API Architectural Styles

4.3.1

Common Architectural Styles

The application defines how third parties interact with it, which means there's no "standard" way to create an API. However, even though an application technically can expose a haphazard interface, the best practice is to follow standards, protocols, and specific architectural styles. This makes it much easier for consumers of the API to learn and understand the API, because the concepts will already be familiar.

The three most popular types of API architectural styles are RPC, SOAP, and REST.

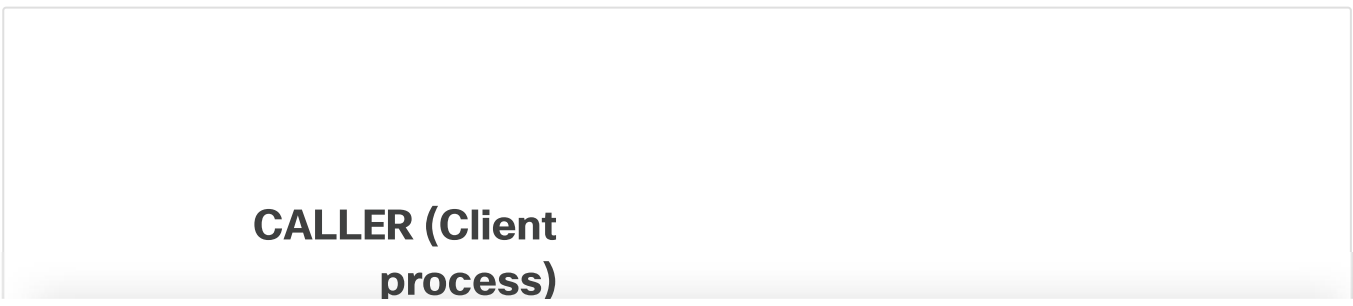
4.3.2

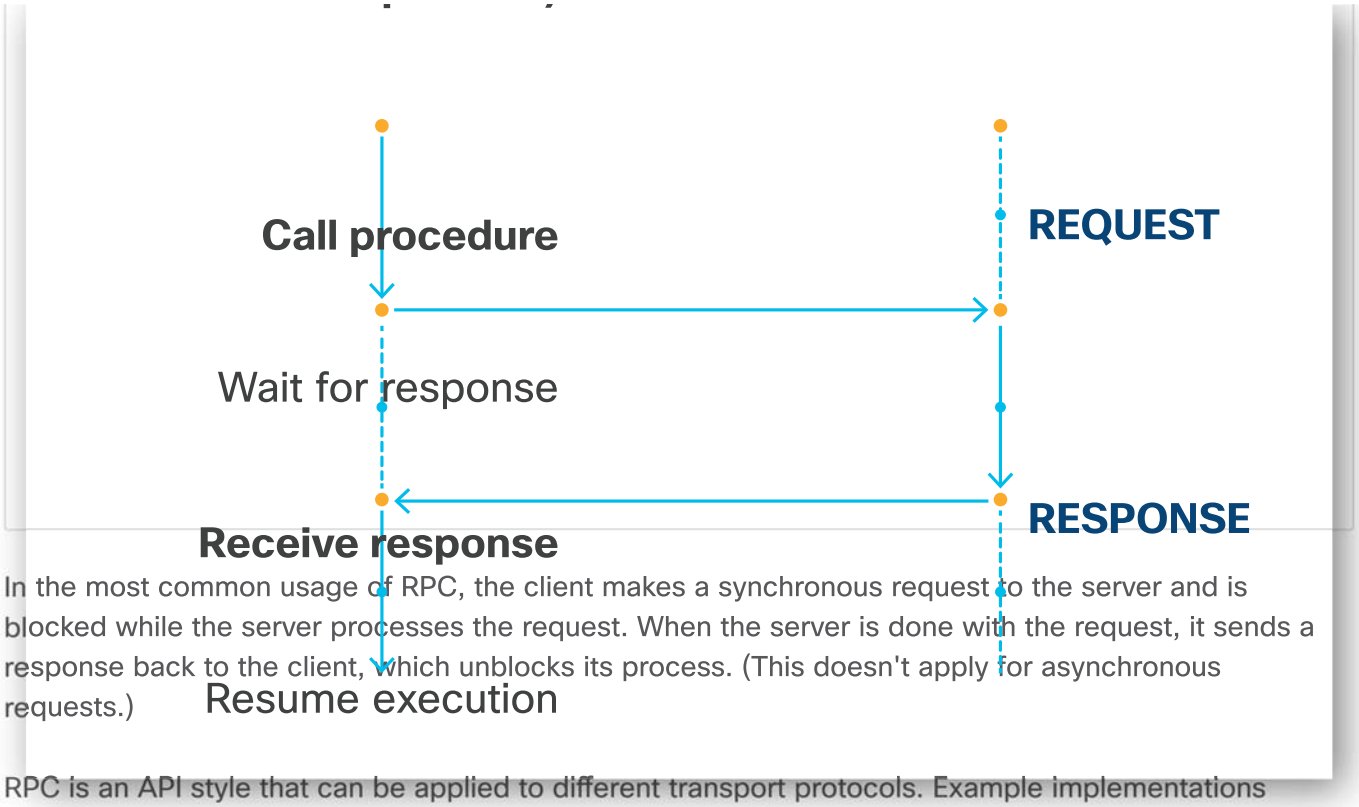
RPC

Remote Procedure Call (RPC) is a request-response model that lets an application (acting as a client) make a procedure call to another application (acting as a server). The "server" application is typically located on another system within the network.

With RPC, the client is usually unaware that the procedure request is being executed remotely because the request is made to a layer that hides those details. As far as the client is concerned, these procedure calls are simply actions that it wants to perform. In other words, to a client, a Remote Procedure Call is just a method with arguments. When it's called, the method gets executed and the results get returned.

Remote Procedure Call client-server request/response model





In the most common usage of RPC, the client makes a synchronous request to the server and is blocked while the server processes the request. When the server is done with the request, it sends a response back to the client, which unblocks its process. (This doesn't apply for asynchronous requests.)

RPC is an API style that can be applied to different transport protocols. Example implementations include:

- XML-RPC
- JSON-RPC
- NFS (Network File System)
- Simple Object Access Protocol (SOAP)

4.3.3

SOAP

SOAP is a messaging protocol. It is used for communicating between applications that may be on different platforms or built with different programming languages. It is an XML-based protocol that was developed by Microsoft. SOAP is commonly used with HyperText Transfer Protocol (HTTP) transport, but can be applied to other protocols. SOAP is independent, extensible, and neutral.

Independent

