POLITECNICO
MILANO 1863

POLITECNICO
MILANO 1863

Software Engineering 2

Introduction to Architectural Styles

Client-server

Interface Design and Documentation

Designing servers to handle multiple requests

Other styles: Multi-tier, Event-driven

M Camilli, E Di Nitto, M Rossi

Software Design

Introduction to Architectural Styles

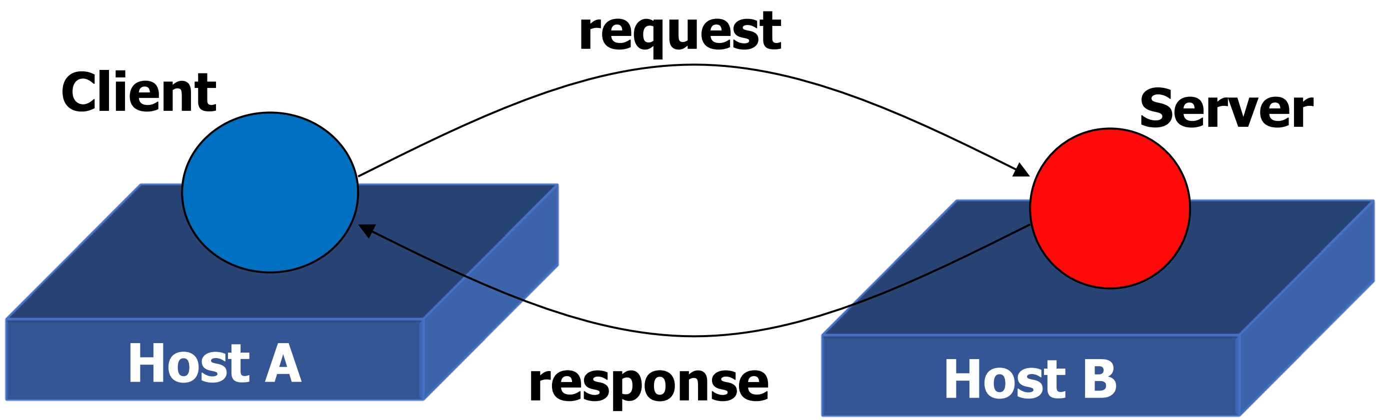
Client-server

Architectural Style

- “an architectural style determines the **vocabulary** of **components** and **connectors** that can be used in instances of that style, together with a set of **constraints** on how they can be combined.
  These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition.” [Garland & Shaw]

[Garland & Shaw] David Garlan and Mary Shaw, “An Introduction to Software Architecture”, Jan 1994, CMU-CS-94-166. http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

Client-server

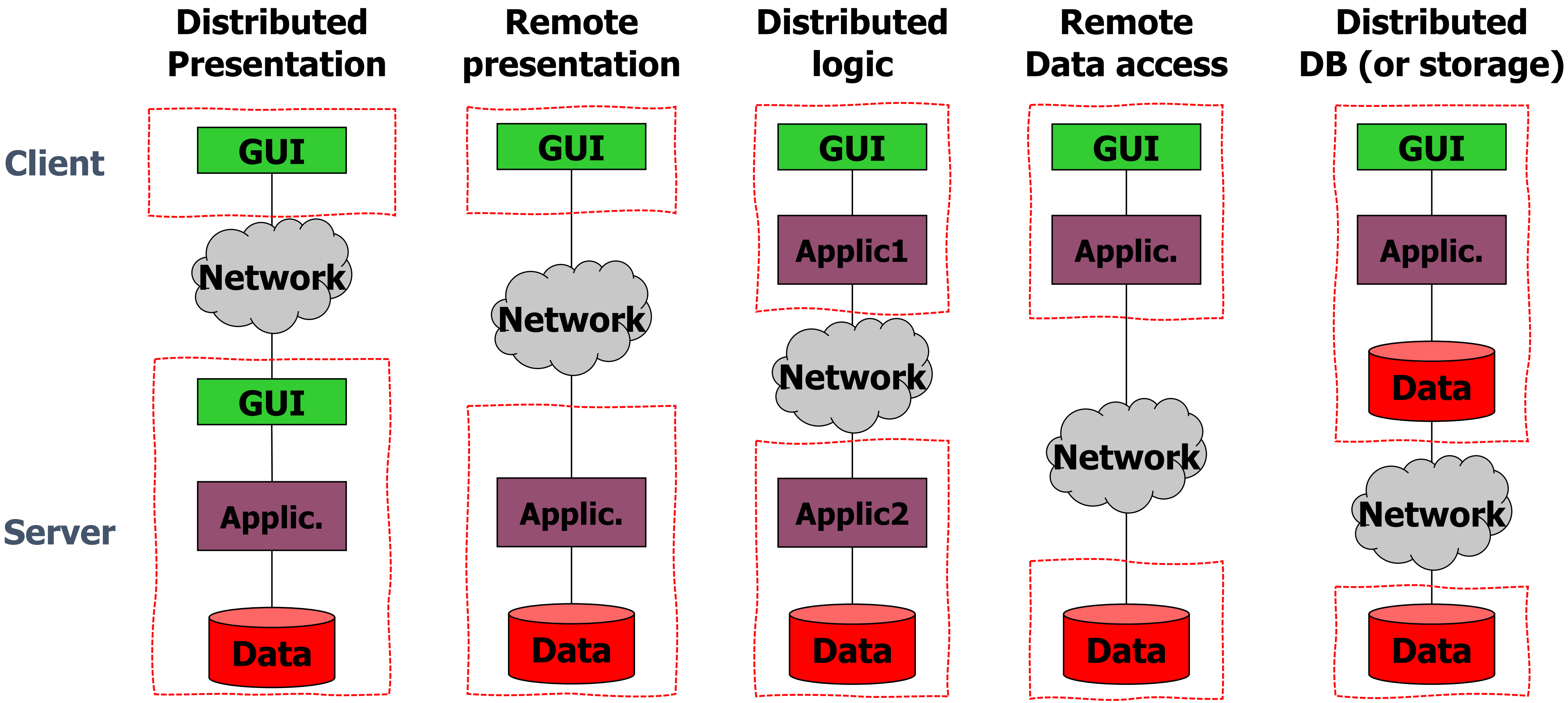


- Two component roles
  - Client issues requests
  - Server provides responses

Why do I use it?

- **Multiple users** need to access a **single resource**
  - e.g., database
- There is a preexisting software we must **access remotely**
  - e.g., email server
- It is convenient to organize the system around a **shared piece of functionality** used by multiple components
  - e.g., authentication/authorization server

Organization of Client-Server software: thin vs fat clients



Client-server: main technical issues

- **Design** and **document** proper interfaces for our server
- Ensure the server is able to **handle multiple simultaneous requests**
  - Forking vs thread pooling

Software Design

Interface Design

Interface design

- An interface is a boundary across which components interact
- Proper definition of interfaces is an architectural concern
  - Impacts maintainability, usability, testability, performance, integrability
- Two important guiding principles
  - **Information hiding**
  - **Low coupling**
- An interface shall encapsulate a component implementation so that this can be changed without affecting other components

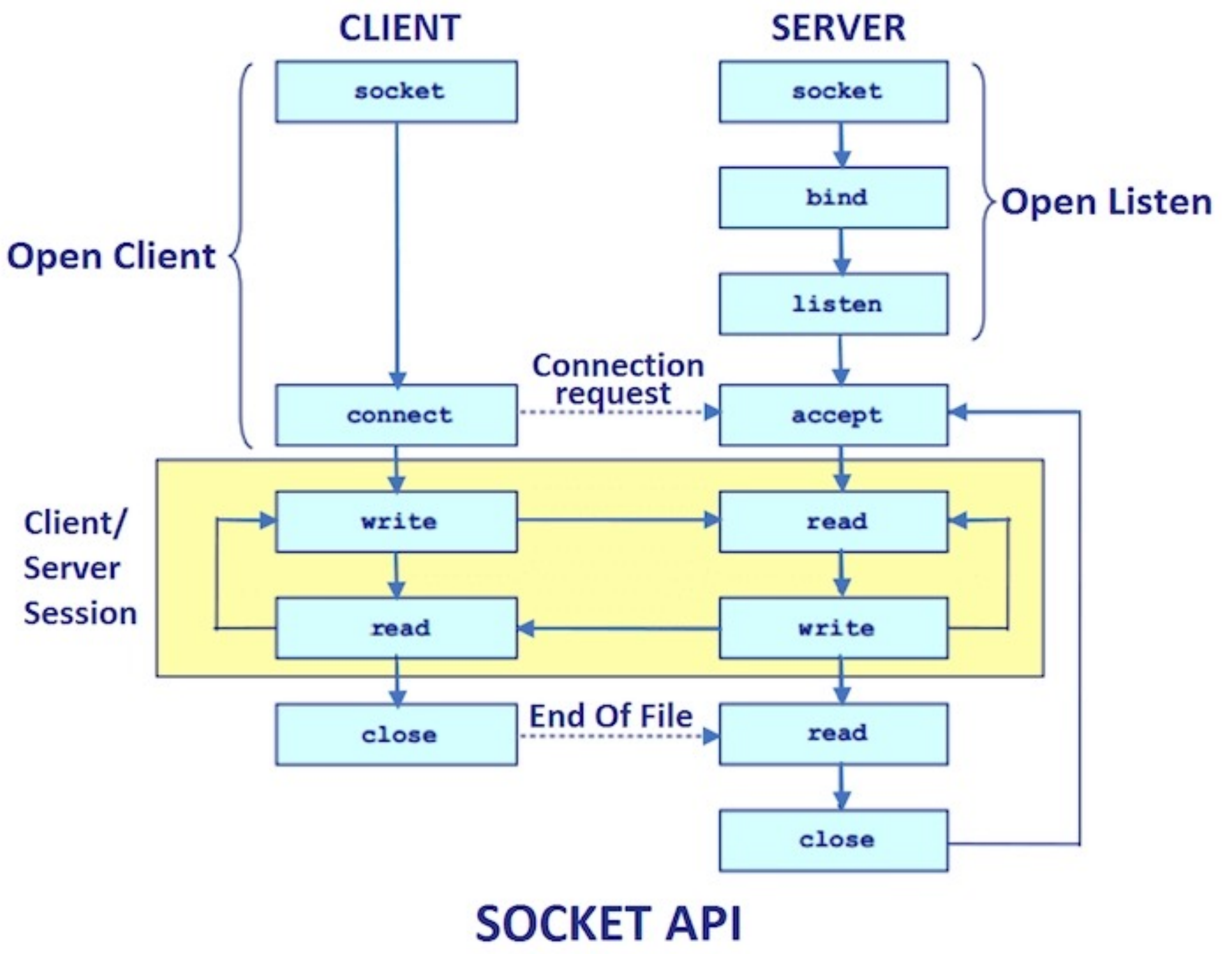
Aspects to consider during interface design

POLITECNICO MILANO 1863

- Contract principle: Any resource (operation, data) added to an interface implies a commitment to maintaining it
- Least surprise principle: Interfaces should behave consistently with expectations
- Small interfaces principle: Interfaces should limit the exposed resources to the minumum
- Important elements to be defined:
  - **Interaction style** (e.g., sockets, RPC, REST)
  - **Representation** and structure of exchanged data
  - **Error handling**

