UNIVERSITY OF PADOVA
DEPARTMENT OF INFORMATION ENGINEERING

Biomedical Wearable Technologies
for Healthcare and Wellbeing

Representational State Transfer (REST)

A.Y. 2023-2024

Giacomo Cappon

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Representational State Transfer (REST)

- **Representational state transfer (REST)**: a software **architectural style** created to guide the design and development of the World Wide Web.
- REST defines a set of **constraints** for how the architecture of an Internet-scale distributed hypermedia system (e.g., the Web) should behave.
- Introduced in 2000 by **Roy Fielding** (co-founder of the Apache HTTP Server project) in his doctoral dissertation (University of California, Irvine).
- The name "Representational State Transfer" evokes an image of how a well-designed Web application behaves:
  - a Web application forms a virtual state machine
  - the user progresses through the application by selecting a link or submitting a data-entry
  - each user action results in a transition to the next state of the application → a representation of that state is transferred to the user.

Architectural style

- **Architecture:**
  - how system elements are identified and allocated
  - how they interact to form a system
  - the amount and granularity of communication needed for interaction
  - the interface protocols used for communication.

- **Architectural style:** a set of architectural constraints that restricts the roles and features of architectural elements and the allowed relationships among those elements.

REST: goal and constraints

- REST is a set of architectural constraints.
- REST goal: to **minimize latency and network communication**, while at the same time **maximizing the independence and scalability of components**.
- This is achieved by placing constraints on **the interface between components**, rather than on each component implementation.
- In total, **6 architectural constraints**:
  1. Client-server architecture
  2. Stateless communication
  3. Cache constraint
  4. Uniform interface between components
  5. Layered system
  6. Code-on-demand (optional)

Constraint 1: Client-server architecture

- **Principle of separation of concerns**: a design principle for separating a computer program into distinct sections, in which each section addresses a separate concern (i.e., a different task) → **modular system**

- The **client-server (CS)** architecture relies on the principle of separation of concern: separation of the user interface concerns (client) from the data storage concerns (server)

- Advantages of the client-server architecture
  - It improves the **portability** of the user interface across multiple platforms
  - It improves **scalability** by simplifying the server components
  - It allows the client and the server components to evolve **independently**

Constraint 2: Stateless communication

- Communication must be **stateless** in nature: the server does not store any state about the client session (i.e., the history of requests) on the server-side.
- **Each request** from client to server must be **standalone**
  - It must contain **all information necessary** to understand the request
  - It cannot take advantage of any stored information on the server.
- The **session state** (i.e., the history of requests) is kept entirely **on the client**.
  - If the server needs any information about previous requests to process the current request, then the client must provide this information in the current request.
- Advantages:
  - Improved **visibility**: no need to look beyond the current request to determine its full nature.
  - Improved **reliability**: recovering from partial failures is easier.
  - Improved **scalability**: the server does not have to manage resource usage across requests
- Tradeoff: it may decrease network **performance** by increasing the repetitive data sent in a series of requests

Constraint 3: Cache constraint

- The data within a response to a request must be implicitly or explicitly **labelled as cacheable or noncacheable**.

- If a response is cacheable, a client cache can reuse that response data for later, equivalent, requests.

- Advantage:
  - Improved **efficiency**: it partially or completely **eliminate some interactions.**

- Tradeoff:
  - **Decrease reliability,** if data within the cache differs significantly from the data that would have been obtained with a new request to the server.

Constraint 4: Uniform interface between components

- The information is exchanged in the same way for every client-server interactions, whatever the client and the server are.

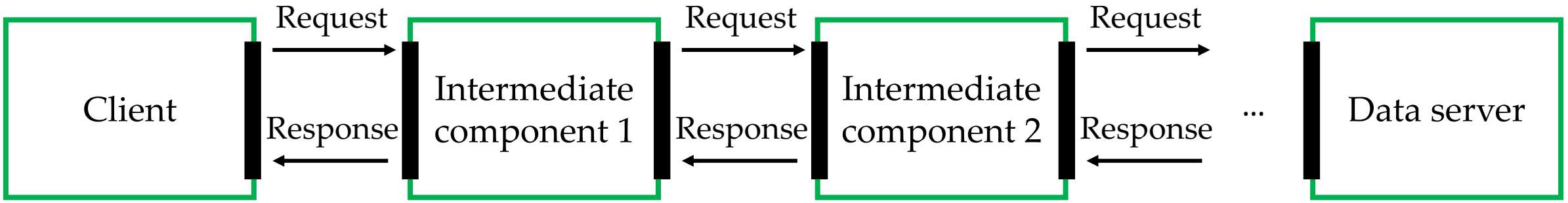
- There is a **common language** between servers and clients that allows each part to be substituted or modified without breaking the entire system.