SOAP was designed so that all types of applications can communicate with each other. The applications can be built using different programming languages, can run on different operating systems, and can be as different as possible.

Extensible

SOAP itself is considered an application of XML, so extensions can be built on top of it. This extensibility means you can add features such as reliability and security.

Neutral

SOAP can be used over any protocol, including HTTP, SMTP, TCP, UDP, or JMS.

SOAP messages

A SOAP message is just an XML document that may contain four elements:

- Envelope
- Header
- Body
- Fault

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Header/>
    <soap:Body>
        <soap:Fault>
            <faultcode>soap:Server</faultcode>
            <faultstring>Query request too large.</faultstring>
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

Example of a SOAP message

Envelope

The `Envelope` must be the root element of the XML document. In the `Envelope`, the namespace provided tells you that the XML document is a SOAP message.

Header

The header is an optional element, but if a header is present, it must be the first child of the Envelope element. Just like most other headers, it contains application-specific information such as authorization, SOAP specific attributes, or any attributes defined by the application.

Body

The body contains the data to be transported to the recipient. This data must be in XML format, and in its own namespace.

Fault

The fault is an optional element, but must be a child element of the Body. There can only be one fault element in a SOAP message. The fault element provides error and/or status information.

4.3.4

REST

REpresentational **S**tate **T**ransfer (REST) is an architectural style authored by American computer scientist Roy Thomas Fielding in Chapter 5 of his doctoral dissertation, *Architectural Styles and the Design of Network-based Software Architectures*, in 2000. Roy Fielding defines REST as a hybrid style, derived from several network-based architectural styles he described elsewhere in the paper, "combined with additional constraints that define a uniform connector interface."

Note: Search for Fielding's dissertation on the internet for more information.

We'll examine REST in detail in the next two sections, but let's look at the basics.

In Fielding's dissertation, he establishes six constraints applied to elements within the architecture:

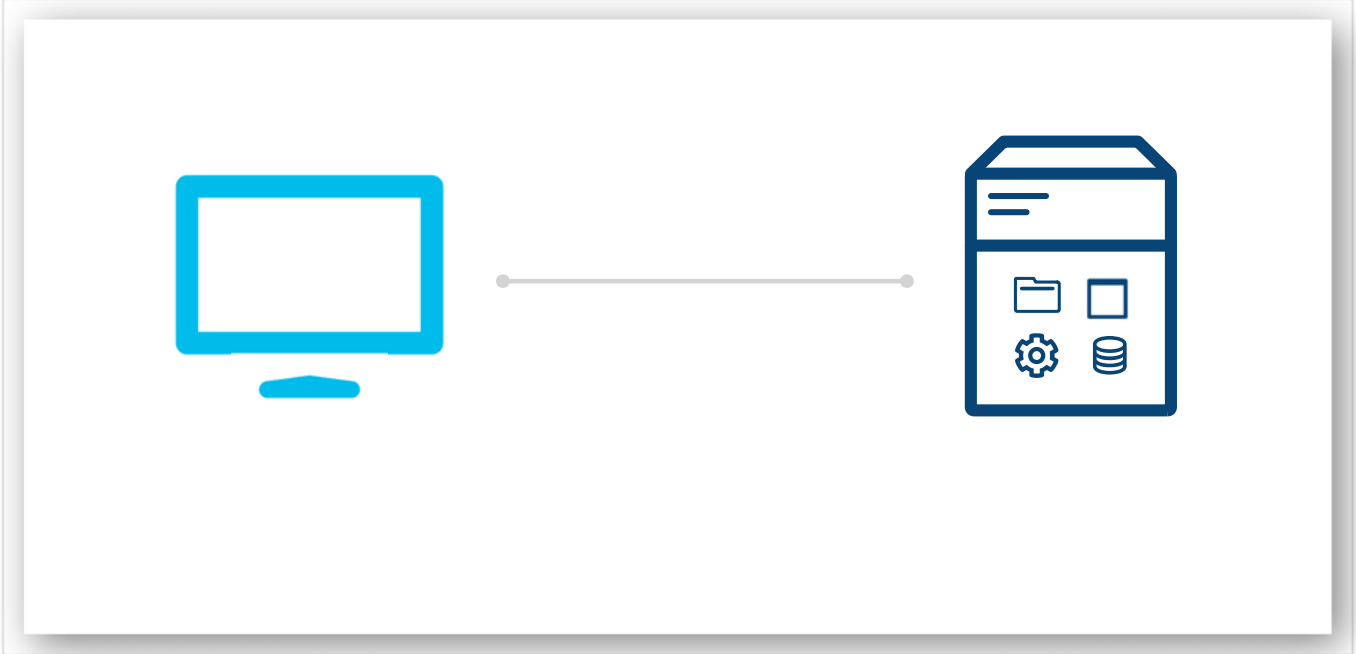
- Client-Server
- Stateless
- Cache
- Uniform Interface
- Layered System
- Code-On-Demand

These six constraints can be applied to any protocol, and when they are applied, you will often hear that it is *RESTful*.

Client-Server

The client and server should be independent of each other, enabling the client to be built for multiple platforms and simplifying the server side components.

REST client-server model

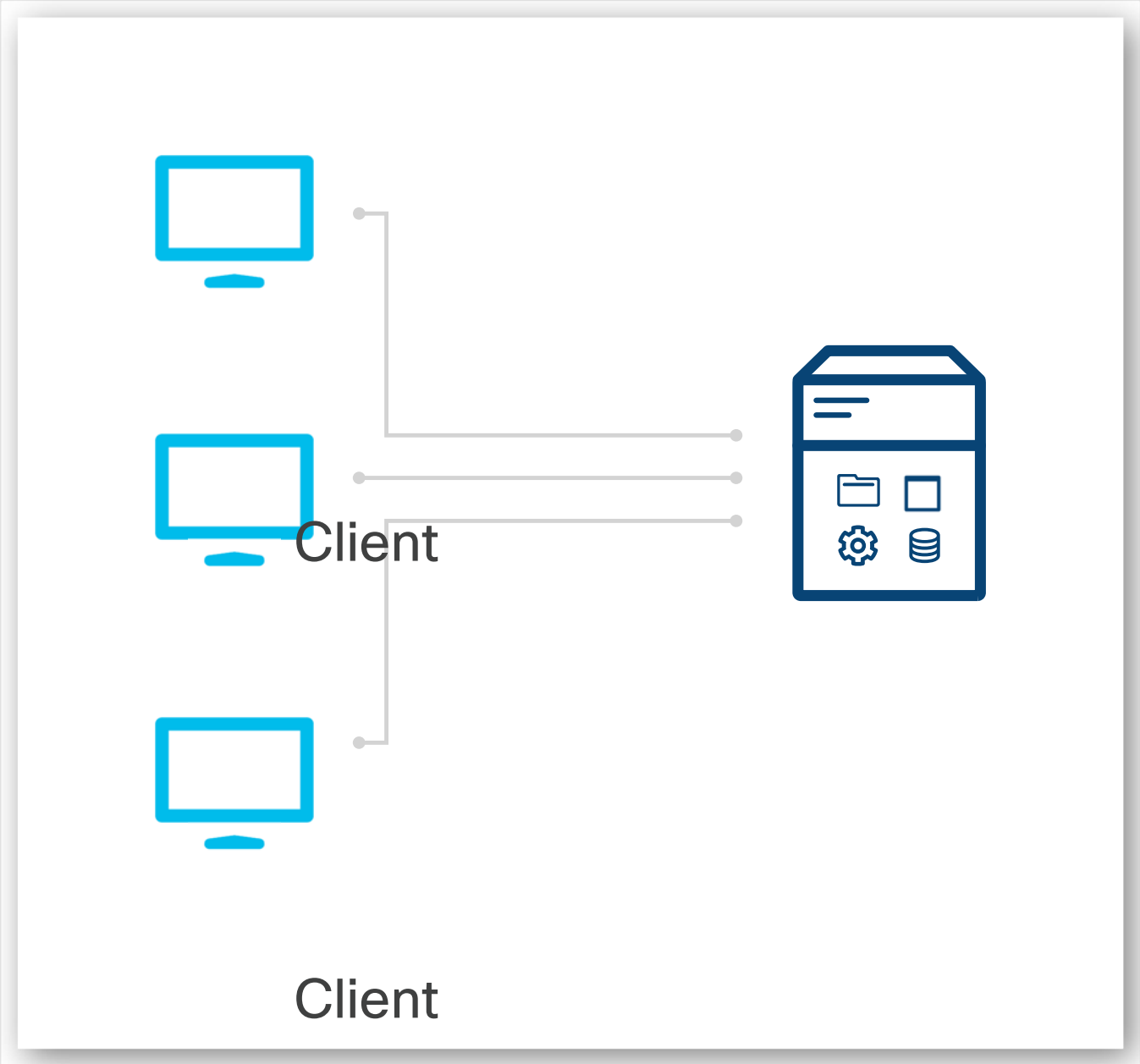


Stateless

CLIENT

Requests from the client to the server must contain all of the information the server needs to make the request. The server cannot contain session states.

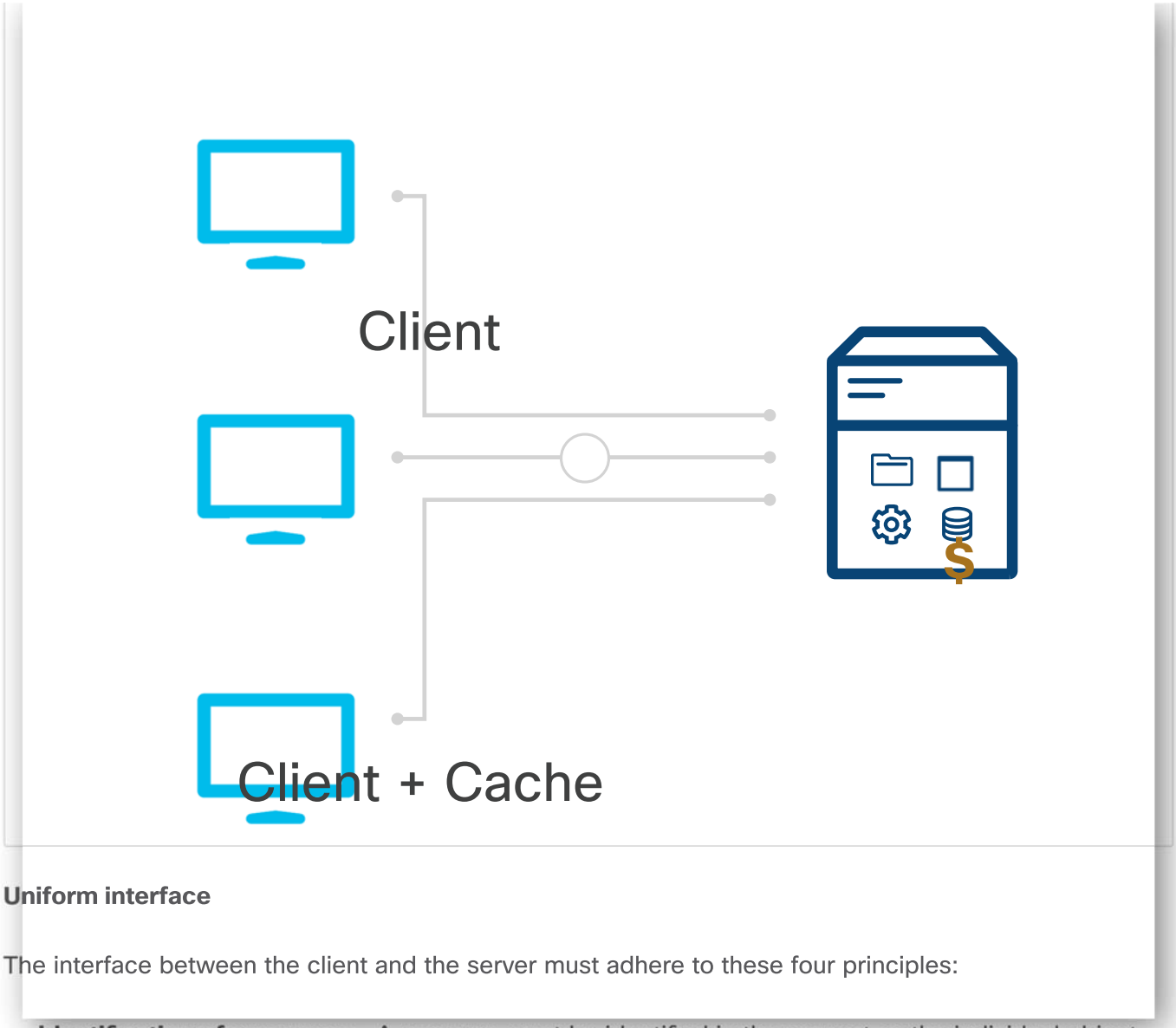
REST stateless model



Cache

Responses from the server must state whether the response is cacheable or non-cacheable. If it is cacheable, the client can use the data from the response for later requests.

REST cache model



Uniform interface

The interface between the client and the server must adhere to these four principles:

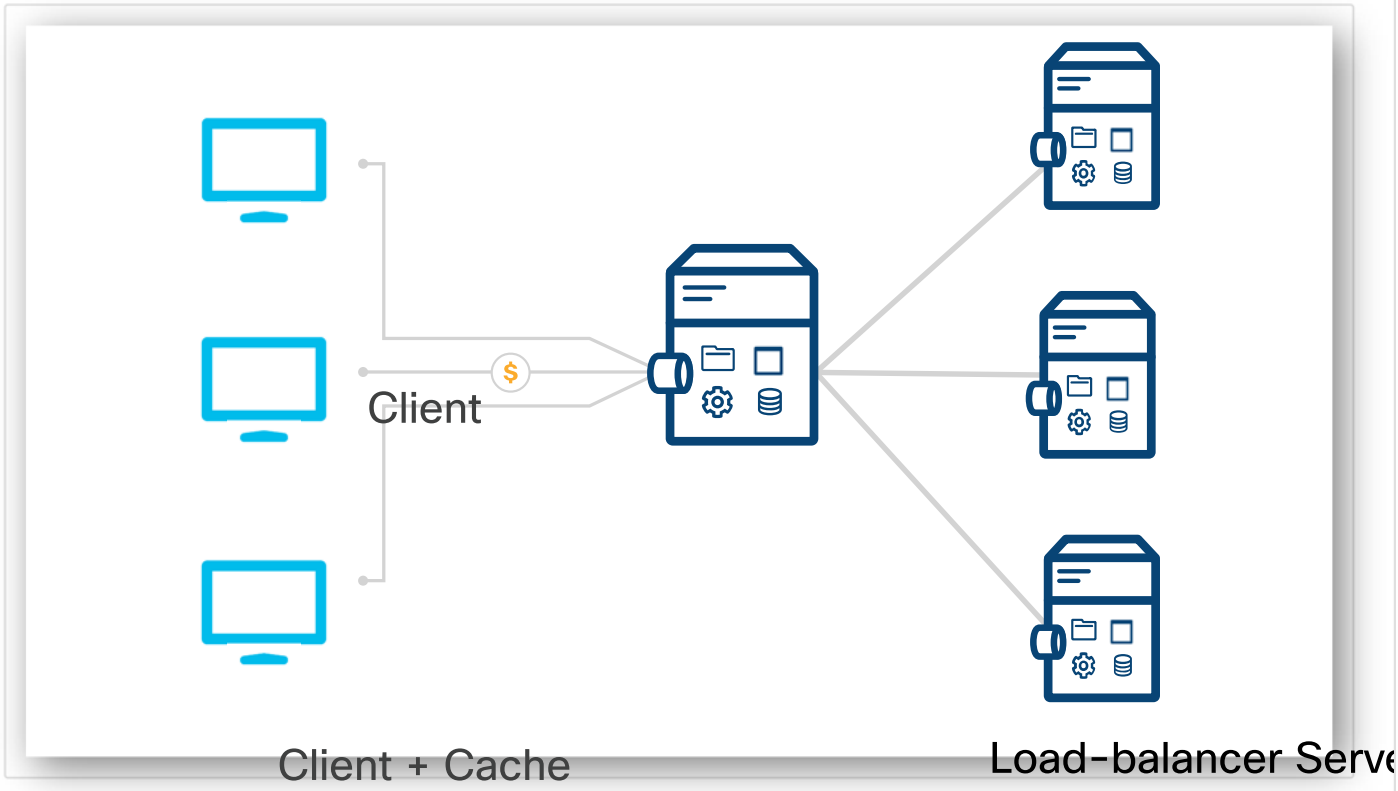
- **Identification of resources -** A resource must be identified in the request as the individual object that the server will access and manipulate. A resource can be any information such as a document, image, person, a collection of other resources, and so on. For example, in the request to change a password for a user, the individual user must be identified.
- **Manipulation of resources through representations -** The client receives a representation of the resource from the server. This representation must contain enough data or metadata for the client to be able to manipulate the resource. For example, a request intended to populate a user's profile must include the profile information. A representation can be an exact copy of the resource on the server or even a simplified version of the resource but not, for example, just an identifier to an additional resource.
- **Self-descriptive messages -** Each message must contain all of the information for the recipient to process the message. Examples of information can be:
  - The protocol type
  - The data format of the message
  - The requested operation

- **Hypermedia as the engine of application state -** The data sent by the server must include additional actions and resources available for the client to access supplemental information about the resource.

Layered system

The system is made up of different hierarchical layers in which each layer provides services to only the layer above it. As a result, it consumes services from the layer below.

REST layered system model



Code-on-demand

This constraint is optional, and references the fact that information returned by a REST service can include executable code (e.g., JavaScript) or links to such code, intended to usefully extend client functionality. For example, a payment service might use REST to make available links to its published JavaScript libraries for making payments. These JavaScript files could then be downloaded and (if judged trustworthy) executed by a client application. This eliminates the need for client developers to create and maintain separate payment-processing code, and manage dependency changes that might break such code from time to time.

Client

The constraint is optional because executing third-party code introduces potential security risks, and because firewalls and other policy-management tools may make third-party code execution impossible in some cases.

4.2
API Design Styles

4.4
Introduction to REST APIs