Sockets

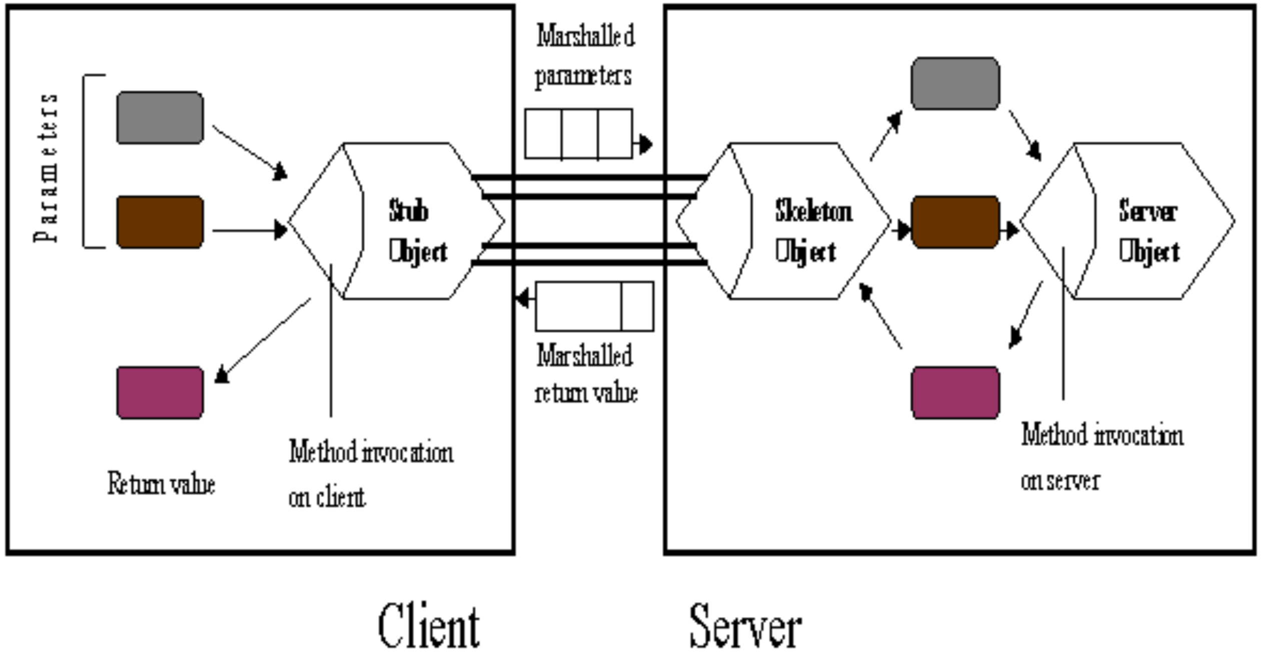
https://www.codingninjas.com/studio/library/socket-programming



- After connection establishment communication is bidirectional
- Both parties must agree on the same protocol (sequence of message exchange)

RPC/RMI

http://infolab.stanford.edu/CHAIMS/Doc/Details/Protocols/rmi/rmi_description.html



- Resembles procedure/method call in a centralized setting
- Stubs and skeletons needed to transform procedure/method calls in messages and vice versa

Communication in RMI: back to the banking example github.com/matteocamilli/bank_example_rmi

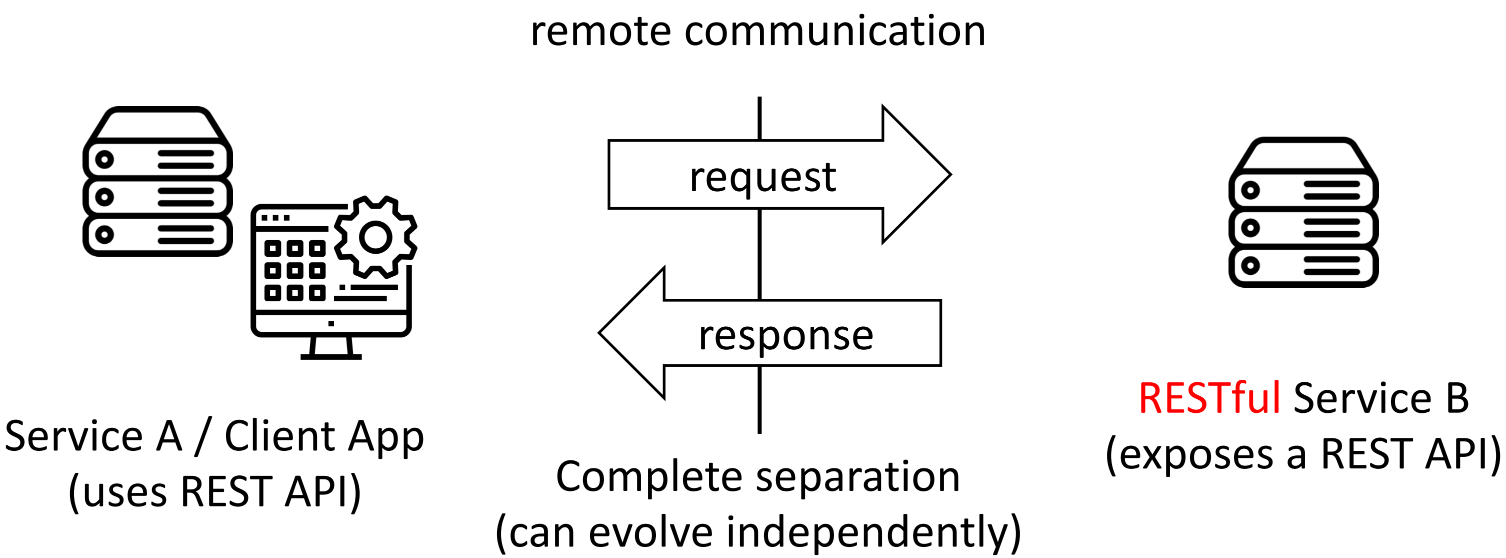
```
import java.rmi.*;
public class Client {
 public static void main (String[] args) {
   try {
     //…
     String name = "Jack B. Quick";
     Account account = manager.open(name);
     float balance = account.balance();
     //…
   }catch(Exception e){e.printStackTrace();}
 }
}
```

`manager` and `account` are local stubs offering the same interface as the corresponding remote objects

```
public class AccountManagerImpl … {
 public Account open(String name) … {
   //… }
 public static void main(String[] args)
 { //…
  AccountManagerImpl server = new
          AccountManagerImpl(_hostName);
  Naming.rebind("//" + _hostName +
          "/AccountManager", server);
  //…
 }
}
```

`server` is the remote object that will execute the `open` operation

REST: REpresentational State Transfer



- REST = REpresentational State Transfer
  - Specific standardized architectural style for Application Programming Interfaces (APIs)
  - Realizes clear separation between distributed, heterogeneous systems/components

Roy T. Fielding and Richard N. Taylor. 2000. Principled design of the modern Web architecture. ICSE '00. ACM, New York, NY, USA, 407–416. https://doi.org/10.1145/337180.337228

REST APIs

- Characteristics
  - Simple and **standardized** → Developers do not have to worry about
    - Communication protocols (HTTP)
    - how to format the data (JSON)
    - how to format requests (request/response encoding)
    - Ready-to-use industry standard
  - **Stateless** → does not keep track of states across server and all clients
    - Lightweight and **scalable**
    - Supports **caching** → high performance

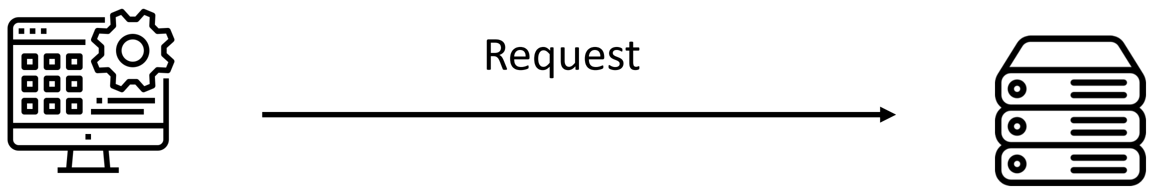
REST APIs: requests

POLITECNICO
MILANO 1863

Known endpoint (public URL that identifies a resource)

`https://petstore.swagger.io/v2/pet/findByStatus`

API v2 of the endpoint (`v2`) — **Resource** (`pet/findByStatus`)



Client App

RESTful Service “pet” of **petstore**

- Types of requests (CRUD)
  - Create (create a new resource) → POST
  - Read (read an existing resource) → GET
  - Update (modify a resource) → PUT
  - Delete (remove a resource) → DELETE

