API implementation

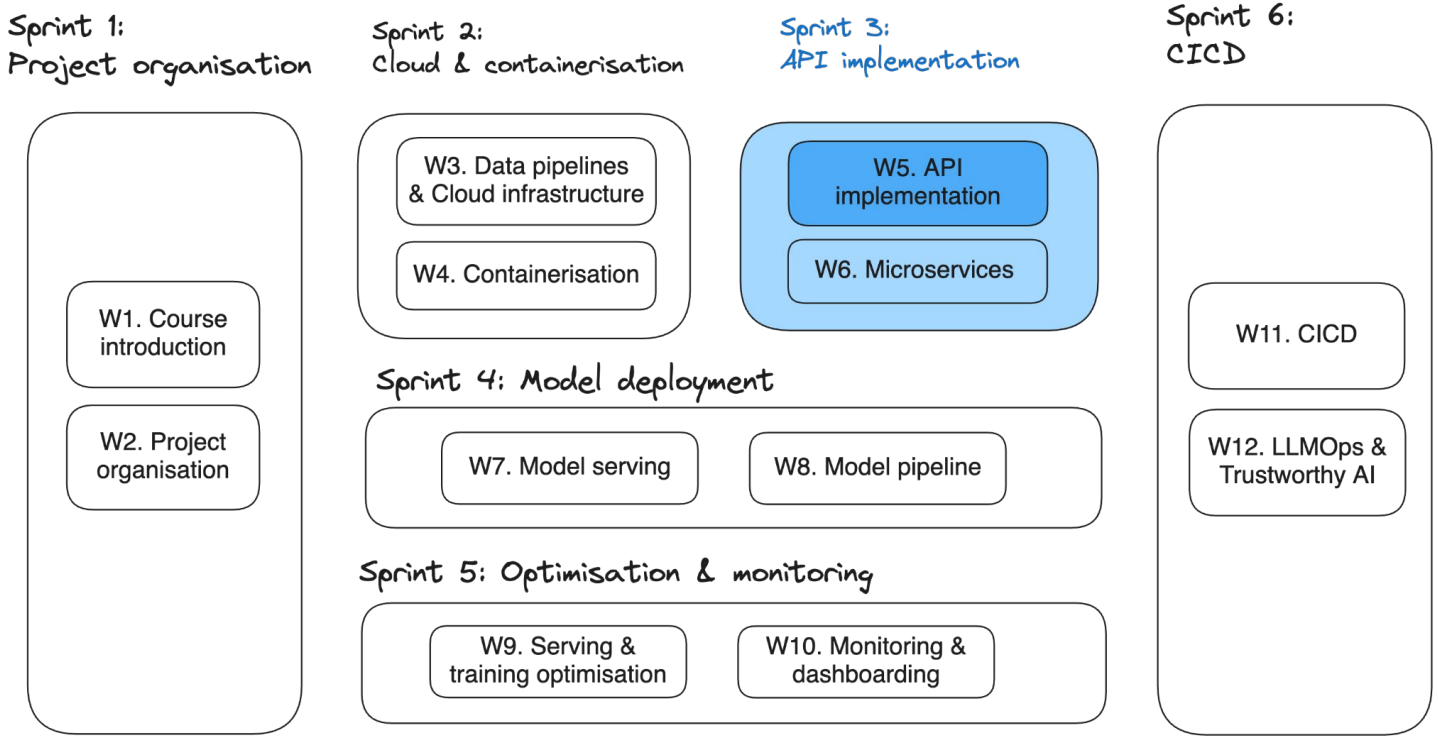
Sprint 3 - Week 5

INFO 9023 - Machine Learning Systems Design

Thomas Vrancken (t.vrancken@uliege.be)
Matthias Pirlet (matthias.pirlet@uliege.be)

2025 Spring

Status on our overall course roadmap



Agenda

What will we talk about today

Lecture (1h15)

1. API
2. REST
3. RPC
4. API Gateway

~~**Guest lecture** (45min)~~

5. ~~Connexion~~ → next week

Lab (45min)

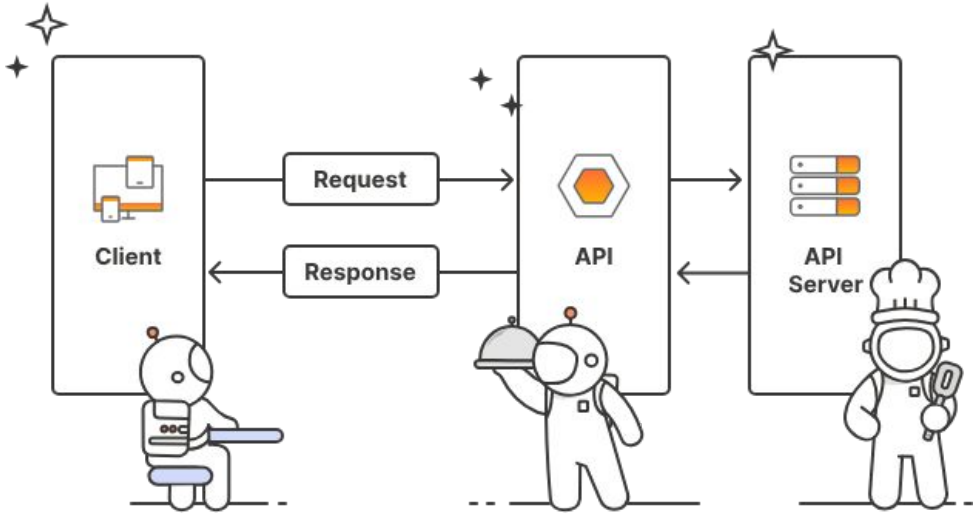
6. Flask

Lecture (30min)

7. *Microservices* → If time allows

API

Application Programming Interface (API)



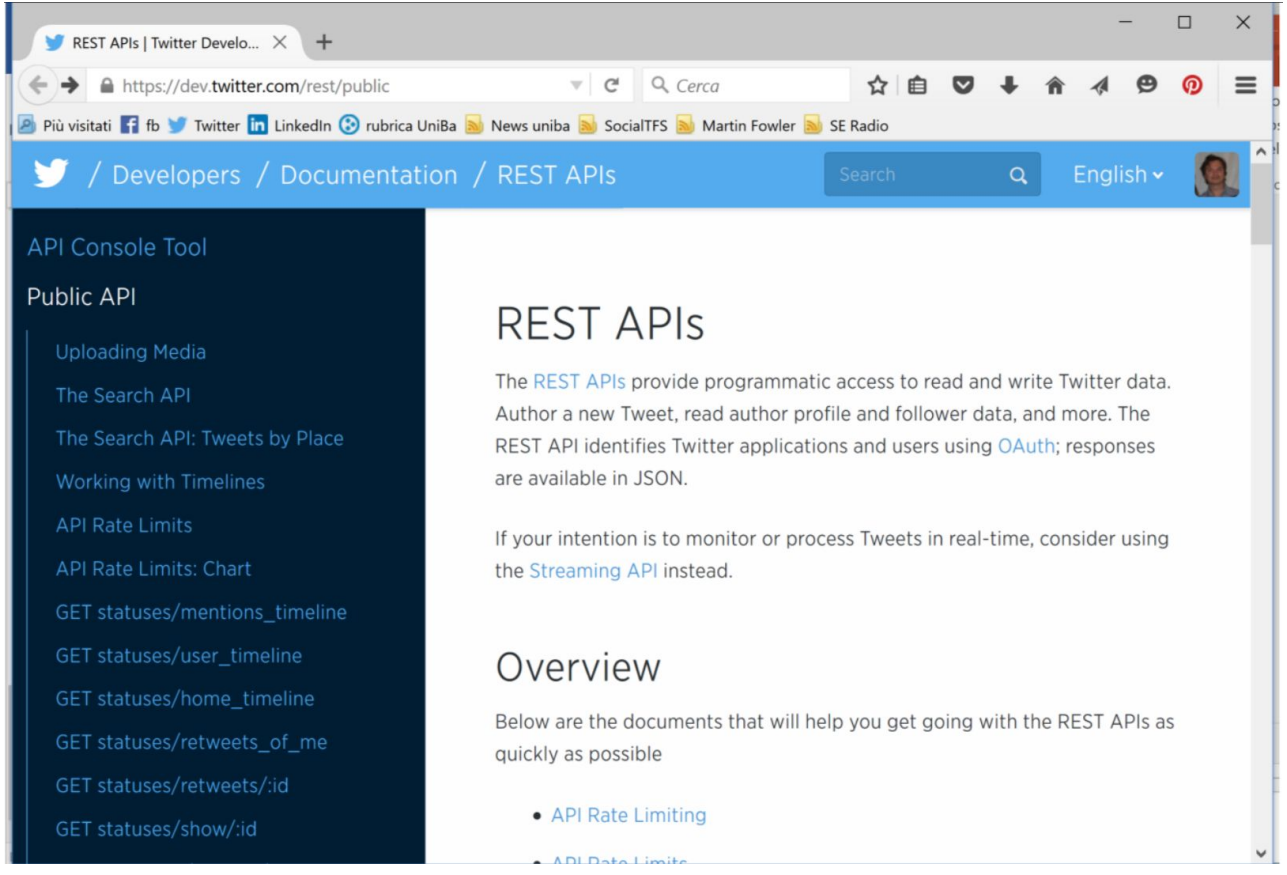
An **Application Programming Interface (API)** is a set of protocols that enable different software components to *communicate* and *transfer data*.

Client sends a **requests** and the API sends a **response**.

https://www.postman.com/what-is-an-api/

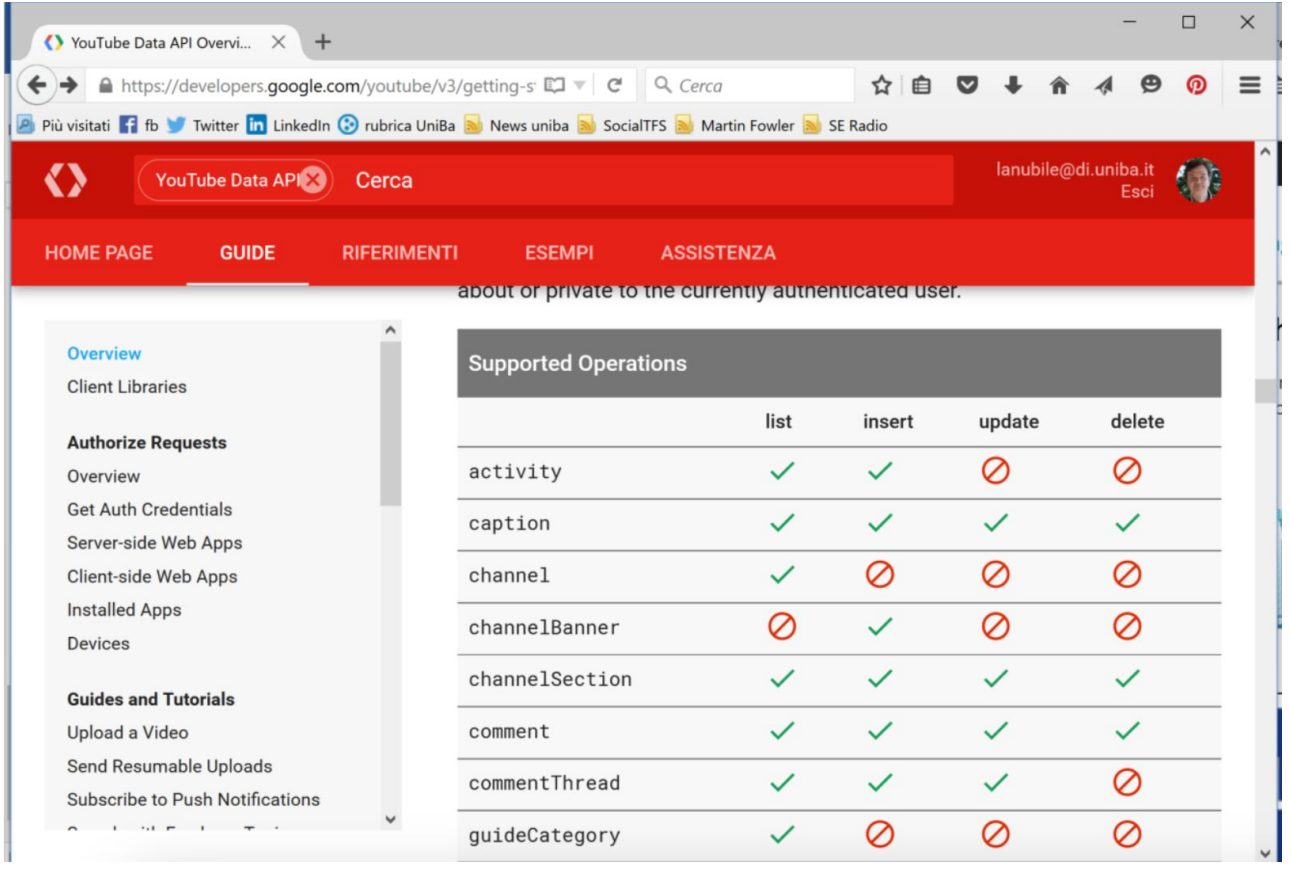

API
LIGHTYEAR
API EVERYWHERE
imgflip.com

APIs examples



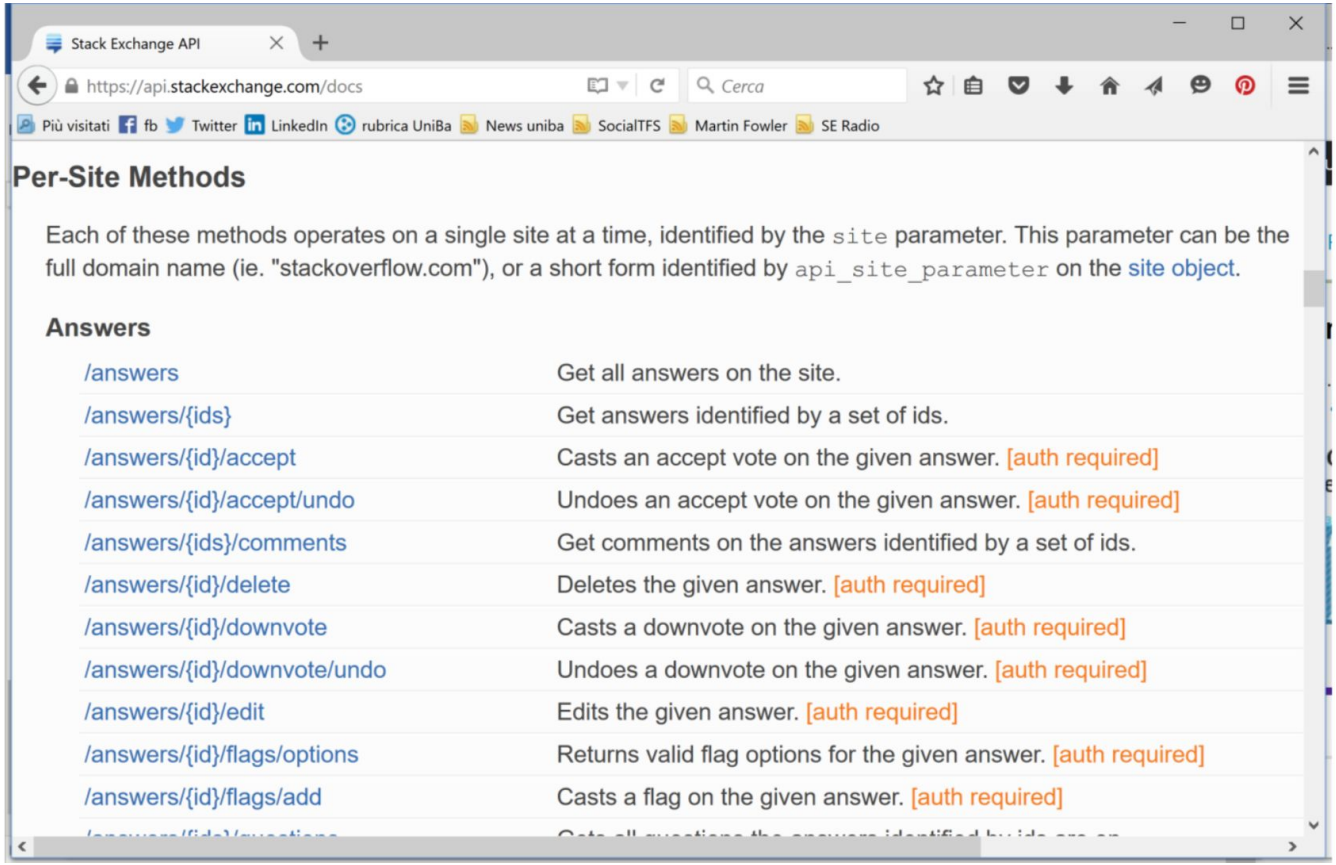
LIÈGE université

APIs examples



LIÈGE université

APIs examples



LIÈGE université

APIs can be accessed differently

Private APIs

- Used to connect different software components within a single organization
- Not available for third-party use
- Some applications may include dozens or even hundreds of private APIs

Public APIs

- Provide public access to an organization's data, functionality, or services
- Can be integrated into third-party applications
- Some public APIs are available for free, while others are offered as billable products

Partner APIs

- Enable two or more companies to share data or functionality in order to collaborate
- Not available to the general public
- Leverage authentication mechanisms to restrict access

https://www.postman.com/what-is-an-api/

Main types of APIs

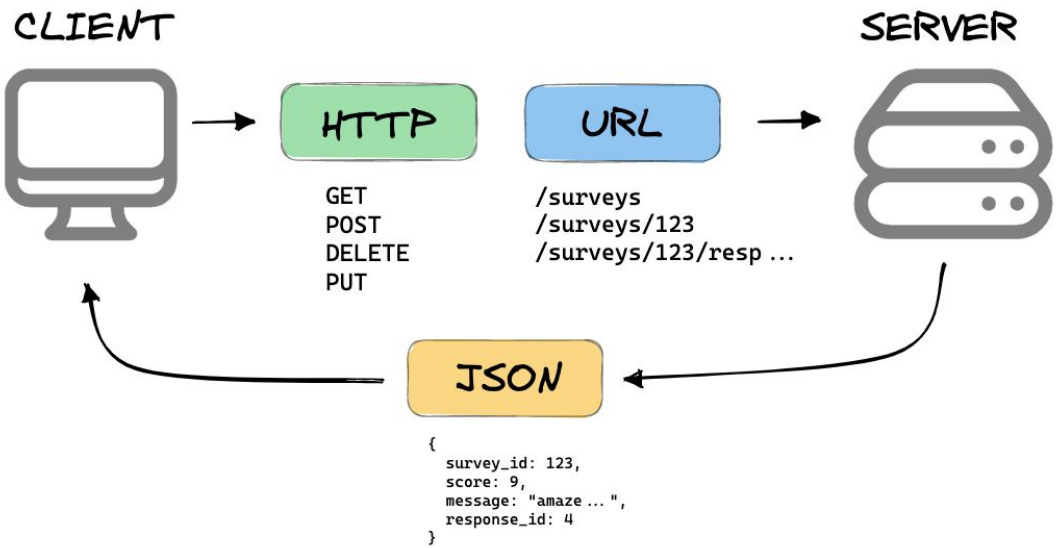
- **SOAP** (Simple Object Access Protocol). Client and server exchange messages using XML. Less flexible API that was more popular in the past.
- **Websocket**: WebSocket API supports two-way communication between client apps and the server. The server can send callback messages to connected clients, which is not possible in REST API.
- **REST** (Representational State Transfer): The client sends requests to the server. The server returns a response (output data) back to the client.
- **RPC** (Remote Procedure Calls). The client completes a function (aka method or procedure) on the server, and the server sends the output back to the client. “Action centric”.

Focus for today!

REST APIs

REpresentational State Transfer (REST) API

Roy Fielding's PhD thesis: [Architectural Styles and the Design of Network-based Software Architectures]



https://mannhowie.com/rest-api

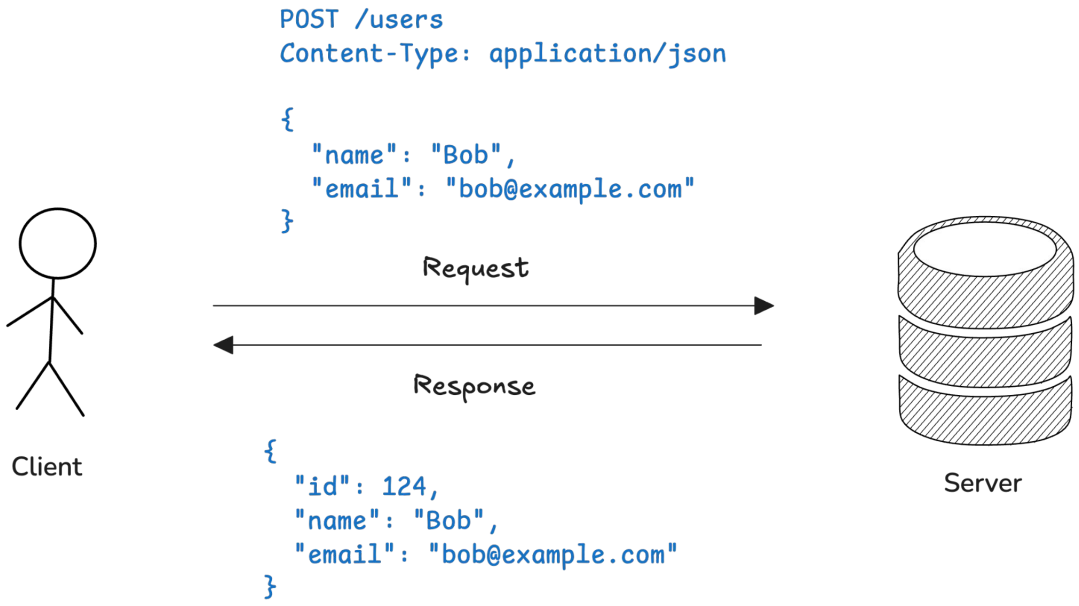
REpresentational State Transfer (REST) API

Retrieving information from server (GET)

Analogy: “Asking the waiter information about the menu at a restaurant.”

REpresentational State Transfer (REST) API

Submits data to the server to create a new resource (POST)



Analogy: “Ordering a new dish at a restaurant and getting it served.”

REpresentational State Transfer (REST) API

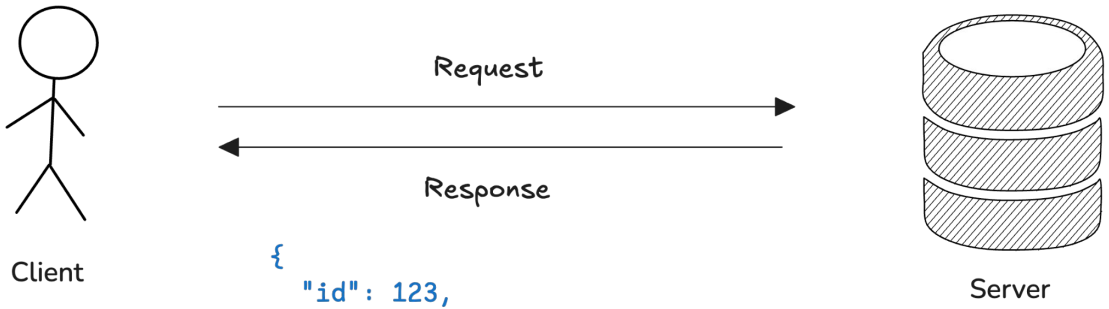
Replaces an existing resource with new data (PUT)

Analogy: “Changing your entire meal order at a restaurant after placing it.”

