-ip-name` and `managed-certificates` which are used to respectively use our static IP, and enable TLS via the Google certificate authority. Those resources are outside the scope of this prototype, they are managed via the `gcloud` console.
  2. A NodePort interfacing the Ingress with our API Deployment monolith Pods
  3. Two ClusterIPs for the databases Pods.
  4. Two LoadBalancers interfacing with the two ClusterIPs to provide back-doors to our databases to monitor them, this is more suited for staging than production, for our prototype.
- Kubernetes Deployments:
  1. The monolithic API Deployment, pulling the image from the Google Container Repository. It gets environment variables from the Kubernetes ConfigMap, which contains the host names that the Kubernetes DNS will resolve to our two databases ClusterIPs
  2. A MongoDB Deployment pulling the official MongoDB image. It picks its relevant config values directly from the Configuration File or Secrets using Helm templating.
  3. A PostgreSQL Deployment pulling the official PostgreSQL image. It picks its relevant config values directly from the Configuration File or Secrets using Helm templating.
- *PersistentVolumeClaims* for both databases to store the data. The Google Kubernetes Engine, not seeing explicit *PersistentVolumes* matching the claims, provisions them for us dynamically.

Putting this together the result of this Helm Chart at the Web API Site will result in such a cluster:

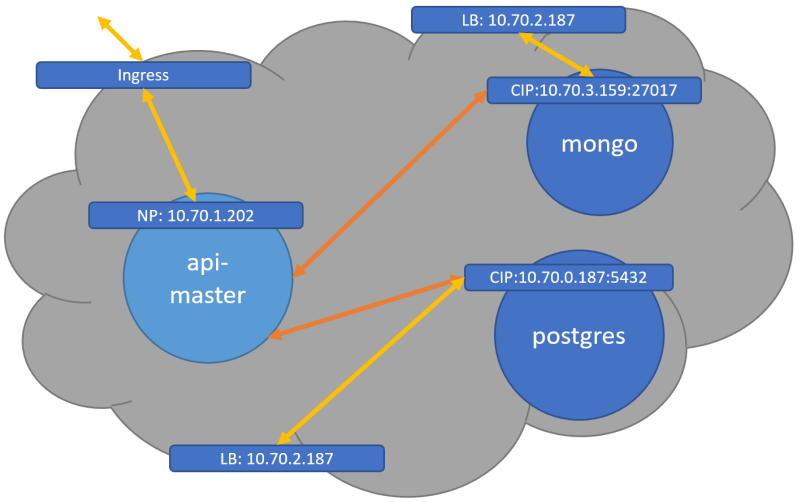

Figure 11: Cluster Result

The API monolith targets the databases by using the ClusterIPs host names which get resolved into the IPs by the internal DNS, reading them from environment variables, set by the Kubernetes ConfigMap, parsed by Helm from a `webapp.prod.yaml`'s `env` entry. It works by simply setting the `SQLALCHEMY_DATABASE_URI` and `MONGO_URI` in the `app.config`, as we use the Flask extensions described in § 2.3.

Another Configuration File is defined, `desktop.prod.yaml`, for the Desktop Deployment. Here we don't have to define the host for MongoDB because our GenericUserStore will target a JSON file instead, as described in § 5.2.3. We can set an initial host name of the Relational database, as `sqlite:///`, but this path will also change when creating/loading SQLite db files from the client, changing `SQLALCHEMY_DATABASE_URI`. This is explained in § 5.5. Because both our database interaction layers have Deployment-Agnosticism, the rest of the API monolith code doesn't care, the Desktop Deployment just launches the compiled server here called "api-master", without ever using Docker/Kubernetes/Helm, which results in:

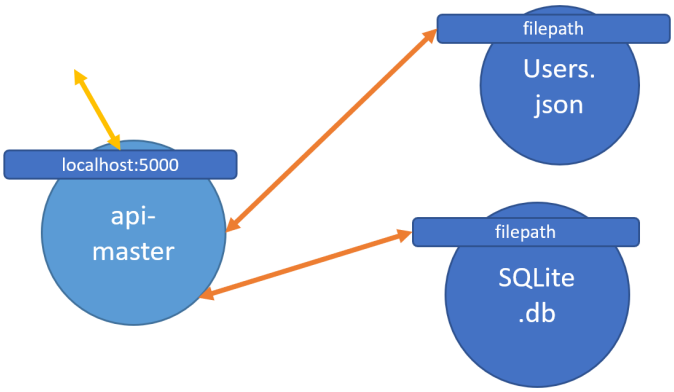

Figure 12: Desktop Result

5.4 Hybrid User System

The User roles are:

1. Support the Authentication/Authorization system for the API.
2. Store *User Preferences*, an arbitrary JSON object for each user.

Using the data implementation described in § 5.2.3, we now describe an implementation of a User system that works for both deployments.

The username is stored in each SQLAlchemy Model the User "owns", or of some parent. The Model View Functions check the JWT of requesting agents to ensure rights of access. For operations on whole collections of Models such as collection-wide Actions or aggregation Transformations View Functions, we are careful to always partition the data, at least by user.

The User entries keep `username` and `password` fields. We then implement the two fundamental view functions for Flask-JWT-Extended (see § 2.5.4), with URLs:

1. `<prefix>/users` upon which `POST` of a valid user registration form creates a new User. The `password` is transmitted in plaintext over HTTPS, the API uses Hashing and Salting and saves the result in the GenericUserStore.
2. `<prefix>/login` upon which `POST` of a valid user login form requests the matching hashed password from the GenericUserStore, and performs a secure password string check via the `bcrypt` package. Success logs the user in by providing him a JWT to store client-side, and include in subsequent requests. Failure returns `401 Unauthorized`.

This is the server-side hybrid system, and the Desktop Deployment client registers a dummy User with `{ "username": "current", "password": "" }` on start-up, or logs in if it exists already, and disables the

registration/login forms inputs that the Web Deployment exposes. With this trick, the Company Desktop Client and Company Web Client can use the exact same logic afterward, thus providing us with Deployment-Agnosticism.

All user-specific Model View Functions URLs get organized by using a `username` prefix, `<prefix>/<username>/<models>`, for instance `api.company.ch/louislandelle/financings` or `localhost:5000/current/financings`.

A minimal demonstration of some of those concepts is provided with the code snippet in § 7.3.

5.5 Hybrid Project System Implementation

Software-as-a-service may need a data "First-layer Partition" concept for each User, perhaps for a *Project* concept like in our case, holding all other SQLAlchemy Models defined in multiple many-to-one relationships. Or, under another name, but with the same underlying idea.

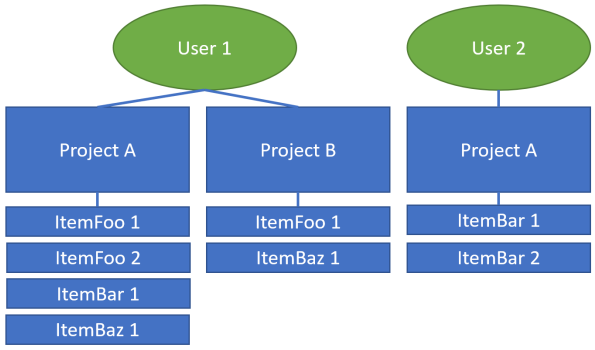

Figure 13: First-layer Partitioning

We decided to have two different modes of storing those Projects to match our needs:

- The Web Deployment implements all Projects for all Users on the same database.
- The Desktop Deployment represents Projects as separate local files which can then be shared, encrypted, etc.