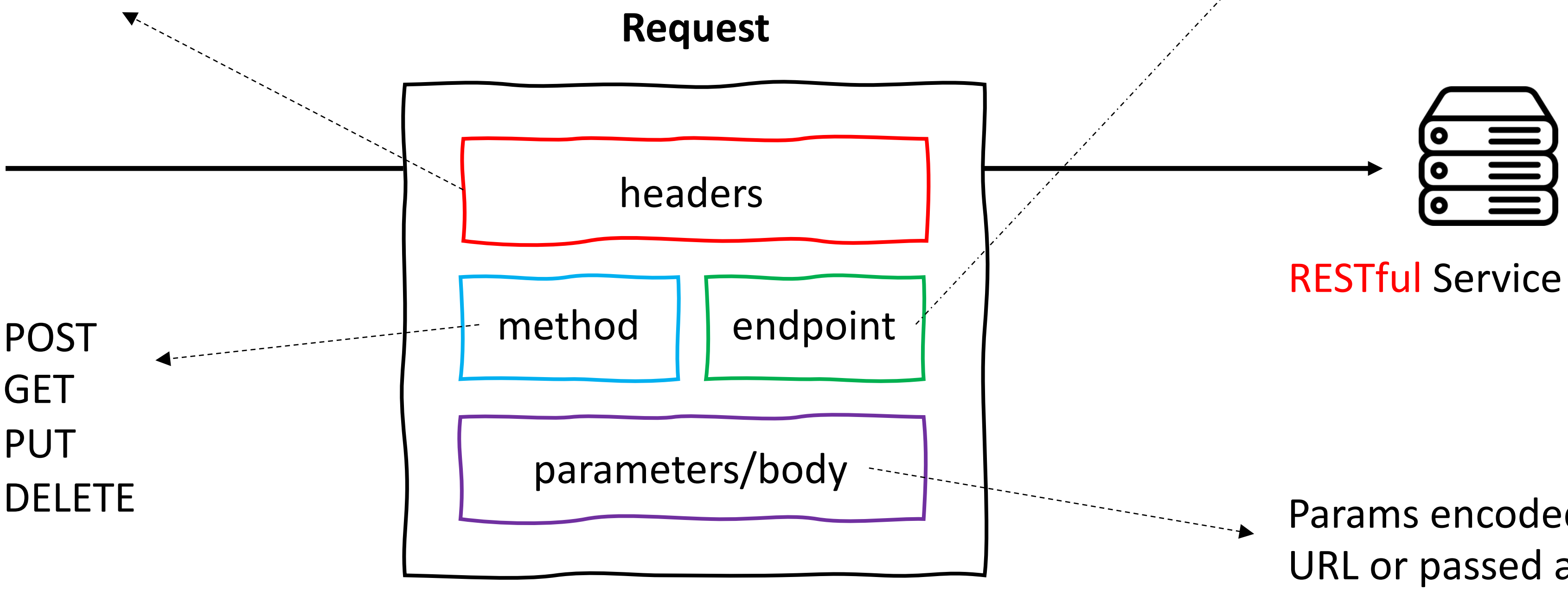
standard HTTP methods

REST APIs: requests

“Special part” contains additional info like auth data and API key

Known endpoint

`https://petstore.swagger.io/v2/pet/findByStatus`



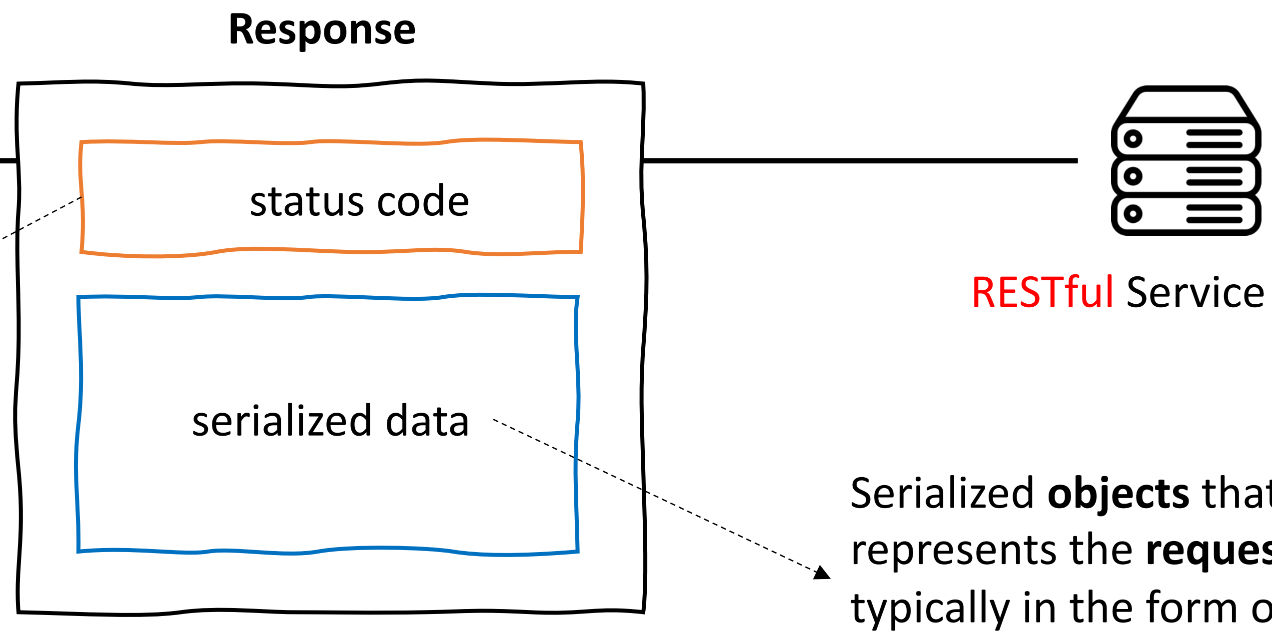
POST
GET
PUT
DELETE

RESTful Service “pet”

Params encoded into URL or passed as part of the body

REST APIs: response

Known endpoint
`https://petstore.swagger.io/v2/pet/findByStatus`



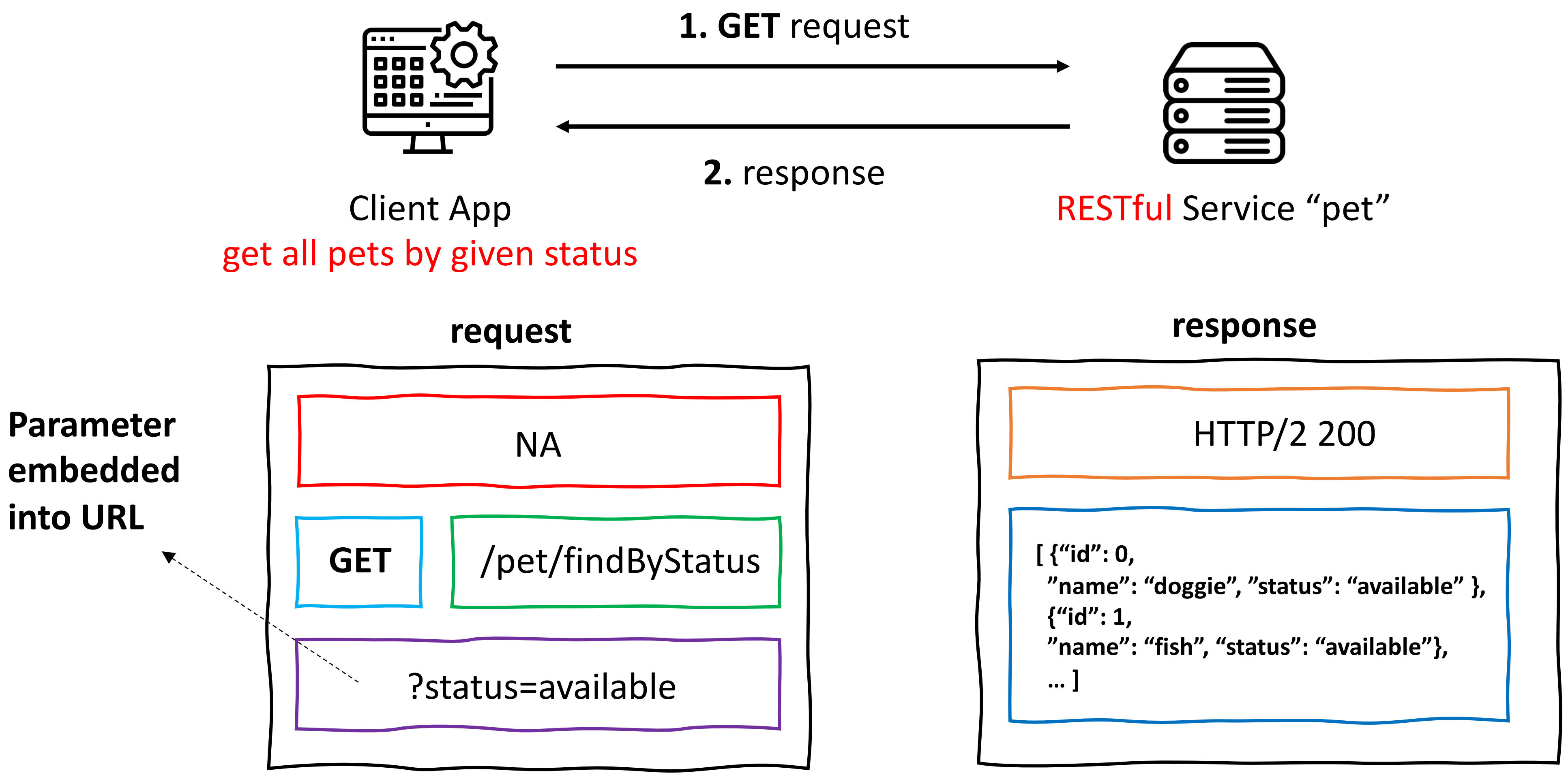
2xx successful
- e.g., 200 = Ok

4xx client error
- e.g., 400 = bad request

5xx server error
- e.g., 500 = internal server error
- e.g., 503 = service unavailable

Serialized **objects** that represents the **request outcome**, typically in the form of **JSON** data (see next slides)

REST APIs: request/response scenario (1)



REST APIs: request/response scenario (1)

POLITECNICO MILANO 1863

Request

HTTP method endpoint parameters

```
curl -X 'GET' \
'https://petstore.swagger.io/v2/pet/findByStatus?status=available' \
-H 'accept: application/json'
```

Header: body format is JSON

Command line request (curl utility)

Response

HTTP/2 200
date: Fri, 13 Oct 2023 16:38:33 GMT
content-type: application/json …

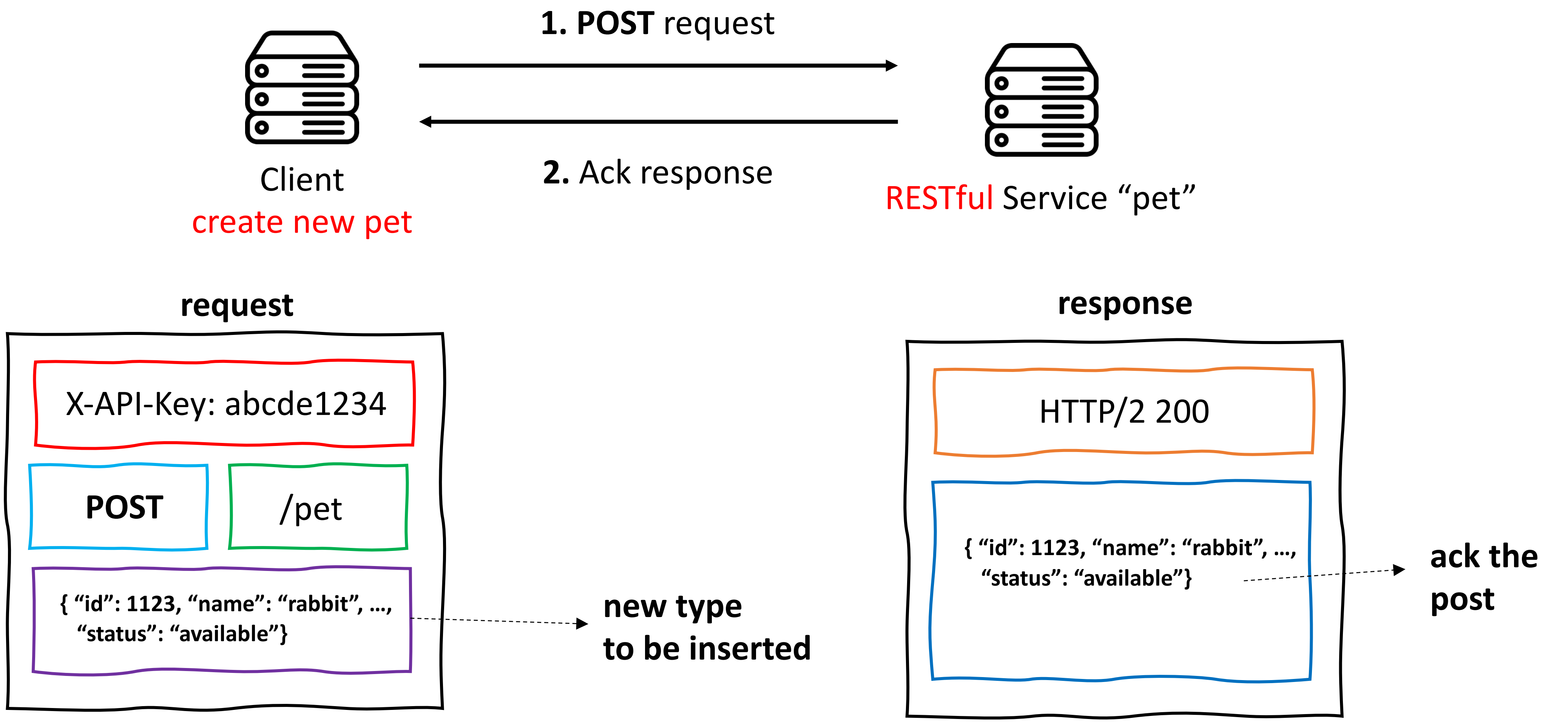
Status code

[{ "id": 9222968140497191000, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available" }, …]

Serialized data

REST APIs: request/response scenario (2)

POLITECNICO
MILANO 1863



REST APIs: request/response scenario (2)

POLITECNICO MILANO 1863

Request

```
curl -X 'POST' \
'https://petstore.swagger.io/v2/pet' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d { "id": 12300, "category": { "id": 0, "name": "string" }, "name": "rabbit", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name": "string" } ], "status": "available" }
```