- Advantage:
  - Independent **evolvability** of the system components: the implementation of interfaces is decoupled from the services they provide.

- Tradeoff:
  - Decreased **efficiency** for some applications: the information is transferred in a standardized form rather than one which is specific to an application's needs.

Constraint 5: Layered system style

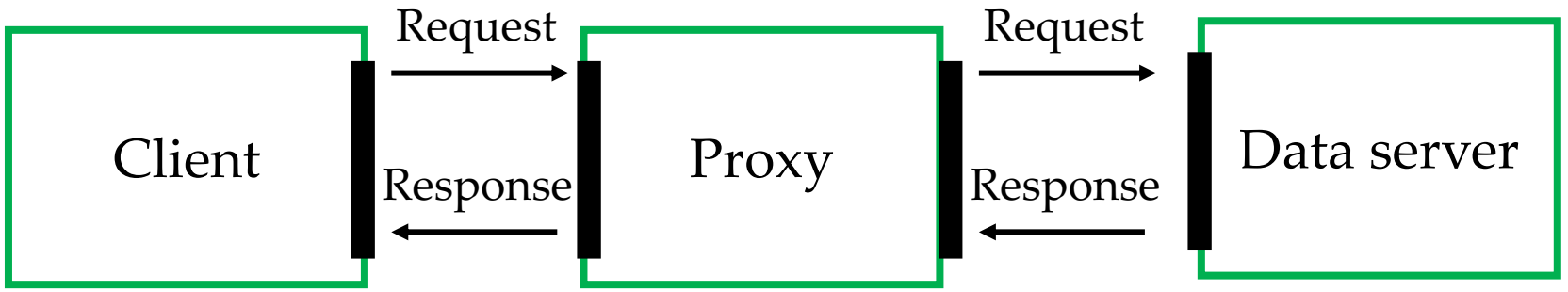
- The architect can inject "layers" of service between the server and the client to efficiently serve the client requests.
- The layering must be transparent to the client: each component cannot "see" beyond the immediate layer with which it is interacting → the client does not know if it is talking with final data server or an intermediate server.



- Advantage: by restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote component **independence**.
- Tradeoff: layers add **overhead** and latency to the processing of data, reducing user-perceived performance.

Example of layered system: proxy

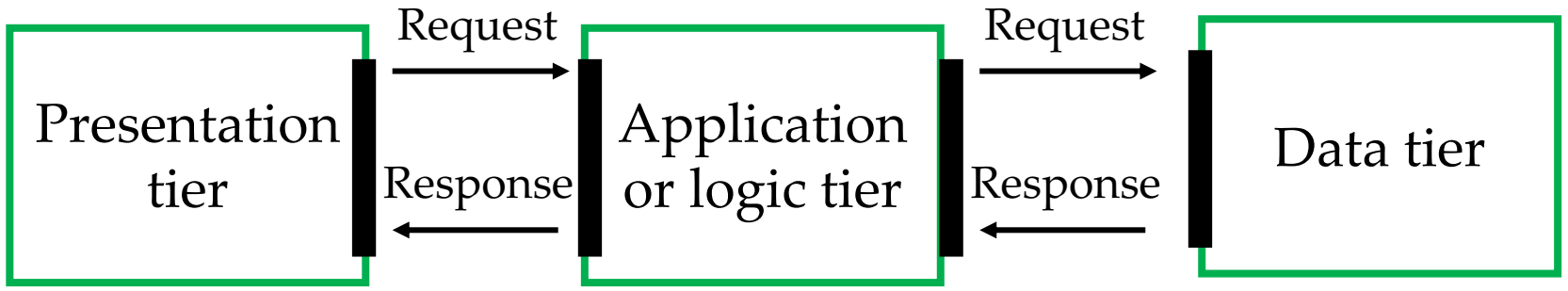
- **Proxy**: a server that acts as an intermediary between a client and a server. The client directs the request to the proxy server, which evaluates the request and performs the required network transactions.