REpresentational State Transfer (REST) API

Partially updates an existing resource (PATCH)

```
PATCH /users/123
Content-Type: application/json

{
  "email": "newalice@example.com"
}
```

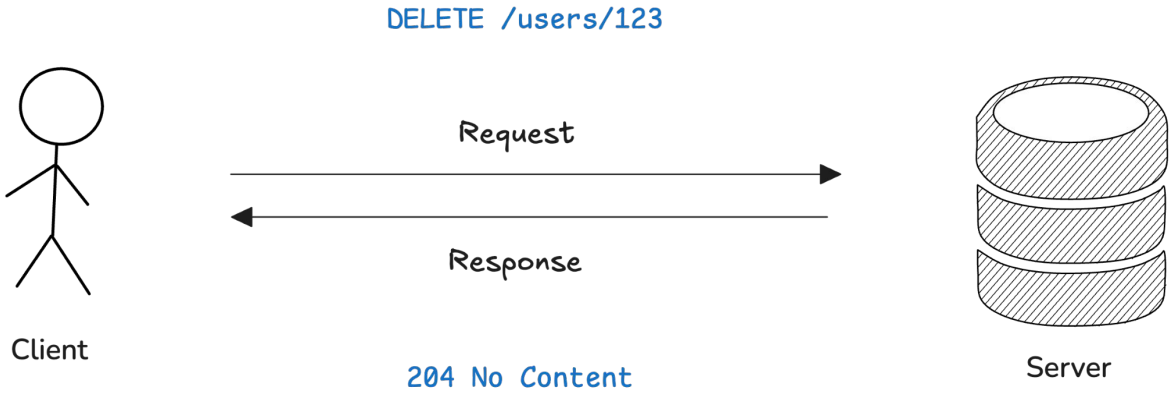



```
{
  "id": 123,
  "name": "Alice Updated",
  "email": "newalice@example.com"
}
```

Analogy: “Asking the waiter to add extra cheese to your existing order.”

REpresentational State Transfer (REST) API

Deletes an existing resource from the server (DELETE)



Analogy: “Canceling your order and removing it from the kitchen.”

REST API: HTTP requests

HTTP is a communication protocol for networked systems

⇒ How your computer accesses a webpage.

REST is based on **CRUD** operations: Able to **Create**, **Read**, **Update** and **Delete** resources

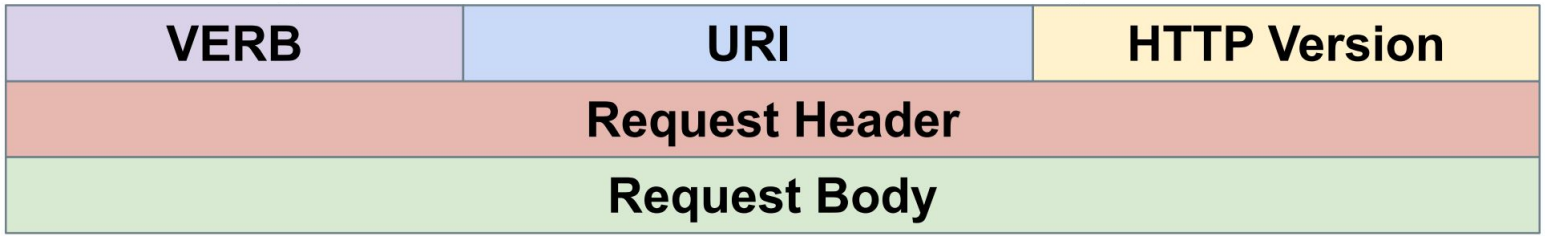
There are 4 basic **verb** commands when making a HTTP request

- **POST**: *Create* a new resource based on the payload given in the body of the request
- **GET**: *Read* data from a single or a list of multiple resources.
- **PUT/PATCH**: *Update* a specific resource (by ID)
- **DELETE**: *Delete* the given resource based on the id provided

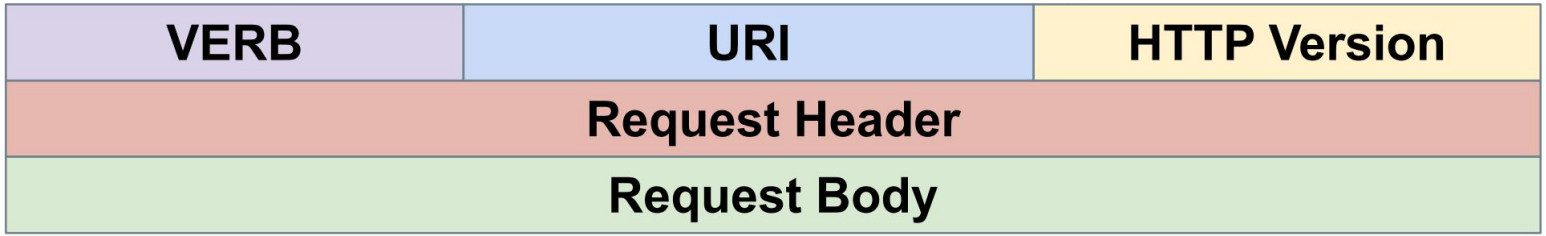
https://mannhowie.com/rest-api

Anatomy of an HTTP Request

- **Verb** (one of PUT, GET, POST, DELETE, …)
- **Endpoint (URI)** Identifies the resource upon which the operation will be performed (e.g. `/users/123`)
- The **Request Header** contains metadata such as
  - Collection of key-value pairs of headers and their values.
  - Information about the **message** and its **sender** like client type, the formats client supports, format type of the message body, cache settings for the response, and more.
    - E.g. "*Content-Type: application/json*"
- **Request Body** is the actual message content (JSON, XML)



Anatomy of an HTTP Request



GET:

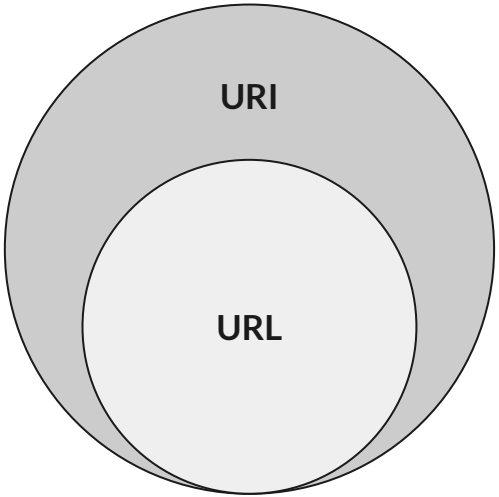
```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org, Accept: text/html,application/xhtml+xml,application/xml; …,
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …,
Accept-Encoding: gzip,deflate,sdch, Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

POST:

```
POST http://MyService/Person/ HTTP/1.1
Host: MyService, Content-Type: text/xml; charset=utf-8, Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person><ID>1</ID><Name>M Vaqqas</Name>
<Email>m.vaqqas@gmail.com</Email><Country>India</Country></Person>
```

What’s the difference between an URI and an URL

What’s the difference between an URI and an URL

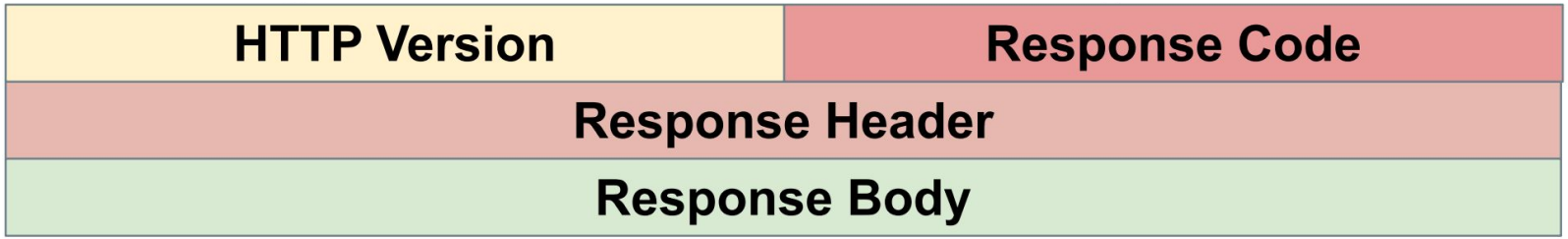


URI (Uniform Resource Identifier) Is just the *identifier* of the resource (e.g. `/users/123`)

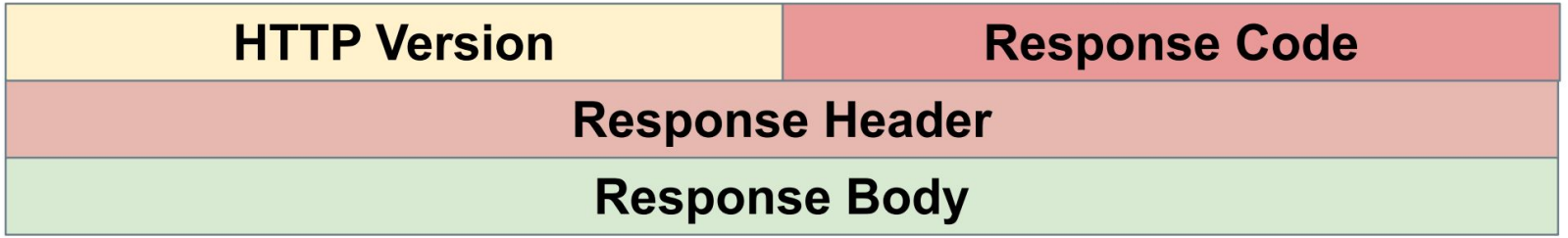
URL (Uniform Resource Locator) is a specific type of URI that not only identifies a resource but also provides the resource *location* (e.g., its network "location"). Essentially, all URLs are URIs, but not all URIs are URLs. (e.g. `https://api.example.com/users/123`)

Anatomy of an HTTP Response

- **Response code** contains request status. 3-digit HTTP status code from a pre-defined list.
- **Response Header** contains metadata and settings about the response message.
- **Response Body** contains the requested resource or data - if the request was successful. Can be as a JSON, XML or other formats such as an HTML or just text.



Anatomy of an HTTP Response



HTTP/1.1 200 OK

Date: Sat, 23 Aug 2014 18:31:04 GMT, Server: Apache/2, Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT, Accept-Ranges: bytes, Content-Length: 32859, Cache-Control: max-age=21600, must-revalidate, Expires: Sun, 24 Aug 2014 00:31:04 GMT, Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns='http://www.w3.org/1999/xhtml'> <head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head> <body> …

Response codes

Status code	Meaning
200 OK	Request was successful.
301 Moved Permanently	For SEO purposes when a page has been moved and all link equity should be passed through.
401 Unauthorized	Server requires authentication.
403 Forbidden	Client authenticated but does not have permissions to view resource.
404 Not Found	Page not found because no search results or may be out of stock.
500 Internal Server Error	Server side error. Usually due to bugs and exceptions thrown on the server side code.
503 Server Unavailable	Server side error. Usually due to a platform hosting, overload and maintenance issue.

Response codes

Status code	Meaning
200 OK	Request was successful.
201 Created	The request has been fulfilled and resulted in a new resource being created.
204 No Content	The request was successful, but there is no representation to return (i.e., the response is empty).
301 Moved Permanently	For SEO purposes when a page has been moved and all link equity should be passed through.
400 Bad Request	The request could not be understood or was missing required parameters.
401 Unauthorized	Server requires authentication.
403 Forbidden	Client authenticated but does not have permissions to view resource.
404 Not Found	Page not found because no search results or may be out of stock.
500 Internal Server Error	Server side error. Usually due to bugs and exceptions thrown on the server side code.
503 Server Unavailable	Server side error. Usually due to a platform hosting, overload and maintenance issue.

Response cc

Status code	
200 OK	
201 Created	g created.
204 No Content	n (i.e., the response
301 Moved Permanentl	y should be passed
400 Bad Request	meters.
401 Unauthorized	
403 Forbidden	ce.
404 Not Found	
500 Internal Server Erro	e server side code.
503 Server Unavailable	maintenance issue.

LIÈGE université

Looking at more endpoints examples

URL endpoint resource	GET	POST	PUT	DELETE
/surveys	Retrieve all surveys	Create a new survey	Bulk update surveys (not advised)	Remove all surveys (not advised)
/surveys/123	Retrieve the details for survey 123	Error	Update the details of survey 123 if it exists	Remove survey 123
/surveys/123/responses	Retrieve all responses for survey 123	Create a new response for survey 123	Bulk update responses for survey 123 (not advised)	Remove all responses for survey 123 (not advised)
/responses/42	Retrieve the details for response 42	Error	Update the details of response 42 if it exists	Remove response 42

https://mannhowie.com/rest-api

Elements of Web Architecture

- **Firewalls** decide which HTTP messages get out, and which get in.
  - These components enforce web *security*.
- **Routers** decide where to send HTTP messages.
  - These components manage web *scalability*.
- **Caches** decide if saved copy of resource used.
  - These components increase web *performance*.

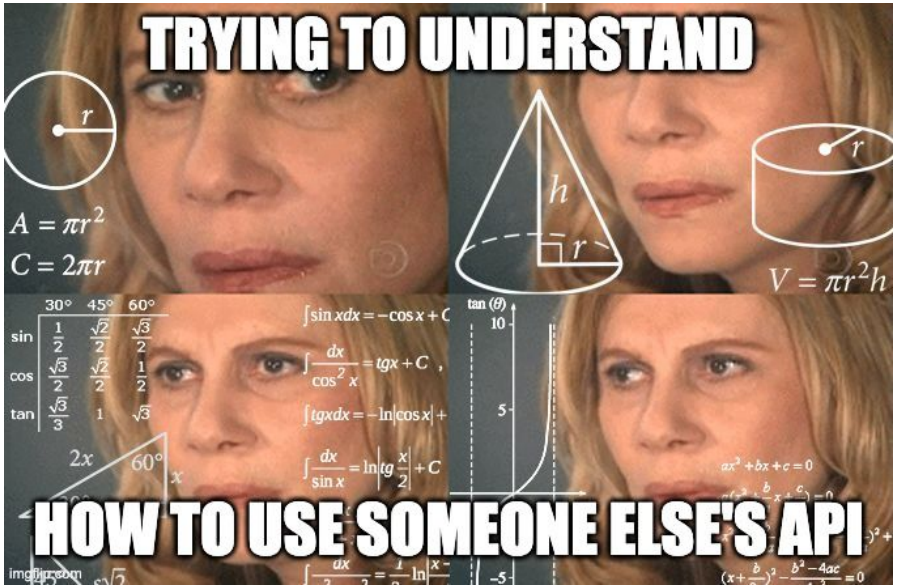
Firewalls

- Firewall decides whether a HTTP message can pass through
- **Important:** All decisions based purely on the **HTTP header**. The firewall **never looks in the payload**
- This message is rejected

Best practices for RESTful APIs

- Avoid using cryptic resource names
- Nouns not verbs when naming resources
  - `GET /users not GET /get_users`
- Plural nouns
  - `GET /users/{userId} not GET /user/{userID}`
- Dashes in URIs for resources and path parameters but use underscores for query parameters
  - `GET /admin-users/?find_desc=super`
- Return appropriate HTTP and informative messages to the user

OpenAPI & Swagger



https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

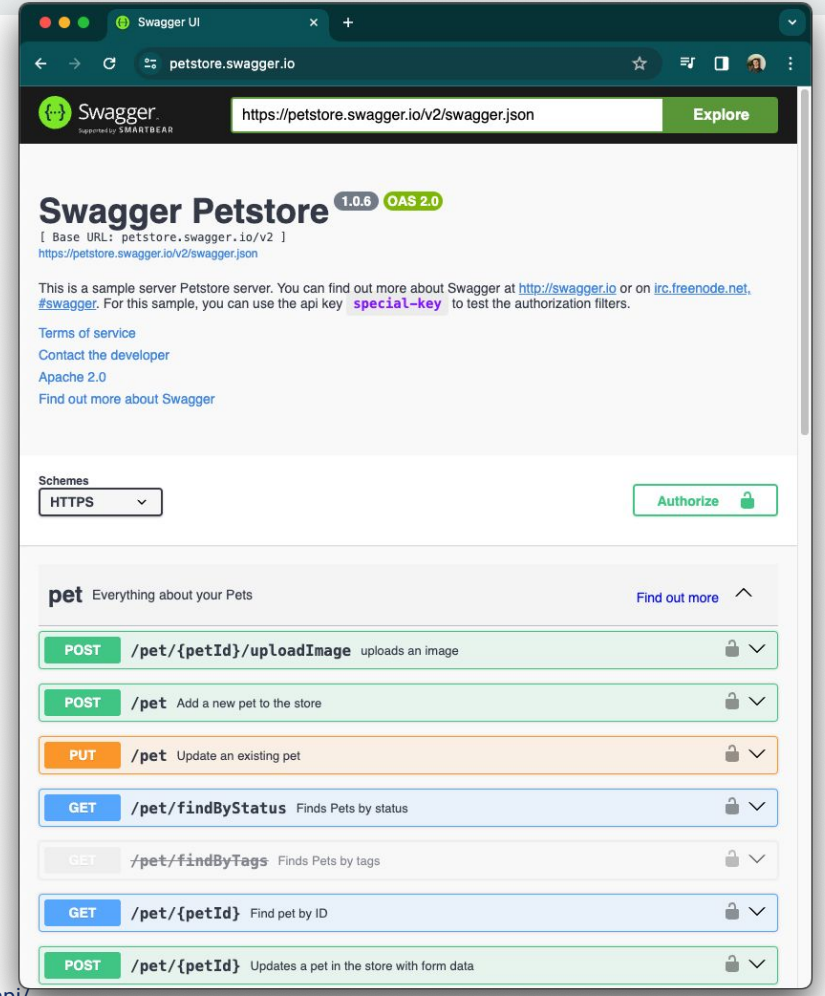
OpenAPI & Swagger

Tools for designing, documenting, and consuming APIs effectively.

OpenAPI Specification (OAS): standardized, language-agnostic format for defining RESTful APIs. It provides a structured way to describe:

- Endpoints (URLs)
- Request and response formats
- Authentication methods
- Parameters and headers
- Response codes

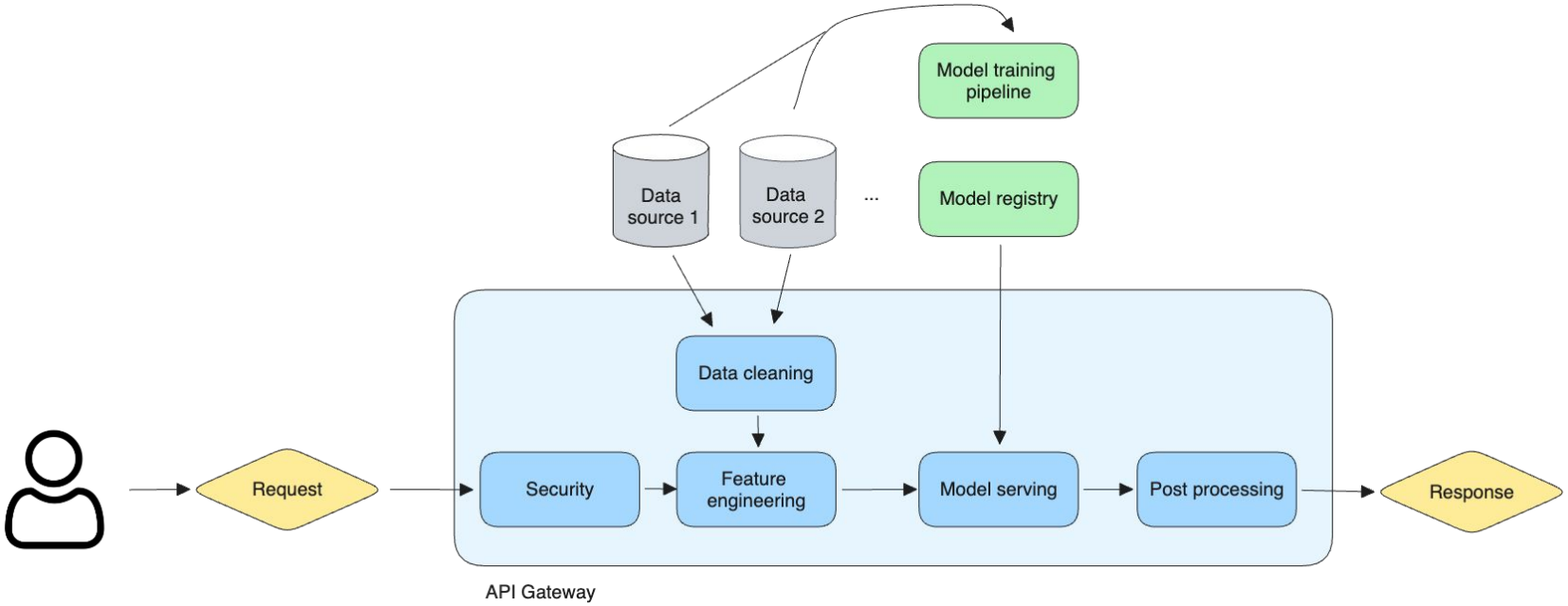
Swagger: suite of tools that work with OpenAPI specifications to create, visualize, and interact with APIs.



https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

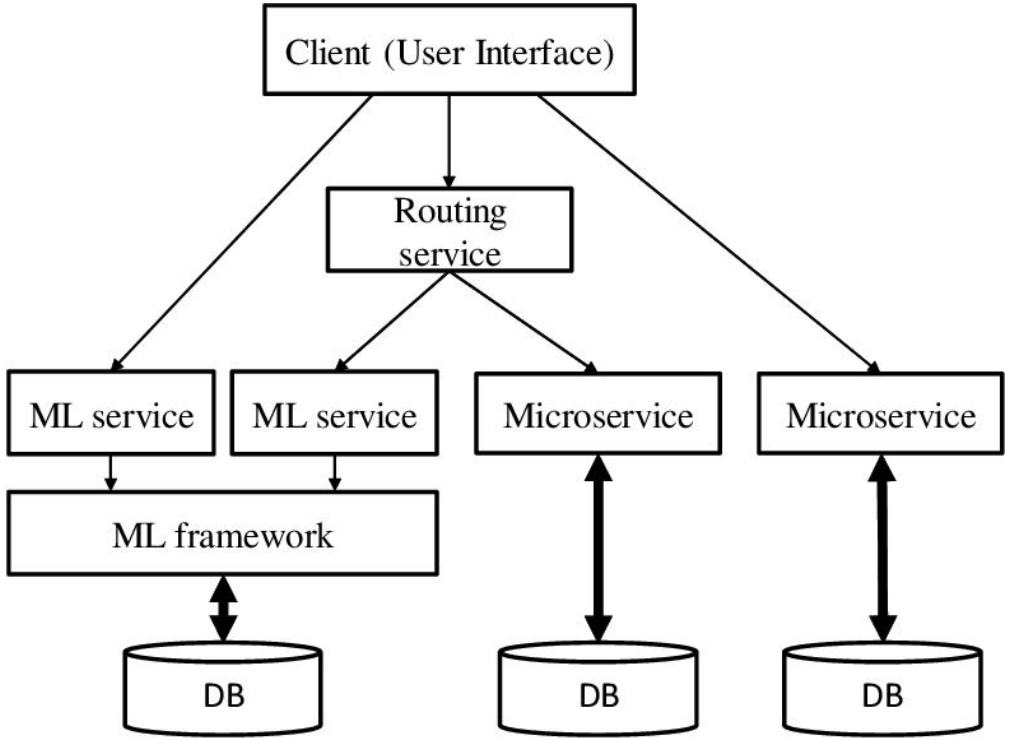
ML API: Might include different logic parts

More components than purely the model serving



You might implement many different independent services:

Microservices



Washizaki, Hironori et al. (2022). **Software-Engineering Design Patterns for Machine Learning Applications**

RPC

RPC is less frequently used but still relevant

RPC (Remote Procedure Calls).

- The client completes a function (aka method or procedure) on the server, and the server sends the output back to the client.
- “**Action centric**”, when a complex action needs to happen on the server side.
- Can be stateful, which allows you to interact with an object on the server side as if you could access it locally.

“While REST web APIs are the norm today, Remote Procedure Call (RPC) hasn't disappeared. A REST API is typically used in applications as it is easier for developers to understand and implement. However, RPC still exists and is used when it suits the use case better.”

https://aws.amazon.com/compare/the-difference-between-rpc-and-rest

Comparing REST & RPC

	REST	RPC

Request	`GET /users/123 HTTP/1.1`
	`Host: example.com`
		`{`
		`  "jsonrpc": "2.0",`
		`  "method": "getUser",`
		`  "params": { "id": 123 },`
		`  "id": 1`
		`}`

Response	`{`
	`  "id": 123,`
	`  "name": "Alice",`
	`  "email": "alice@example.com"`
	`}`
		`{`
		`  "jsonrpc": "2.0",`
		`  "result": {`
		`  "id": 123,`
		`  "name": "Alice",`
		`  "email": "alice@example.com"`
		`  },`
		`  "id": 1`
		`}`

Comparing REST & RPC

	REST	RPC

Request	`PUT /users/123 HTTP/1.1`
	`Host: example.com`
	`Content-Type: application/json`
	
	`{`
	`  "email": "newalice@example.com"`
	`}`
		`{`
		`  "jsonrpc": "2.0",`
		`  "method": "updateUserEmail",`
		`  "params": { "id": 123, "email": "newalice@example.com" },`
		`  "id": 2`
		`}`

Response	`{`
	`  "id": 123,`
	`  "name": "Alice",`
	`  "email": "newalice@example.com"`
	`}`
		`{`
		`  "jsonrpc": "2.0",`
		`  "result": { "success": true },`
		`  "id": 2`
		`}`

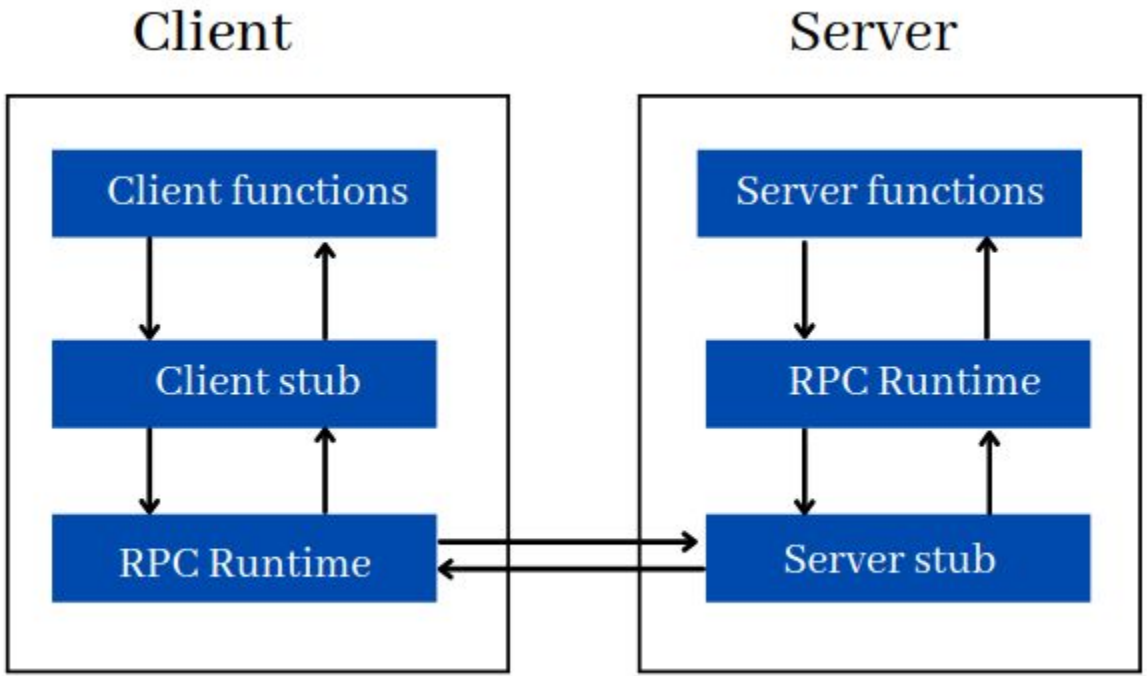
An example of an RPC call