This is implemented with a `Project` SQLAlchemy Model. It gets a `username` key to know to which user it belongs, as a user-specific Model. Then, all project-specific SQLAlchemy Models get a `db.relationship` with the Project SQLAlchemy Model, providing the basis for project-partitioning, cascading deletes, and a back-reference to the `Project`.

All project-specific Model View Functions get routed from:
`<prefix>/<username>/<project_id>/<model>`

This way, the view functions know directly from the URL which `Project` to target for project-specific Models, to implement project data partitioning.

In the Web Deployment, this all occurs on the single PostgreSQL database, for all project, for all users. However in the Desktop Deployment we want to have a separate SQLite `.db` file for each project. We have a specific route for the Desktop Deployment creating and reading such files. We augment the file creation by also creating a singleton `Project` in it, with the name of the file as project name. We also disable the generic Project create-read-update-delete of the Web Deployment from the Desktop Deployment. This provides a "Project = File" concept to the Company Desktop Client user.

5.6 CI/CD With Github Actions

5.6.1 Testing Implementation

To disable data permanence while testing, one can use the `sqlite:///` URL for Flask-SQLAlchemy and `"mem"` mode for our GenericUserStore. This way, both data access layers are emulated in memory.

This can be defined in a `desktop.test.yaml` Configuration File, for example, for local testing.

We use the Python testing package `pytest` and its *Fixtures* feature to help in testing. They are methods that `yield` some object which we can use in other tests, where the `scope` parameter of the decorator specifies for how long the object is kept in memory.

The most important fixture is the `client` fixture, which we can define in the special file `conftest.py`. Defined in this file, fixtures are made available in all other testing files. This client fixture provides the built-in `test_client` Flask test client which interfaces well with `pytest`. It is made from the result of the same `create_app` function that produces the production Flask server. This function is outlined in § 5.3.

```
@pytest.fixture(scope="session")
def client(env):
    app = server.create_app(env)
    yield app.test_client()
```

Another convenient fixture is the `auth` fixture yielding the JWT needed for protected endpoints. It uses the same principle as the client script example in § 7.3. The point of such a fixture is to register then login a temporary "tester" user to test the correct rejection of users without Authorization, partitioning between users, and the results of protected resources themselves.

```
@pytest.fixture(scope="session")
def auth(client):
    creds = {"username" : "tester", "password" : ""}
    ans = client.post("users", json=creds)
    assert ans.status_code == 201 and "access_token" in ans.json()
    yield (_auth := {"headers": {"Authorization" : f"Bearer {ans.json()["access_token"]}"}})
    ans = client.delete("users/tester", **_auth)
    assert ans.status_code == 204
```

Those two fixtures give us a mechanisms to perform integration tests conveniently, by calling functions replicating requests using standard HTTP Verb syntax:

```
def test_post_get_resource(client, auth):
    ans = client.post("tester/project1/resources", json={"field1": 1e6, ...}, **auth)
    assert ans.status_code == 201
    ans = client.get("tester/project1/resources", **auth)
    assert ans.status_code == 200 and ans.json()[0]["field1"] == 1e6
    ...
```

5.6.2 Continuous Integration with Github Actions

We use Github Actions to automate testing. Continuing after the generic steps described in § 2.4.7, we can define jobs to run `pytest` tests, aggregated in an `api/test` directory, and to use our testing Configuration File, via:

```
- name: Install requirements
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt
- name: Integration tests
  run: |
    python -m pytest -svv api/test --env=api/conf/desktop.test.yaml
```

This installs the required Python environment and then will generate a comprehensive output on Github with the test results. We set its trigger with pull requests or commits on the `development` branch, upon which the developer can decide to accept or reject the pull request, or hotfix a `development` commit.

This methodology is straightforward to adapt to perform the same kind of automated tests for the client codebase as well, using node unit tests.

5.6.3 Continuous Deployment with Github Actions

Using a modern git-based hosting solution such as Netlify is straight-forward in order to serve Static Files from a Github repository to the Web Client Site, as described in § 5.1. The user just needs to specify to the service the branch, triggers, and the location with our Static Files and have those files built, in our case via `npm run build`, within the CD Workflow.

Deploying on the Web API Site requires more overhead. Fortunately, using our stack, this can be done easily as well. Before writting this workflow, we generate a GKE Service Account Key representation, a JSON document storing a base64 encoded private key (and metadata) matching a generated key pair where the public key is stored on the GCP, to authenticate the account, and store it as a Github Secret.

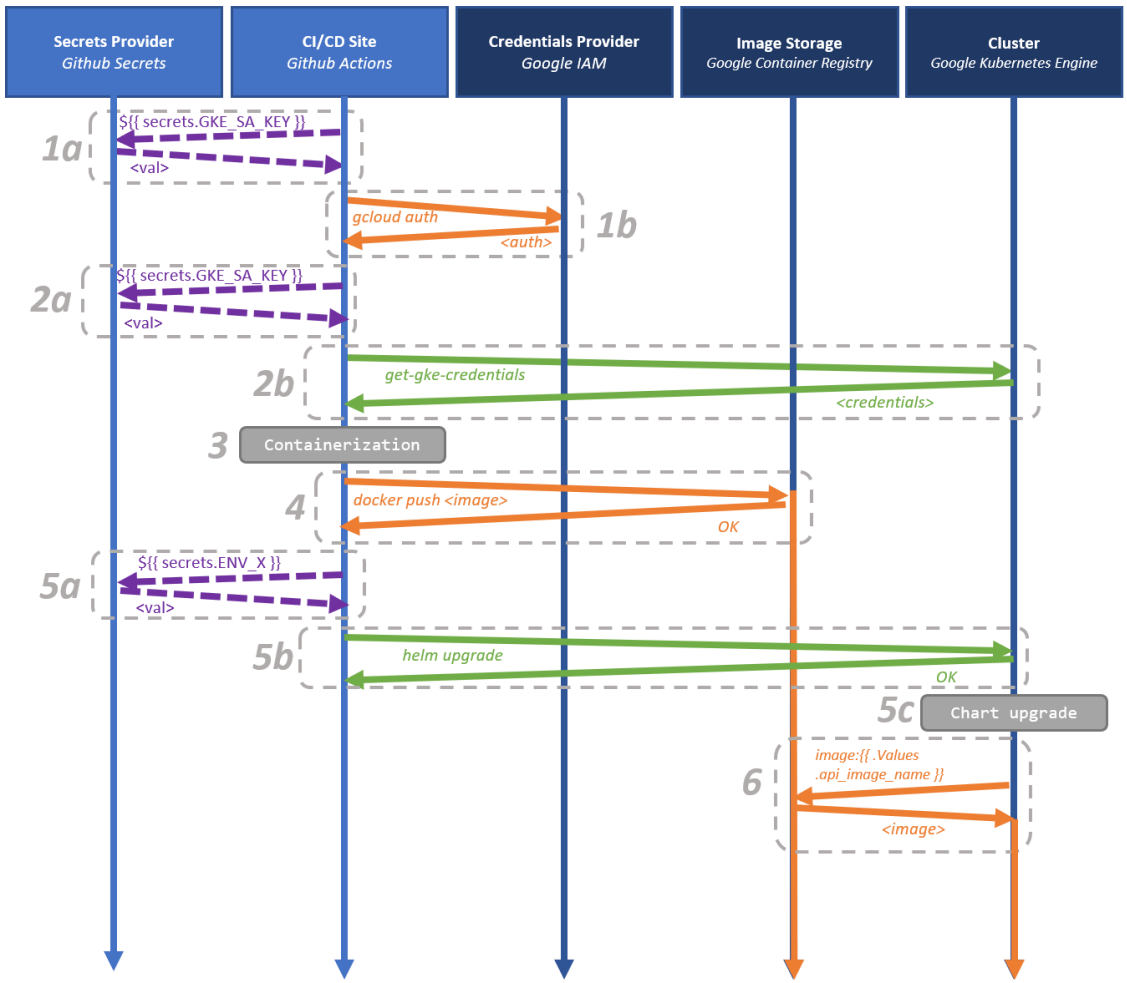

Figure 14: Details of the Continuous Deployment on the GKE

1. This shows the relationship between the Continuous Deployment and the secrets managed through RBAC. The RBAC protocol is discussed in § 5.7, the end result of using Github Secrets is to make secret values available within our Github Actions workflows via the templating mustache/double-curly