Response

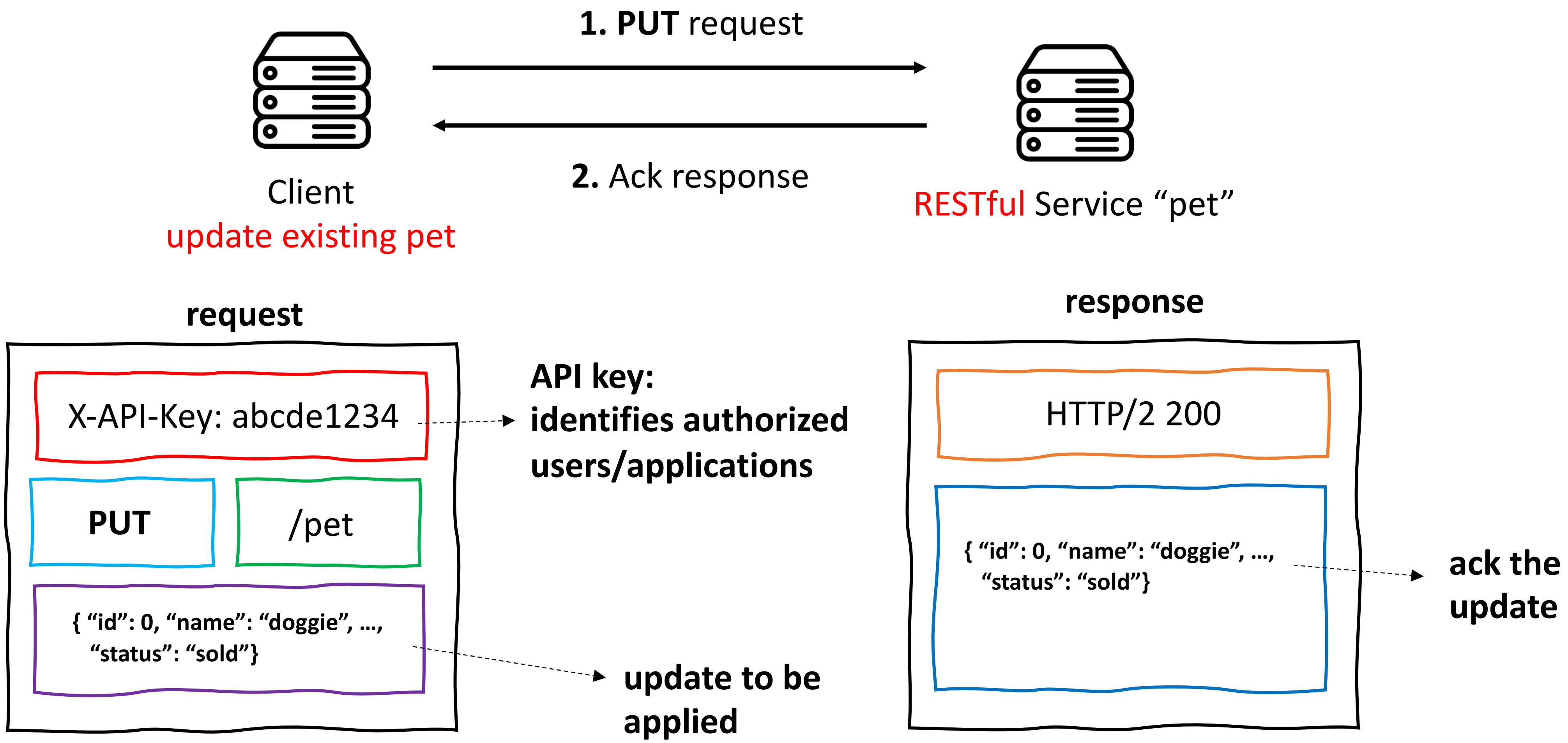
HTTP/2 200
date: Fri, 13 Oct 2023 16:38:33 GMT
content-type: application/json …

```
{ "id": 12300,
  "category": { "id": 0, "name": "string" },
  "name": "rabbit",
  "photoUrls": [ "string" ],
  "tags": [ { "id": 0, "name": "string" } ],
  "status": "available" }
```

EXTRA MATERIAL

REST APIs: request/response scenario (3)

POLITECNICO MILANO 1863



EXTRA MATERIAL

REST APIs: request/response scenario (3)

POLITECNICO MILANO 1863

Request

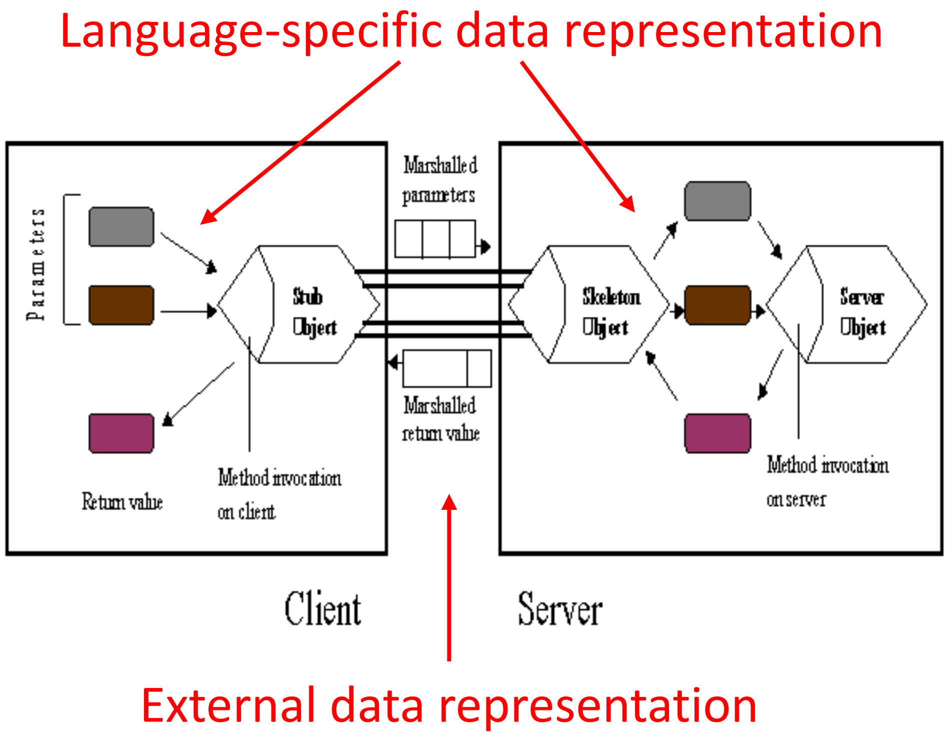
```
curl -X 'PUT' \
'https://petstore.swagger.io/v2/pet' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{ "id": 0, "category": { "id": 0, "name": "string" }, "name":
"doggie", "photoUrls": [ "string" ], "tags": [ { "id": 0, "name":
"string" } ], "status": "sold" }'
```

Response

HTTP/2 200
date: Fri, 13 Oct 2023 16:38:33 GMT
content-type: application/json …

```
{ "id": 0,
  "category": { "id": 0, "name": "string" },
  "name": "doggie",
  "photoUrls": [ "string" ],
  "tags": [ { "id": 0, "name": "string" } ],
  "status": "sold"  }
```

Representation and structure of exchanged data



- **Representation** impact on
  - Expressiveness
  - Interoperability
  - Performance
  - Transparency

Representation and structure of exchanged data – JSON, XML, Protocol Buffer

POLITECNICO
MILANO 1863

XML

```
<guests>
 <guest>
  <firstName>John</firstName> <lastName>Doe</lastName>
 </guest>
 <guest>
  <firstName>María</firstName> <lastName>García</lastName>
 </guest>
 <guest>
  <firstName>Nikki</firstName> <lastName>Wolf</lastName>
 </guest>
</guests>
```

Message instance

Interface description

JSON

Message instance

```
{"guests":[
 { "firstName":"John", "lastName":"Doe" },
 { "firstName":"María", "lastName":"García" },
 { "firstName":"Nikki", "lastName":"Wolf" }
]}
```

Proto definition

```
message Guest {
  string firstName = 1;
  string lastName = 2;
}
message Guests {
 repeated Guest guests = 3; }
```

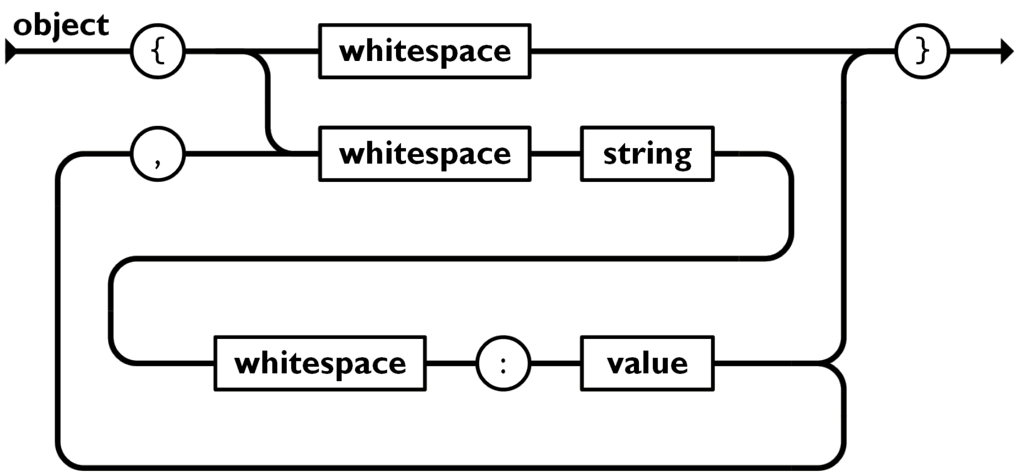
Representation and structure of exchanged data – JSON, XML, Protobuffer

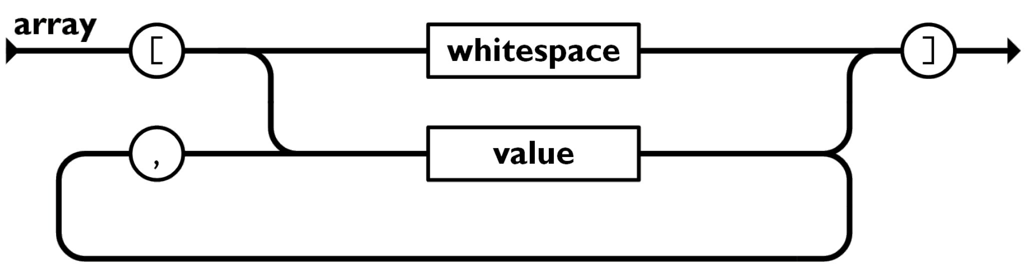
	XML	JSON	Protocol Buffers
Expressiveness	Good	Good	Good
Interoperability	Good	Good	Good
Performance	Verbose, requires multiple parsing passes	More compact that XML, single pass parsing	The most compact one
Transparency	Good, data passed as text	Good, data passed as text	Binary format

EXTRA MATERIAL

JSON standard (json.org)

- Language-independent **text format** to **encode** data
- Two main constructs: Object (name-value pairs), Array (ordered list)





EXTRA MATERIAL

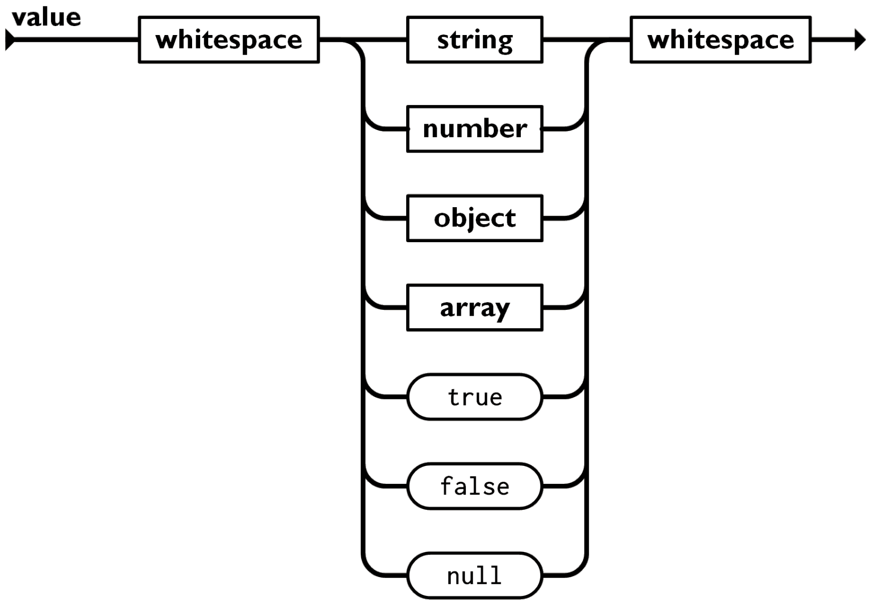
JSON standard (json.org)

- Possible JSON representation describing pets in our pet store

Array of objects

```
[  { "id": 9222968140497189000,
     "category": {  "id": 42,  "name": "Chihuahua" },
     "name": "doggie",
     "photoUrls": [ "http://..." ],
     "tags": [ {  "id": 0,  "name": "small" } ],
     "status": "available"
   },
   { "id": 9222968140497189111,
     "category": { … },
     "name": "fish",
     "photoUrls": [ "http://..." ],
     "tags": [ {  "id": 0,  "name": "small" } ],
     "status": "sold"
   },  …
]
```

Key-value pairs



Error handling

- **Examples of issues**
  - An operation is called with invalid parameters
  - The call does not return anythong
    - The component cannot handle the request in the current state
    - Hardware/software errors prevent successful execution
    - Misconfiguration issues (e.g., the server is not correctly connected to the database)
- **Possible reactions**
  - Raising an exception
  - Returning an error code
  - Log the problem

Multiple interfaces and separation of concerns

POLITECNICO MILANO 1863

- A server can offer multiple interfaces at the same time
- This enables
  - Separation of concerns
  - Different levels of access rights
  - Support to interface evolution (see next)

Interface evolution

POLITECNICO
MILANO 1863

- Interfaces constitute the contract between servers and clients
- Sometimes interfaces need to evolve (e.g., to support new requirements)
- Strategies to support continuity
  - **Deprecation**: declare well in advance that an interface version will be eliminated by a certain date
  - **Versioning**: maintain multiple active versions of the interface
  - **Extension**: a new version extends the previous one

POLITECNICO
MILANO 1863

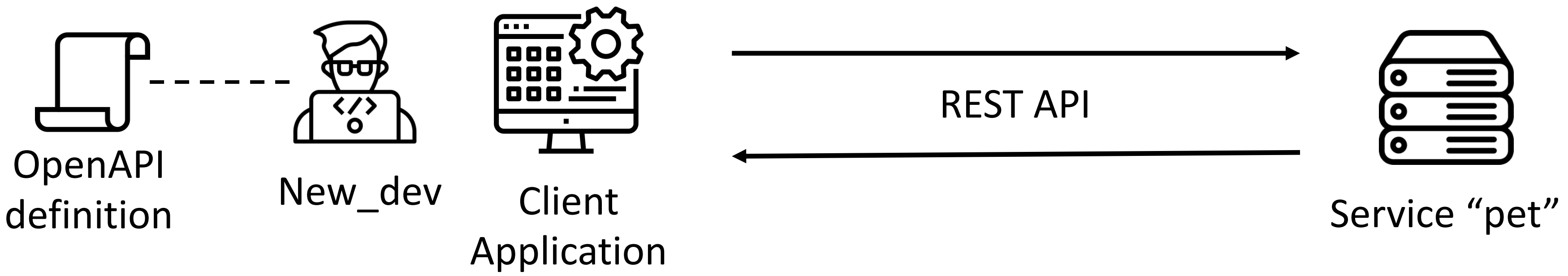
Software Design

Interface Documentation

Documenting interfaces

- Interface documentation
  - explain how to use it
  - should not include information about the internals of a component (otherwise users may assume behavior that could change in the future)
- Audience of an interface documentation
  - **Developers** and maintainers of the component **offering** the interface
  - **Developer** and maintainers of the component **using** the interface
  - **QA teams** for system **integration and testing**
  - **Software architects** (including those looking for reusable components)

OpenAPI Specification



- Scenario
  - **New_dev** should get started developing and maintaining the application using the "pet" service
  - How can New_dev **understand the API** and become productive **without digging** into the sources of "pet"?
    - New_dev can refer to the so-called OpenAPI definition

OpenAPI Specification

POLITECNICO
MILANO 1863

- OpenAPI Specification
  - **Defines** how to describe a REST API interface through an **OpenAPI definition**
- OpenAPI definition
  - File (JSON or YAML)
  - Describes what a service can do using its interface
- Benefits
  - Standardized format, public, and well-known
  - Readable by
    - **humans** to **understand** and **use** the REST API
    - **machines** to **automate** tasks like testing or code generation

OpenAPI Specification

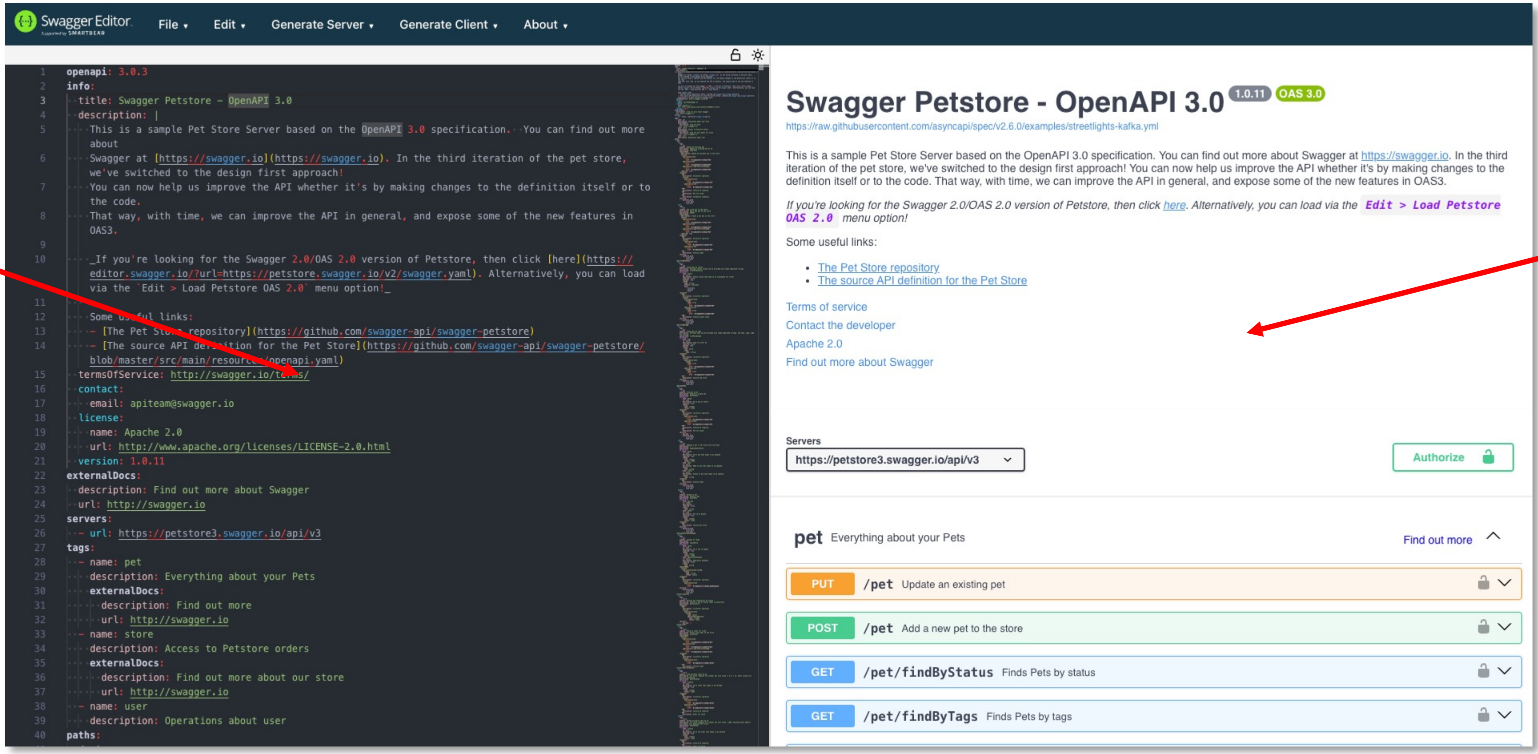
- OpenAPI definition describes
  - Endpoints
  - Resources
  - Operations
  - Parameters including data types
  - Authentication/authorization mechanism
- Tool support
  - API **validator** — conformance to the standard
  - **Documentation** generator — clear human readable description
  - **SDK** generator — automated creation of client libraries in a programming language of choice

OpenAPI Spec — petstore.yaml

Try it out at:

https://editor.swagger.io/?_ga=2.155528316.612942566.1697644409-1947143061.1697644409

OpenAPI definition (YAML file)



Generated documentation

EXTRA MATERIAL

OpenAPI Spec — petstore.yaml

OpenAPI version

Basic REST API info

```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about Swagger at [https://swagger.io](https://swagger.io).

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master/src/main/resources/openapi.yaml)
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
```

version of the API itself

Textual description

Base URL of the endpoints

EXTRA MATERIAL

OpenAPI Spec — petstore.yaml

Description of API requests (“paths” section)

```yaml
paths:
...
  /pet/findByStatus:
    get:
      tags:
        - pet
      summary: Finds Pets by status
      description: Multiple status values can be provided with comma separated strings
      operationId: findPetsByStatus
      parameters:
        - name: status
          in: query
          description: Status values that need to be considered for filter
          required: false
          schema:
            type: string
            default: available
            enum:
              - available
              - pending
              - sold
```

Resource

HTTP method

Textual description

Can be also header/path

Parameters

Enumerative type

EXTRA MATERIAL

OpenAPI Spec — petstore.yaml

Possible responses

```yaml
  /pet/findByStatus:
...
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
            application/xml:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid status value
      security:
        - api_key: []
```

Ok response

Either JSON
or XML format
depending on request

Each item of the array
has the referenced schema

Error in case of bad request

Authentication with API key

mandatory

```
Pet ⌄ {
    id                  > [...]
    name*               > [...]
    category            Category > {...}
    photoUrls*          > [...]
    tags                > [...]
    status              > [...]
}
```

array

object

Generated representation
of the pet schema

EXTRA MATERIAL

OpenAPI Specification

POLITECNICO
MILANO 1863

- Automated generation of client code example
  - Codegen tool: https://swagger.io/tools/swagger-codegen/

```
swagger-codegen generate -i https://petstore.swagger.io/v2/swagger.json \
    -l python -o /petstore
```