- Possible uses of proxies:
  - Load balancing: a resource collection is stored in multiple servers, the proxy server receives all the requests to that resources and redirect each request to the competent data server.
  - Anonymous proxy: a server that forward the client's requests to the data server without revealing the IP address of the client that originated the request.
  - Caching: a proxy can be used to cache static contents to improve the network efficiency
  - Security: a proxy can implement the cryptographic protocols of secure websites

Example of layered system: three-tier application

- **Three-tier application**: a web application that consists of three tiers (or layers) implemented in physically different components



Constraint 6: Code-on-demand (optional)

- The client functionalities can be extended by downloading and executing code in the form of applets or scripts.

- Advantage:
  - Clients are **simplified** by reducing the number of features required to be pre-implemented.
  - Allowing features to be downloaded after deployment improves system **extensibility**.

- This constraint is not mandatory but **optional**: the architecture should support this constraint in the general case, but this may be disabled within some contexts.

The definition of the uniform interface

- The REST uniform interface (constraint 4) is defined by putting 4 requirements on the interface between components (clients and servers):
  - Hypermedia as the engine of application state (HATEOAS)
  - Identification of resources
  - Manipulation of resources through representations
  - Self-descriptive messages

Hypermedia as the engine of application state

- Web applications are based on **distributed hypermedia**:
  - When a client interacts with a Web application, the application servers provide information dynamically through **hypermedia.**
  - **Hypermedia:** any content that contains links to some other form of media (e.g., an image, a movie, a text) → generalization of hypertext for a general media.
  - When a link is selected, information is moved from the location where it is stored in the server to the location where it will be used in the client → application state transition.
  - A REST client needs no prior knowledge about how to interact with an application server beyond a generic understanding of hypermedia.

Identification of resources

- **Resource**: any information or concept that can be named and be the target of a hypertext (a document, an image, a collection of other resources, a temporal service, e.g., "today's weather in Padova"...).
- More precisely, the resource is the conceptual mapping to an entity or a set of entities, it is not the entities that correspond to the mapping in a specific time.
- Formally, a resource R is a temporally varying membership function $M_R(.)$, which for time t maps to an entity or a set of entities.

$$E = M_R(t)$$

Entity Resource Time

- The **entity** is the specific realization of the concept identified by the resource.
- The entity to which a resource points is stored and manipulated in a specific format or representation → **resource representation.**
- Each resource is identified by a label called **resource identifier.**

Resources types

- Example:
  - «the first version of document X»
  - «the revision 1.0 of document X»
  - «the revision 1.1 of document X»
  - …
  - «the latest version of document X»

Static resources: they always point to the same entity (with a specific representation), in this case a precise document version.

Dynamic resource: the pointed entity can change over time. At some time, the resource «the latest version of document X» will point to the same entity of «the revision 1.1 of document X».

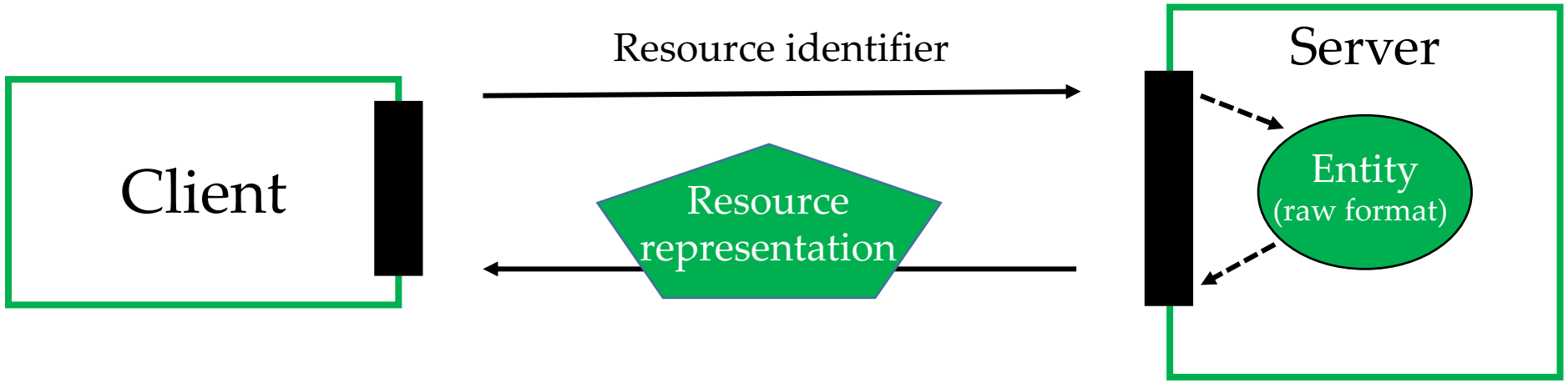
- A resource can also point to the empty set → this allows to map a concept, i.e., to make references to that concept, before any realization of that concept exists.