brackets, referencing them by key. This fetches the secret value **(1a)**. The secret value is returned to be used in the workflow, in this case the GCP service account key JSON (base64 encoded) which allows us to authenticate to push our image to the google container repository. A task is written to log in our project, fetching the credentials needed for subsequent `gcloud` operations. We configure Docker to use the auth in order to push the image to the Google Container Registry **(1b)**.

```
      - uses: google-github-actions/setup-gcloud@master
        with:
          project_id: ${{ secrets.GKE_PROJECT }}
          service_account_key: ${{ secrets.GKE_SA_KEY }}
          export_default_credentials: true
      - run: |-
          gcloud --quiet auth configure-docker
```

2. We also need access to our cluster's Kubernetes API to use `kubectl`/`helm` on it. This task needs the same service account key **(2a)**, to fetch the GKE credentials **(2b)**.

```
      - uses: google-github-actions/get-gke-credentials@main
        with:
          cluster_name: auto-cluster-1
          location: europe-west6
          credentials: ${{ secrets.GKE_SA_KEY }}
```

3. `docker build` is ran to containerize our codebase, using our Dockerfile. It takes commit metadata to generate a unique image. We use the `GITHUB_SHA` workflow environment variable which is set to the commit SHA that triggered the workflow, providing us with a unique tag each commit.

```
  - run: |-
      docker build -f api/Dockerfile . \
        --tag "gcr.io/$PROJECT_ID/$IMAGE:$GITHUB_SHA" \
        --build-arg GITHUB_SHA="$GITHUB_SHA" \
        --build-arg GITHUB_REF="$GITHUB_REF"
```

4. `docker push` publishes the image to the Google Container Repository, which is accessible due to our previous authentication. From this point, the GCR holds our commit-specific image.

```
  - run: |-
      docker push "gcr.io/$PROJECT_ID/$IMAGE:$GITHUB_SHA"
```

5. `helm upgrade --install` is used to install the Helm chart. To provide values to the templating engine, the non-sensitive YAML configuration file is passed, and sensitive key-values can be passed as well using the Github Secrets. This is the bridge to transfer the secrets from the RBAC'd secrets provider to the kubernetes cluster's image env. In **(5a)** the secrets are fetched, then `helm upgrade` triggers a chart upgrade in the cluster **(5b)**. During this operation, the Helm chart templates have access to the secret values (from the `--set x=y` directives) and can forward them into the environments of the Kubernetes Deployment, referencing them by name once again **(5c)**. The secrets plaintext is thus contained into the HTTP requests from Github to the GKE, their protection secured by TLS.

```
    run: |-
      helm upgrade --install webchart api/chart -f api/conf/webapp.prod.yaml \
```

```
3           --set api_image_name="gcr.io/$PROJECT_ID/$IMAGE:$GITHUB_SHA" \
4           --set env_x={{ secrets.ENV_X }} --set env_y={{ secrets.ENV_Y }} ...
```

Those secrets would typically hold the master credentials to our MongoDB and PostgreSQL databases, JWT secret key, and other sensitive data for security, which we cannot version control in our Configuration file `webapp.prod.yaml`.

6. The last step is automatically performed by the chart upgrade, the image of the Deployment is set by the templating engine to the value of `api_image_name` which is an URL to the `gcr.io` image - it fetches this image to deploy it in the cluster. The API is running, its environment contains the secrets.

5.7 RBAC Human Implementation

Secrets are needed in CD workflows. We create an implicit, "human RBAC", through Cloud tools credentials access for them. This is a simple human process matching the definition of RBAC in § 4.7.1 to avoid the proliferation of technologies in our Solution Stack for this prototype. *Roles* are defined as having access:

- *ProjectOwner*: to the Github repository settings page
- *ClusterAdmin*: to the GCP Console
- *Developer*: to neither

With this system, the *ProjectOwner* defines the secrets keys mapped to values. Then, the *Developer* can use secrets in CI/CD pipelines and in the API codebase, by knowing its key but never its value. Finally, we have to keep in mind that the *ClusterAdmin* can read the secrets, so this role needs to be kept as small as possible. Using secrets along the CI/CD pipeline can thus be viewed like:

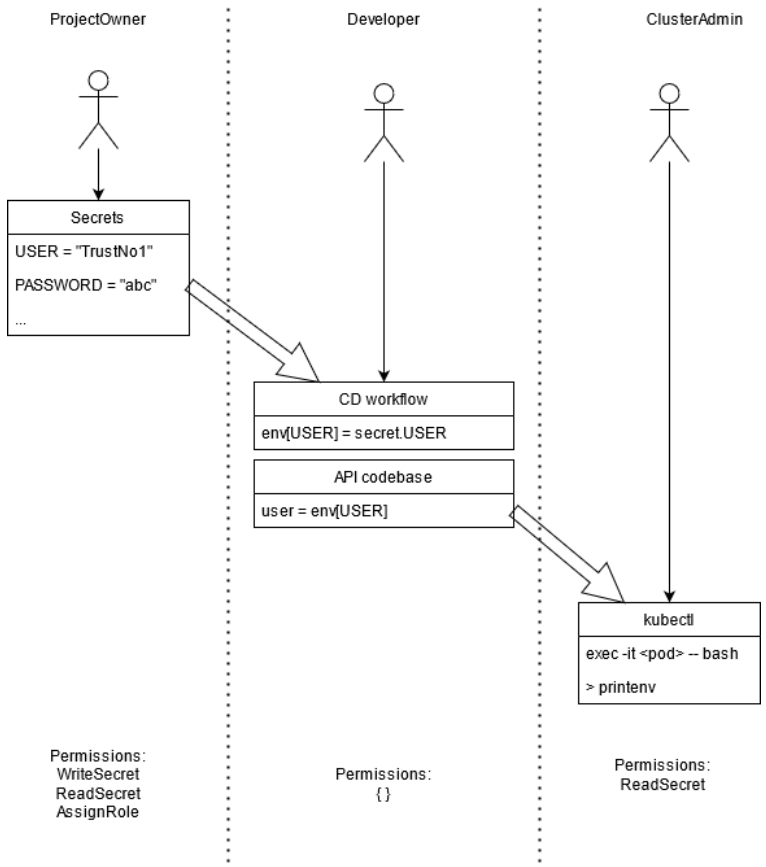

Figure 15: RBAC Users, Roles and Permissions involved in using secrets

As a result we circumvent using a separate RBAC tool, restricting only the access to Cloud tools. The transfer of access to the tools can be scheduled by human interaction.

5.8 Database Administration Tools

Two databases live in the Web Deployment cluster. We want to use GUI administration tools on the databases, for monitoring, custom queries, etc. We discuss in § 5.3 the two LoadBalancer services that are interfacing with the databases. Let's name the databases deployments `postgres` and `mongo`. The LoadBalancers can either be defined in the Helm Chart, or created ad-hoc via the commands:

```
kubectl expose deployment mongo --type=LoadBalancer --name=lb-mongo
kubectl expose deployment postgres --type=LoadBalancer --name=lb-postgres
```

Running `kubectl get services`, one can see the pending and then ready external IPs that get assigned to the load balancers. When ready, assuming PostgreSQL database is accessible at `postgres://<admin>:<pwd>@postgres:5432` within the cluster, we access our database using an administration tool, *pgAdmin4*, using URL `postgres://<admin>:<pwd>@<external_ip>:5432`.

For MongoDB, we use *MongoDB Compass* for a similar purpose, using URL `mongodb://<admin>:<pwd>@<external_ip>:27017`.

6 Discussion

One of the first class citizen of an API implementation is the relationship between Resources and Flask View Functions. They form a tight couple, where the Resource is the abstraction of the server-side concept exposed, and the Flask View Function is the Python implementation accepting requests for and serving representations of that Resource.

We have introduced a classification of those couples in the API Specification § 4.5. They form the first class citizens of the API, and hopefully with our architecture they become the main point of focus of the developer's attention. This is important for a company resources because this is where the domain logic is implemented. We also discussed how those Resources/View Functions are grouped within Blueprints which are then aggregated within a monolithic application (for organization purposes).

The first kind of Resources/View Functions implemented are typically Models. Here, there should be no hurdles, using our stack this is a straight-forward process. Simple Actions that only mutate the database models are often straight-forward as well. Complexity comes when implementing Transformations, because at this point the developer might need to "request" data from the same or other Blueprints to transform.

This is not trivial using Flask, because of the Application and Request global proxies, which are introduced in § 2.2.3. It means that to directly call a view function from another view function, the developer needs to prepare and push Application and Request contexts on top of those stacks, and configure them well to retrieve the right answer. This is technically possible but seems to be discouraged, by the design decisions of Flask and poor documentation on the topic.

6.1 Cross-view calls

Let's imagine an example where a Model view function `view_financings(id)` is implemented, and we need to implement a Transformation view function `view_cashflows(id)` that transforms `view_financings` data and returns the result.

First, `view_financings(id)` can accept an optional `id` to either work upon a single financing, or to work on the whole collection. It will implement conditional logic based on the HTTP Verb, using mainly SQLAlchemy and Marshmallow to load/dump representations and work with it on the underlying database of financings. Maybe `view_financings` performs additional, intermediary logic on the financings as well.

Second, `view_cashflows(id)` would only be used with `GET`, and has to fetch either on one or the collection of financings data to transform it. We don't want to repeat ourselves, like re-implementing the whole `view_financings` logic, so it makes sense that we want to "call" the `view_financings` as a first step here.

Two methods can be used:

- *Decoupling* the Resource logic from the View Function callable: we implement the logic of `view_financings(id)` in a separate function which is not coupled with the Flask blueprint and does not use Application or Request contexts. It accepts arguments describing the request we want to perform, like `resource_financings(method, req_json, id)`. Then, `view_financings(id)`, which is coupled with the Flask blueprint this time, calls `resource_financings(request.method, request.json, id)`, which means we don't have to care about the Request Context in the logic anymore. Then, we do the same for `view_cashflows(id)` calling some `resource_cashflows("GET", {}, id)`. Then, `resource_cashflows` calls `resources_financings("GET", {}, id)` and goes on with its own logic, like any other function in our program.

- *Self-requests*, using the `requests` module for example. The server can create a `GET` HTTP Request to `<prefix>/financings` on itself and transform the HTTP Response's data. This works out-of-the-box with a monolithic server, but it does add unnecessary overhead compared to the previous method. Here, the Application and Request contexts will be properly handled by Flask for us in the call of the other view function, as with receiving any other typical HTTP traffic.

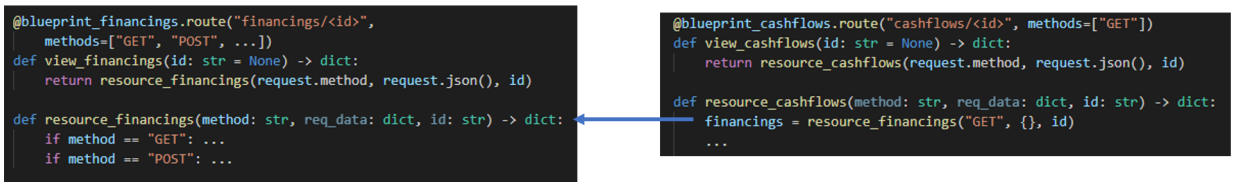

Figure 16: Using Decoupling

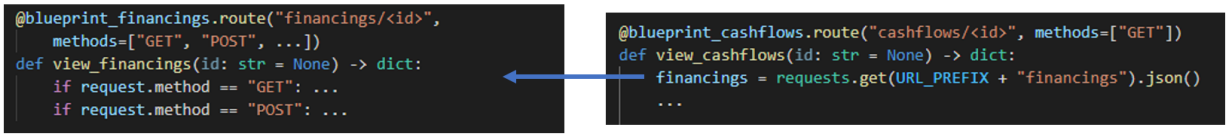

Figure 17: Using Self-Requests

For a monolithic server, it's not a real debate on which method to use. Decoupling feels natural, it organizes the code by itself, and doesn't add complexity. However, this breaks down when using a micro-service based architecture.

6.2 Micro-service architecture

Going towards a micro-service architecture implies to replace a monolithic server with a network of independent servers that communicate with each other typically over HTTP, eventually providing the same interface. An advantage of this is the ability to scale those independent services heterogeneously, depending, for example, on their current load. Another one is the ability to provide a clearer separation of concerns between the sub-parts of the API, as well as the ability to replace them with whatever respect their own specification without disrupting the greater software system. For example, a high load service could be rewritten in a lower-level compiled language to improve performance. In our clusterized system, micro-services would take the form of independent Deployments on separate Pods/ReplicaSets. In some way, having two database Pods external to the main monolithic Server Pod is already a start in this direction.

Through Blueprints, it's convenient to compose micro-services coming from the monolithic server codebase. We can just have small apps that import only the Flask extensions we need, and create the Flask app with only the Blueprints it handles. Let's imagine we have a four-blueprint application: $Blueprint_{data}$ for relational data, $Blueprint_{users}$ for user non-relational data, $Blueprint_{prefs}$ that updates some user preferences from the data of a third-party server, and $Blueprint_{cashflows}$ for transformations of our $Blueprint_{data}$.

Now let's imagine we want to split the user-related blueprint into a separate service for our Web Deployment, "`ServiceUsers`", away from a second service with the rest of blueprints, "`ServiceMaster`".

In `server_master.py`:

$$app_{master} := Blueprint_{data} + Blueprint_{cashflows} + Blueprint_{prefs}$$

In `server_users.py`:

$$app_{users} := Blueprint_{users}$$

Then we make two Docker images with those, deploy them as two Kubernetes Deployments, which will have their separate internal IPs, GUnicorn servers running, etc.

Here we are forced to use `requests` to communicate, with the `ServiceMaster` requesting data from `ServiceUsers` using the cluster-DNS mapped host, as they do not exist on the same Docker container, Kubernetes Pod and Services anymore.