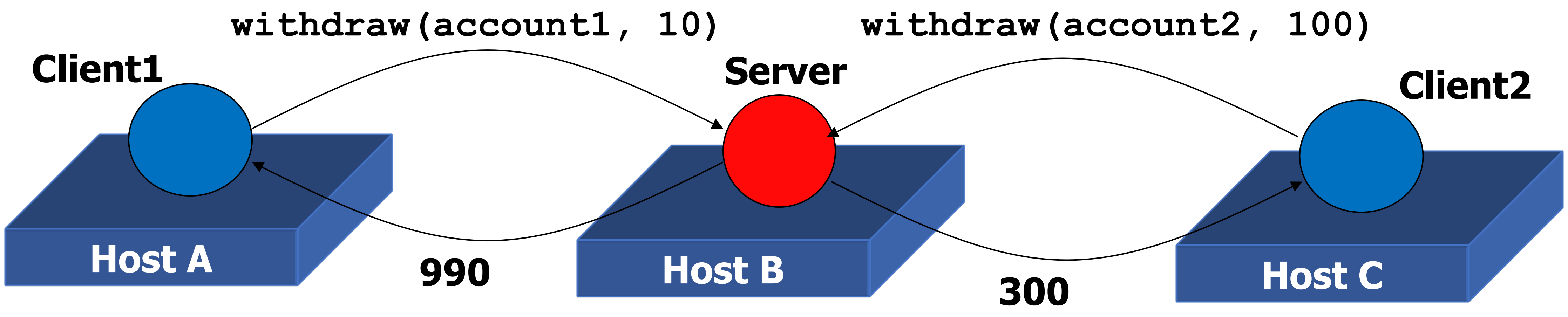
```python
configuration = swagger_client.Configuration()
# create an instance of the API class
api_instance = swagger_client.PetApi(swagger_client.ApiClient(configuration))
status = ['pending'] # list[str] | Status values that need to be considered for filter

try:
    # Finds Pets by status
    api_response = api_instance.find_pets_by_status(status)
    pprint(api_response)
except ApiException as e:
    print("Exception when calling PetApi->find_pets_by_status: %s\n" % e)
```

Software Design

Clinet-server style: handling multiple requests

Handling multiple requests

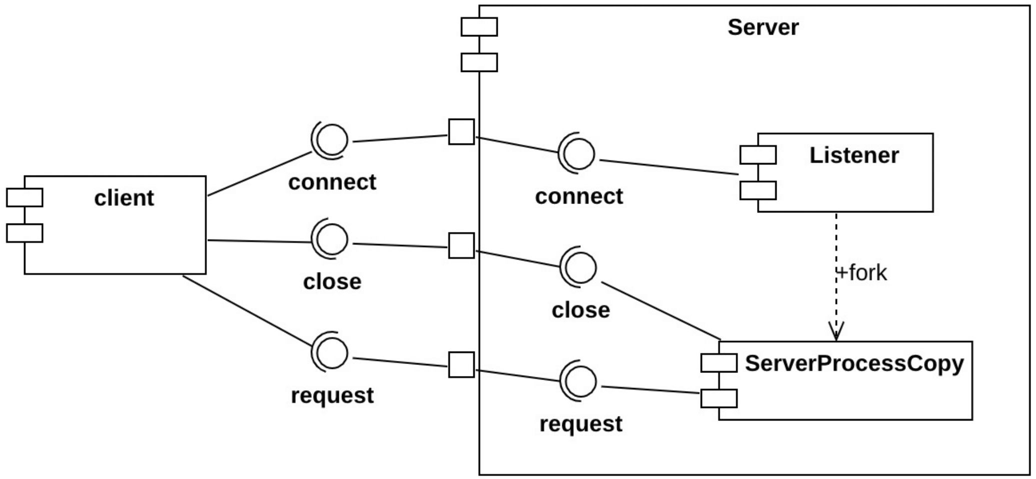
- The server must be able to receive and handle requests from multiple clients, example:

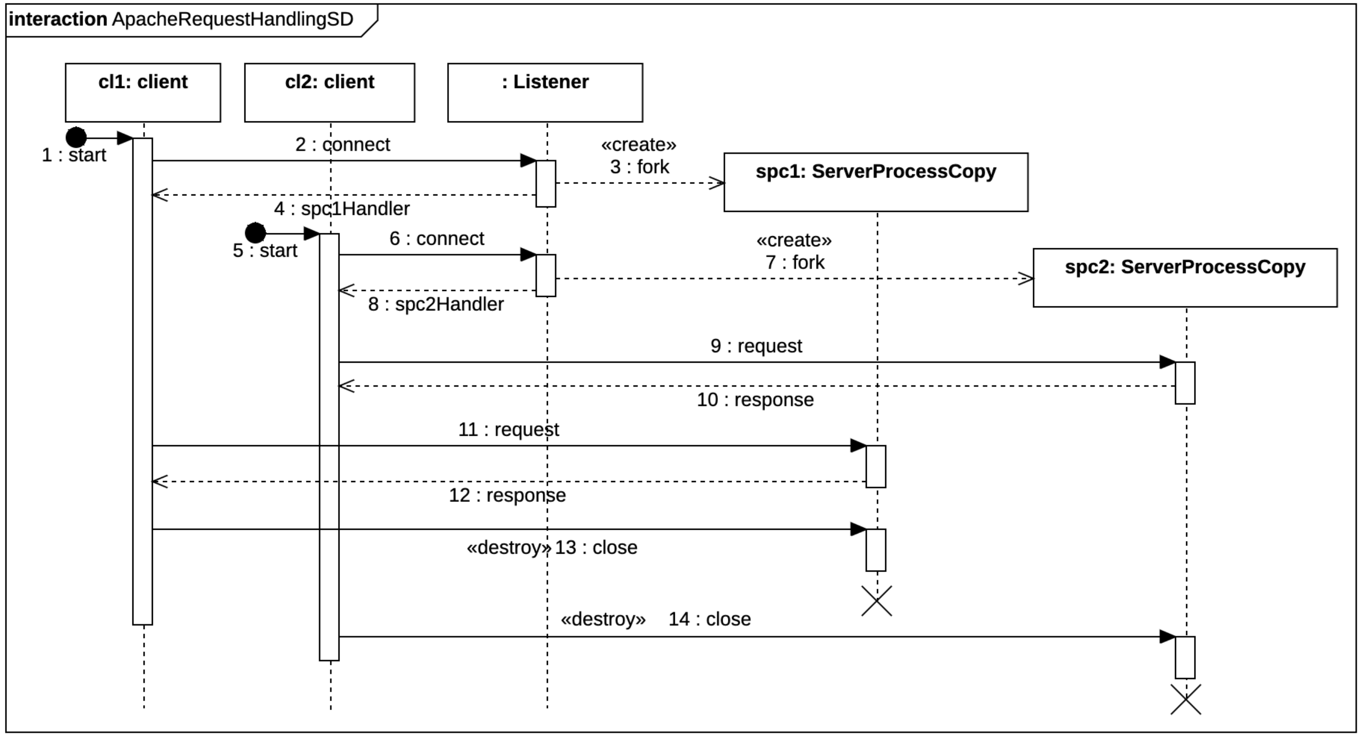


- How can the server serve the second `withdraw` if the first one is still ongoing?

Handling multiple requests: a first approach (forking)

- Simple approach used by Apache Web Server
  - **One process per request** or per client





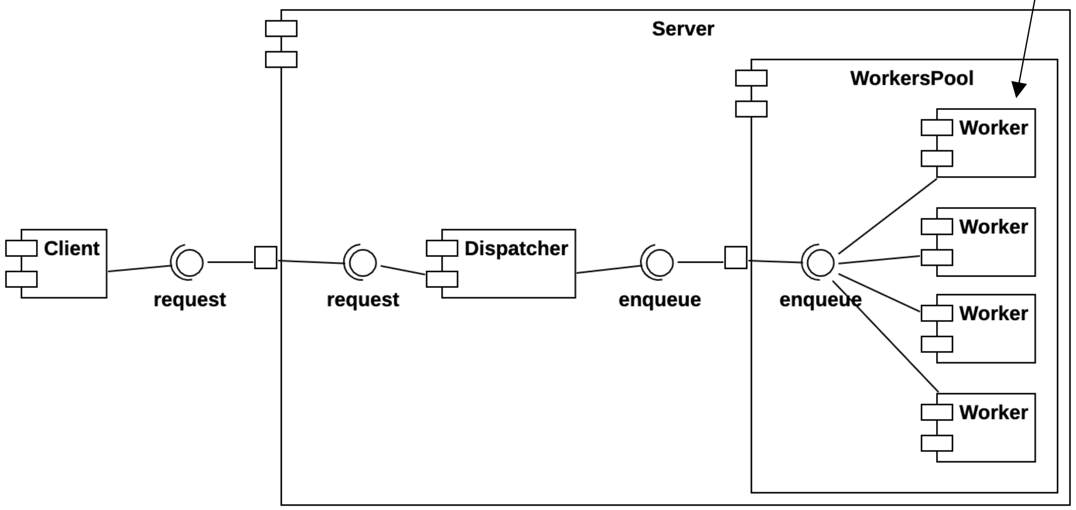
Handling multiple requests: a first approach (forking)

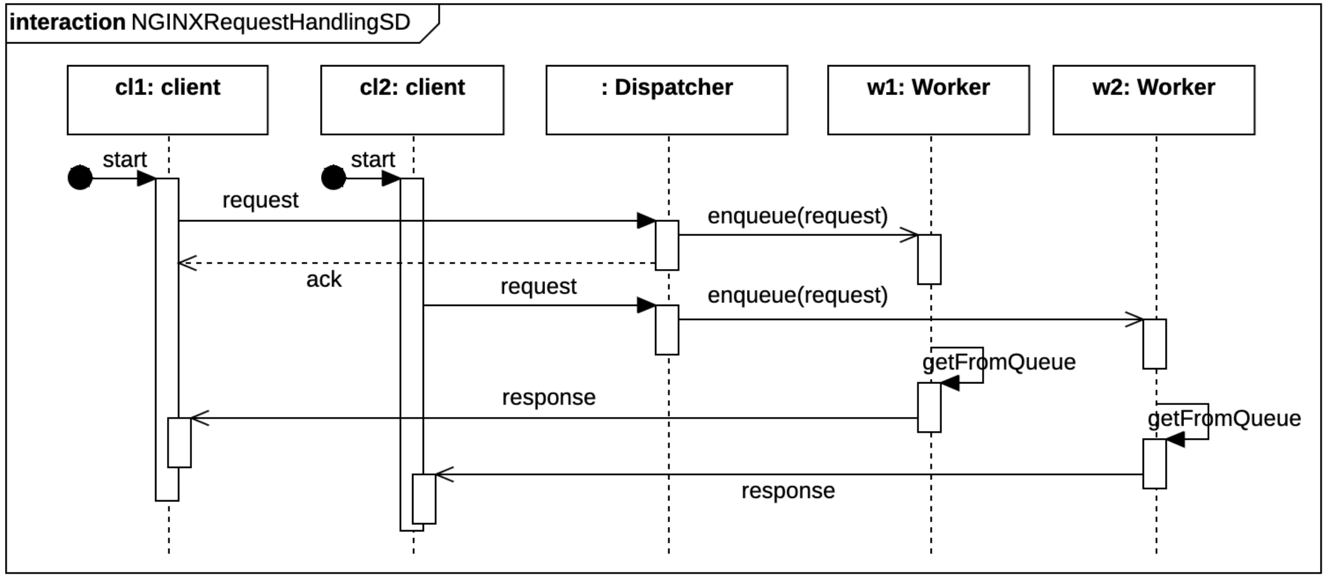
- Strengths
  - Architectural simplicity
  - Isolation and protection given by the “one-connection-per-process” model
    - Slow processes do not affect other incoming connections
  - Simple to program
  - Effective solution till ~2000
- Issues
  - Growth of the WWW over the past 20 years (#users and weight of web pages)
  - #active processes at time *t* difficult to predict and may saturate the resources
  - Expensive fork-kill operations for each incoming connection
  - → **Scalability issues!**