```
GET /getMovie/12 HTTP/1.1
Host: api.moviedb.com
Content-Type: application/json
```

```
Python
/* Function definition */
def getMovie(id) {
  // ...
}

/* Function call */
getMovie(id)
```


https://iq.opengenus.org/rpc-vs-rest/

Breaking down an RPC process



https://iq.opengenus.org/rpc-vs-rest/

Stateless vs Stateful

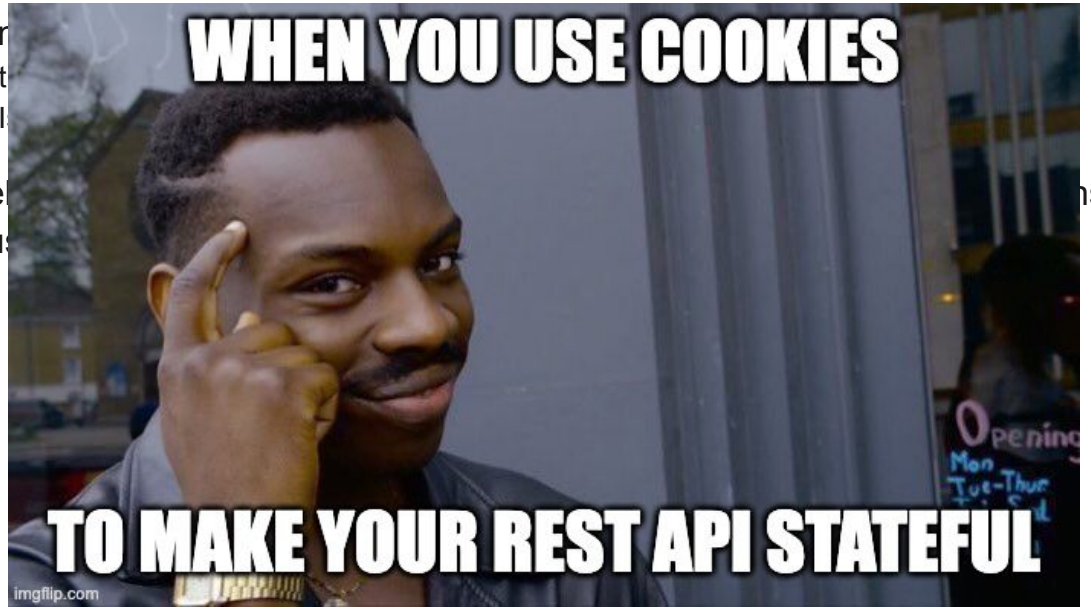
- **Stateless API** does not maintain information about previous requests
  - 👉 REST APIs
- **Stateful API** maintains information about previous requests
  - For most complex systems
  - RPC APIs *can* be stateful

Stateful APIs are helpful for complex workflows where previous interactions need to be considered, while stateless APIs are useful for simpler and more scalable systems.

Stateless vs Stateful

- **Stateless API** does not maintain information about previous requests
  - 👉 REST APIs
- **Stateful API** r
  - For most
  - RPC API

Stateful APIs are hel nsidered, while stateless APIs are u



RPC principles

Remote invocation: An RPC call is made by a client to a function on the remote server as if it were called locally to the client.

Passing parameters: The client typically sends parameters to a server function, much the same as a local function.

Stubs: Function stubs exist on both the client and the server. They are responsible for marshalling and unmarshalling (aka serializing and deserializing) the function arguments and returns.

IDL (Interface Definition Language): Language used to describe the interface that an RPC service provides. It is a way to define the functions, methods, and data types that are available for remote invocation, ensuring that both the client and server understand the format and types of data that will be sent and received.

gRPC

- A more modern version of RPC developed by Google
- Includes key improvements such as
  - Protocol Buffer (protobuf) as IDL
  - use of HTTP/2
  - Bidirectional streaming and flow control
  - Authentication

GenAI API Gateways

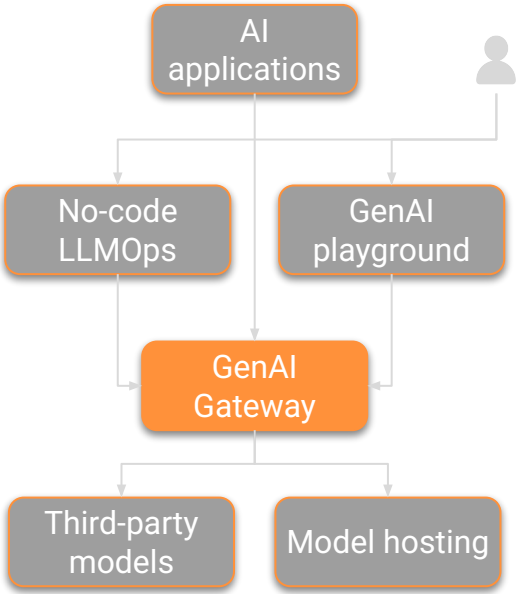
Credits where credits are due

Robbe Sneyders
Office of the CTO @ ML6

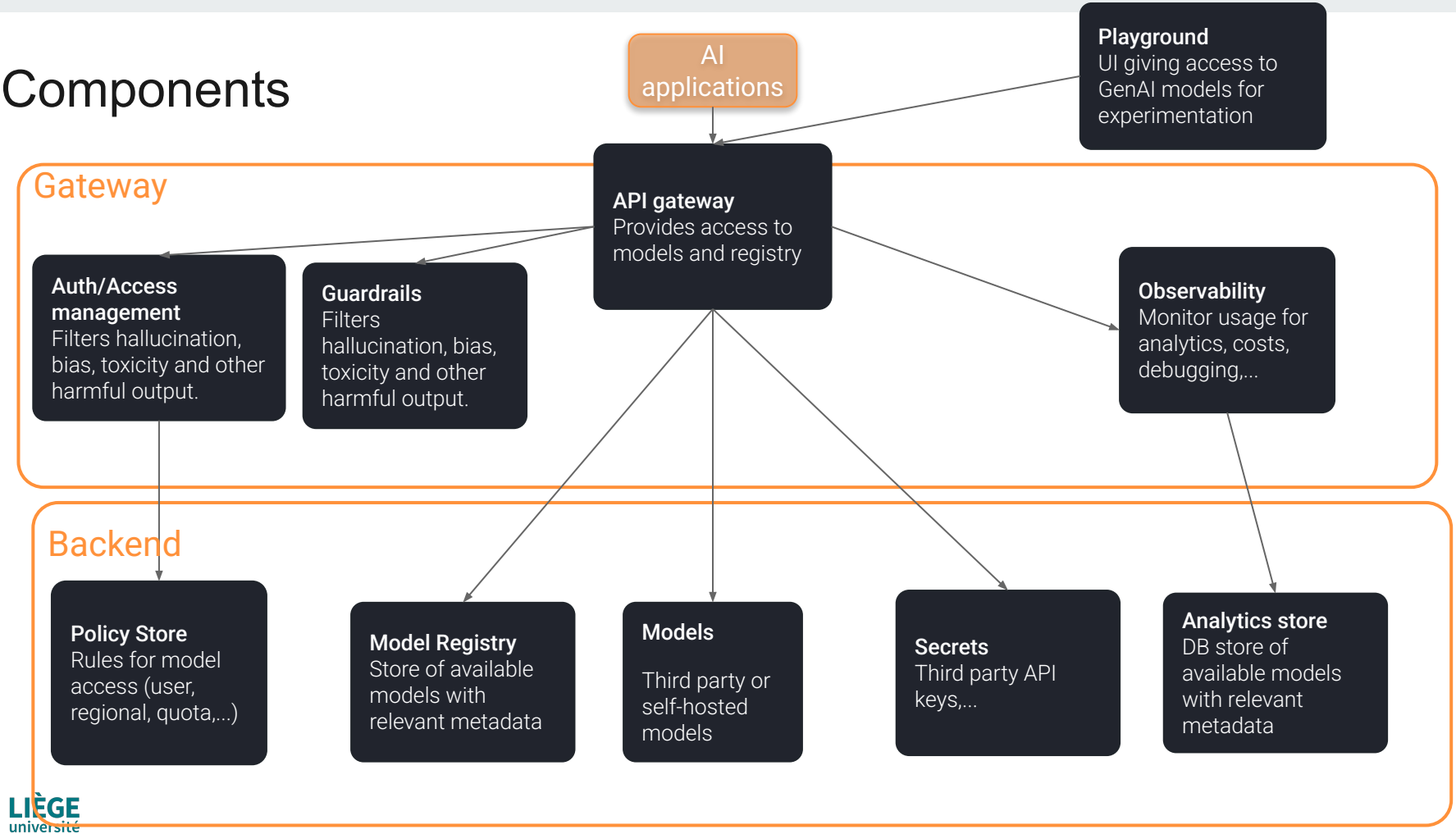
Arne Pannemans
Machine Learning Engineer

What is a GenAI Gateway?

- Central interface to manage access to GenAI models cross the organization
- Consumer AI applications send all model calls to the gateway using a unified, abstracted API
- The gateway routes all calls to the requested model, translating the requests to the selected model endpoint's specification
- The gateway monitors the traffic, manages authentication and access control, maintains, stores logs and implements any central guardrails



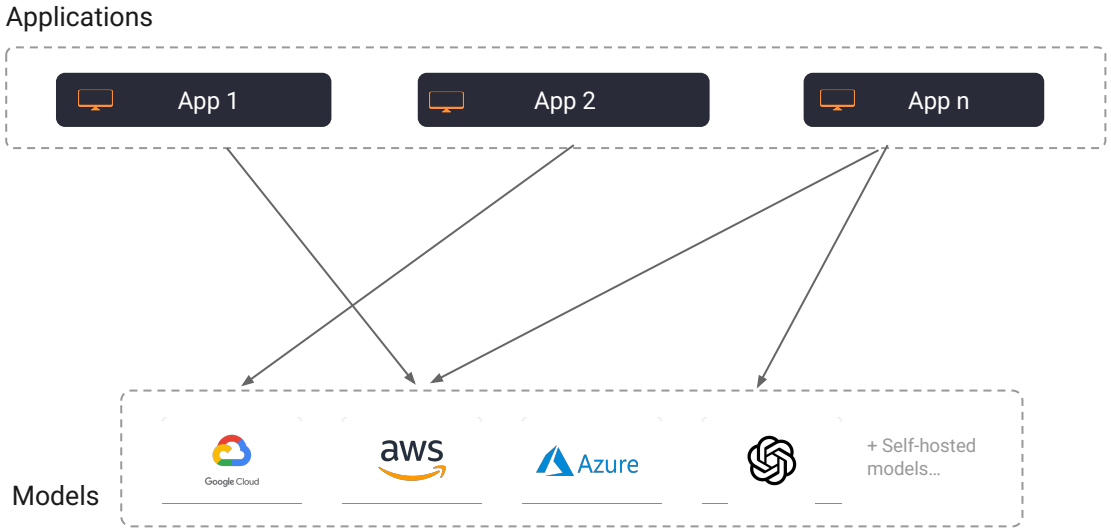
Components


AI applications
Playground
UI giving access to GenAI models for experimentation
Gateway
API gateway
Provides access to models and registry
Auth/Access management
Filters hallucination, bias, toxicity and other harmful output.
Guardrails
Filters hallucination, bias, toxicity and other harmful output.
Observability
Monitor usage for analytics, costs, debugging,...
Backend
Policy Store
Rules for model access (user, regional, quota,...)
Model Registry
Store of available models with relevant metadata
Models
Third party or self-hosted models
Secrets
Third party API keys,...
Analytics store
DB store of available models with relevant metadata

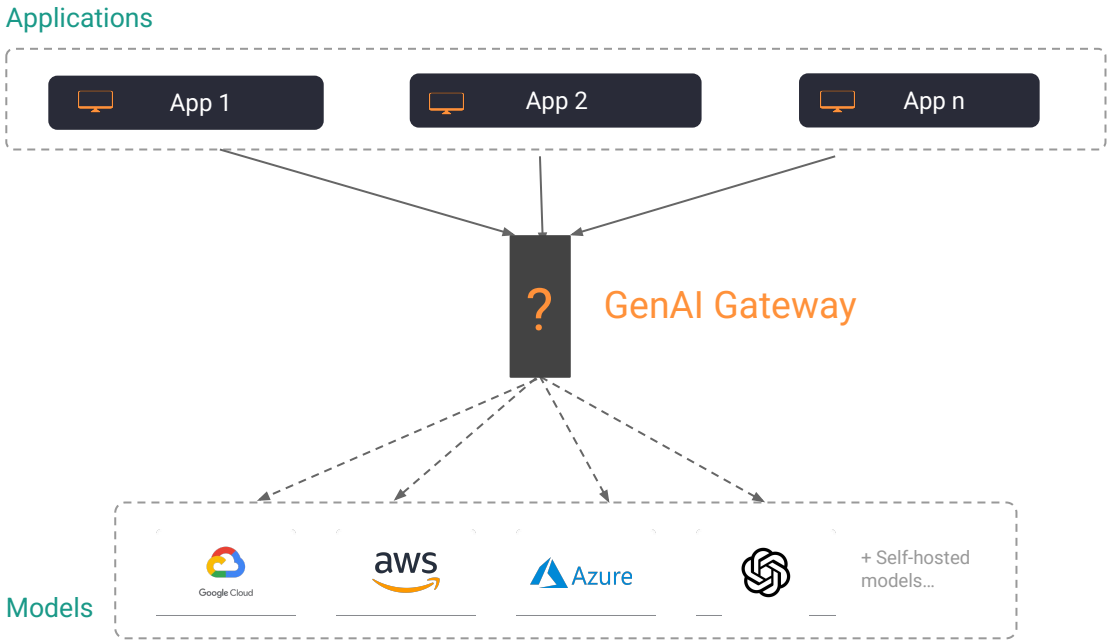
Key advantages of the GenAI Gateway

- **Model selection**: The models to be offered as part of the gateway can be chosen based on quality, cost, or contracting needs, removing this work from the individual use case teams.
- **Specific tooling**: Tooling related to LLM usage, such as caching, monitoring, credential management, etc, can be implemented once and used across use cases.
- **Cost management**: A GenAI gateway provides insights into the costs and tools to manage them. The platform centralizes licences and API keys for third party model providers, as well as infrastructure for self-hosted models.
- **Sensitive data**: Providing easy access to GenAI models (also through a playground UI) across the organization lowers the risk of sensitive data being entered into unapproved third party tools.
- **Guardrails to manage risks**: Routing GenAI through a central gateway makes it possible to implement organization-wide guardrails against hallucination, bias, toxicity and other harmful output.

Putting the API in GenAI API Gateway



Putting the API in GenAI API Gateway



Reference architecture

High level overview

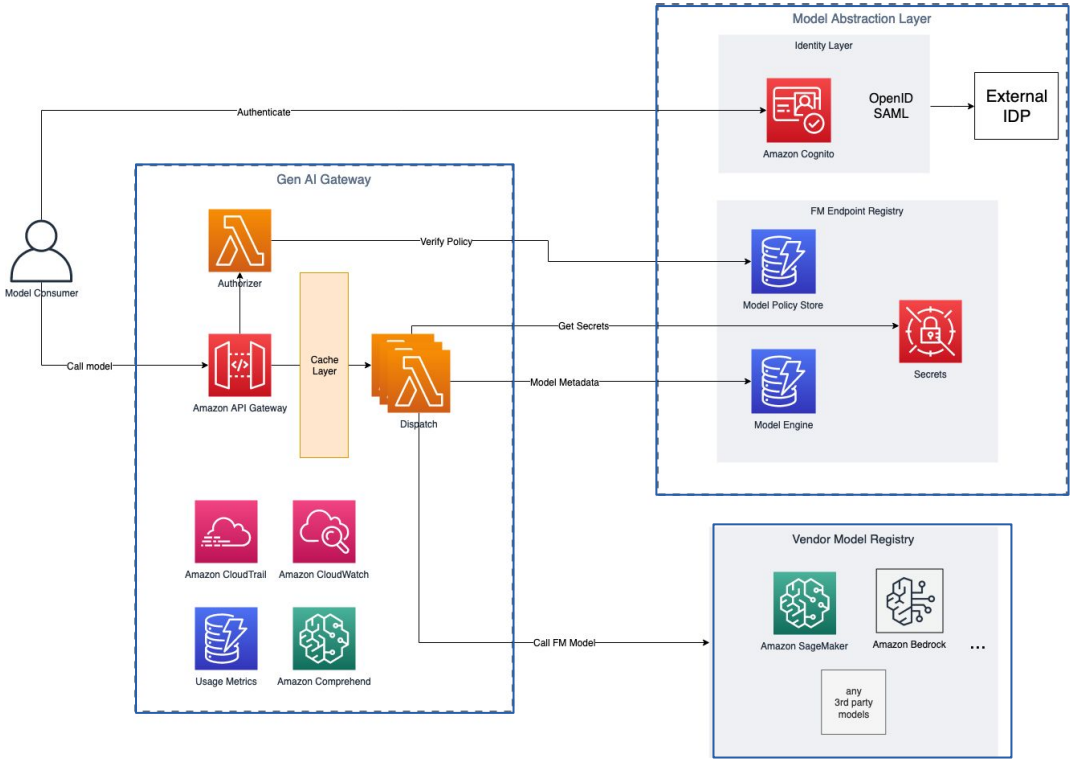
Model abstraction layer

- Model management
- Governance
- Secure access control

GenAI Gateway

- Consumer-facing API
- Request forwarding
- Monitoring, guardrails,...

Model abstraction layer



GenAI API Gateway

https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Reference architecture

Model abstraction layer (MAL)

Endpoint registry

- Registry of endpoints for models that were added to the gateway

Model policy store

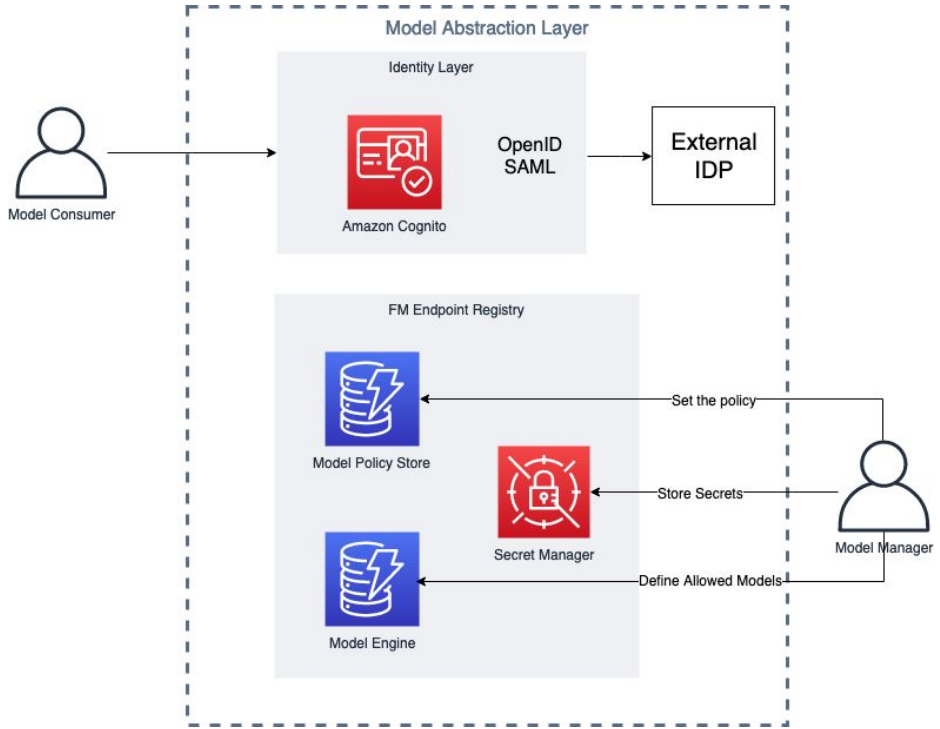
- Quantitative and qualitative rules for how models can be used (e.g. per location, content filters, …)

Identity layer

- Role-based access control (RBAC) to models. Granular user access permissions to the models.

Secret manager

- Store required keys & secrets for the endpoint to actually serve vendor models



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Reference architecture

GenAI Gateway

Unified API interface

- Simplifies interaction with all models

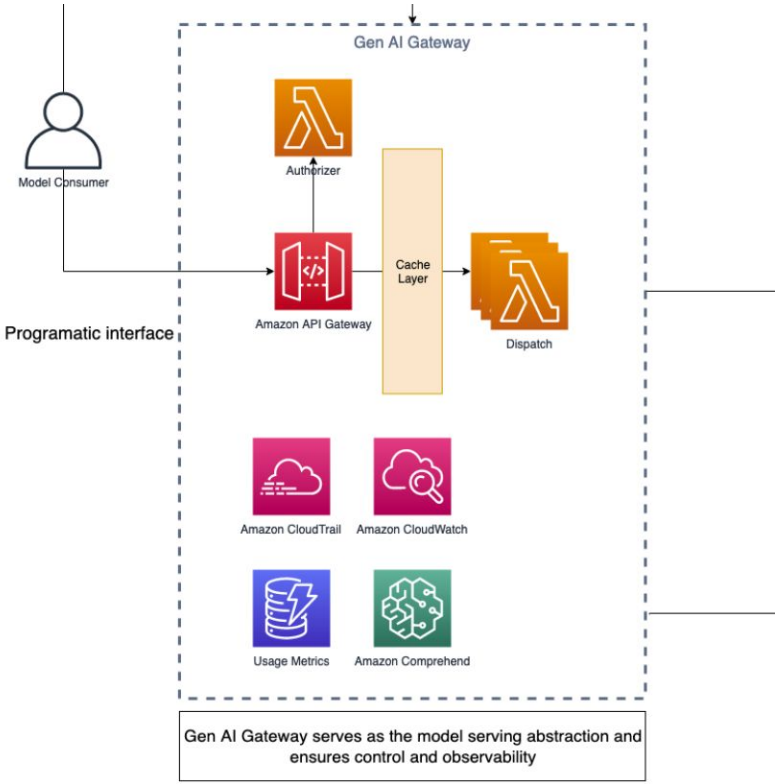
Cost control

- API quota, limits, and usage management
- Requests for dedicated hosting

Content, privacy, and responsible AI policy enforcement

- Guardrails for pre and post-processing of content
- Control of data flowing in the system

Monitoring and logging



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Reference architecture

Provides benefits attributed to general API gateways

Cost control mechanism: A robust system helps optimize resource use and manage costs. It tracks resource usage, model inference costs, and data transfer expenses. This enables cost insights, savings, and better resource allocation.

Cache: Inference can be costly, especially during testing and development. A cache layer reduces costs and speeds up responses. It also offloads the inference burden, freeing resources for other requests.

Observability: Logs capture all activities on the AI Gateway and Discovery Playground. They track user interactions, model requests, and system responses. These logs aid troubleshooting, user behavior analysis, and transparency.

Quotas, rate limits, and throttling: These controls manage AI resource usage. Quotas limit requests per user or team within a set period. Rate limits prevent excessive usage. Throttling controls request frequency to avoid system overload.

Audit trails and usage monitoring: These track and record all AI interactions. They ensure accountability and help analyze system usage.

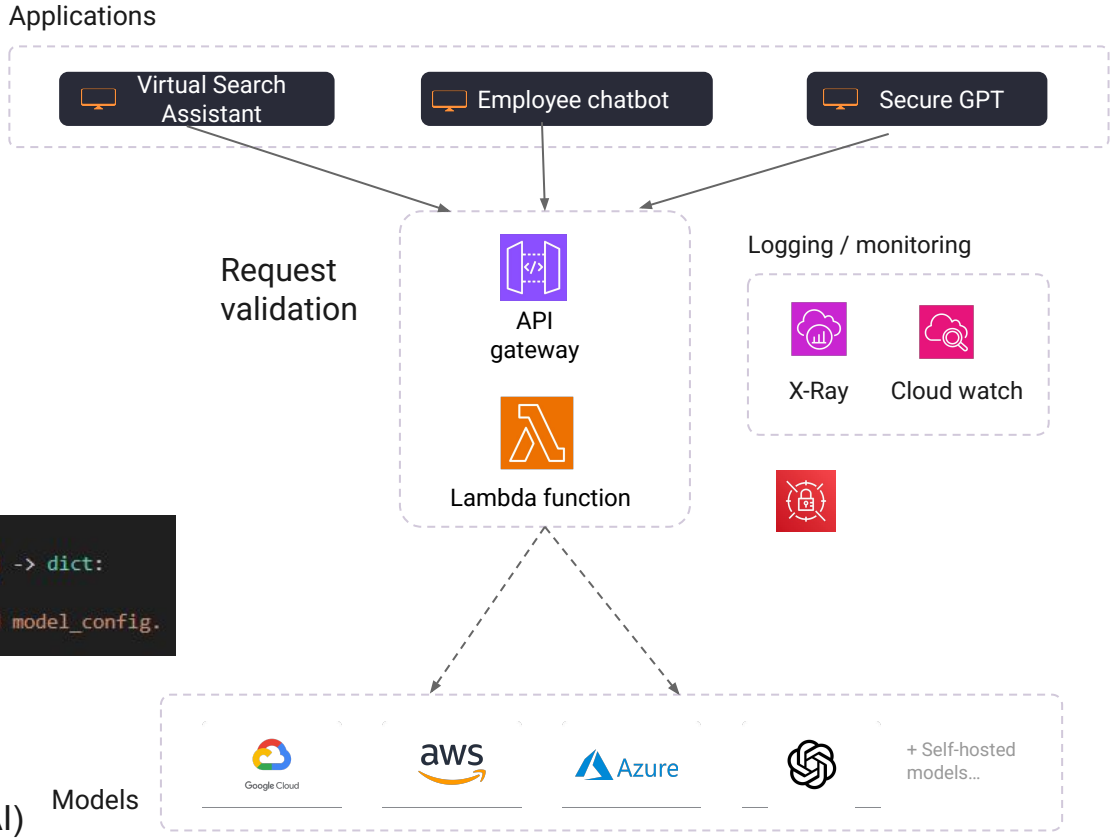
https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

GenAI gateway

A basic implementation