But now, this breaks our Deployment-Agnosticism principle. To restore it, we *could* have in our config the hosts of all services, with the Webapp Deployment having the true cluster hosts, and the Desktop Deployment

having `localhost` everywhere. This implies making all of our "cross-blueprint" calls to be made with HTTP requests to the relevant host. What we really want instead, is a structure that will either:

- use HTTP requests to the relevant view functions, *if the remote Blueprint is on a different service than the emitting Blueprint.*
- Directly call the relevant Decoupled resource functions, *if the remote Blueprint is on the same service (same address space) as the emitting Blueprint.*

Our entire application can be viewed as a graph of Blueprints with needs to request data from each other (we also have relational and non-relational databases that are already generic based on the methods we implemented with SQLAlchemy and GenericUserStore).

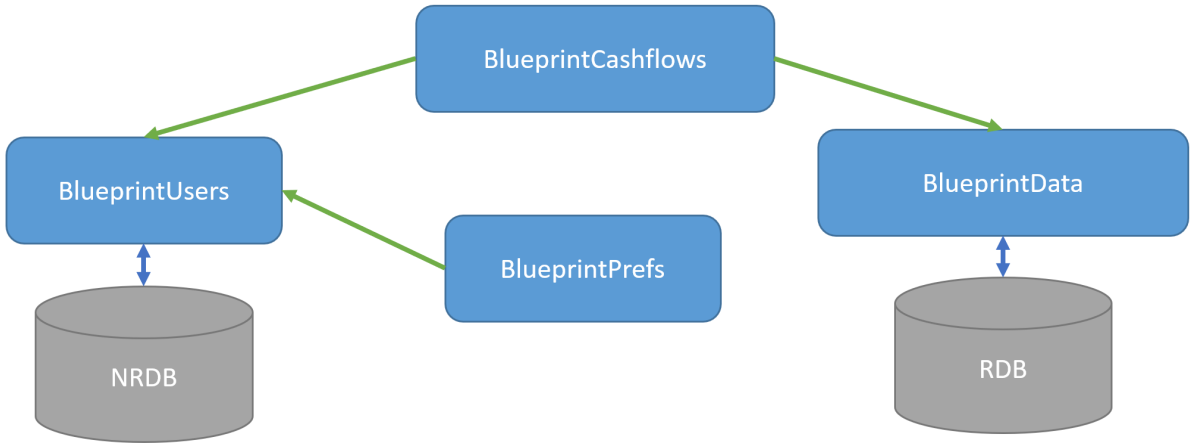

Figure 18: Blueprint graph with request edges and databases

The edges needs to be agnostic on whether they have to remotely fetch across a service partition between the blueprints, or if they have to fetch data from a local `resource_*` function.

We implement this using a kind of Bridge pattern that can turn into a Proxy. It can be named `GenericBlueprintProxy` and depending on said predicates, it will either use HTTP requests to the remote receiver Blueprint, or call directly the functions from the receiver Blueprint, using a secondary Adapter pattern [28].

This adapter can be named `DirectInstanceRequests` and it adapts the interface of the `requests` package into direct calls to the `resource_*` functions, so that the `GenericBlueprintProxy` can bridge either interchangeably.

We also add a `GenericBlueprintResponse` for wrapping both `requests` JSON response payload or `DirectInstanceRequests` `dict` returned data.

The `GenericBlueprintProxy` needs to get the instance of the blueprint it points to, when they live on the same service. When they don't, instead we need the service host URL. In both cases the `GenericBlueprintProxy` exposes verbs `get`, `post`, `put`, etc. applied on some `url` (which is actually a "url suffix", which we append to the service host URL when remote, or we match to get the function name and path args when local).

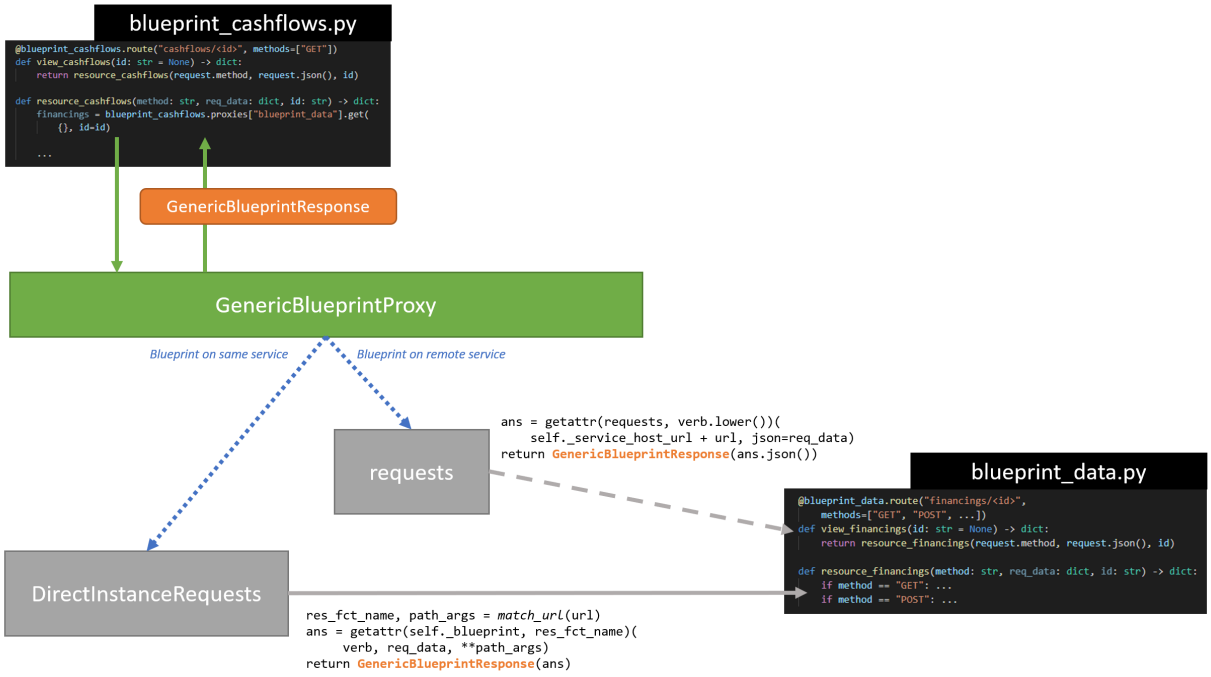

Figure 19: Flow of the GenericBlueprintProxy bridge depending on the context with snippets illustrating how to implement genericity. Here with $Blueprint_{cashflows}$ and $Blueprint_{data}$ on the same service, it will use DirectInstanceRequests as a result

Some `proxies` dictionary is provided to each blueprint, mapping their needed-blueprint's names to the provisioned `GenericBlueprintProxy`. We can do that by monkey-patching the `proxies` dictionary into the blueprint instances for example. Or more conventionally we could wrap blueprint components into actual classes with routes as `@classmethod`s.

Some linking algorithm is needed to assign `GenericBlueprintProxy` to the right back-end on start-up, based on the arbitrary partition of our Blueprints between services in our graph:

```
for BluepA → BluepB in Graph:
    if ( BluepA in service ) and ( BluepB not in service ):
        BluepA.proxies["blueprint_b"] = GenericBlueprintProxy(requests, url=get_url( BluepB ))

    if ( BluepA in service ) and ( BluepB in service ):
        BluepA.proxies["blueprint_b"] = GenericBlueprintProxy(DirectInstanceRequests( BluepB ))
```

Figure 20: Pseudo-code to link Blueprints with proxies

For the Relational databases, multiple SQLAlchemy targets are supported using URL entries as `app.config["SQLALCHEMY_BINDS"]`.Then it needs to be target with `bind="..."` in `db.create_all()` and similar operations, as well as specifying a `__bind_key__` field in Models to assign them to a specific RDB.

For the Non-Relational databases, we re-use our `GenericUserStore` described in § 5.2.3, giving it a more general name like `GenericNrdbStore`. For local JSON file mode we just have to specify a different "url" (filepath), and for the MongoDB mode we can create separate `PyMongo(app, uri="mongodb://...")` instances, specifying the `uri` directly instead of using `app.config["MONGO_URI"]`. Those `GenericNrdbStore` instances will have to be stored in the Blueprints to perform transactions, unlike the SQLAlchemy managed dbs.

Therefore, the linking algorithm is extended to expose RDBs and NRDBs to the Blueprints that need them:

```
for BluepA → RdbA in Graph:
    if (BluepA in service):
        app.config["SQLALCHEMY_BINDS"] |= { "rdb_a" : os.environ["rdb_a_url"] }

for BluepA → NrdbA in Graph:
    if (BluepA in service):
        BluepA.nrdbs["nrdb_a"] = GenericNrdbStore(
            mode=os.environ["nrdb_a_mode"],
            url=os.environ["nrdb_a_url"])
```

Figure 21: Pseudo-code to link databases

Deployment-Agnosticism is restored: in the case of our Desktop Deployment, the Company Desktop API is a monolith with all blueprints on the same server, so all `GenericBlueprintProxy` use the `DirectInstanceRequests` to the relevant Blueprints, and we never have actual `localhost` HTTP traffic, while the databases get deployed as local files.

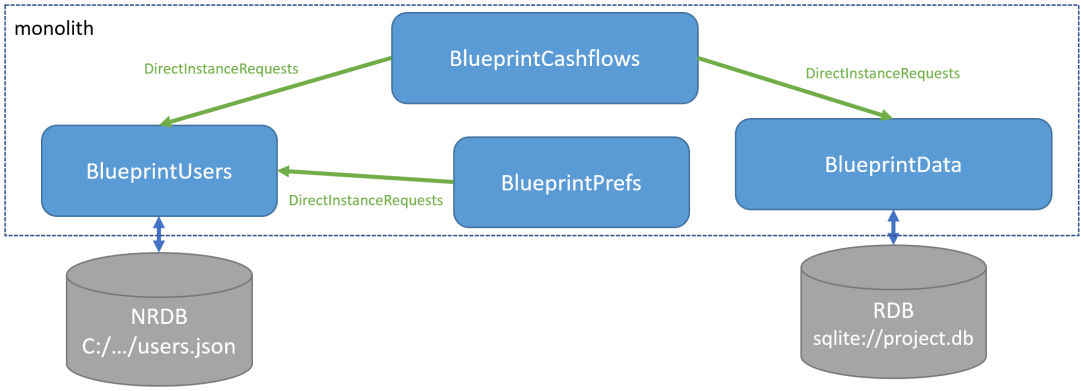

Figure 22: Desktop Deployment of our components graph

At the same time, we have the option to arbitrarily split our Web Deployment in micro-services within our cluster, allowing Blueprints to communicate in an efficient manner, and having our databases are external services as well, from the same domain codebase as with our Desktop Deployment.

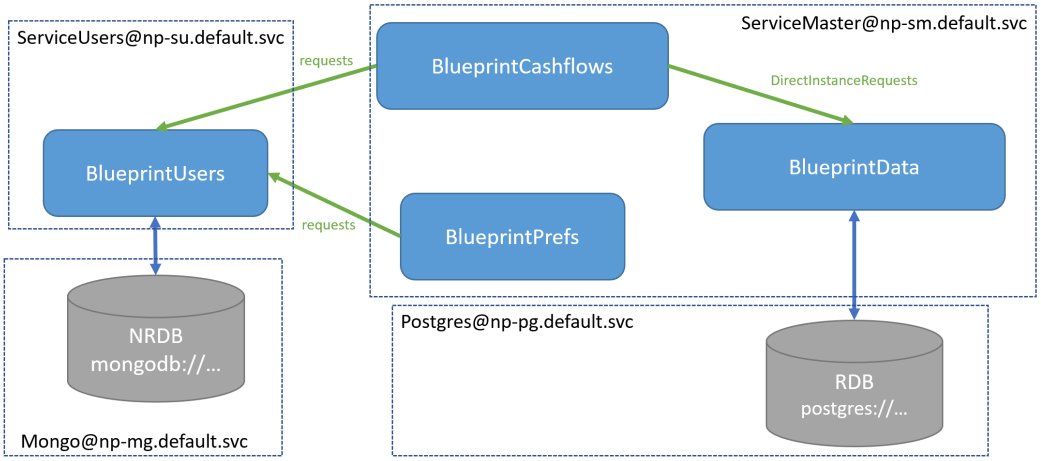

Figure 23: Webapp Deployment of our components graph, with the Kubernetes service names exposing them

6.3 Micro-Services Remarks

A prototype of this architecture was tried for our project. This approach increases complexity, and the advantages of micro-services were not sufficient to justify it. As it stands currently it appears that micro-services, cluster based applications that would be written as sub-components entirely agnostic of their cluster topology and deployment-agnostic as well is not practical for just a few developers, and many other topics have to be considered such as synchronicity, redirects, etc. Perhaps a better-suited Python web framework than Flask with this design philosophy in mind, potentially twinned with a tailor-made platform-as-a-service supporting its deployment, can enable this development process at human scale.

6.4 Closing Remarks

Overall, it is possible to achieve our outlined needs, with the technologies available today, using approaches presented in this report. Going the monolithic server route spares a lot of trouble for the extremely small team, while preserving the Deployment-Agnosticism. However, for extremely small teams, even the monolithic server results in a large increase of complexity compared to only going for Desktop-only or Web-only Deployment.

We observed from our real-world implementation of the contents of this report that this hybrid architecture is functional. Is the complexity is greater or smaller than the sum of two parallel projects that achieve the same goal? Impossible to say without having tried said alternative as well. However, we can imagine that having more specialized tools and frameworks for this purpose could save development time and reduce complexity, and they would be worth creating, if other teams face the same needs.

Trends in software, cloud technology, and internet coverage outline a future where user-facing software are increasingly Web applications, without any local installation. This report also shows that with some planning and some implementation detours, a Web application can be hybridized into a Desktop application.

7 Appendices

7.1 Appendix: Flask Internal mechanism

- In the `flask.app.Flask` class we have the WSGI Application Callable `Flask.__call__(self, environ, start_response)`
- This method directly returns `Flask.wsgi_app(self, environ, start_response)` which actually implements the WSGI Application Callable logic.
- When we use the `@app.route(endpoint)` decorator on view functions, or `@blueprint.route(endpoint)` followed by a `app.register_blueprint(blueprint)`, internally Flask will save those within the `self.view_functions` mapping, from endpoints to the view functions.
- `wsgi_app()`, which `Flask.__call__` calls, performs the following:
  - pushes a `flask.ctx.RequestContext` on the Request context stack
  - calls `Flask.full_dispatch_request()` which pre-processes the request and calls `Flask.dispatch_request()`
  - `dispatch_request()` observes the RequestContext from the stack top, saves it in `req`
  - `dispatch_request()` calls `self.view_functions[rule.endpoint](**req.view_args)`
- Through this, our endpoints-mapped view functions execute to treat the request, and the Request proxy gets provisioned for its usage within them.

7.2 Appendix - Minimal Flask Data Server

```python
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from marshmallow_sqlalchemy import SQLAlchemyAutoSchema

app = Flask(__name__)
db = SQLAlchemy(app)

class Financing(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    notional = db.Column(db.Integer)
    currency = db.Column(db.String)

class FinancingSchema(SQLAlchemyAutoSchema):
    class Meta:
        model = Financing
        load_instance = True

@app.route("/financings", methods=["GET", "POST"], defaults={"uid": None})
@app.route("/financings/<uid>", methods=["GET", "DELETE"])
def view_financing(uid: int) -> dict:
    if uid is None:
        if request.method == "GET":
            financing_models = Financing.query.all()
            financings_dump = FinancingSchema(many=True).dump(financing_models)
            return jsonify(financings_dump)
        if request.method == "POST":
            financing_model = FinancingSchema(session=db.session).load(request.json)
            db.session.add(financing_model)
            db.session.commit()
            return "Created", 201
    else:
        if request.method == "GET":
            financing_model = Financing.query.filter(Financing.id == uid).first()
            financing_dump = FinancingSchema().dump(financing_model)
            return financing_dump
        if request.method == "DELETE":
            Financing.query.filter(Financing.id == uid).delete()
            db.session.commit()
            return "Deleted", 204

if __name__=="__main__":
    app.config |= { "DEBUG" : True, "SQLALCHEMY_DATABASE_URI" : "sqlite://" }
    with app.app_context():
        db.create_all()
    app.run()
```

This can be used, for example for posting an entry and fetching it back:

```
curl -X POST -H "Content-Type: application/json" \
    -d '{"id":1, "notional":2500000, "currency":"CHF"}' http://localhost:5000/financings
curl http://localhost:5000/financings # db entry model representation fetched
```

7.3 Appendix - Minimal Flask User Server

```python
from flask import Flask, json, request, jsonify
from flask_jwt_extended import current_user, jwt_required, JWTManager, create_access_token

app = Flask(__name__)
jwt = JWTManager(app)
_users = []

@jwt.user_identity_loader
def user_identity_lookup(user):
    # Creates the identity from the user
    return user["username"]

@jwt.user_lookup_loader
def user_lookup_callback(_jwt_header, jwt_data):
    # Loads the identity and returns the user
    identity = jwt_data["sub"]
    u = next((u for u in _users if u["username"] == identity), None)
    return u

@app.route("/users", methods=["GET", "POST"])
def view_users():
    if request.method == "GET":
        return jsonify(_users)
    if request.method == "POST":
        # Registers the user. In production, the password is hashed and salted
        _users.append(request.json)
        return "Registered", 201

@app.route("/login", methods=["POST"])
def view_login():
    if request.method == "POST":
        username = request.json["username"]
        u = next((u for u in _users if u["username"] == username), None)
        # Checks if user exists and the password matches
        # We would use a hash, salt, secure string comparaison in production
        if not u or u["password"] != request.json["password"]:
            return "Wrong username or password", 401
        token = create_access_token(identity=u)
        return jsonify(access_token=token)

@app.route("/protected_resource")
@jwt_required()
def view_protected_resource():
    if request.method == "GET":
        # Here we implement whatever user-partitioning of our data
        result = 1 if current_user["username"] == "louislandelle" else 0
        return jsonify({"result": result, "accessed_by": current_user["username"]})

if __name__=="__main__":
    app.config |= { "JWT_SECRET_KEY" : "some_secret" }
    app.run()
```

This can be used, for example with the following Python client script:

```python
import requests
URL = "http://localhost:5000"
requests.post(URL + "/users", json={"username":"louislandelle", "password":"123"})

# Wrong password
ans = requests.post(URL + "/login", json={"username":"louislandelle", "password":"456"})
print(ans.content.decode()) # > Wrong username or password

# Correct password
ans = requests.post(URL + "/login", json={"username":"louislandelle", "password":"123"})
print(ans.content.decode()) # > {"access_token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ey \
#JmcmVzaCI6ZmFsc2UsImlhdCI6MTYyOTExNzc1NiwianRpIjoiNWNiODM5ZmUtNDQ0My00ZmI1LTg3ODUtMmM5Z \
#Dg2NDkyNzIxIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6ImxvdWlzbGFuZGVsbGUiLCJuYmYiOjE2MjkxMTc3NTYs \
#ImV4cCI6MTYyOTExODY1Nn0.ovpvBJ_ISBOFCeqYPgHxRrXDXDHa8t17soHRXv1sm-I"}
_auth = {"headers": {"Authorization" : "Bearer " + ans.json()["access_token"]}}

# Without Authorization
ans = requests.get(URL + "/protected_resource")
print(ans.json()) # > {'msg': 'Missing Authorization Header'}

# With Authorization
ans = requests.get(URL + "/protected_resource", **_auth)
print(ans.json()) # {'accessed_by': 'louislandelle', 'result': 1}
```

7.4 Appendix: GUnicorn / Flask Interaction based on WSGI

Having laid out the purpose and specification of the WSGI, it would be useful to understand how an actual WSGI HTTP Server (GUnicorn) interacts with an actual WSGI Web Framework (Flask), behind the scenes. Unfortunately documentation on this topic is sparse, so we find it useful to go through the GUnicorn source code following the trace from running it via the CLI:

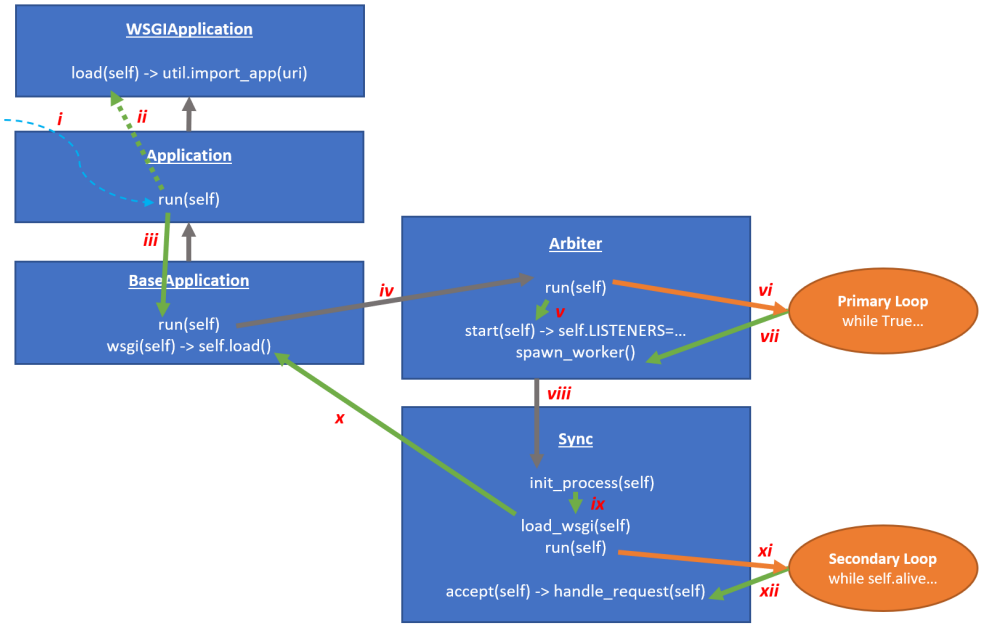

Figure 24: Steps after running GUnicorn

7.4.1 Running GUnicorn

(i) Running gunicorn from the command line runs `Application.run()` inherited by `WSGIApplication`. `WSGIApplication` is a child class of `Application`, itself a child class of `BaseApplication`.

7.4.2 Loading the WSGI Web Framework's WSGI Application Callable

- `Application.run()`'s basic mechanism it to call (ii) `self.load()` then (iii) `super().run()`
- (ii) `self.load()` calls `WSGIApplication.load()` because it overrides `BaseApplication.load()` , but this is only to TRY if the WSGI Application Callable import works. `WSGIApplication.load()` loads the WSGI Web Framework's WSGI Application Callable, that gunicorn was called on (`Flask.__call__` in our case), and returns it (this operation uses Python introspection functionalities)
- (iii) `super().run()` calls `BaseApplication.run()`
- (iv) `BaseApplication.run()` calls `Arbiter(self).run()`, which sets `Arbiter.app := self` (pointing to the `WSGIApplication` instance)

7.4.3 Arbiter, Primary Loop

- Arbiter manages workers (spawn, kill based on the required worker amount, etc.)
- (v) `Arbiter(self).run()` calls `Arbiter.start()` which will create `self.LISTENERS`, which are the TCP sockets for incoming requests
- (vi) `Arbiter(self).run()` will start the Primary Infinite Master Loop, `while True:...`
- (vii) `Arbiter(self).run()`'s Loop will call `Arbiter.spawn_worker()` when needed

7.4.4 Arbiter, Spawning workers

- (viii) `Arbiter.spawn_worker()` instantiates workers objects with type `worker_class`, which needs to be a child-class `gunicorn.workers.base.Worker`.
- `Arbiter.spawn_worker()`'s instantiation passes `self.app` (the instance of `WSGIApplication`), the `self.LISTENERS` sockets, and other parameters.
- Then it calls the instantiated worker's `worker.init_process()`
- Some worker classes override this `worker.init_process()` , some don't defaulting with `base.Worker.init_process`. This forks the parent process to create a new worker process.

7.4.5 Worker initiation

- The default worker class is `gunicorn.workers.sync.SyncWorker`: we use it in this scenario
- `SyncWorker` does not override `base.Worker.init_process()`
- `init_process()`'s basic mechanism is to save a pointer `self.PIPE := os.pipe`, (ix) call `self.load_wsgi()` and `self.run()`
- (x) `self.load_wsgi()` sets `self.wsgi := self.app.wsgi()`, which is `BaseApplication.wsgi()`, which caches `WSGIApplication.load()` (in our case, returning the Flask instance, the WSGI Application Callable) and returns it
- (ix) `self.run()` returns `NotImplementedError` in `workers.base.Worker`: it must be overridden by `Worker` child classes
- (xi) `workers.sync.SyncWorker.run()` starts the Secondary Loop, `while self.alive:...` (in reality there are two different loop functions for single and multi socket modes, but very similar)

7.4.6 Worker, Secondary Loop

Each loop cycle `SyncWorker`:

1. notifies the `Arbiter` of his aliveness
2. (xii) calls `self.accept(self.sockets[0])` (in single socket mode)
3. `self.accept()` calls `self.handle()`, which unwraps the potential TLS, parses the HTTP Request, and calls `self.handle_request(parsed)`
4. `self.handle_request()` calls `wsgi.create(req, _, _, socket, _)`
   - `wsgi.create()` creates a `gunicorn.http.wsgi.Response` instance and creates an environ dictionary, WSGI compliant, based on the request.
   - `Response` has the method `start_response(status, headers)`, which processes status and headers, and saves processed forms in the `Response`
   - `Response.write(arg: bytes)` is a function accepting bytes and sending them over the socket
   - it returns the `resp: Response` and the `environ: dict`
5. then it calls the central piece: `respiter = self.wsgi(environ, resp.start_response)`, in our case, remember that `self.wsgi := Flask(__name__)`, which gets forwarded to the `wsgi_app()` discussed in § 7.1.
   - the WSGI Application Callable of the WSGI Web Framework processes the WSGI environ.
   - the resulting status code and headers of the processing will be returned to the WSGI Web Server via `start_response(status, headers)`

- the resulting request body is returned by the WSGI Web Framework's WSGI Application Callable `self.wsgi`, and stored in `respiter`

6. `Response.write()` is called on the `respiter` content (the request body) which sends the headers stored in the `Response`, then the content (potentially chunked), over the socket

7. finally the `Response` is closed with `resp.close()`, the HTTP Response is concluded

7.5 Appendix - Ideal Production Sites

For our prototype we don't use staging sites, but a real production product would need them as a last-barrier testing ground, not only for the application but also to observe the result of our Continuous Deployment workflow.

This last validation can then be applied to the actual production site. For the Company Web API this would be implemented by creating a twin cluster, the staging cluster, on the Google Kubernetes Engine.

For a production product we can also imagine that the Desktop Installer make script is used with Continuous Deployment workflow as well, and the resulting binary gets uploaded to a third hosting site from where the customers can download it.

The more complete sites architecture would then resemble something more like this:

Figure 25: Sites architecture

References

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol-http/1.1," RFC 2616, RFC Editor, 1999.

[2] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," RFC 5246, RFC Editor, 2008.

[3] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011.

[4] G. Mulligan and D. Gračanin, "A comparison of soap and rest implementations of a service based interaction independence middleware framework," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pp. 1423–1432, IEEE, 2009.

[5] R. Fielding, "The design of network-based software architectures," *Doctoral Dissertation, School of Information and Computer Science*, 2000.

[6] R. Fielding, "Rest apis must be hypertext-driven," *Untangled, musings of Roy T. Fielding*, 2008.

[7] R. McCool, "Server scripts," *WWW-talk mailing list*, 1993.

[8] K. C. D. Robinson, "The common gateway interface (cgi) version 1.1," RFC 3875, RFC Editor, 2004.

[9] P. Eby, "Python web server gateway interface v1.0," PEP 0333, Python Software Foundation, 2003.

[10] P. Eby, "Python web server gateway interface v1.0.1," PEP 3333, Python Software Foundation, 2010.

[11] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A survey and comparison of relational and non-relational database," *International Journal of Engineering Research & Technology*, 2012.

[12] T. M. Connolly and C. E. Begg, *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.

[13] M. Lamburg, "Python database api specification v1.0," PEP 248, Python Software Foundation, 2001.

[14] M. Lamburg, "Python database api specification v2.0," PEP 249, Python Software Foundation, 2001.

[15] J. M. Barnes, "Object-relational mapping as a persistence mechanism for object-oriented applications," *Mathematics, Statistics, and Computer Science Honors Projects. 6.*, 2007.

[16] M. Portnoy, *Virtualization essentials*, vol. 19. John Wiley & Sons, 2012.

[17] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393, IEEE, 2015.

[18] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[19] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[20] G. Sayfan, *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[21] J. Shah and D. Dubaria, "Building modern clouds: using docker, kubernetes & google cloud platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0184–0189, IEEE, 2019.

[22] N. S. M. Jones, J. Bradley, "Json web token (jwt)," RFC 7519, RFC Editor, 2015.

[23] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, and A. L. Wolf, "A characterization framework for software deployment technologies," tech. rep., Colorado State Univ Fort Collins Dept of Computer Science, 1998.

[24] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[25] A. Contan, C. Dehelean, and L. Miclea, "Test automation pyramid from theory to practice," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–5, IEEE, 2018.

[26] D. Ferraiolo, J. Cugini, and D. Kuhn, "Role-based access control: features and motivations," *Proceedings of the 11th Annual Computer Security Applications Conference*, 1995.

[27] B. Warsaw, E. Bendersky, and E. Furman, "Adding an enum type to the python standard library," PEP 435, Python Software Foundation, 2013.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.