Handling multiple requests: alternative approach (worker pooling)

- Alternative approach adopted by NGINX Web Server
- Designed for high concurrency — deals with scalability issues

Predefined range of instances (n..m)





Handling multiple requests: alternative approach (worker pooling)

- Note (1): NGINX tackles the previous issues by introducing a new **architectural tactic**
  - **Tactic = Design decisions that influence the control of one or more quality attributes**
- Strengths of the new approach
  - Number of workers is fixed → they do not saturate available resources
  - Each worker has a queue
  - When queues are full the dispatcher drops the incoming requests to keep high performance
  - Dispatcher balances the workload among available workers according to specific policies

Handling multiple requests: alternative approach (worker pooling)

- Note (2): architectural tactics introduce **quality attribute trade-offs**
- NGINX decided to **optimize scalability** and **performance** by **sacrificing availability (in some cases)**
  - When all worker queues are full, the dispatcher drops incoming requests rather than spawning new workers

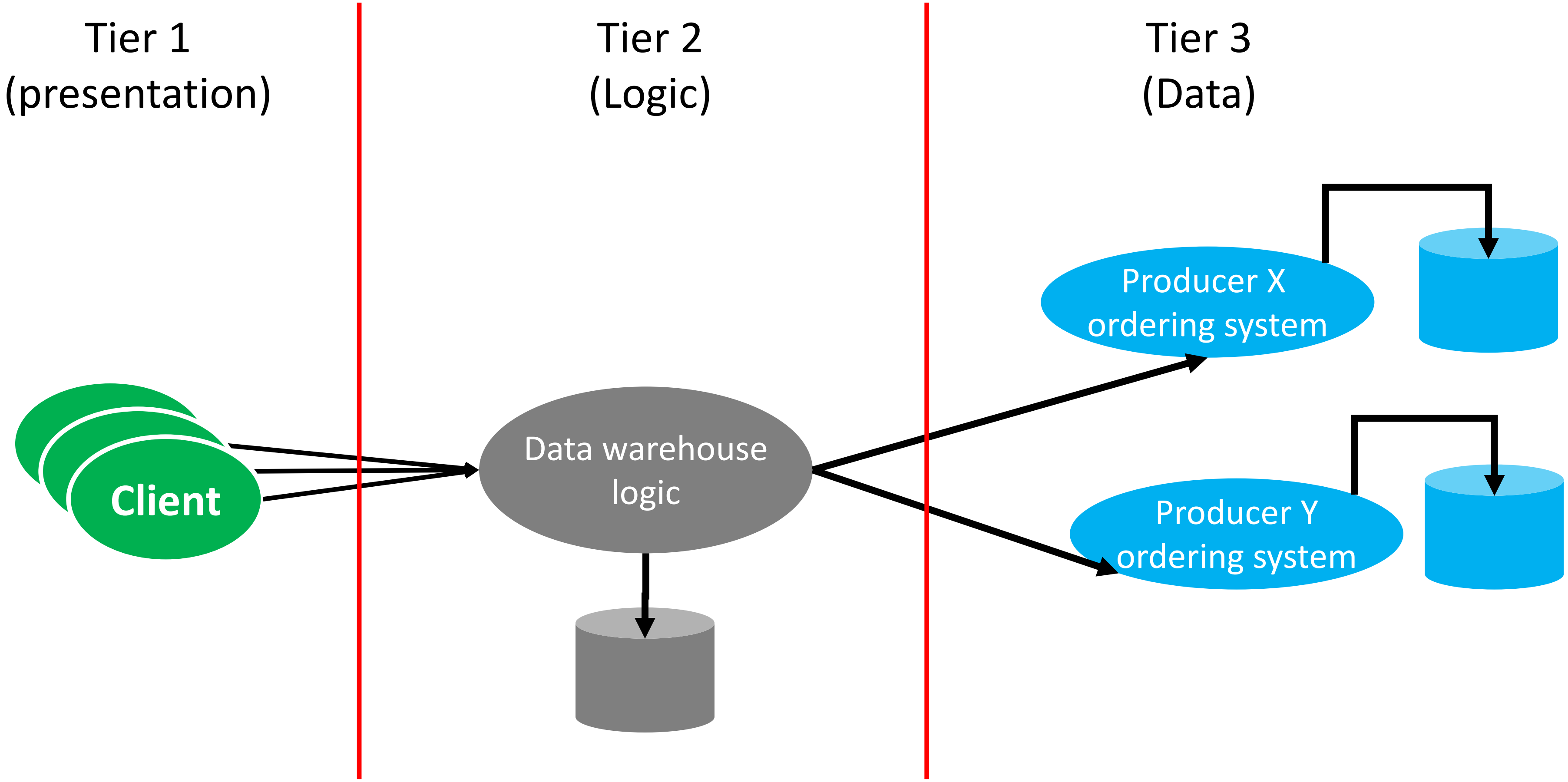
POLITECNICO
MILANO 1863

Software Design

Other styles: Multi-tier, Event-driven

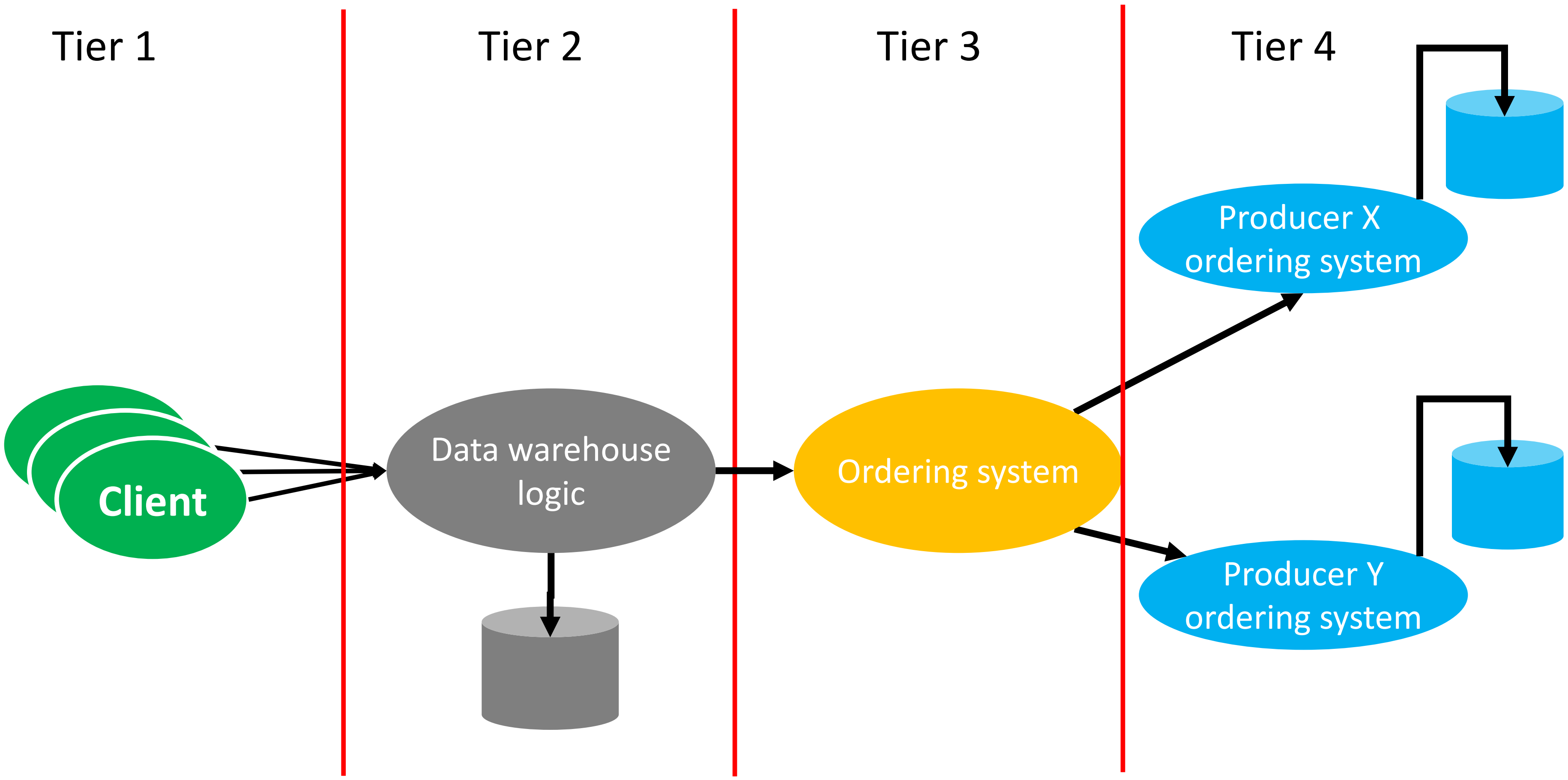
Three-tier Architecture – an example

Data warehouse for a supermarket



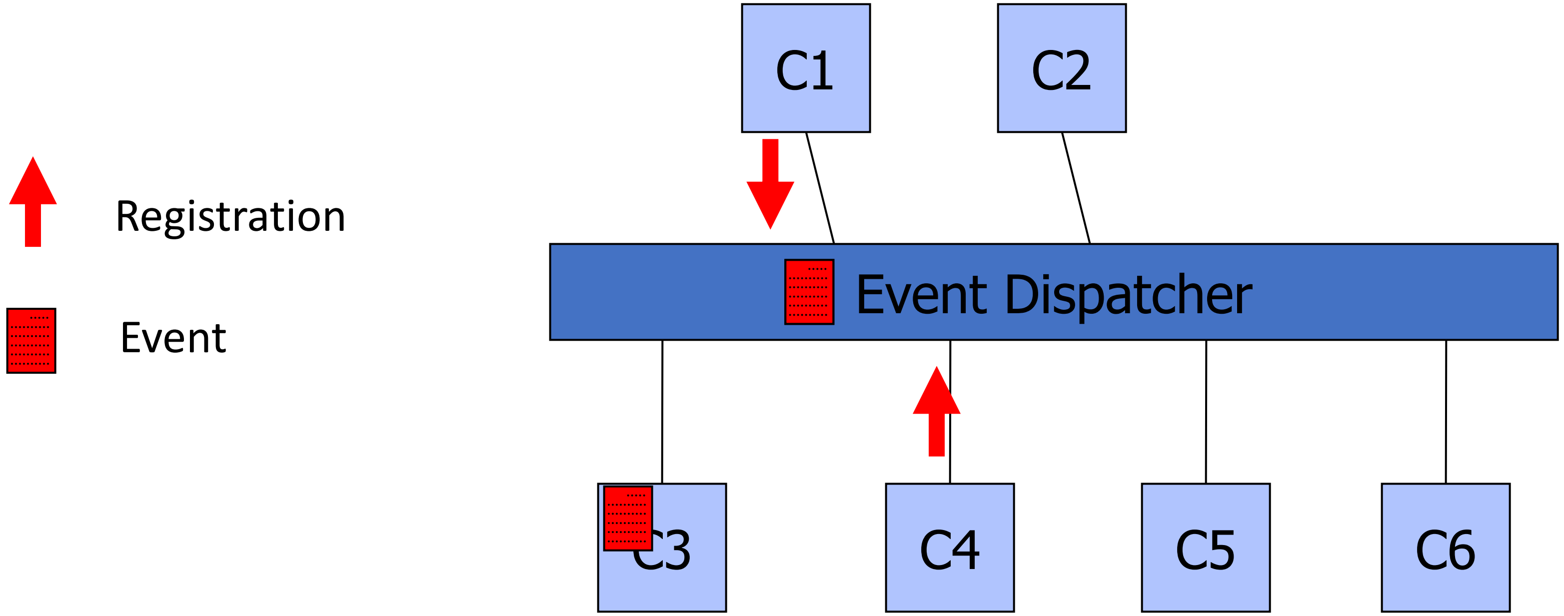
From 3 to N tiers

Data warehouse for a supermarket



Event-driven Architecture

POLITECNICO MILANO 1863



Logical view

- Components can **register** to / **send events**
- Events are **sent to all registered components**
- **No explicit naming** of target component

Event-driven paradigm

- Often called publish-subscribe
  - **Publish** = event generation
  - **Subscribe** = declaration of interest
- Different kinds of event languages possible

Event-driven systems Pros & Cons

+ **Very common in modern development practices**
  + E.g., continuous integration / deployment (such as GitHub Actions)
+ **Easy addition/deletion of components**
  + Publishers/subscribers are decoupled
  + The event dispatcher handles this dynamic set

– **Potential scalability problems**
  – The event dispatcher may become a bottleneck (under high workload)

– **Ordering of events**
  – Not guaranteed, not straightforward

Other characteristics

- Messages/events are asynchronous (send and forget)
- Computation is reactive (driven by receipt of message)
- Destination of messages determined by receiver, not sender (location/identity abstraction)
- Loose coupling (senders/receivers added without reconfiguration)
- Flexible communication means (one-to-many, many-to-one, many-to-many)

Examples of relevant technologies

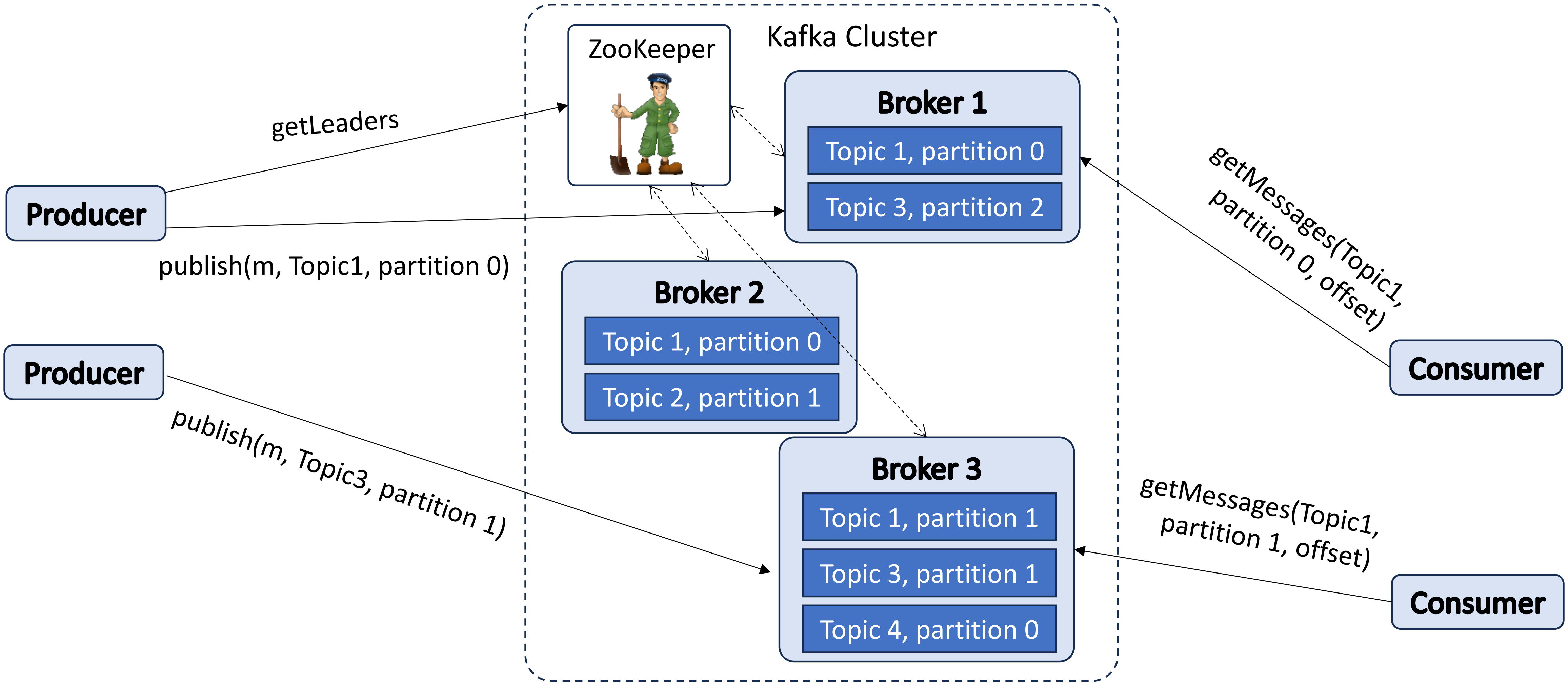
POLITECNICO MILANO 1863

- Apache Kafka
  - https://kafka.apache.org/
- RabbitMQ
  - https://www.rabbitmq.com/

Apache Kafka

- Kafka is a framework for the event-driven paradigm:
  - Includes primitives to create **event producers** and **consumers** and a runtime infrastructure to handle **event transfer** from producers to consumers
  - **Stores events** durably and reliably
  - Allows consumers to **process events** as they occur or retrospectively
- These services are offered in a distributed, highly scalable, elastic, fault-tolerant, and secure manner

Kafka architecture

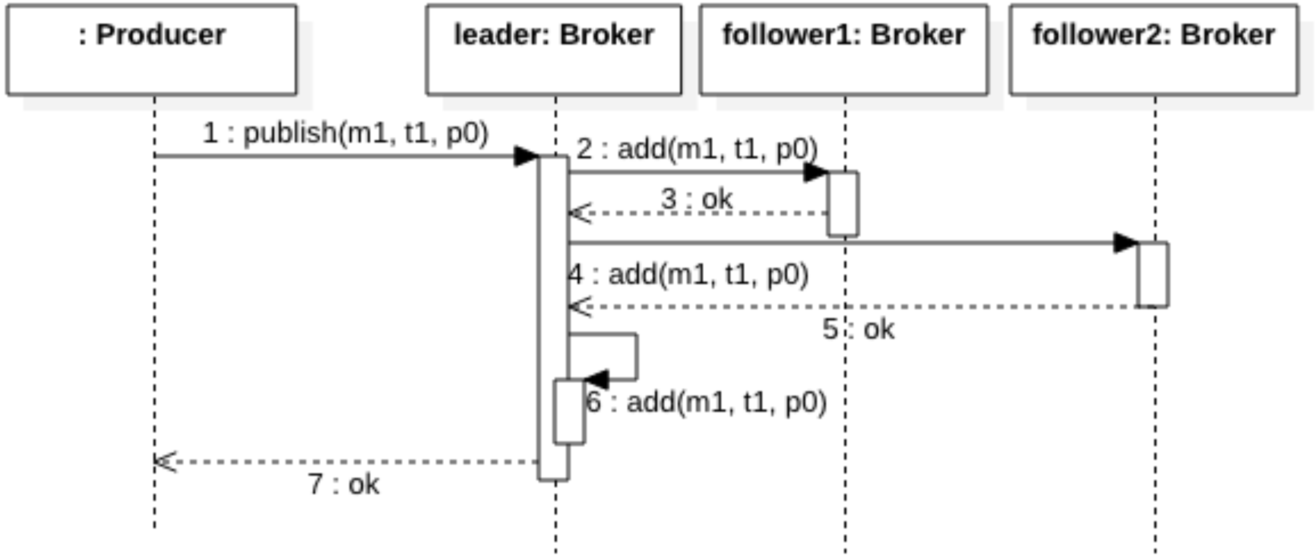


Kafka main characteristics

- Each broker handles a set of topics and topic partitions, parts including sets of messages on the topic
- Partitions are independent from each other and can be replicated on multiple brokers for fault tolerance
- There is one leading broker per partition. The other brokers containing the same partition are followers
- Producers know the available leading brokers and send messages to them
- Messages in the same topic are organized in batches at the producers' side and then sent to the broker when the batch size overcomes a certain threshold
- Consumers adopt a pull approach. They receive in a single batch all messages belonging to a certain partition starting from a specified offset
- Messages remain available at the brokers' side for a specified period and can be read multiple times in this period
- The leader keeps track of the in-synch followers
- ZooKeeper is used to oversee the correct operation of the cluster. All brokers send heartbeat to ZooKeeper. ZooKeeper will replace a failing broker by electing a new leader for all the partitions the failing broker was leading. It may also start/restart brokers

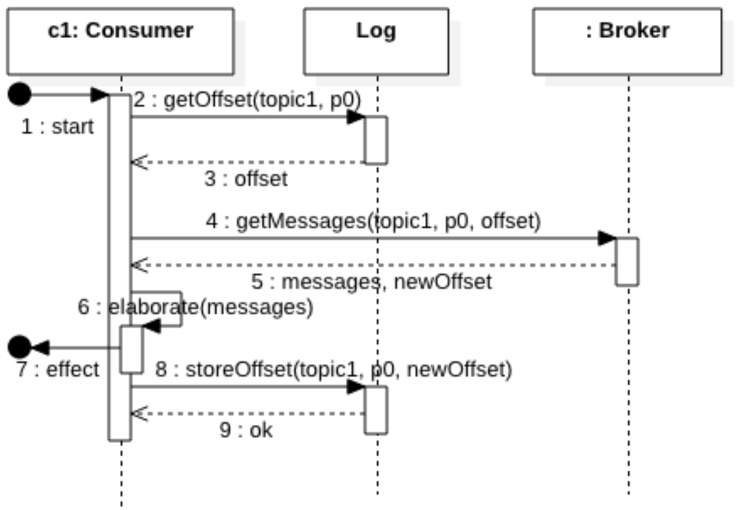
Message delivery semantics — Producer

- Brokers commit messages by storing them in the corresponding partition
- Leader adds the message to followers (replicas) if available
- **Issue**: in case of failure, the producer may not get the response (msg #7)
    - The producer has to resend the message
    - Kafka brokers can identify and eliminate duplicates
- Synchronization with replicas can be transactional
    - Exactly-once semantics is possible but long waiting time
    - At-least-once can be chosen by excluding duplicates' management
    - At-most-once can be chosen by publishing messages asynchronously



Message delivery semantics — Consumer

- Each **consumer** can rely on a **persistent log** to keep track of the **offset** so that it is not lost in case of failure
- If the consumer fails after having elaborated messages and before storing the new offset in the log, the same messages will be retrieved again
  - → at-least-once semantics
- The delivery semantics can be changed if the new offset is stored before the elaboration
  - → at-most-once semantics because, if failing after storing the offset, the effect of the received messages does not materialize
- Transactional management of log allows for exactly-once semantics



Kafka architectural tactics

POLITECNICO
MILANO 1863

- Scalability
  - **Multiple partitions and multiple brokers** → possibility to distribute producers/consumers on different partitions handled by different brokers
  - **Scale of operation**: Kafka supports the creation of clusters of brokers
    - Each clusters includes up to 1K brokers able to handle trillions of messages per day
- Fault tolerance
  - Partitions are **stored persistently**
  - **Replication** significantly reduces the chances of losing data
  - Cluster management takes care of restarting brokers and setting leaders as needed