```python
@property
def model_info(self) -> list[dict]:
    return [
        {
            "model_id": "text-embedding-3-large",
            "description": "Most performant text embedding model from
            "documentation_url": "https://platform.openai.com/docs/gu
        },
        {
            "model_id": "gpt-4o",
            "description": "OpenAI's most performant Language model.
            "documentation_url": "https://platform.openai.com/docs/mo
        }
    ]
```

```python
def invoke(self, *, model_id: str, model_config: dict) -> dict:
    """
    Invoke an OpenAI model with the given model_id and model_config.
```



- List available models
- Call a specific model (here OpenAI)

https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

GenAI gateway

How to communicate with it?

OpenAI

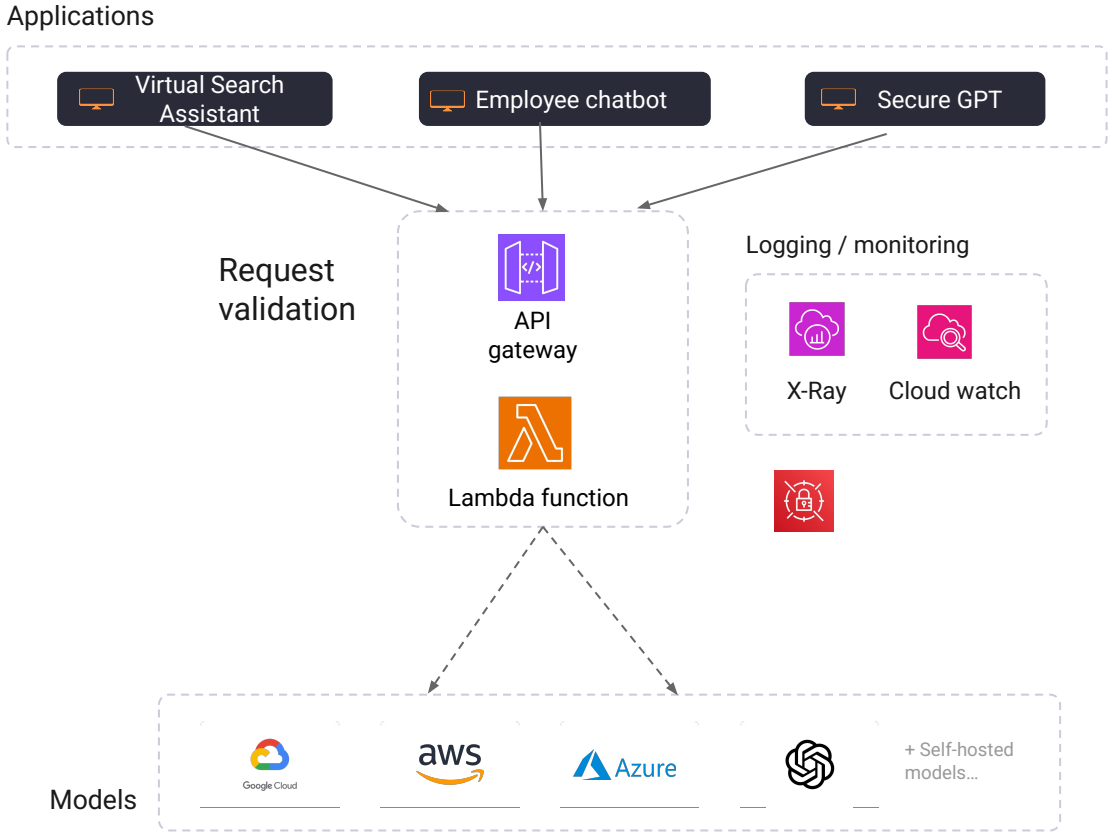
- /chat/completions
- /embeddings

AWS bedrock

- /invokeModel
- /converse

GCP

- /{model-id}:predict



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Endpoint design

A game of trade-offs

Endpoint design

Phase 1: Invoke

```
/models/{model_id}/invoke"
```

```
model_id = "cohere.embed-multilingual-v3"
model_config = {
    "texts": ["Hello, world!"],
    "input_type": "search_document",
    "truncate": "START",
}
```

User is responsible for correctly configuring model parameters.
No validation is done in the gateway.

Advantages

- Model agnostic
- Flexible
- Easy to set- up
- Low maintenance

Disadvantages

- High user responsibility
- Free-form is error prone
- Error handling and logging is hard

Endpoint design

A game of trade-offs

- *High flexibility*
- *Low maintenance*

Endpoint design

Phase 2: /converse

```
/models/{model_id}/converse
```

```
model_id="anthropic.claude-3-sonnet-20240229-v1:0"
messages = [
    {
        "role": "user",
        "content": "Hello there!"
    },
    {
        "role": "assistant",
        "content": "Hi! How can I help you today?"
```

Fixed set of expected fields.
Easy to validate in API gateway.

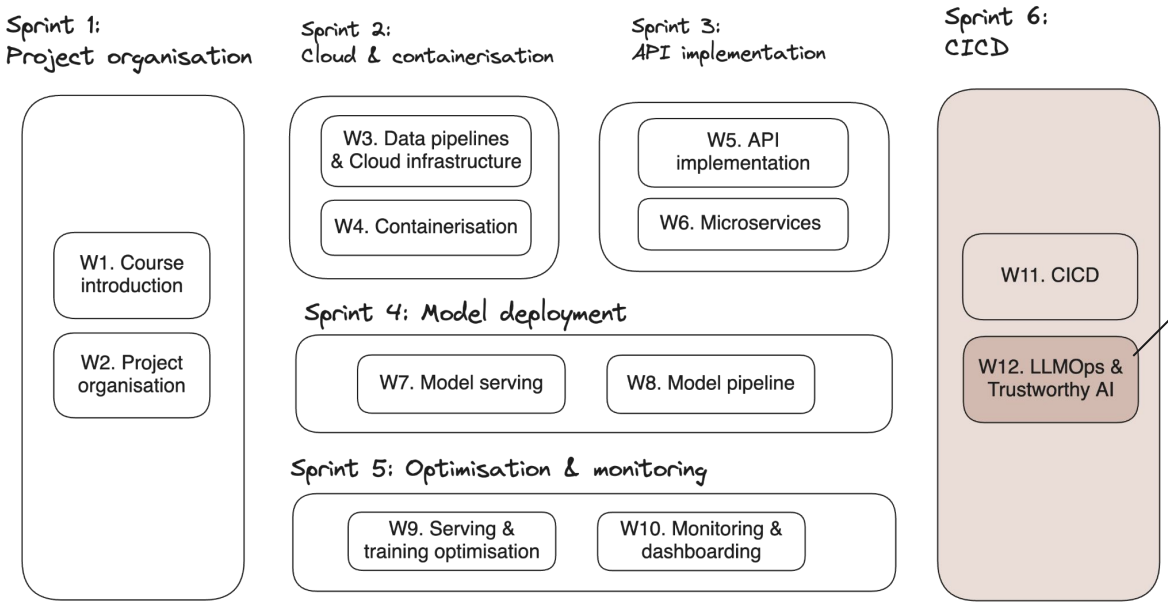
Advantages

- Consistent
- Simple
- Easy to validate input

Disadvantages

- Less flexibility
- Support only limited models
- Harder to maintain

We will come back to some aspects of it…



During the LLMOps course:

- Off-the-shelf GenAI platforms
- Guardrails

Microservice architecture

Credits where credits are due

Robbe Sneyders
Office of the CTO @ ML6

Ruwan Lambrichts
Senior ML Engineer @ ML6

Topic of this section

Defining a general software architecture for microservices in your project/organisation.

Topic of this section

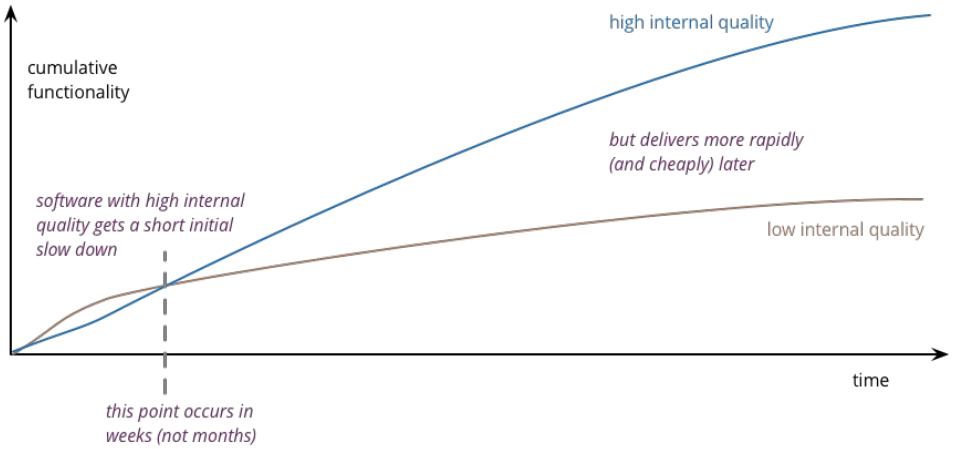
Defining a general software architecture for microservices in your project/organisation.

- The shared understanding that the expert developers have of the system design.
- The decisions you wish you could get right early in the project.

Why does software architecture matter?

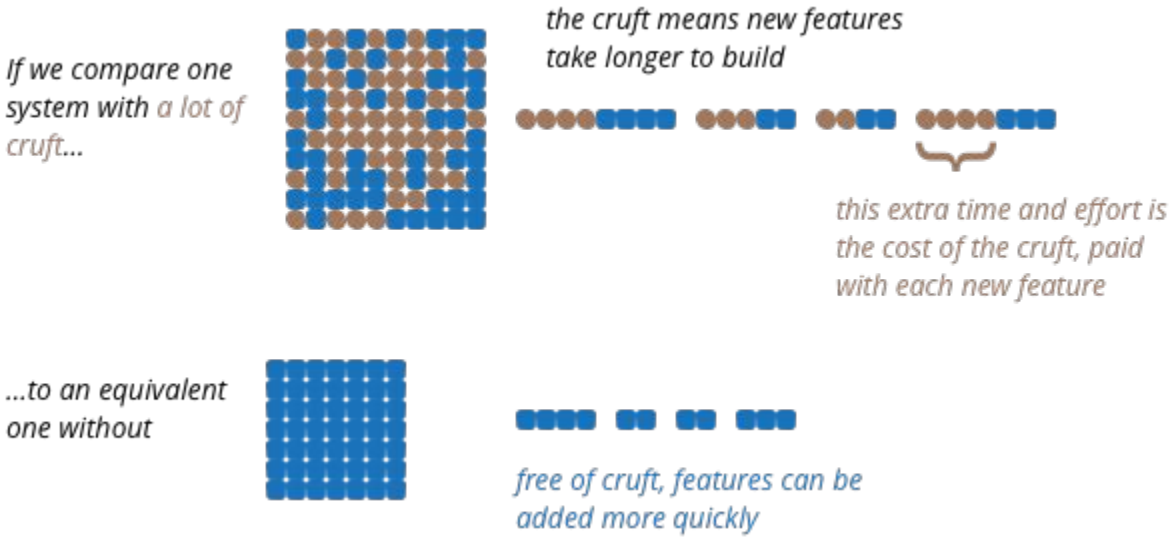
Software architecture enhances the **quality** of applications developed by your team, which is something stakeholders don't immediately perceive (and which might seem less important)

However, poor architecture is a major contributor to **cruft** - leading to code that is much harder to modify, causing features to arrive more slowly and with more defects (which the client cares about a lot)



Cruft causes features to arrive more slow and with more defects

Cruft refers to unnecessary, outdated, or redundant code and design elements that accumulate over time in software projects. It can degrade maintainability, increase technical debt, and lead to inefficiencies in system performance and development velocity.



LIÈGE université

Ideal state: Your team uses/builds boilerplates.

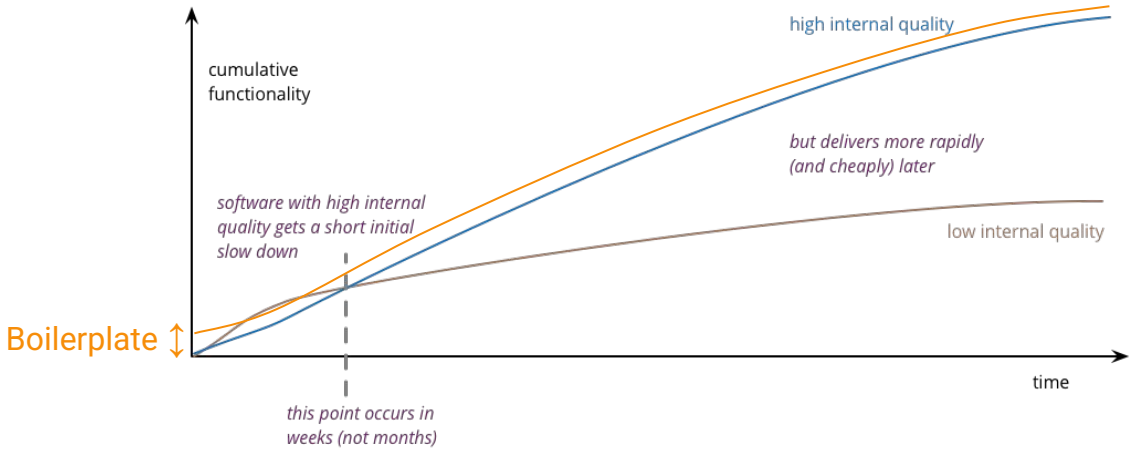
Creating a shared understanding of the microservice design .

Captured in the **microservice boilerplate**, so every project can start from the tried and tested decisions already made on previous projects.

Ideal state: Your team uses/builds boilerplates.

Take ‘correct’ decisions based on combined knowledge of developers and projects, implemented by the microservice boilerplate

- Saves time at start of project
- Projects don’t run into problems down the line, can evolve fast, and stay maintainable
- Consistent design across projects facilitates understanding of other projects



What is a microservice?

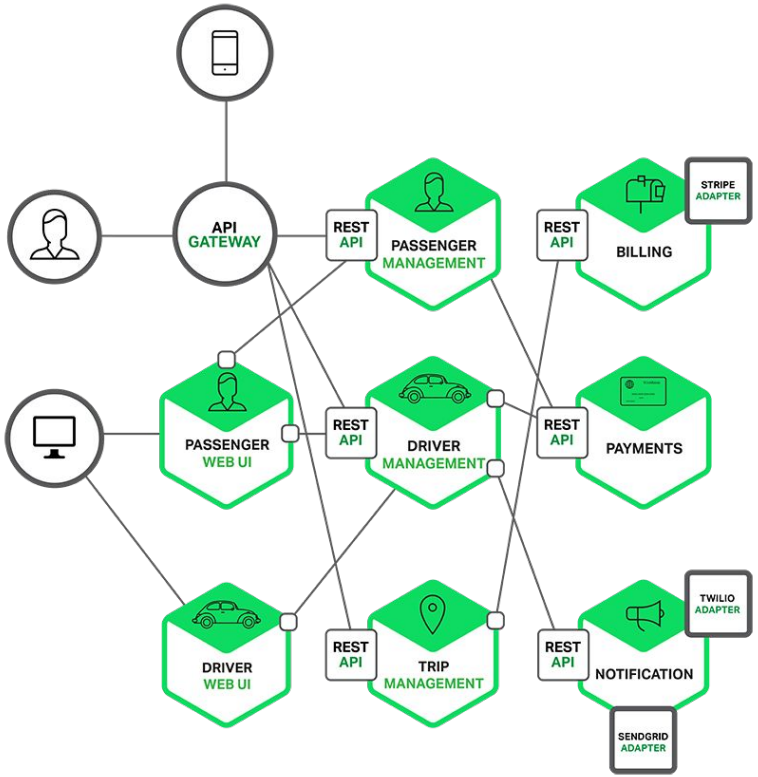
Some context

What is a microservice?

*Using a microservices approach, software is composed of **small services** that communicate over **well-defined APIs** that can be **deployed independently**. These services are owned by small autonomous teams.*

What is a microservice?

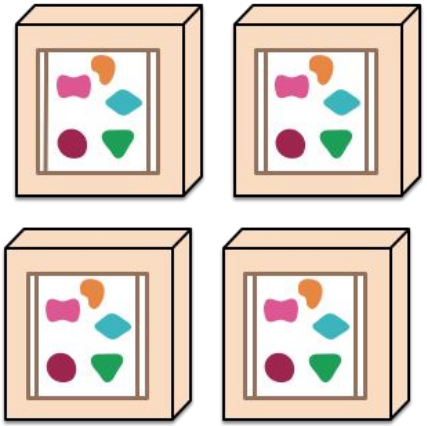
- Applying single responsibility principle at the architectural level
- As small as possible, but as big as necessary
  - “No bigger than your head”
- Advantages compared to monolithic architectures:
  - Independently deployable
  - Language, platform and technology independency between components
  - Distinct axes of scalability
  - Provides firm module boundary
  - Increased architectural flexibility
- Often integrated using REST over HTTP
  - Business domain concepts are modelled as resources



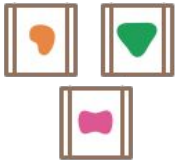
Monolith vs microservice

A monolithic application puts all its functionality into a single process...

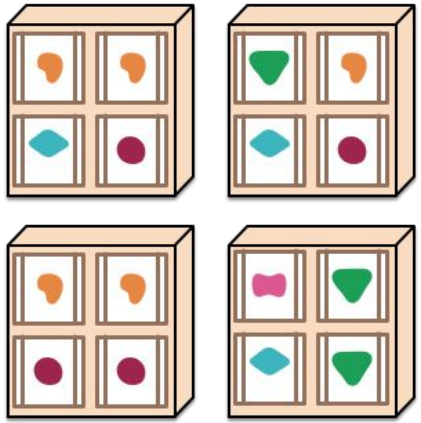
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.

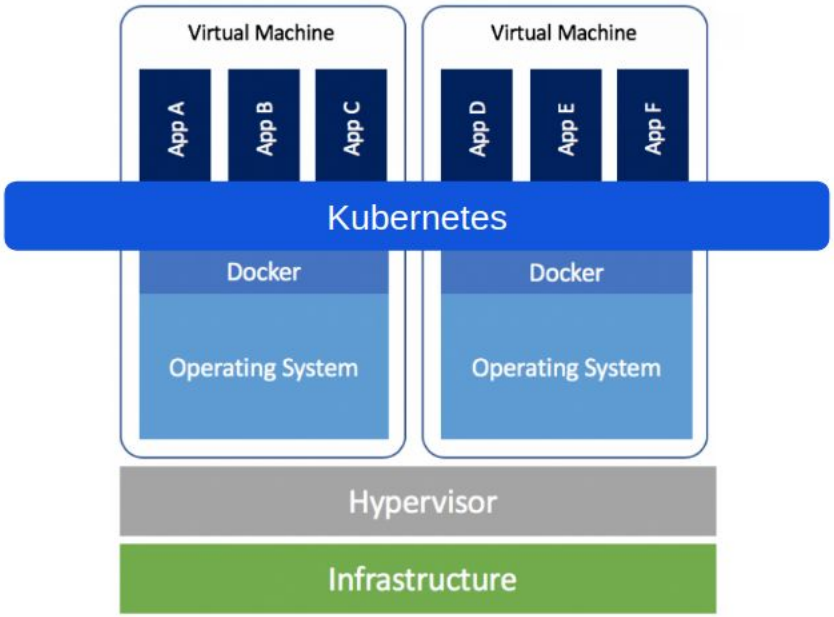


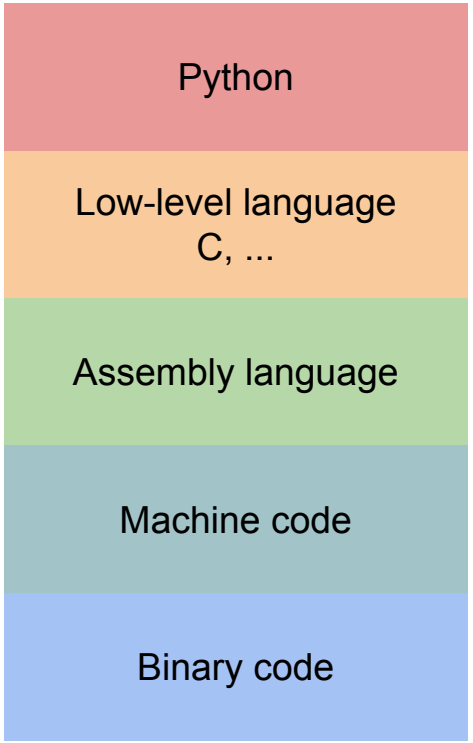
Abstractions
&
Separation of concerns

Abstraction

- Abstraction is the act of **representing essential features without including the background details** or explanations.
- Reduce complexity and allow efficient design and implementation of complex software systems.

Abstractions are everywhere

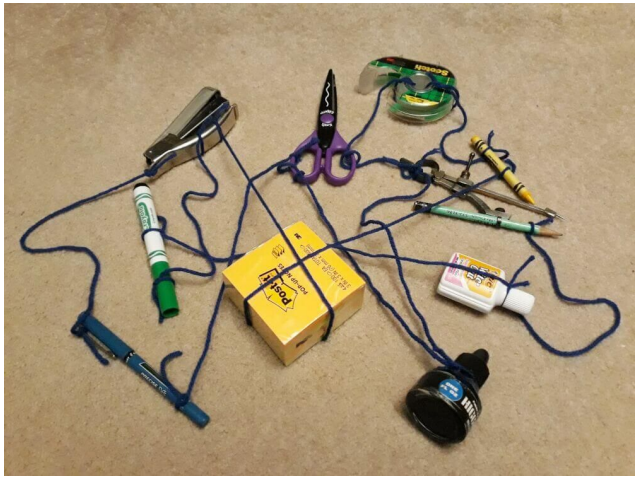




Abstractions are everywhere

Separation of concerns

- Separate code into sections that each address a separate concern
- Sections can evolve independently of other components
- Separation of concerns is a form of abstraction



Images: https://pusher.com/tutorials/clean-architecture-introduction

LIÈGE université

Interfaces

- Describes the public actions and attributes that a type of component supports and implements.
- Defines the boundary across which two components can communicate.
- Components with an identical interface are interchangable.

Interfaces

- In traditional OOP languages, an Interface defines a contract which is explicitly implemented.
- In Python, you don't have to explicitly declare an interface.

```
interface Pet {
    public Owner owner();
    public void pat();
}

class Dog implements Pet {

    public Owner owner() {
        return this.get_owner();
    }

    public void pat() {
        return love;
    }
}
```

```
class Dog:

      @property
      def owner(self):
            return self._get_owner()

      def pat(self):
            return love


dog = Dog()
try:
      dog.owner()
      dog.pat()
      print("dog is a pet")
except:
      print("dog is not a pet")
```

LIÈGE université

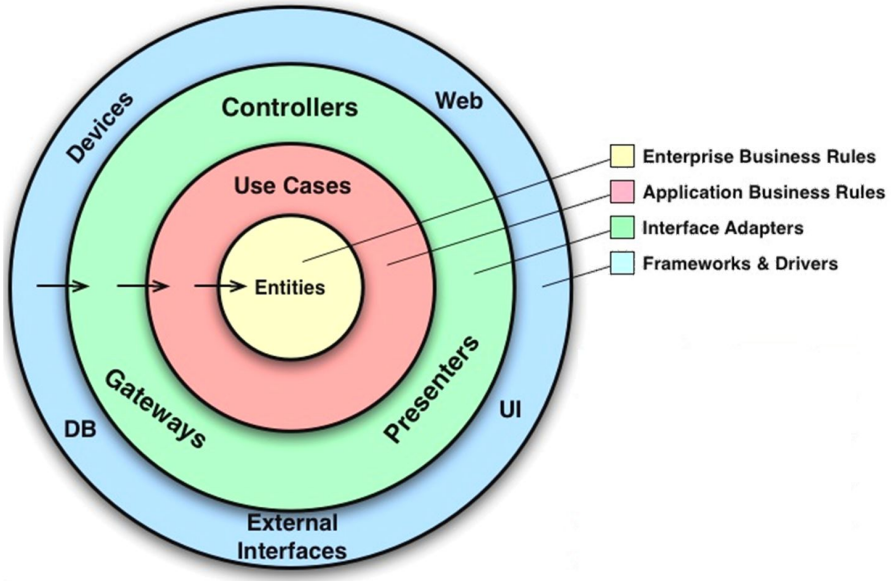
Clean architecture

The Clean architecture

By Robert C. Martin (Uncle Bob)

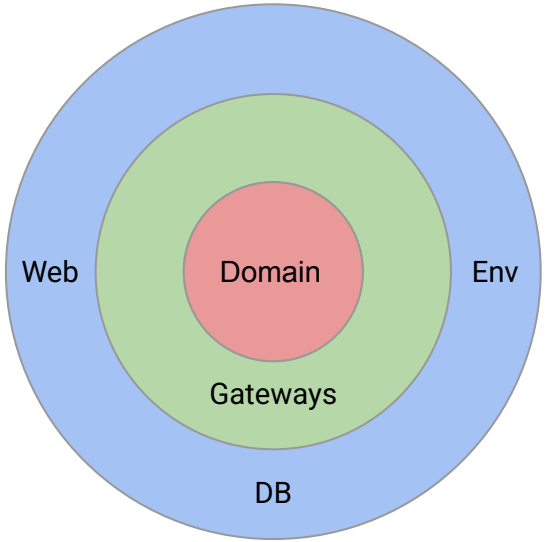
- Clear separation of of concerns between components.
- Dependency rule:
  Dependencies can only point inwards. Inner circle should not be dependent of outer circle specifics.
- Interfaces between each layer.
- Only isolated, simple data structures are passed across the boundaries.

Very OOP focused



https://en.wikipedia.org/wiki/Robert_C._Martin

The Clean architecture - tailored



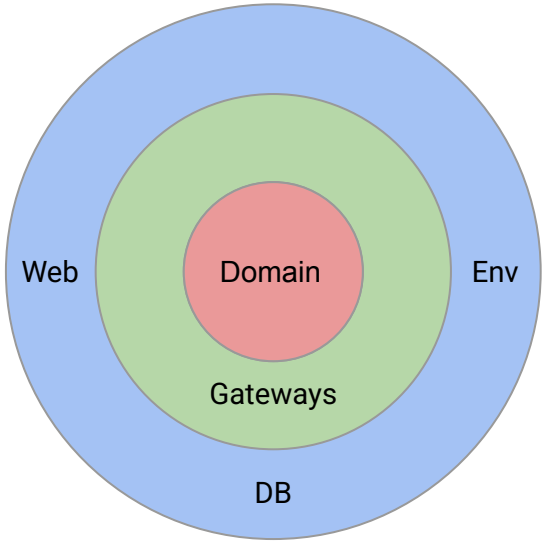
Infrastructure:
Frameworks, environment, and storage and web access.

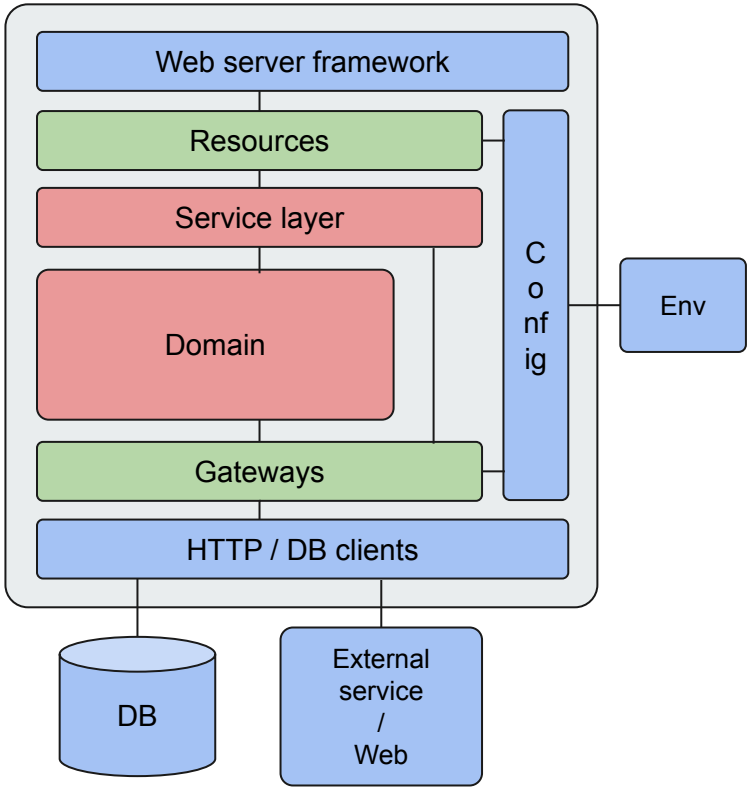
Interface adapters:
Encapsulate the infrastructure layer, and provide a Python API for the domain.

Domain:
All logic core to the microservice.

The anatomy of a microservice

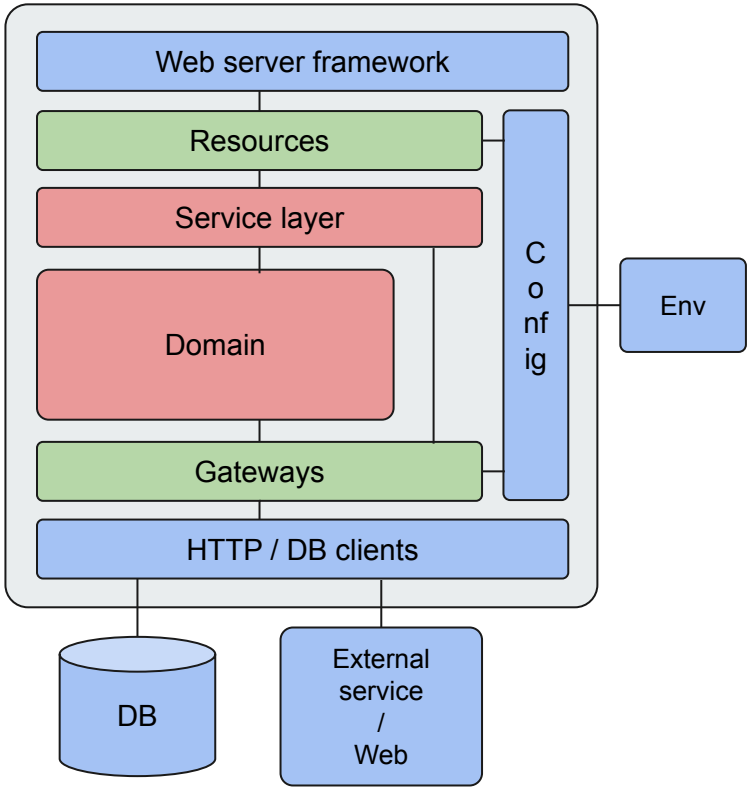
The clean anatomy of a microservice





The clean anatomy of a microservice

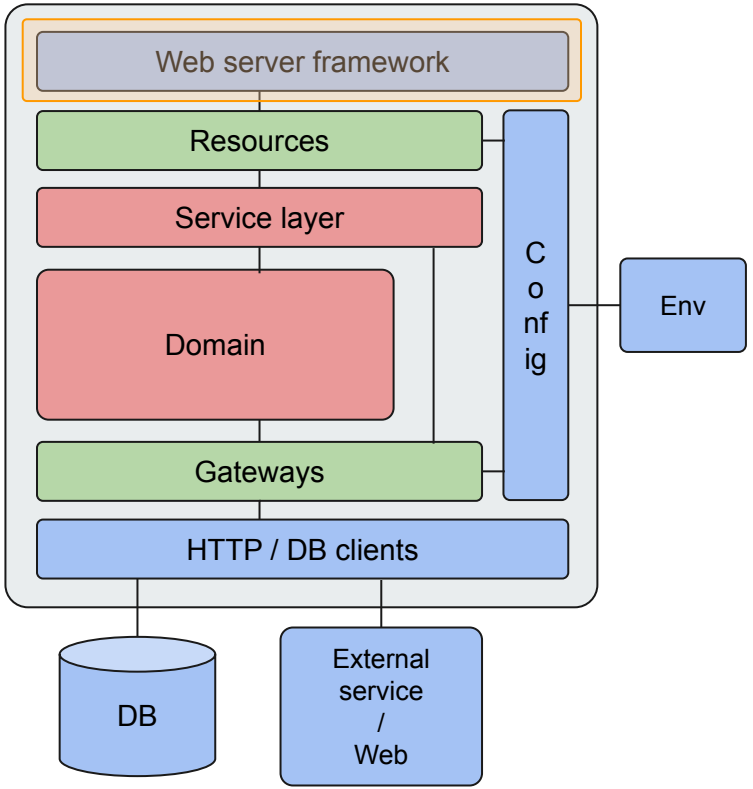
- Microservices display consistent internal structure composed of some or all of the displayed layers.
- The microservice boilerplate aims to provide minimal viable structure that is shared by all microservices.



The anatomy of a microservice

Web server framework

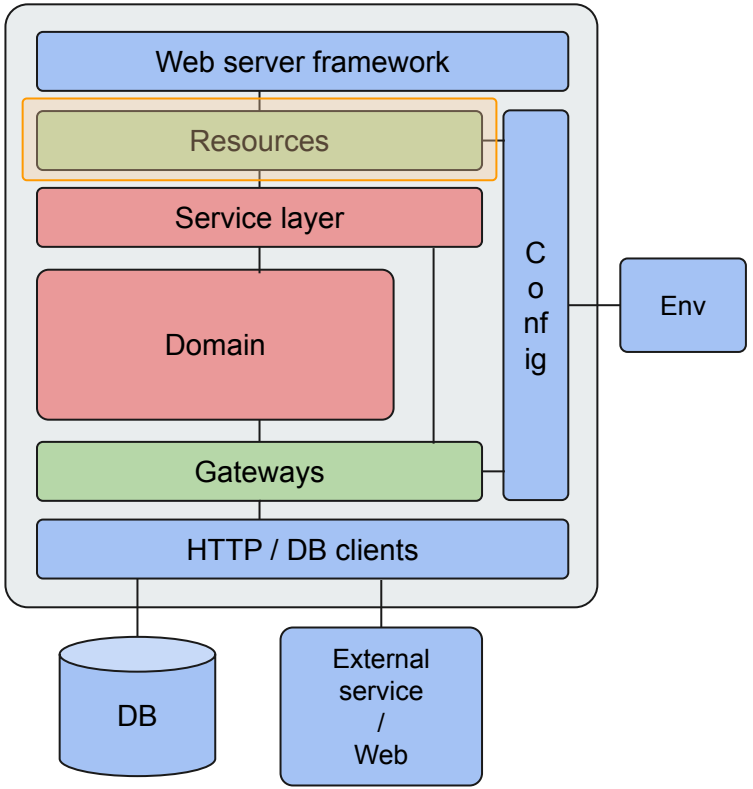
- Act as a mapping between the exposed application protocol (eg. HTTP) and Python.
- Validate requests and responses against application protocol.



The anatomy of a microservice

Resources

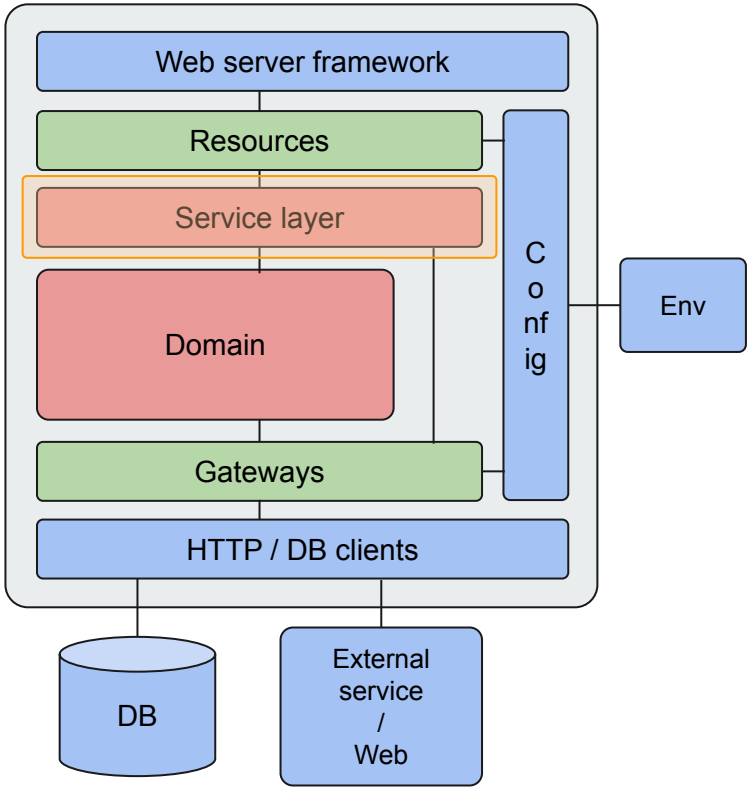
- Act as a mapping between the web server framework and the service domain.
- Thin layer for sanity checking requests and providing protocol specific responses.



The anatomy of a microservice

Service layer

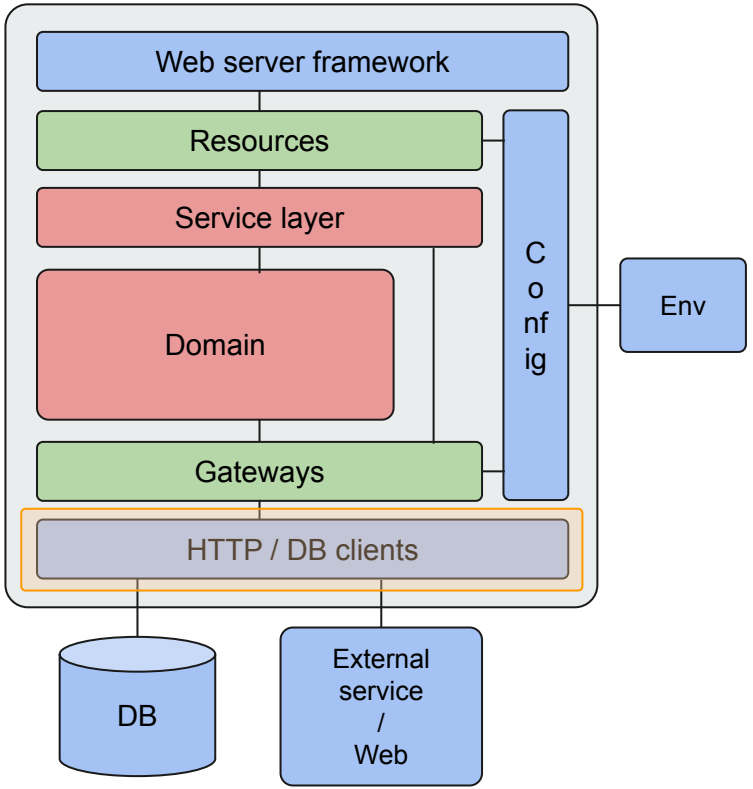
- Defines the interface of the domain with its set of available operations.
- Can be reused by multiple protocol clients; HTTP, GRPC, Queue reader, ...



The anatomy of a microservice

HTTP / DB clients

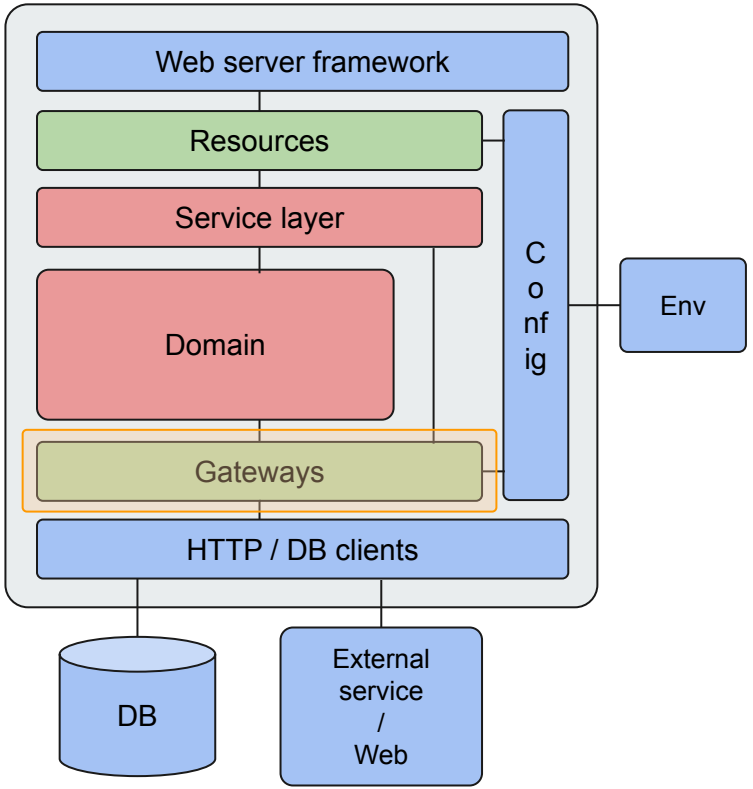
- Clients to handle connections to external datastores and services
- HTTP-client:
  Client that understands the underlying protocol to handle the request-response cycle.
  (eg. python requests)
- DB-client:
  Provide (often client-specific) python representation of data in datastores



The anatomy of a microservice

Gateways

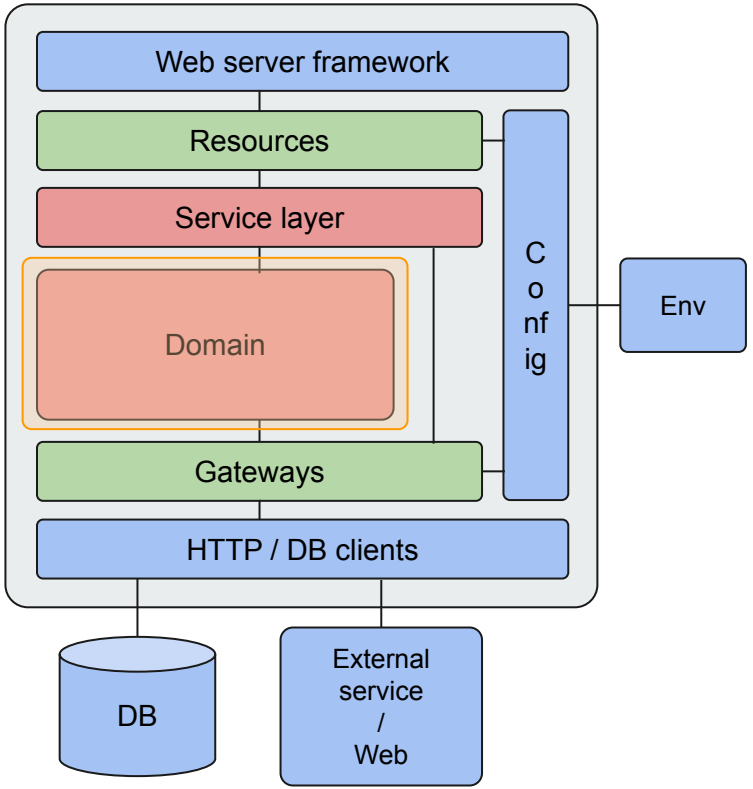
- Isolates domain from details in the database access code.
- Groups query construction code to minimize duplicate query logic.
- Encapsulates message passing with a remote service, marshalling requests and responses from and to Python.



The anatomy of a microservice

Domain

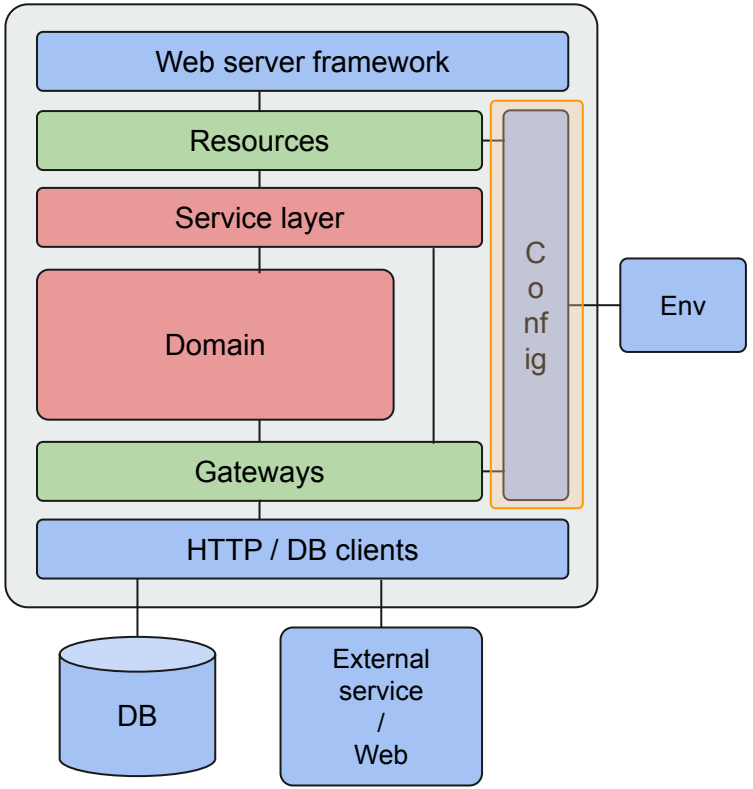
- Contains the actual service logic.
- The domain consists of pure Python with some core libraries. It is independent of any used frameworks and IO clients, and only depends on the interface of the resources and gateways.



The anatomy of a microservice

Config

- Reads and groups the configuration of the service.
- All configurable parameters of the service should be extracted as config, so they can be set and changed in the environment at run time.



Advantages of the clean anatomy

- **Independent of externals**
  When any of the external parts of the system become obsolete, like the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

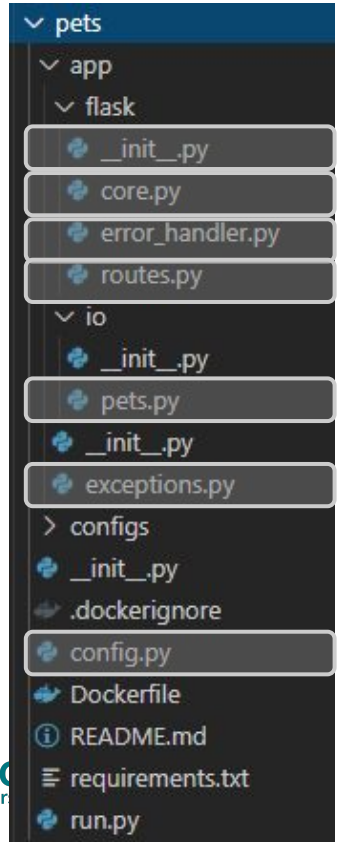
- **Scalable and maintainable**
  A modular architecture allows for scaling without an explosion in complexity. New modules can be added independently of others. Multiple developers can work in parallel on different modules.

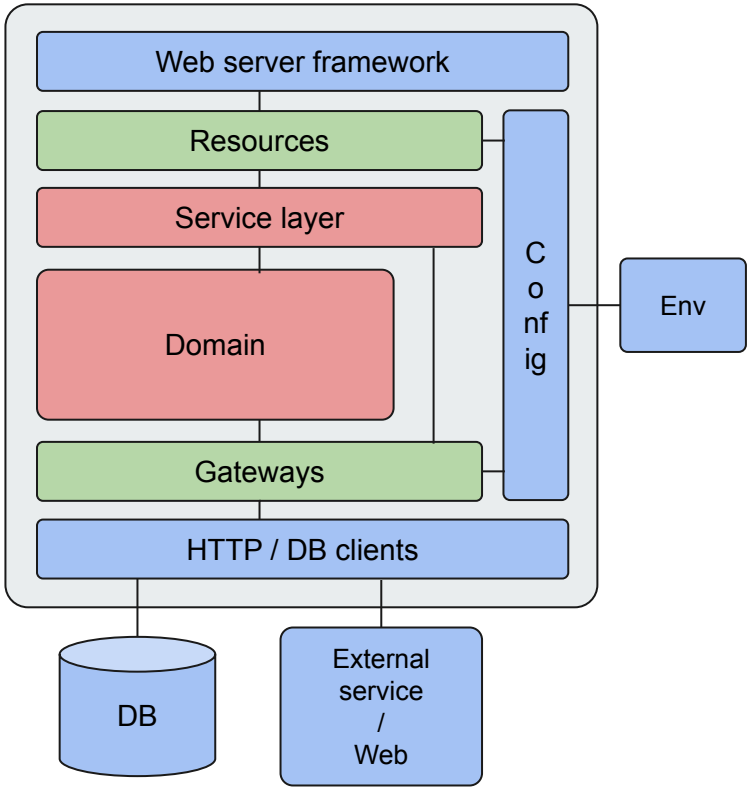
- **Deploy everywhere**
  Since the configuration is separated, the same code can be deployed in different environments without changes.

The anatomy of an example

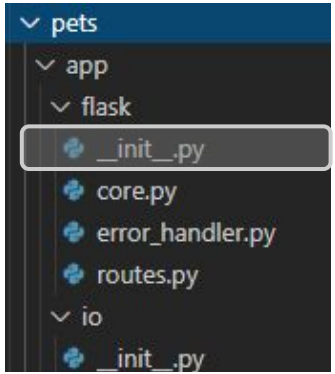
Mapping the structure on the pets example

The anatomy of a microservice: pets example

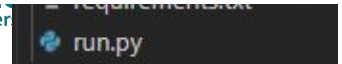


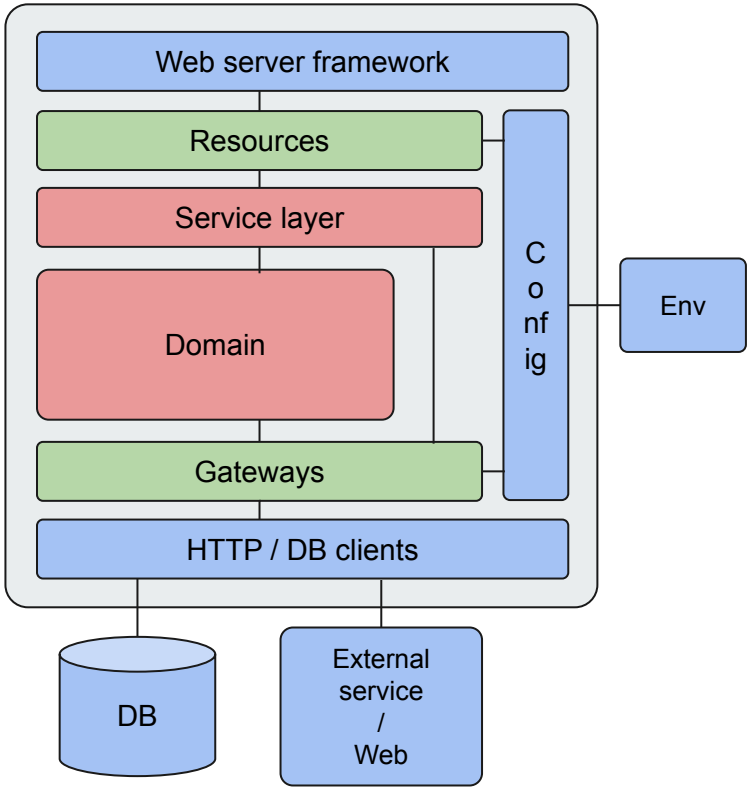


The anatomy of a microservice: pets example

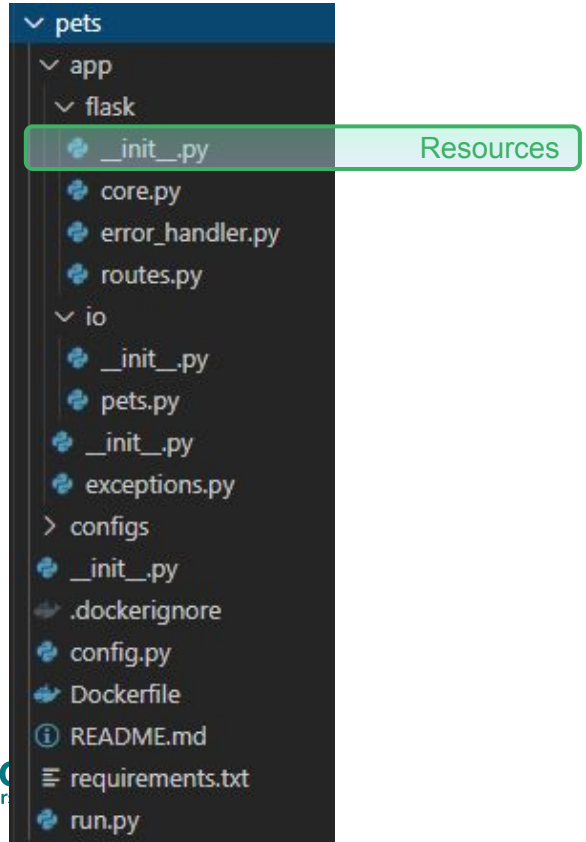


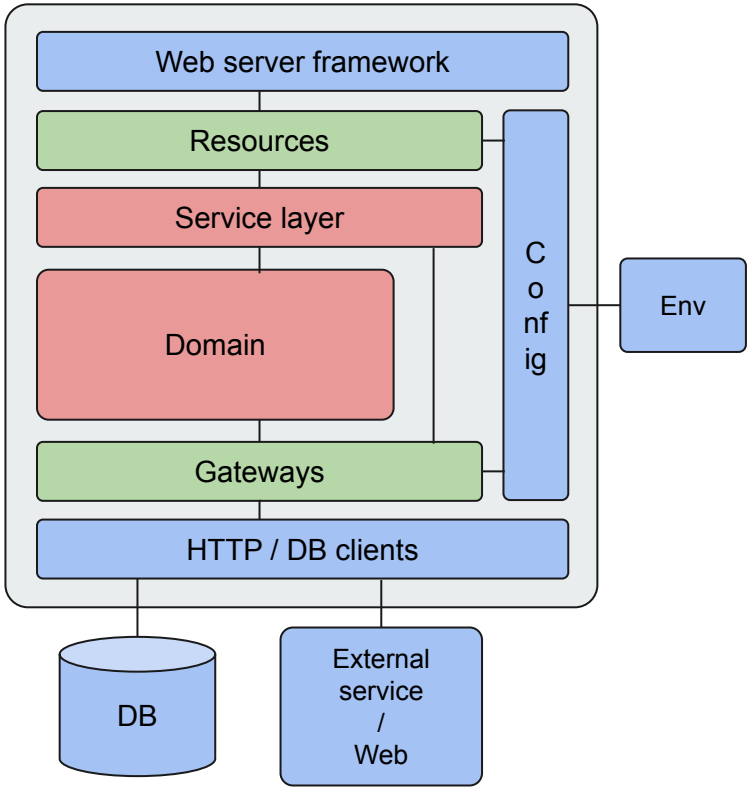

```
20      # Use newer version of swagger ui
21      options = {'swagger_path': swagger_ui_path}
22
23      app = connexion.App(__name__,
24                          specification_dir=config.SPECIFICATION_DIR,
25                          options=options)
26
27      app.add_api('swagger.yaml',
28                  strict_validation=True,
29                  validate_responses=True)
30
31      error_handler.register_error_handlers(app)
```



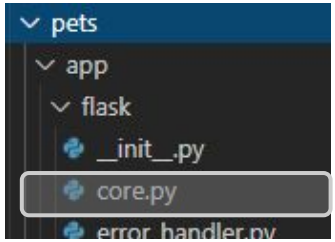


The anatomy of a microservice: pets example

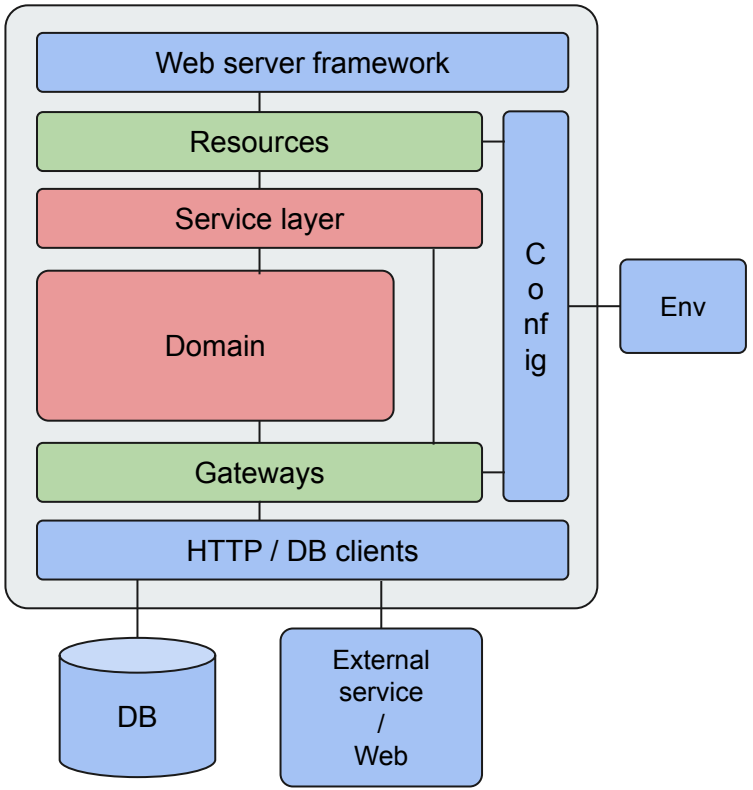




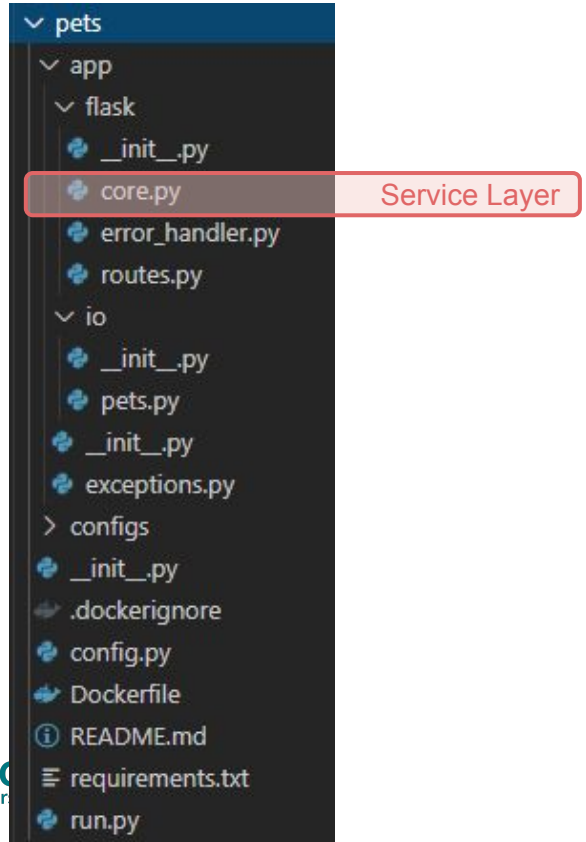
The anatomy of a microservice: pets example

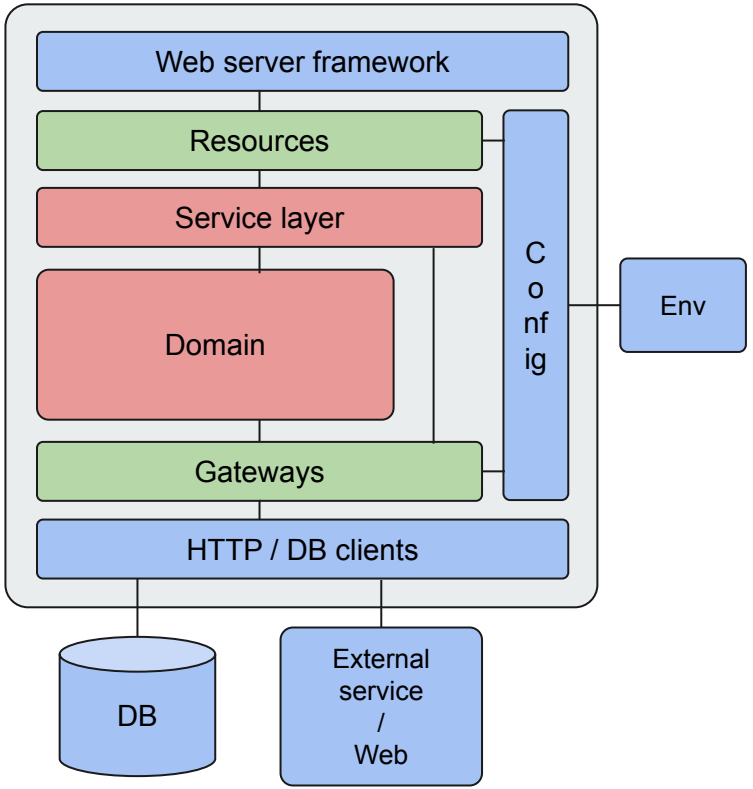



```
7   from app.io import pets
8
9
10  def get_all_pets():
11      """Get array of all pets."""
12      return pets.get_all()
13
14
15  def get_pet_by_id(pet_id):
16      """Get single pet by id."""
17      return pets.get(pet_id)
18
19
20  def add_pet(*, name, owner, description):
21      """Add a new pet."""
22      return pets.create(name=name, owner=owner, description=d
```

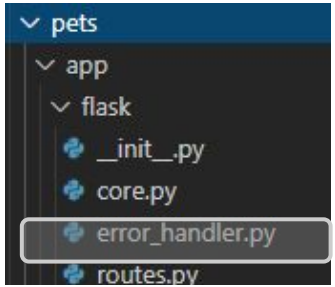


The anatomy of a microservice: pets example

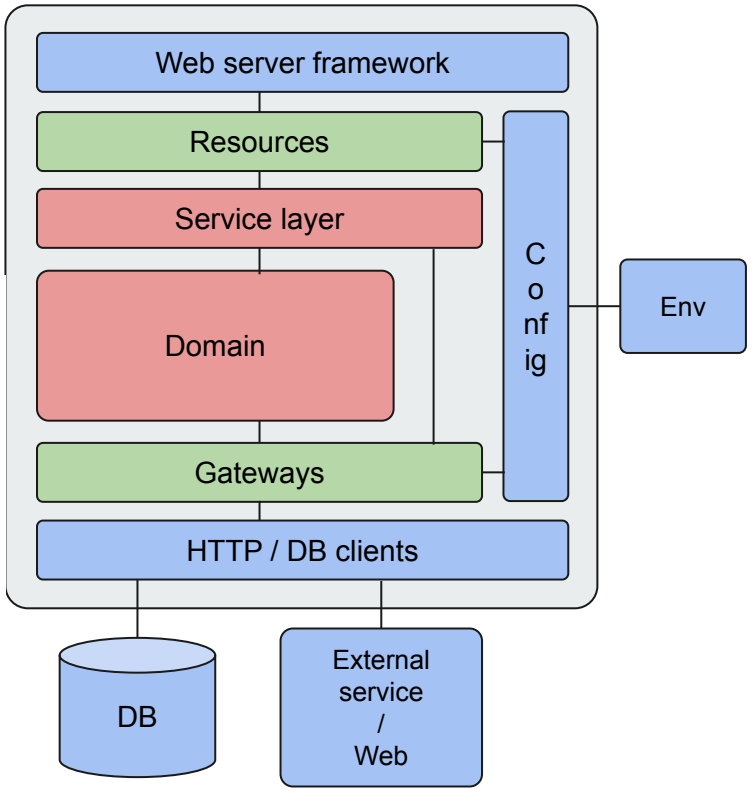




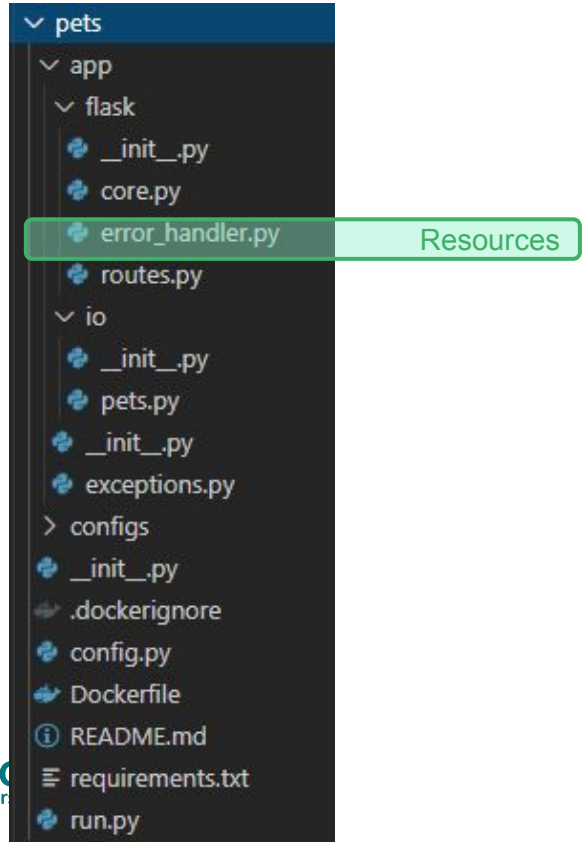
The anatomy of a microservice: pets example

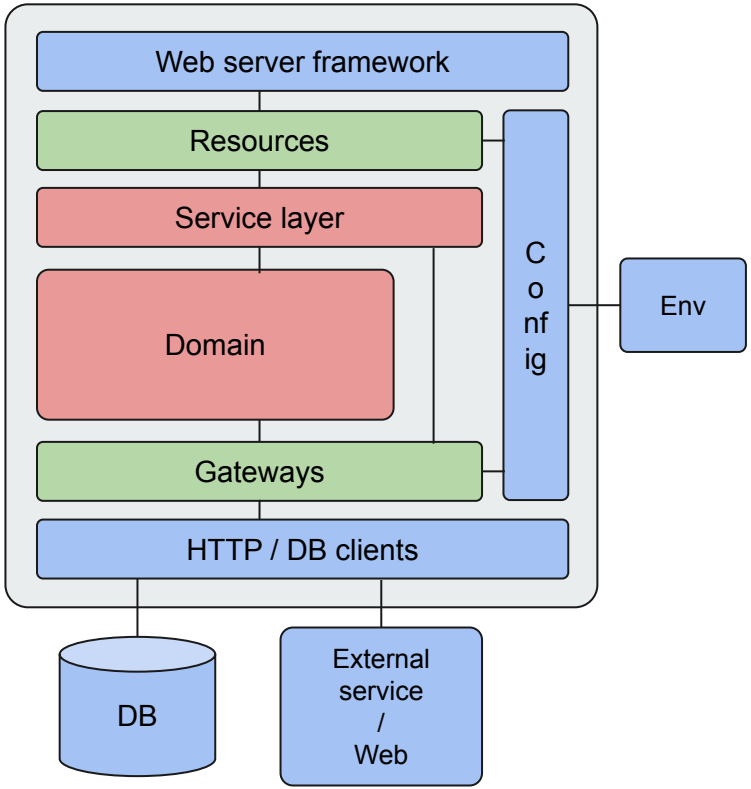



```
8  from flask import jsonify
9
10 from app import exceptions
11
12
13 def handle_custom_error(error):
14     """Handle custom errors."""
15     response = jsonify({'message': error.message})
16     response.status_code = error.status_code
17     return response
18
19
20 def register_error_handlers(app):
21     """Add error handlers to the app."""
22     app.add_error_handler(exceptions.PetNotFound, handle_custo
```

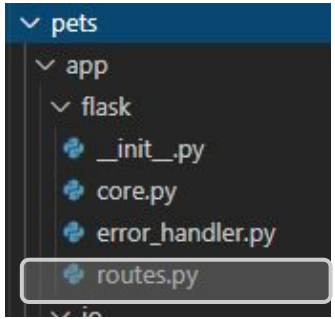


The anatomy of a microservice: pets example

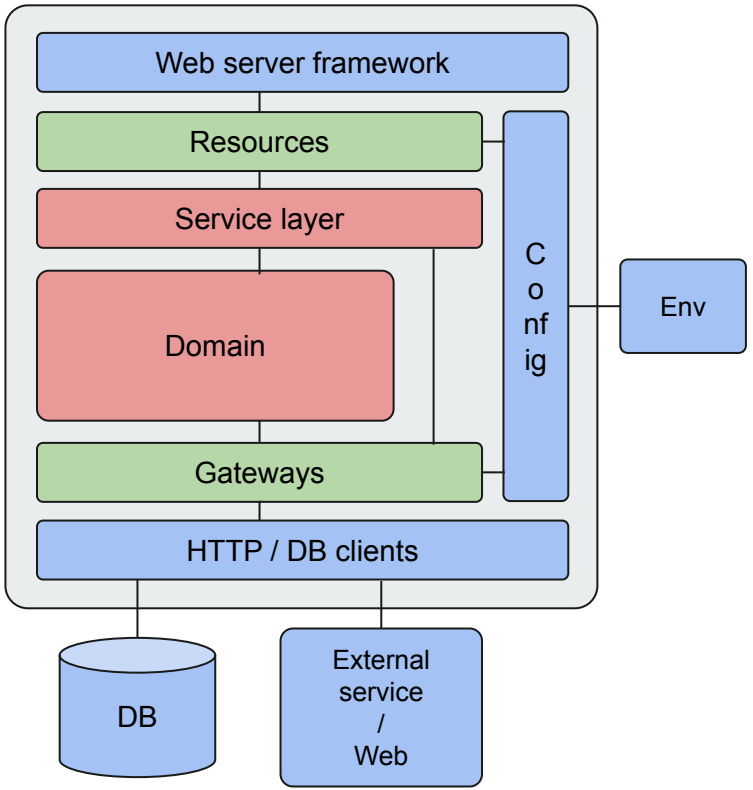




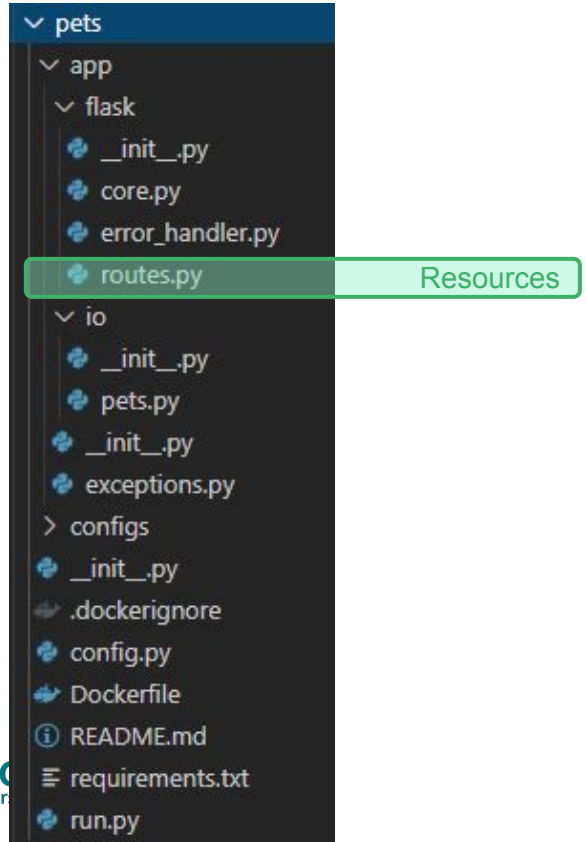
The anatomy of a microservice: pets example

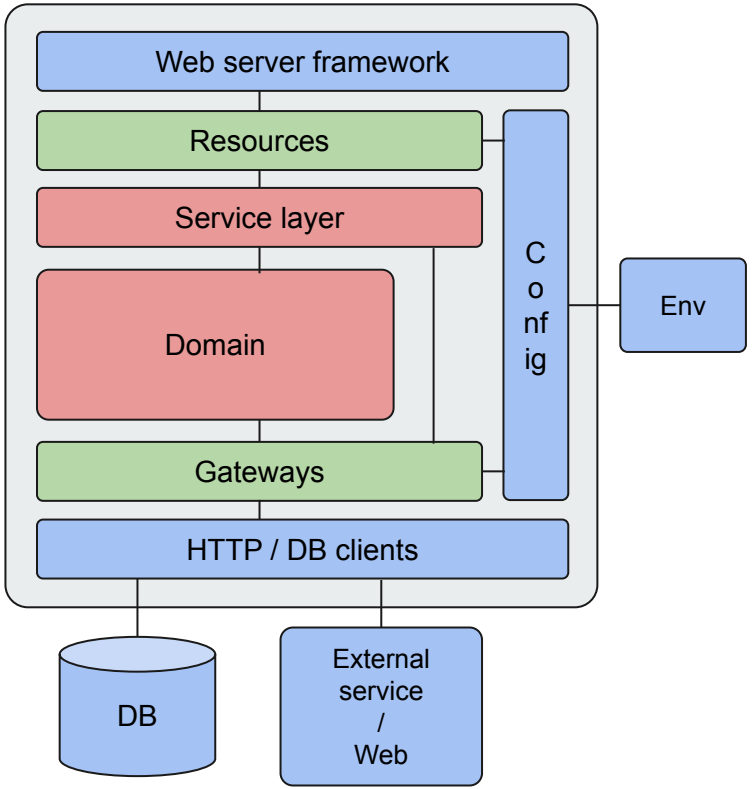



```
7   from app.flask import core
8
9
10  class Pet:
11      """Includes all HTTP methods for /pets/pet_id>"""
12
13      @staticmethod
14      def get(pet_id):
15          """Get an existing pet."""
16          return core.get_pet_by_id(pet_id), 200
17
18      @staticmethod
19      def delete(pet_id):
20          """Delete an existing pet."""
21          return core.delete_pet_by_id(pet_id), 204
```

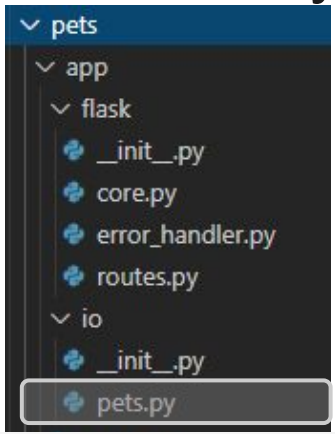


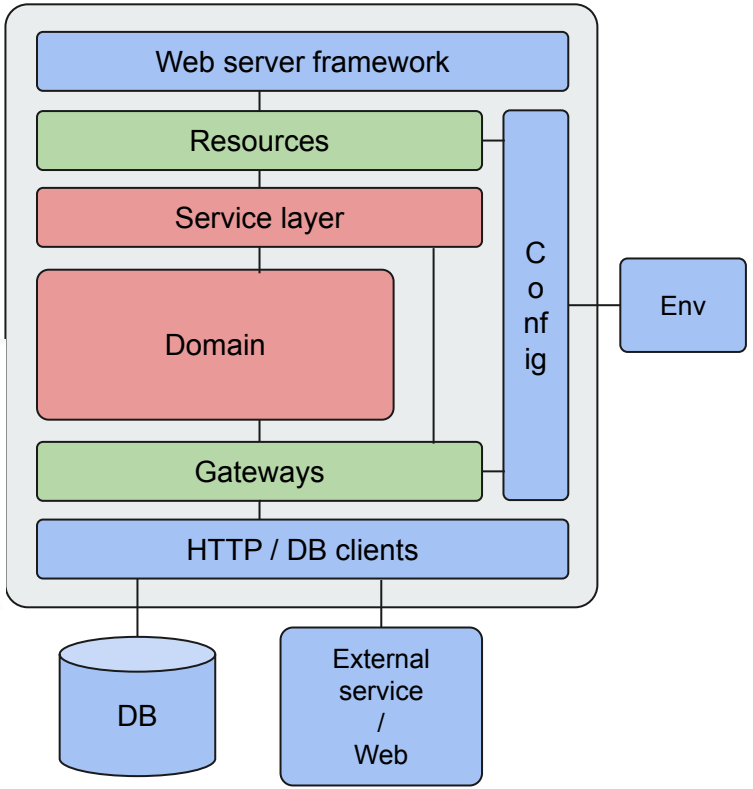
The anatomy of a microservice: pets example





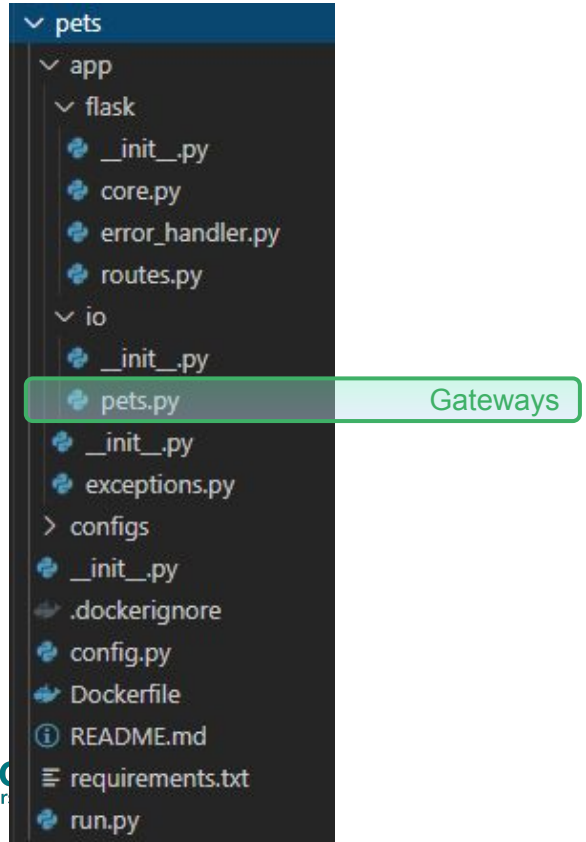
The anatomy of a microservice: pets example

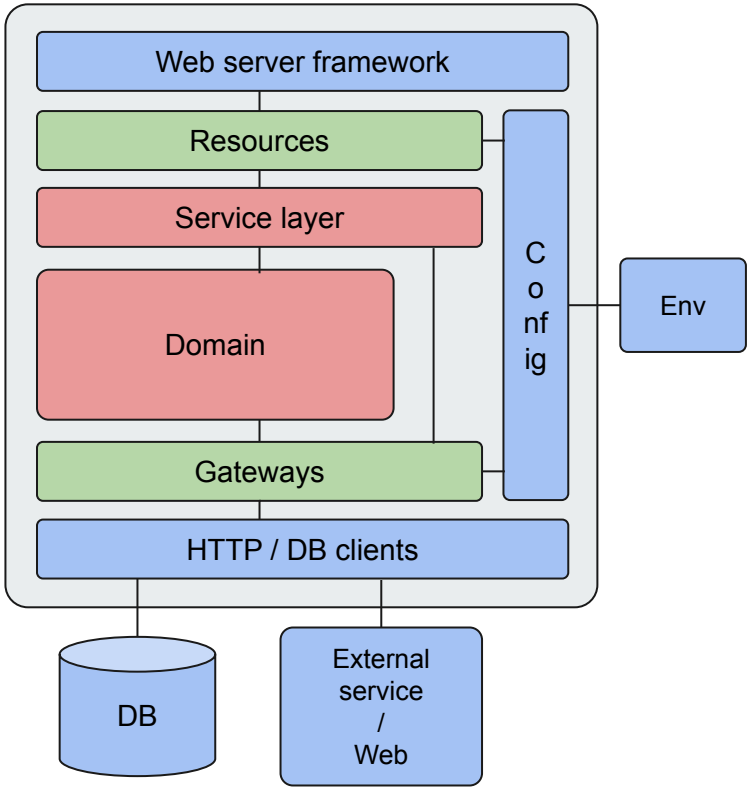




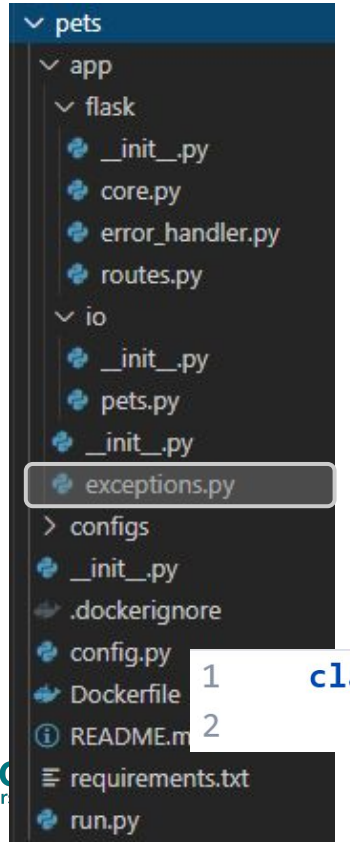

```
11  client = firestore.Client()
12  pets = client.collection('pets')
13
14
15  def get(pet_id: str) -> dict:
16      """Get a pet by id.
17
18      Raises:
19          exceptions.PetNotFound: If no pet with the specified id can be found.
20
21      """
22      doc_ref = pets.document(pet_id)
23      pet = doc_ref.get()
24      if not pet.exists:
25          raise exceptions.PetNotFound('Pet with id {pet_id} was not found.')
26      return doc_ref.get().to_dict()
```

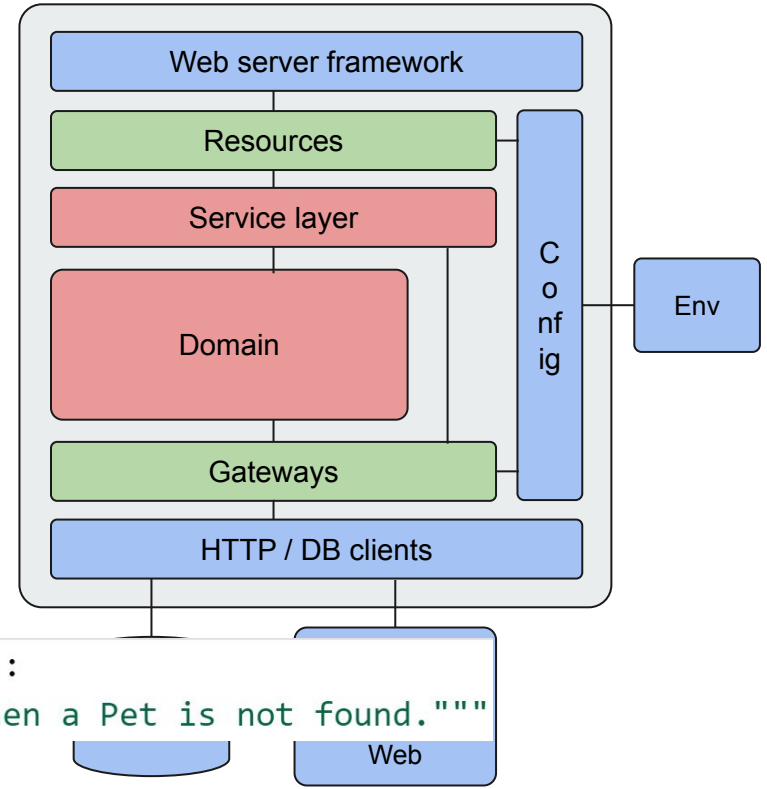
The anatomy of a microservice: pets example





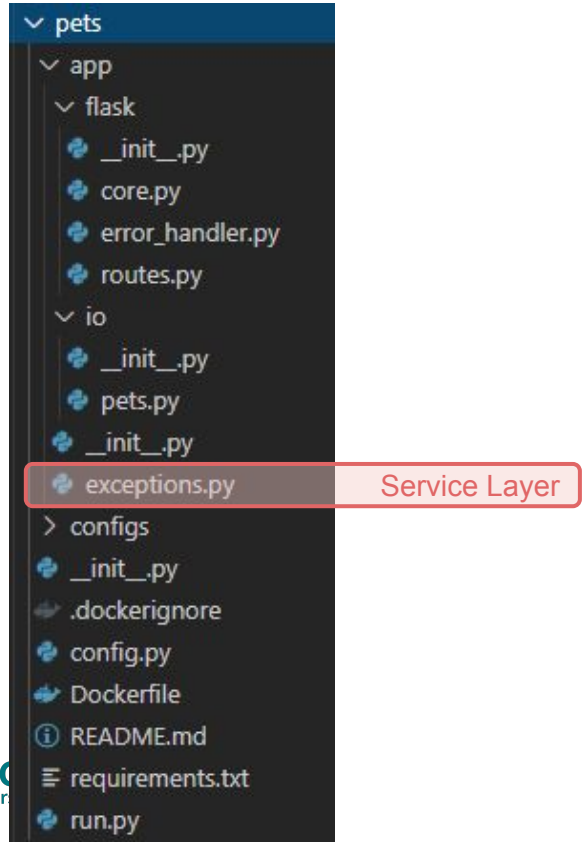
The anatomy of a microservice: pets example

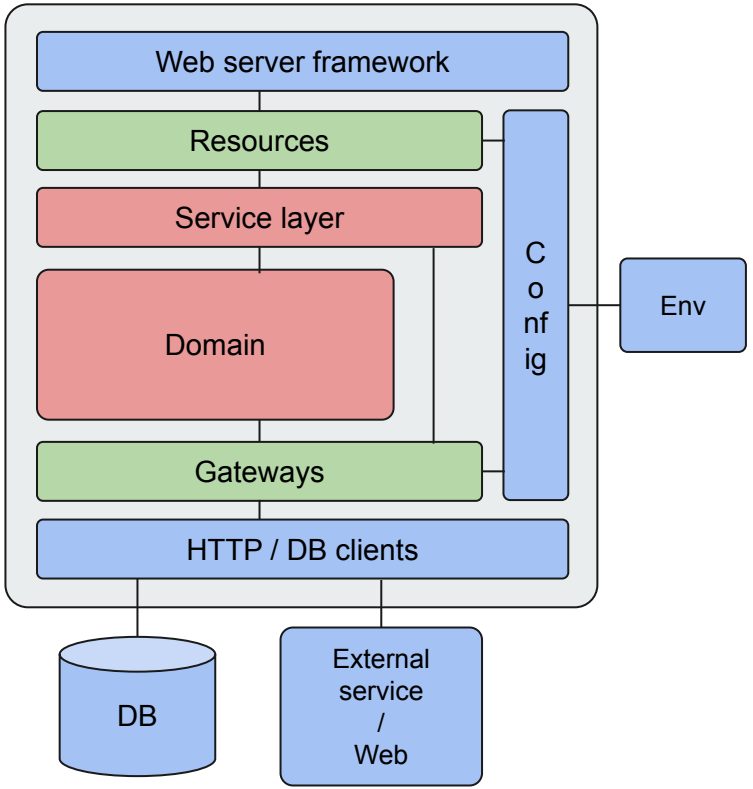




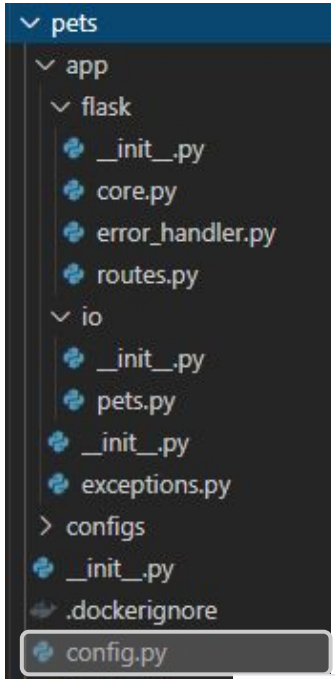

```python
class PetNotFound(Exception):
    """Exception to raise when a Pet is not found."""
```

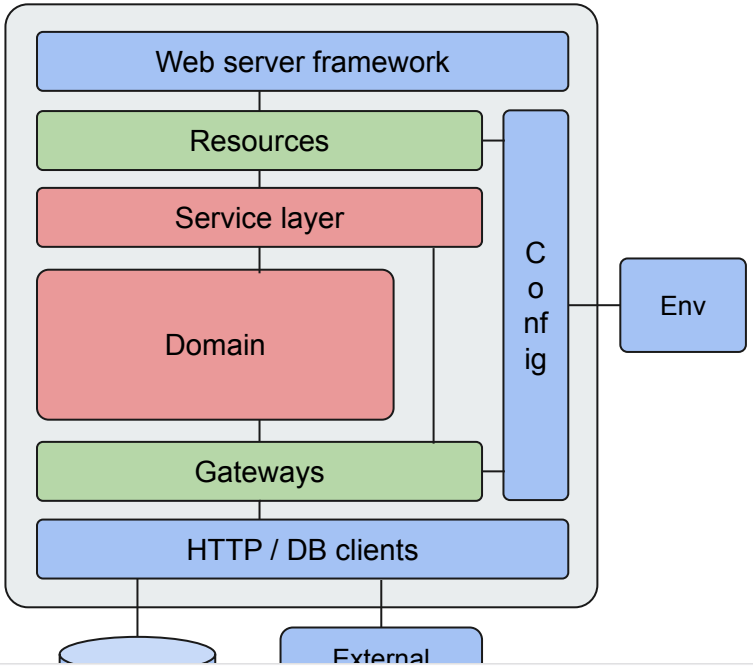
The anatomy of a microservice: pets example





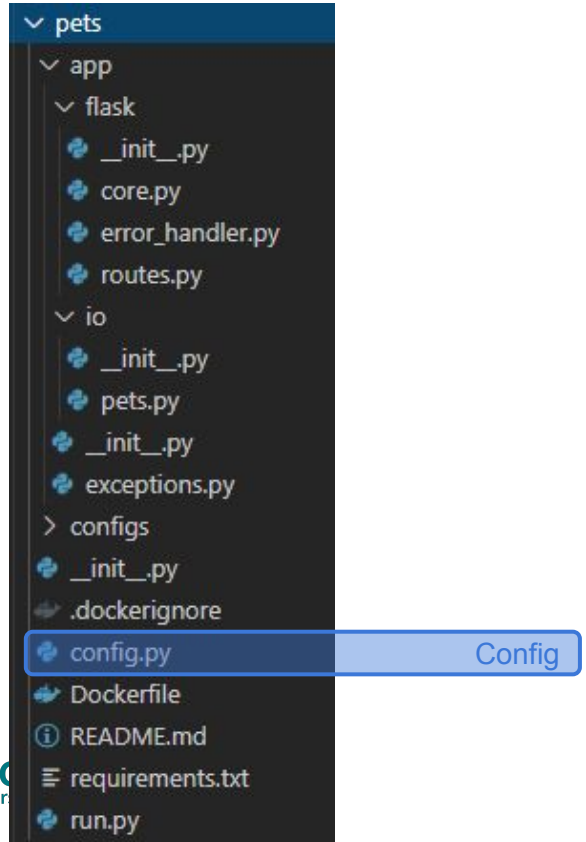
The anatomy of a microservice: pets example

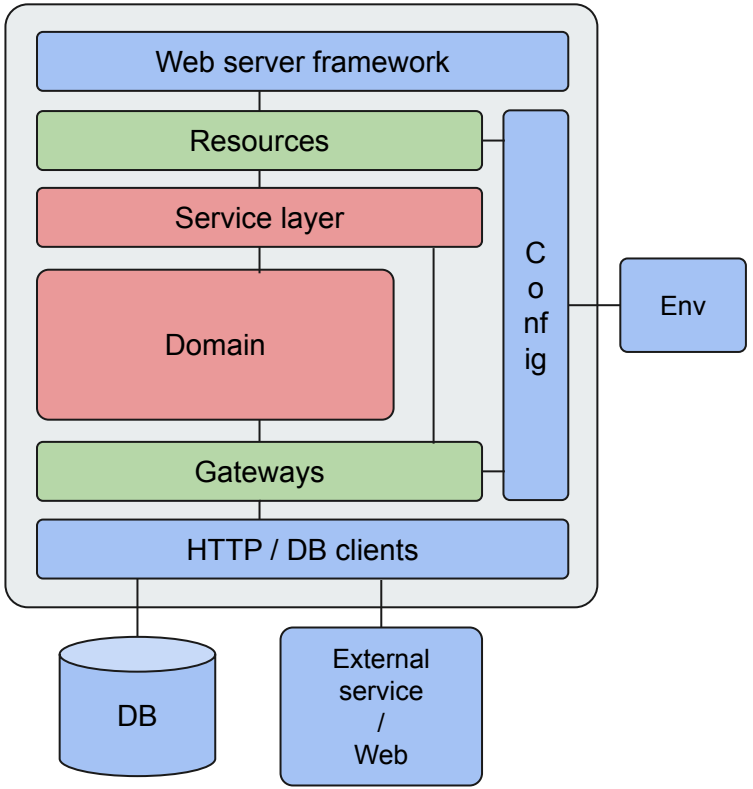




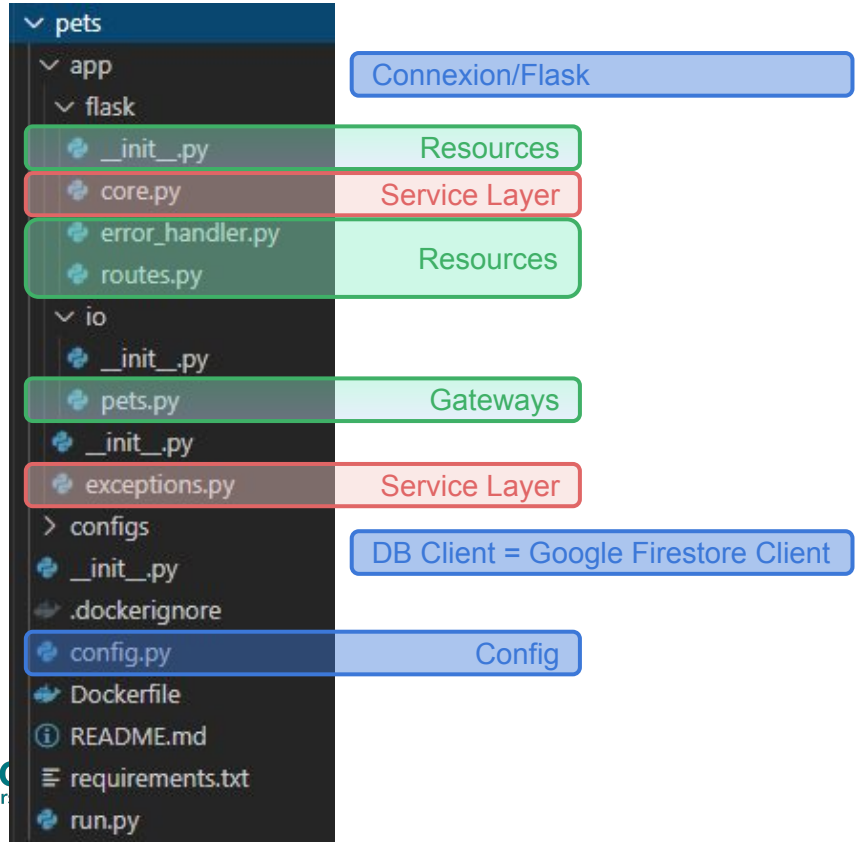

```python
import os

SPECIFICATION_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)),
                                 'configs')
```

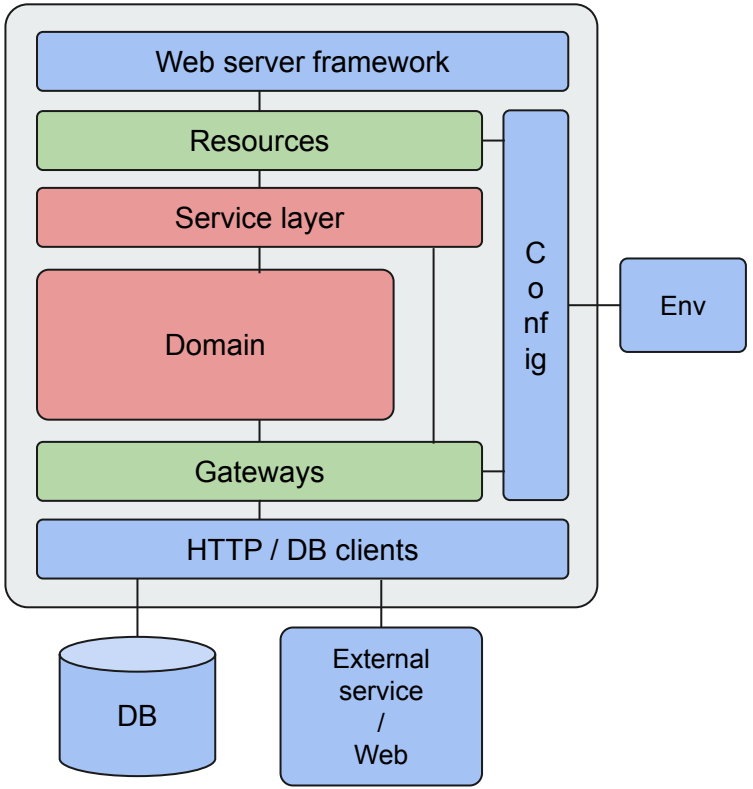
The anatomy of a microservice: pets example



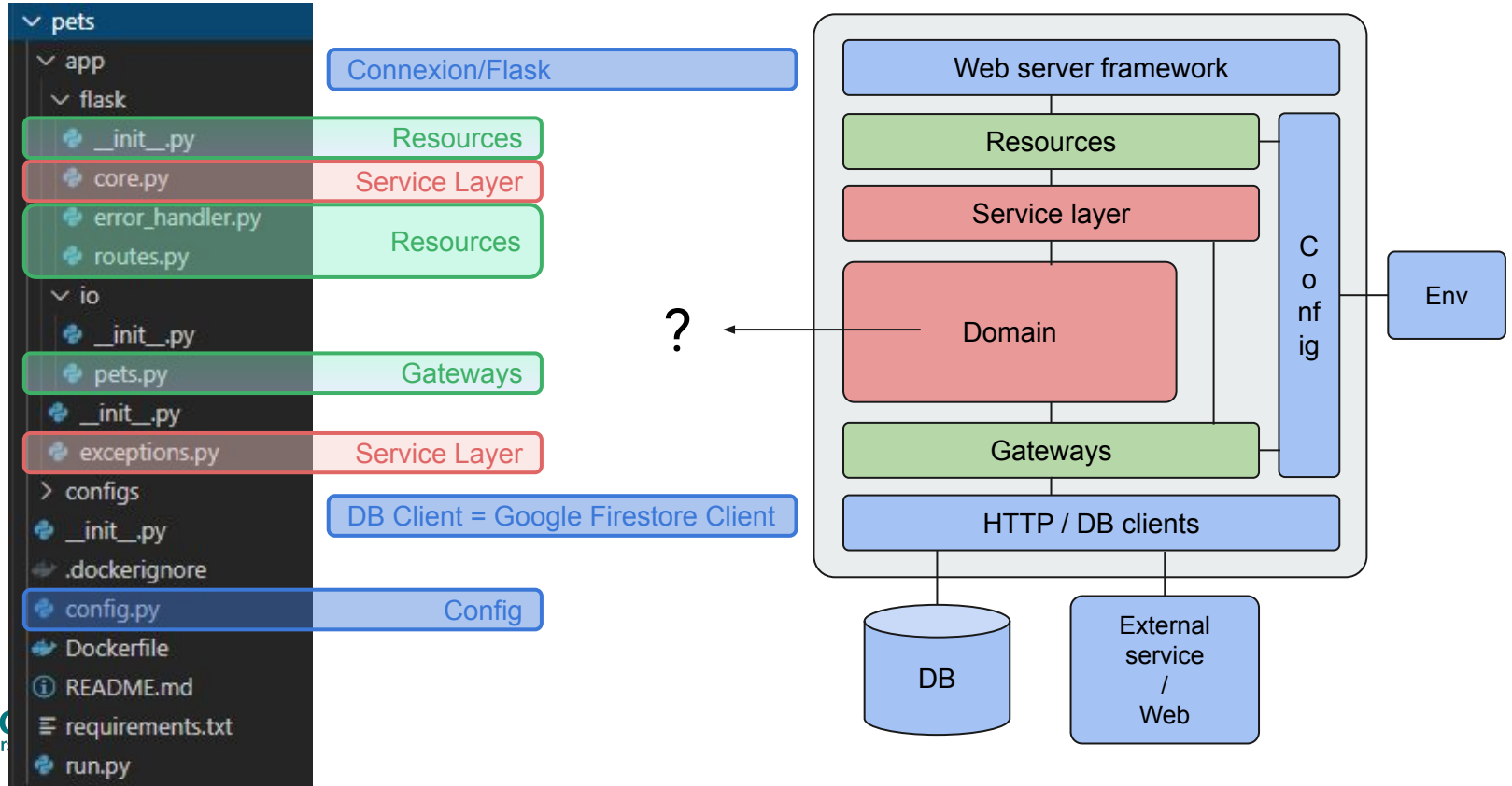


The anatomy of a microservice: pets example

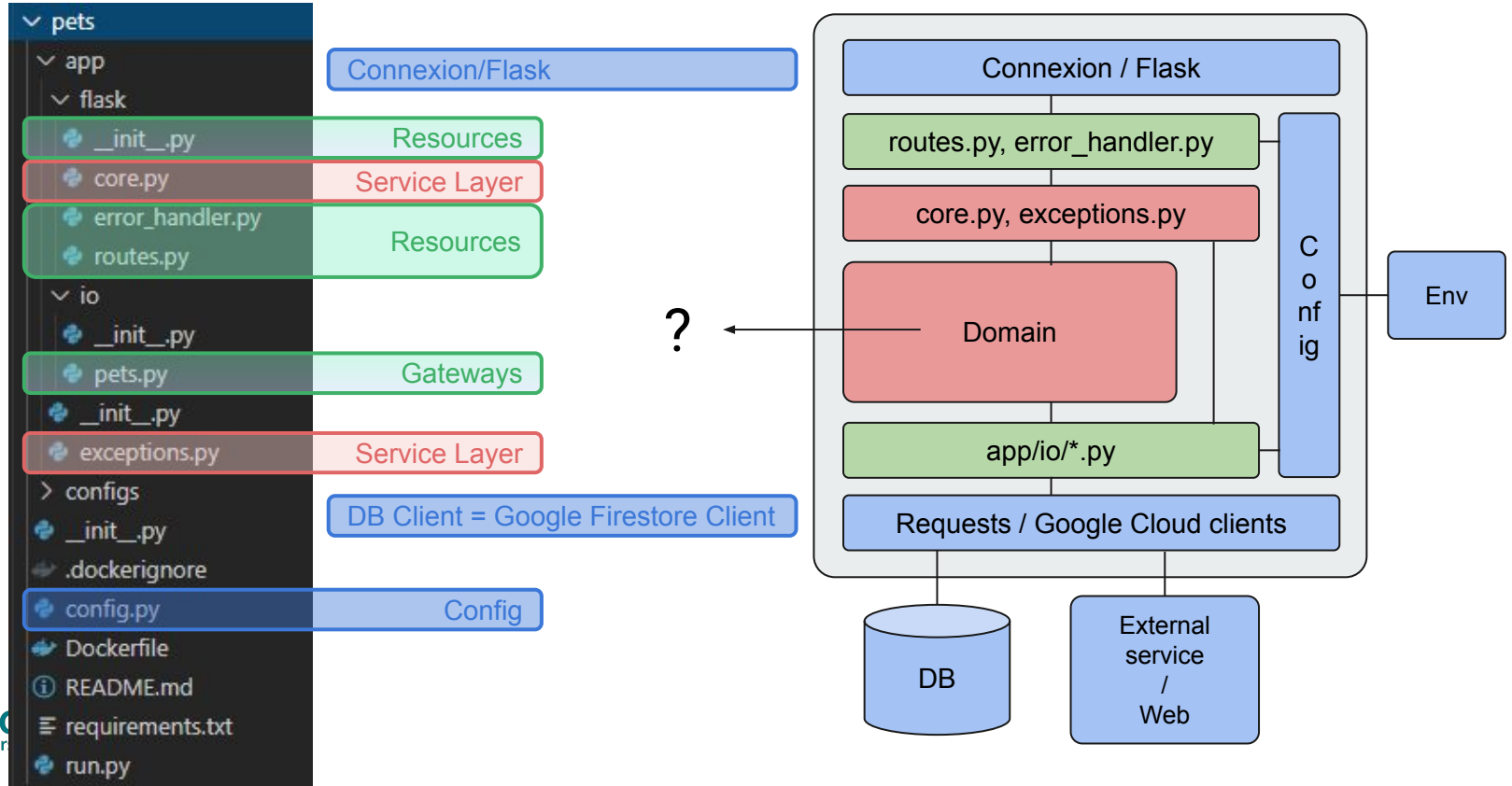




The anatomy of a microservice: pets example



The anatomy of a microservice: pets example



Directed work: Flask

Wrap-up

Lecture summary

Topic	Concepts	To know for…	
		Project	Exam
API	• API • REST • RPC		Yes
GenAI Gateway	• GenAI Gateway		Yes
Microservices	• Microservices • Monolith • Anatomy of a microservice	Possibly	
Flask		Possibly	

Project objective for sprint 3

#	Week	Work package	Requirement
3.1	W05	Build an **API** to **serve your model** and any extra logic that is needed to serve it (e.g. using Flask). You should be able to run the API locally.	Required
3.2	W05	Package your model serving API in a **Docker container**. This too should be run locally.	Required
3.3	W06	**Deploy** your model serving API in the Cloud. You should be able to call your model to generate new predictions from another machine. **Attention**: This can incur Cloud **costs**. Make sure to use a platform where you have credits and not burn through them. You can ask for support from the teaching staff in that regard.	Required

Next week…

- Today we saw how an API works and how to build a local Flask API
- Next week we will see how you can deploy an API in the Cloud

That’s it for today!

LIÈGE université