Resources

- The abstract definition of resources in REST allows to:
  - manage many sources of information without distinguishing them by type or implementation
  - it allows to reference the concept rather than some singular representation of that concept → no need to change all existing links whenever the representation changes
  - late bind specific representations to resources, enabling content negotiation to take place based on characteristics of the request, rather than on the specific content to be requested

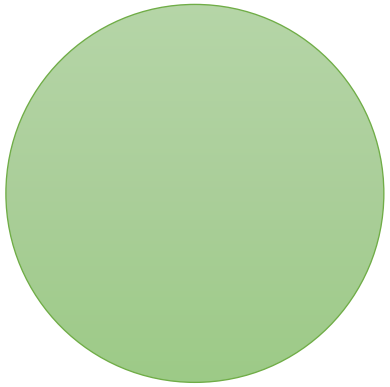
Resource representations

- **Resource representation**: a view of the current state (or realization) of the resource at the time of the request.
- In a client-server interaction, the client asks to the server a resource (by the resource identifier), and the server answers by sending the corresponding resource representation.
- The representation sent to the client is not necessarily the same used to store the data in the server.

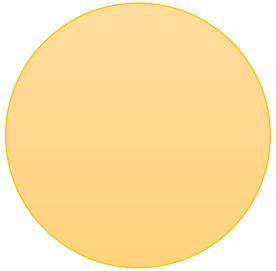


Representations: example

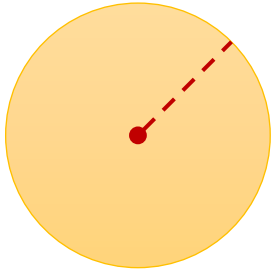
Resource: circle



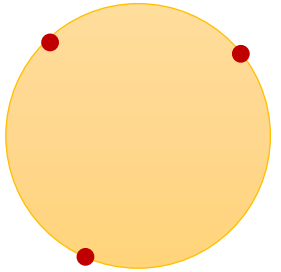
Entity: a circle realization



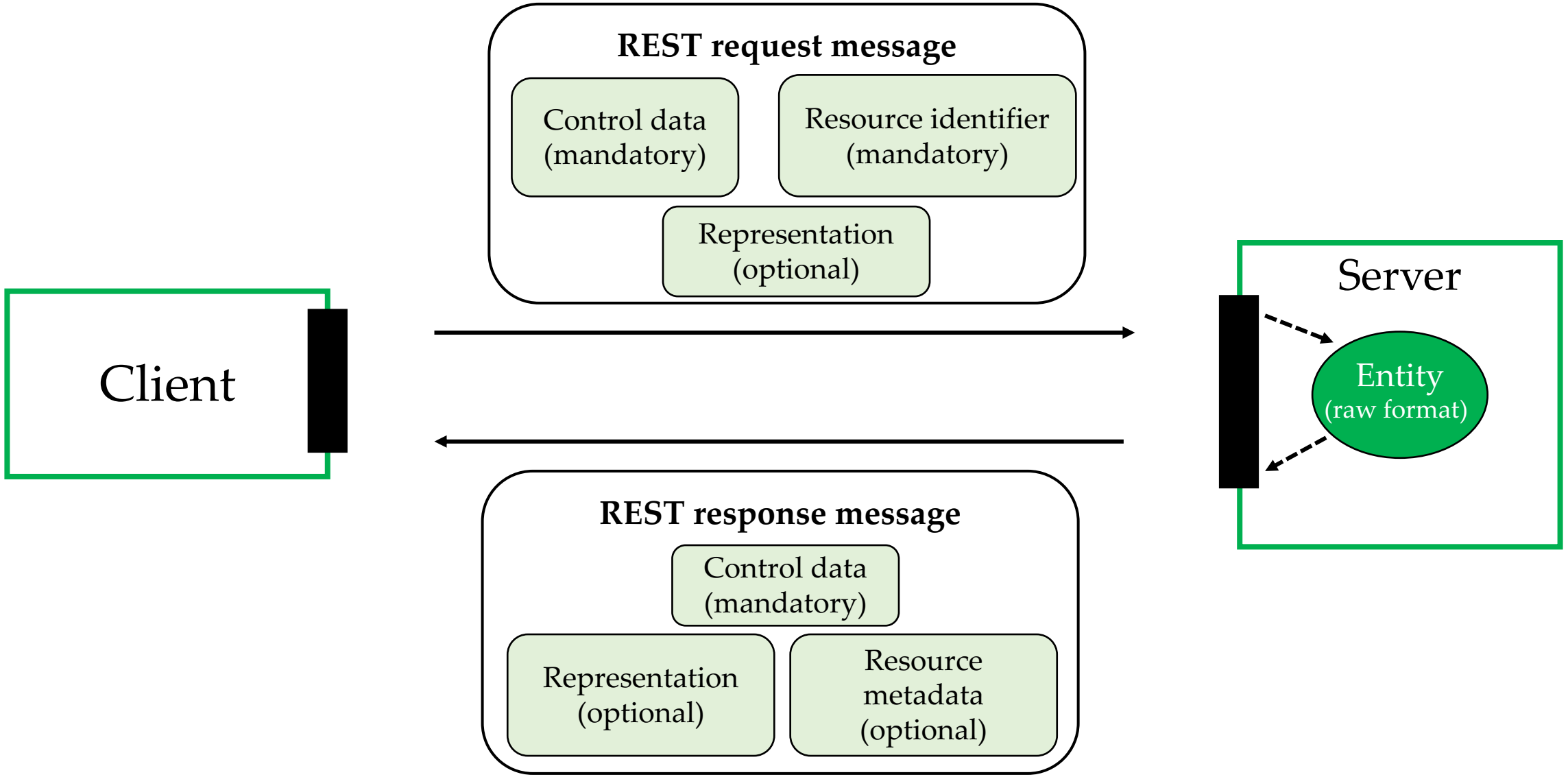
Representation 1: one point for the circle center and one vector for the circle radius in SVG format



Representation 2: coordinates of three points of the circumference in a csv file



REST self-descriptive messages



Resource representations and REST messages

- A resource **representation** includes:
  - **Data** (the actual representation)
  - **Metadata** (description of the representation) → typically, not displayed to the end user (e.g., data type, creation date, version number, …)
- The resource representation is provided to the client within a **response message** that contains some additional contents:
  - **Resource metadata (optional)**: additional information about the resource that exists on the server (e.g., alternate text to be displayed in the event an image representation cannot be displayed for some reason).
  - **Control data (mandatory)**: it deals with the validity of the resource and its representation on the client (e.g., if data is cacheable, the expiry time, a checksum for checking integrity, …).

Summary of REST

- An architectural style for **distributed hypermedia** systems
- 6 constraints
  - **Client-server** architecture
  - **Stateless** communication
  - **Cache** constraint
  - **Uniform interface**
  - **Layered** system
  - **Code-on-demand** (optional)

- The uniform interface is realized through the definition of abstract **resources**
  - Each resource is identified by a **resource identifier**
  - The realization of a resource is an **entity**
  - A resource is manipulated by resource **representations** that reflect the current realization of the resource
  - Representations can be sent from server to client (or viceversa) encapsulating them in a **self-explained message**
  - A request message must include **control data** (mandatory), resource identifier (mandatory) and representation's data and metadata (optional)
  - A response message must include control data (mandatory), representation's data and metadata (optional) and **resource metadata** (optional)

The REST paradigm increases performance, scalability, simplicity, modifiability, visibility, portability, and reliability of the system.

References

- Roy T. Fielding and Richard N. Taylor, “Principles and design of the modern Web architecture”, ACM Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150. https://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf