

Patterns in Software Engineering



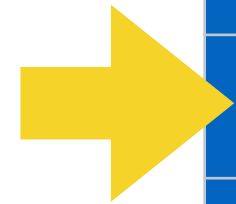
05 Architectural Patterns II

Stephan Krusche

28 November 2022
Technical University of Munich



Course schedule



#	Date	Subject
	17.10.22	No lecture, repetition week (self-study)
1	24.10.22	Introduction
	31.10.22	No lecture, repetition week (self-study)
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	21.11.22	Architectural Patterns I
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	12.12.22	Antipatterns II
	19.12.22	No lecture
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	Microservice Patterns I
11	30.01.23	Microservice Patterns II
12	08.02.21	Course Review

Roadmap of the lecture



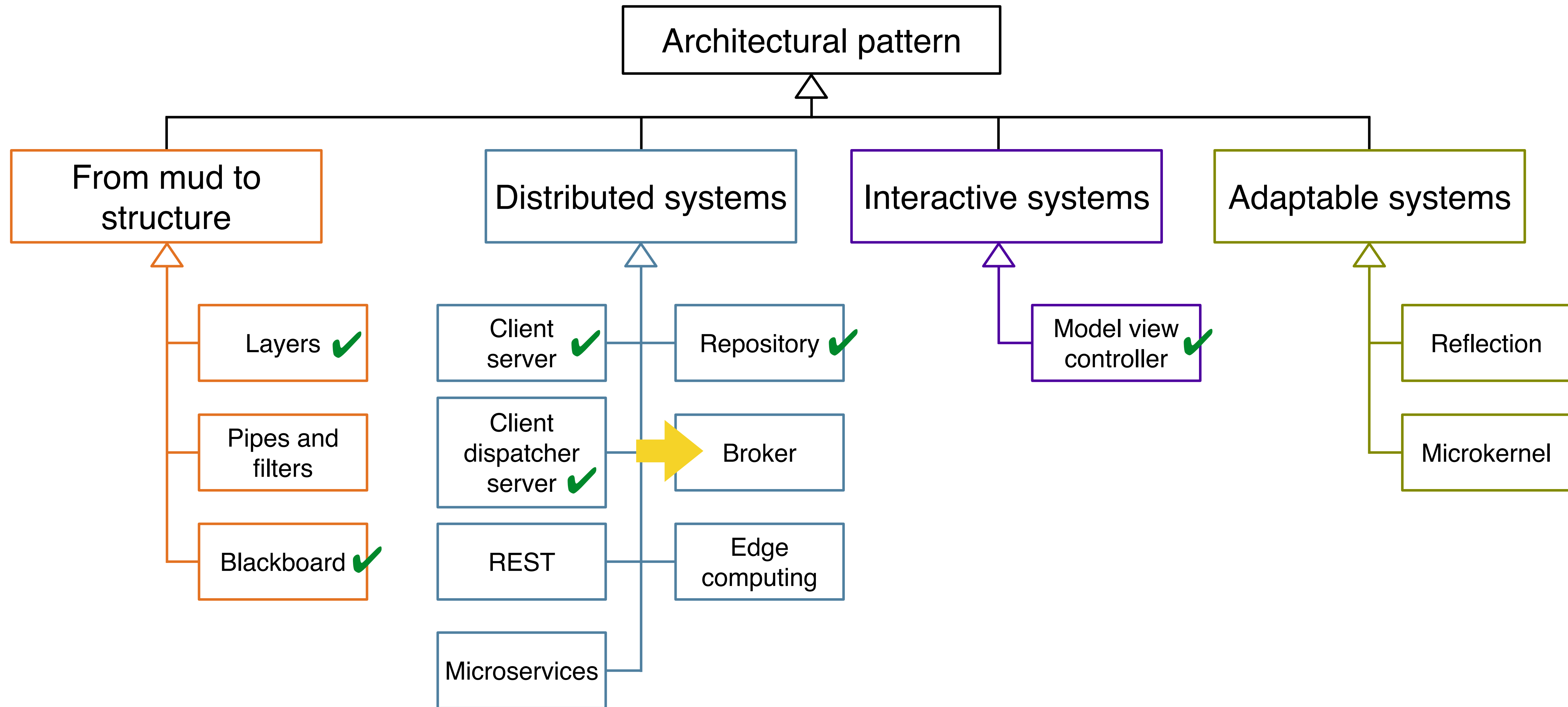
- **Context and assumptions**
 - You implemented many different design patterns and a couple of architectural patterns
 - You are aware of distributed systems and the need to connect distributed components
- **Learning goals: at the end of this lecture you are able to**
 - Differentiate between the covered architectural patterns
 - Choose the right architectural style for a specific problem
 - Implement a RESTful client and web service in Java
 - Explain the idea of GraphQL and gRPC
 - Model the application of an architectural pattern

Outline

Broker

- REST as architectural style
- REST resources
- RESTful APIs: HTTP and HATEOAS
- GraphQL
- gRPC

Architectural pattern overview

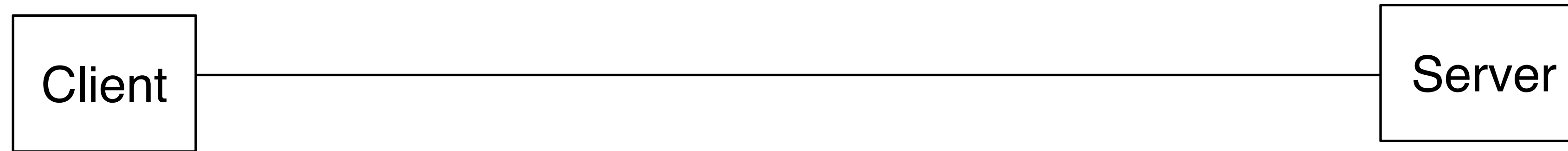


Design goals addressed by the broker



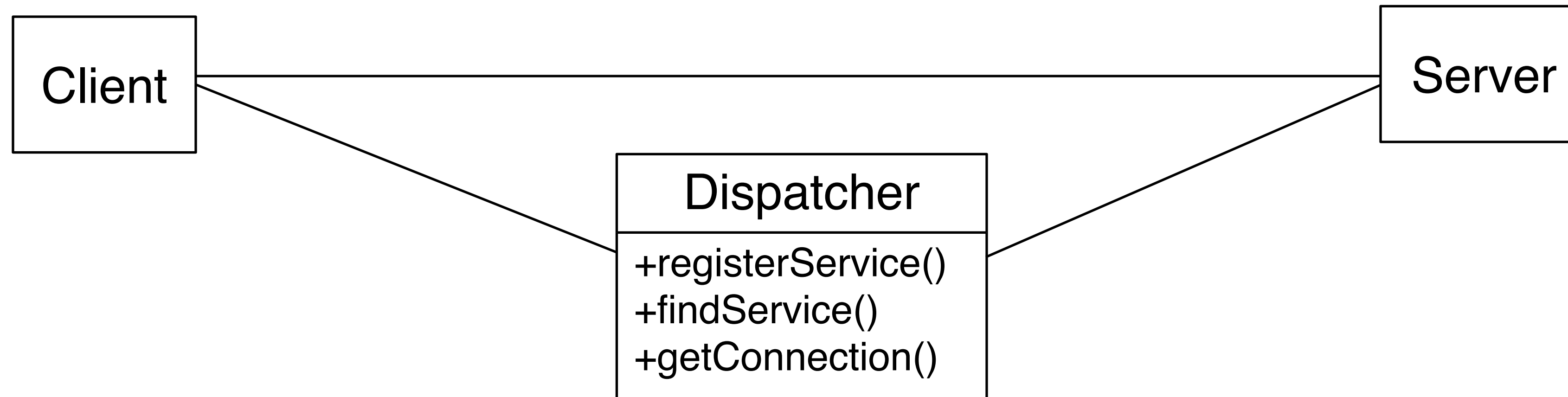
- **Low coupling**
 - Decoupling of service implementation and presentation mechanisms
 - Decoupling of service implementation and communication mechanisms
- **Location transparency**
 - A client can use a service independently of the location of the server that offers it
- **Runtime extensibility**
 - Being able to add, remove and exchange components at runtime

From client server ...



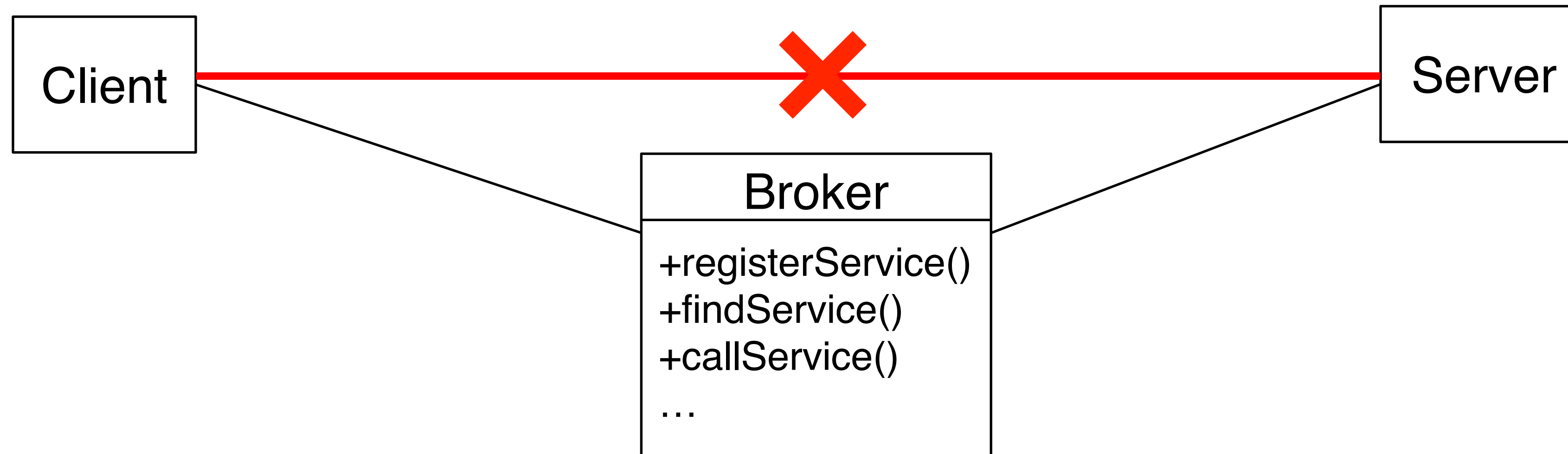
- **Client**: any application that accesses the services provided by a remote server (**example**: web browser)
- **Server**: responsible for providing remote **services**
 1. Provision of common services to many domains (**example**: web server)
 2. Specific functionality for an application domain or task (address book, yellow pages, ...)
- Nonfunctional requirement achieved: **low coupling**

... via client dispatcher server ...



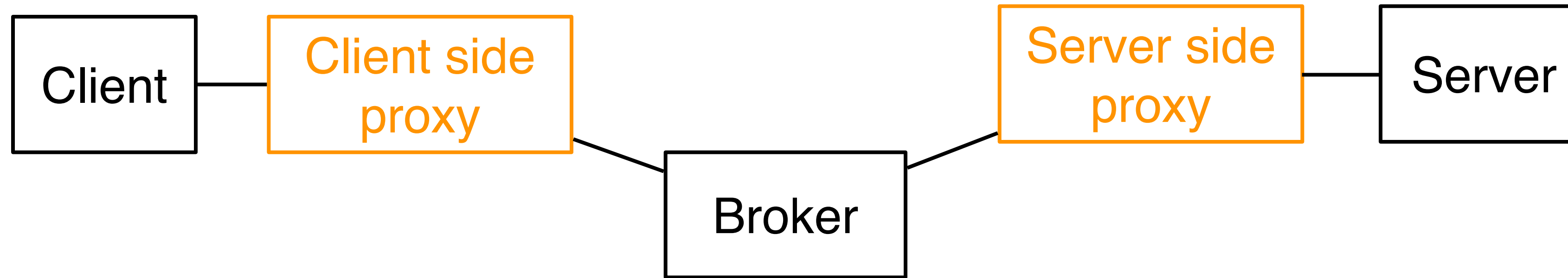
- **Dispatcher**: responsible for providing the name service
- The client directly communicates with the server
- Nonfunctional requirements achieved: **location transparency**, **runtime extensibility**

... towards broker ...



- **Broker**
 - Provides the name service
 - Transmission of requests to services (client → server)
 - Transmission of responses and exceptions (server → client)
- There is **no** direct connection between client and server
- Nonfunctional requirement achieved: **platform independence**

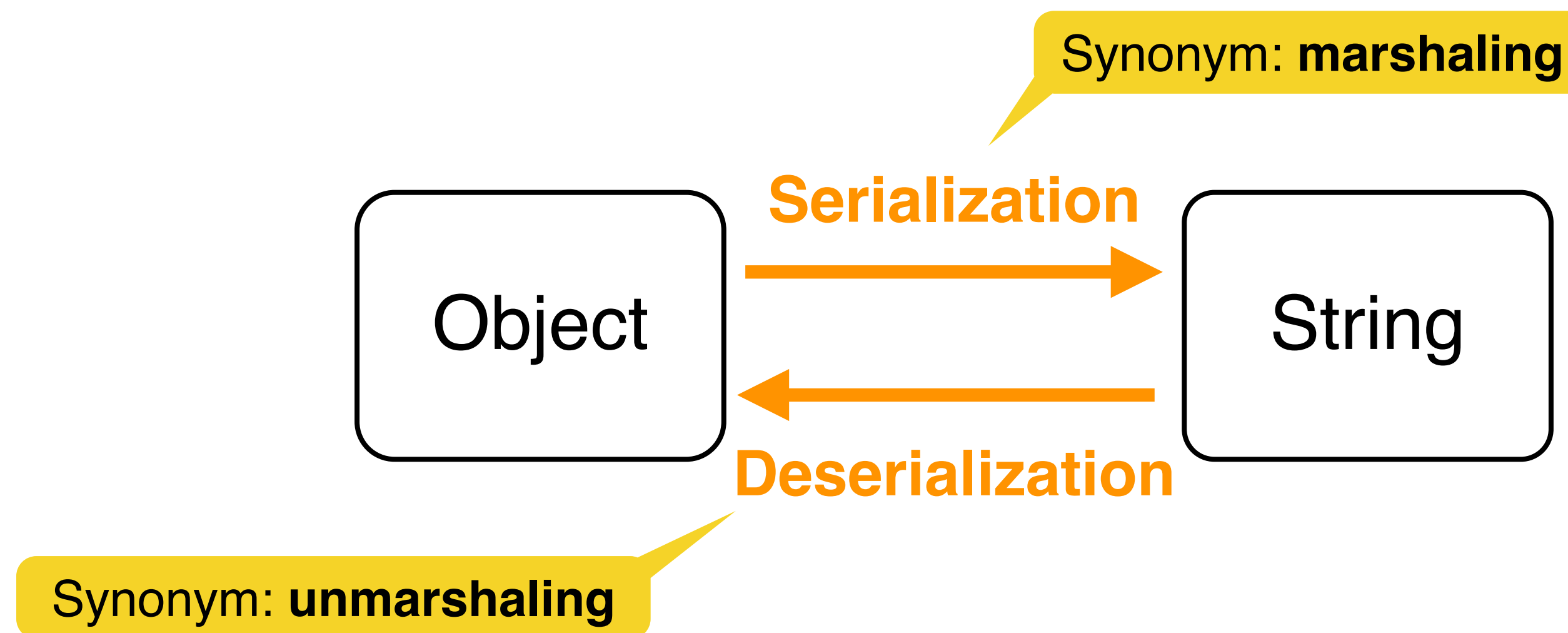
... to broker with proxies



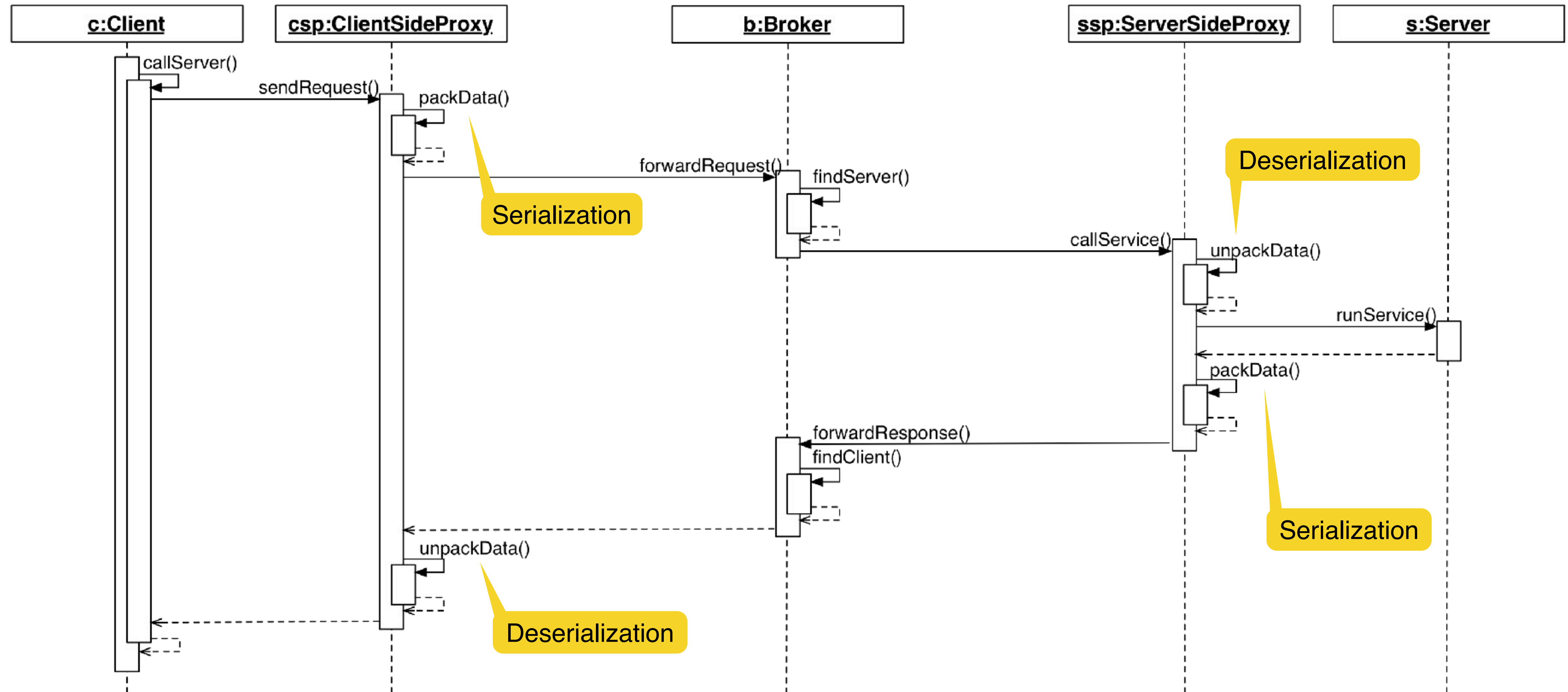
- **Client side proxy**: layer between client and broker
 - Remote objects (in the server) appear as local ones
 - Translates the object model specified by the server into an object model in the client
 - Hides interprocess communication details between **client** and **broker**
- **Server side proxy**: layer between broker and server
 - Receives requests from the broker
 - Hides interprocess communication details between **broker** and **server**
 - Calls the services in the server

Serialization and deserialization

- **Conversion** to/from byte stream suitable for transmission across a network
- Used when the state of an object must be **moved** from a program on one machine to another program on another machine (**heterogeneous architecture**)
- Used in implementations for **remote procedure calls**: Java RMI, CORBA and REST
- Why do we need those?
 - **Platform specificities**: endianness (big endian vs. little endian), different type systems, different object representations (e.g. json, xml, string, binary)



Use of serialization and deserialization in the broker



Synchronous vs. asynchronous communication



- Synchronous
 - The client issues the method call and waits (**blocks**) until the result is returned
- Asynchronous
 - The client issues the method call, and continues (a **non-blocking** method call)
 - It gets notified by broker when the result is ready
 - This is usually implemented using **callbacks**

Steps to realize a broker pattern

1. Provide the **object model** and **service definitions**

- Define client and server objects: the state of the server objects should be private
- Clients may change or read the server's state only by passing requests to the broker
- Define the service interfaces

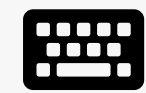
2. Define the **broker** service

- Specify the services offered by the Broker (register, call, ...)

3. Implement the **broker component** and **proxy objects** at client and server side

- Serialize and deserialize objects
- Pass service calls to the broker
- Implement the interface access routines for the proxies

4. Implement the **client** and the **server**



L05E01 Broker

Start exercise

Medium

Not started yet.

Due Date: in 7 days



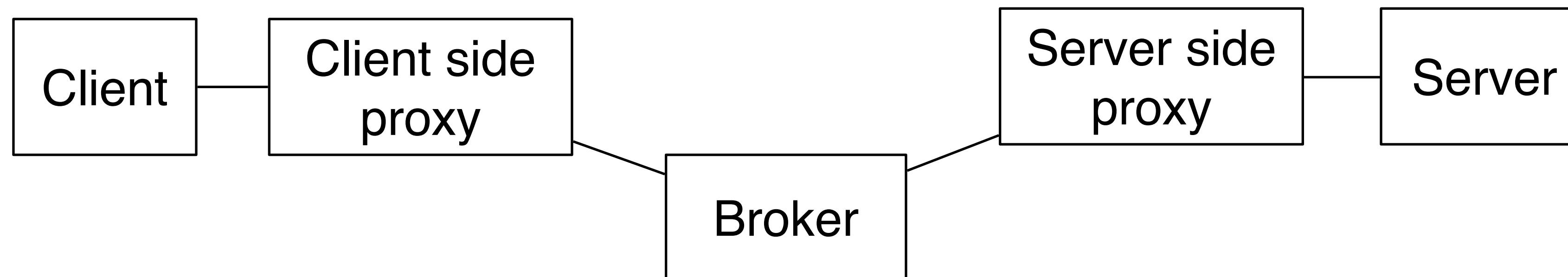
20 min



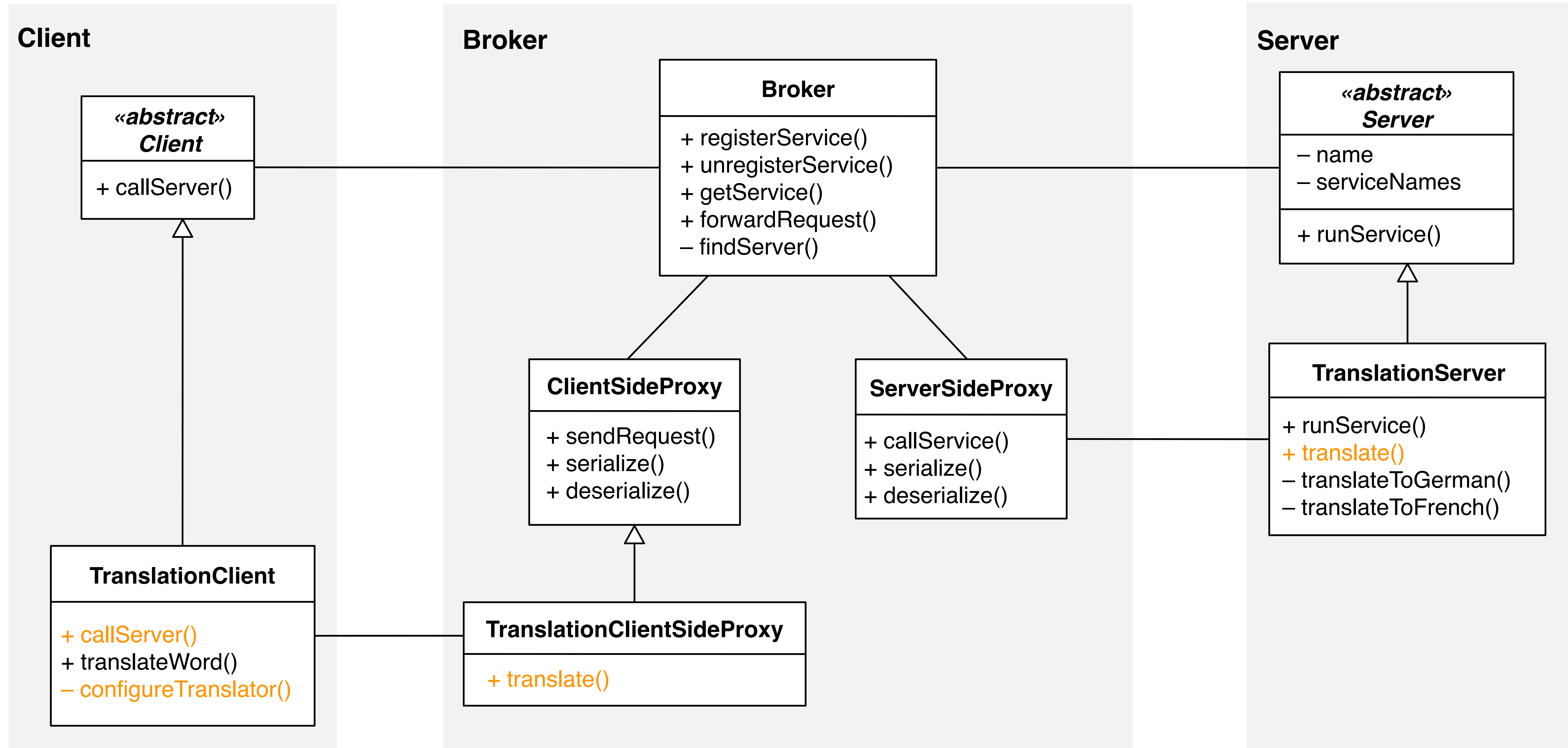
8 pts



- Problem statement: **translation service**
 - The client wants to translate an **English** word to **German** or **French**
 - The English → German translation has already been implemented using the broker architectural style and is being used by the client
 - Now English → French translation needs to be supported



Hint: UML class diagram



Outline

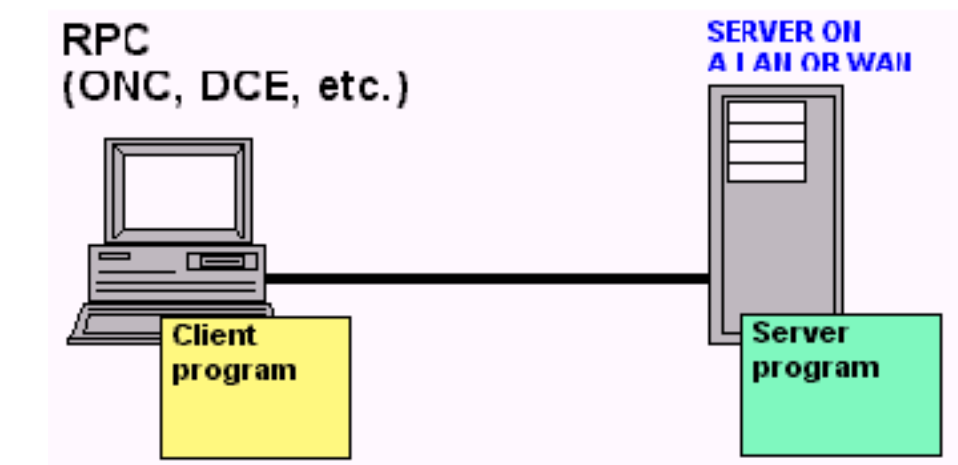
- Broker

REST as architectural style

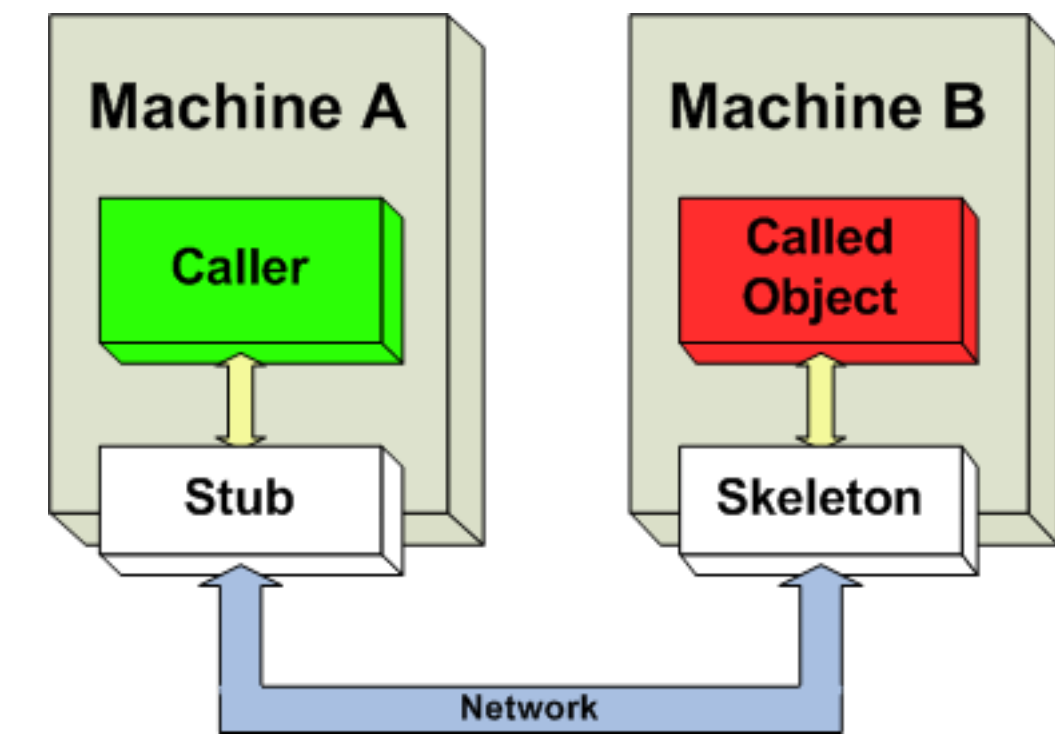
- REST resources
- RESTful APIs: HTTP and HATEOAS
- GraphQL
- gRPC

History: RPC, RMI, and CORBA

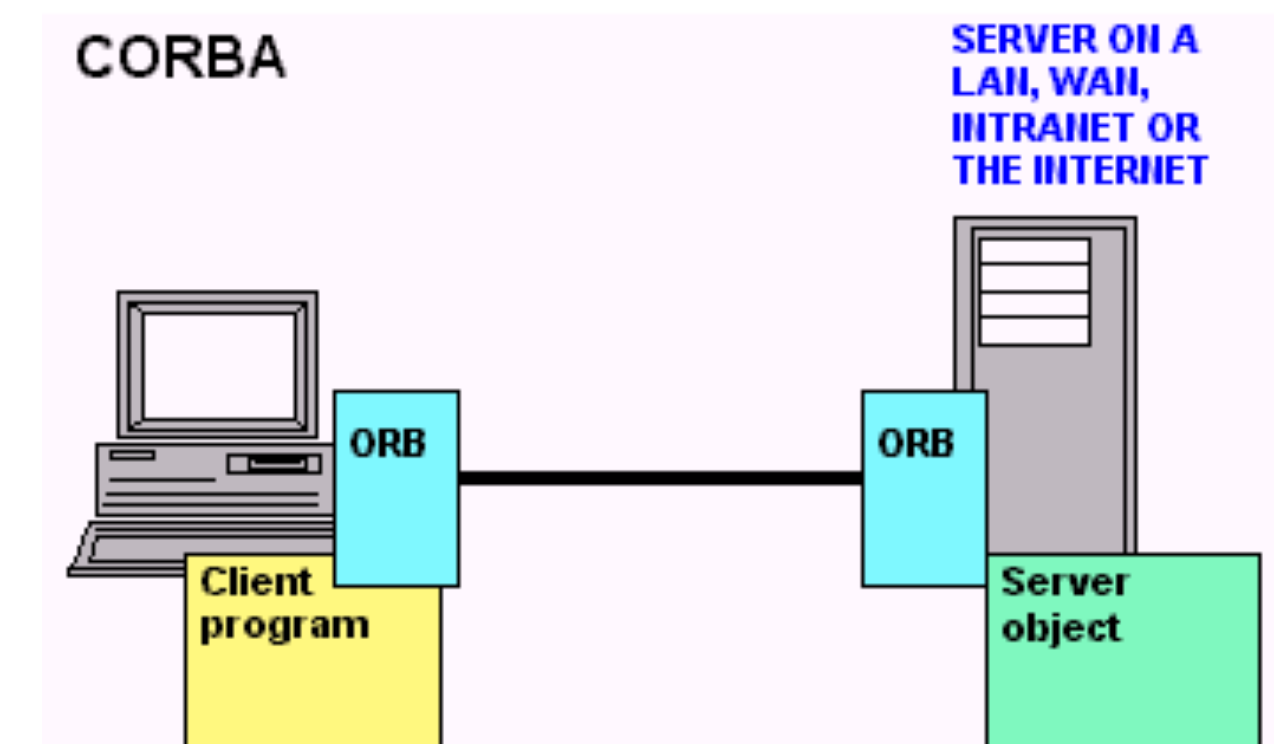
- **RPC**: remote procedure calls enable the execution of code in a different address space
- **RMI**: remote method invocation
- **JAVA RMI**
 - Support to transfer serialized Java classes
 - Usage of the Java remote method protocol (JRMP) for looking up and referencing remote objects
- **CORBA**: a cross platform and programming language implementation of the RPC concept
 - **C**ommon **O**bject **R**equest **B**roker **A**rchitecture



RPC (1981)



Java RMI (1995)



CORBA (1991)

CORBA → SOAP → REST

- Problems with RPC frameworks like **CORBA**
 - Proprietary protocols and IDLs (interface definition language)
- **SOAP** (simple object access protocol)
 - Released as XML-RPC in 1998 by Microsoft
- Solution by **Roy T. Fielding** in his dissertation in 2000
REST (Representational State Transfer)
 - Communication using a layered architecture
 - Protocol agnostic but often used and described with HTTP
 - Use links between endpoints to navigate a web API (HATEOAS)
 - Uses URL endpoints to provide resources and manipulate resources
- Since 2012: **gRPC**, **GraphQL**, **Apache Thrift**



Roy T. Fielding

Covered at the end of the lecture

- From a programmer's perspective you can broadly differentiate between **message based** and **method based** communication styles and mechanisms

1. Message based

- The developer structures APIs as a set of messages that can be exchanged and encapsulate all necessary functionality
- **Example:** HTTP, REST, WebSockets, GraphQL

2. Method based

- The developer structures APIs as a set of methods that have parameters and return types encapsulating remote functionality
- **Example:** RPC, RMI, CORBA, gRPC

Method based communication is eventually implemented using message based communication

REST as architectural style



- Context

- Shared resources and services with large numbers of distributed clients
- Access control and quality of service are important

- Problem

- Manage a set of shared web resources and services
- Make them **modifiable** and **reusable**,
- Support **scalability** and **availability** while spreading the resources across multiple distributed components

- Solution

- Provide an abstraction that models the structure and behavior of the world wide web
- **RE**presentational **S**tate **T**ransfer (REST)
- REST was originally called the "HTTP object model"

- Overview

- The REST architectural style defines a set of constraints on distributed architectures to simplify the exchange of resources between distributed components

- Elements

- **Web resource**: any entity that can be accessed on the Internet.
- Web resources are identified by their **URI (Uniform Resource Identifier)**
- The most common form of URI is URL used to identify documents or files
- **RESTful web service**: offers access to textual representations of web resources with a predefined set of operations
- **Client**: component that invokes a RESTful web service

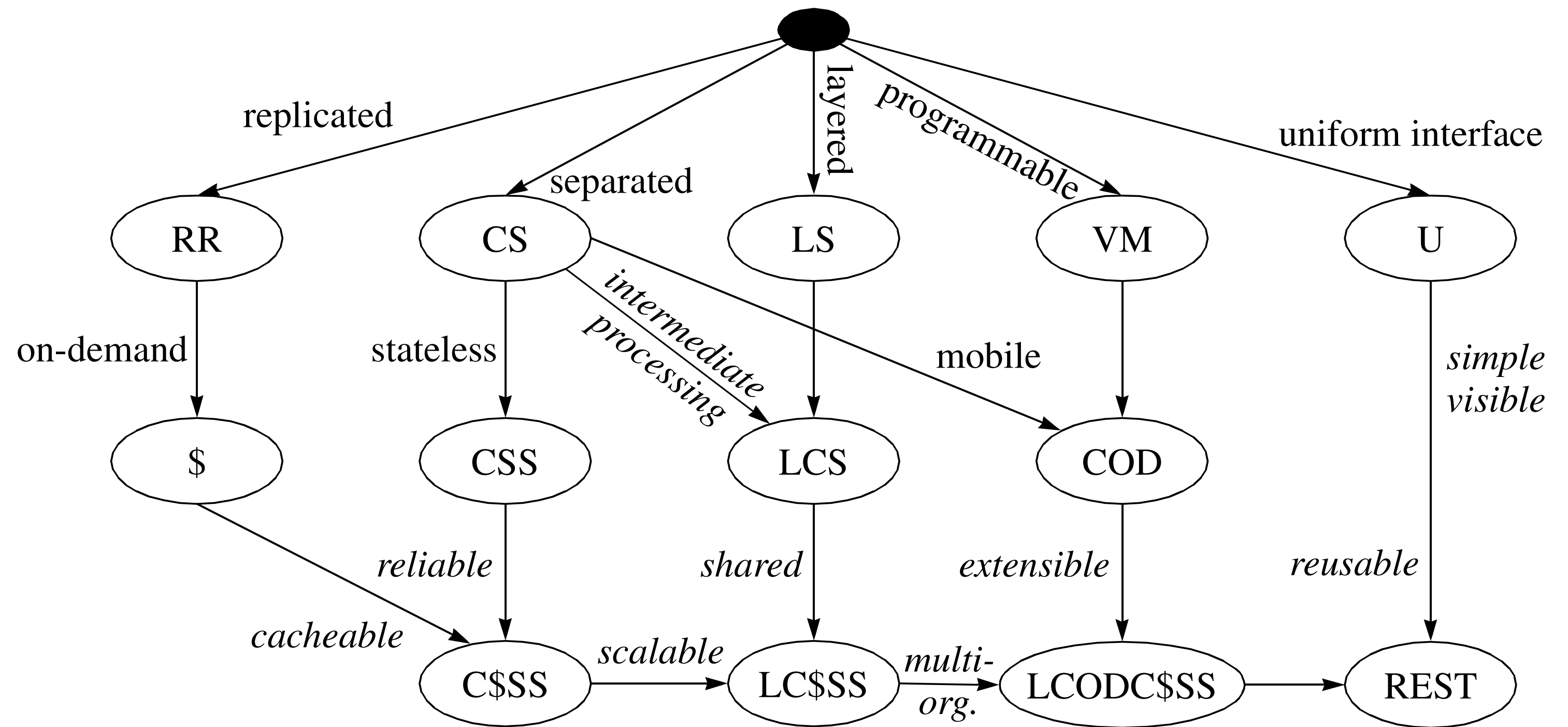
REST as architectural style



- **Connector**
 - Request / response
 - A request made to a web resource elicits a response with a body encoded in e.g. XML or JSON
 - The protocol is **stateless**: no additional state information between two requests
- **Constraints**
 - Clients can connect to RESTful web services only via the request/response connector
- **Weaknesses**
 - Decisions where to locate functionality (in the client or in the web service) are often complex and costly to change after a system has been built
 - Larger, nested web resource exchanges often require multiple independent requests
 - The stateless nature prohibits e.g. publisher subscriber use cases

REST requirements

1. Client server (CS)
2. Stateless (CSS)
3. Cacheable (C\$SS)
4. Uniform interface (U)
5. Layered system (LS)
6. Code on demand (COD)



Client server (CS)

The first requirement is to use the **client server architectural style**

- **Benefits**
 - Separation of concerns
 - Portability of the user interface across multiple platforms
 - Scalability



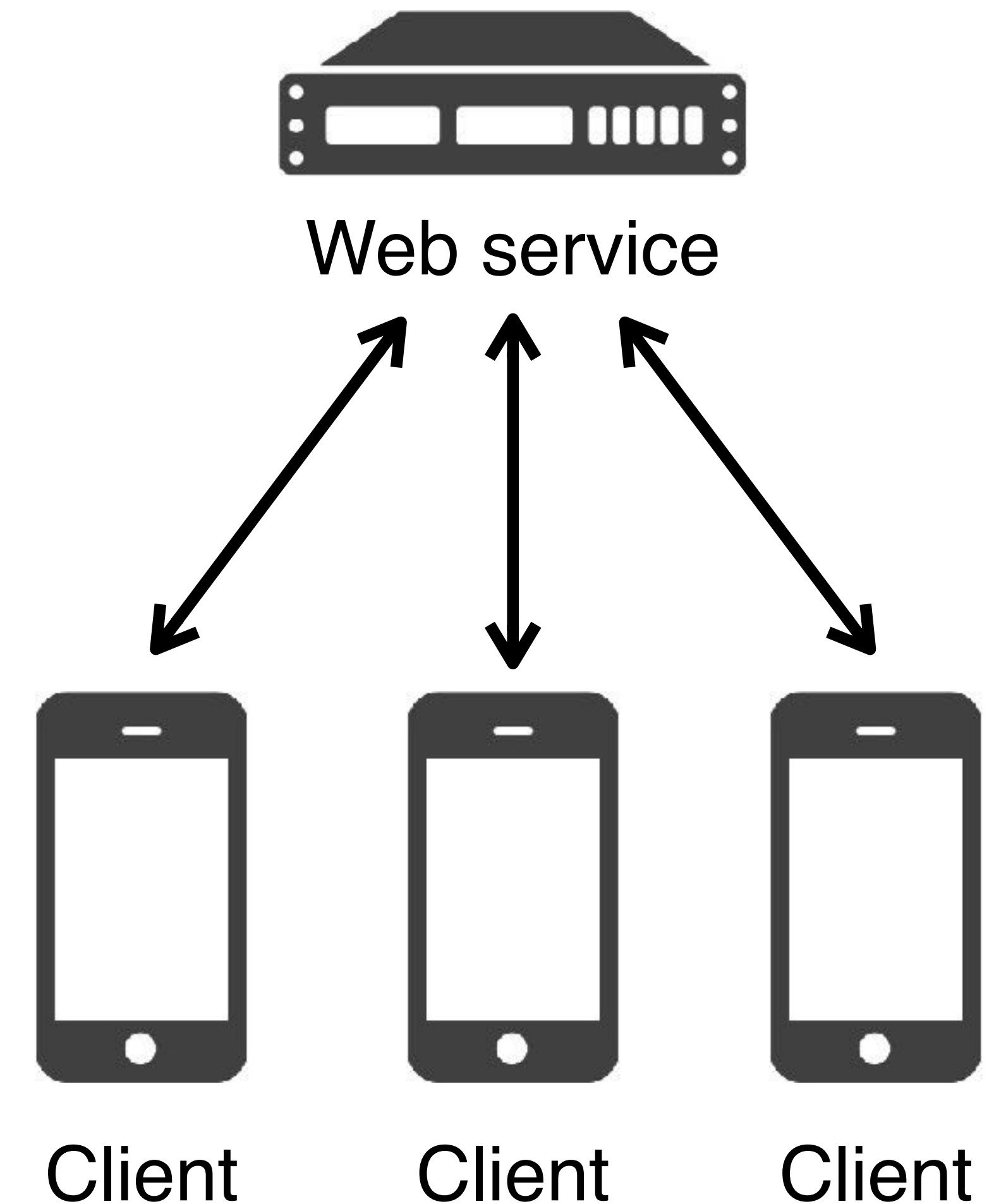
Web service



Client

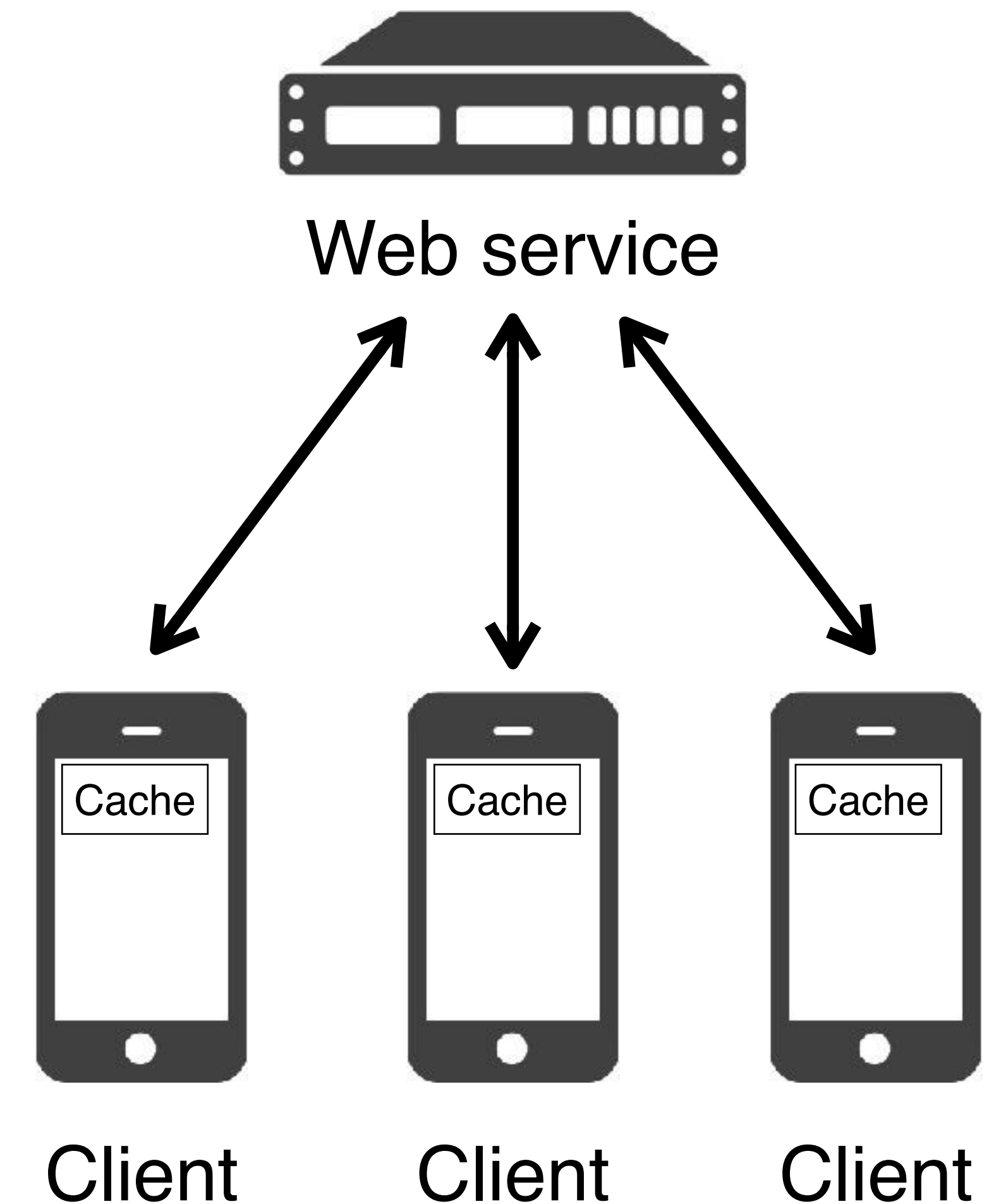
Stateless (CSS)

- Communication must be **stateless** in nature
- Each request from client to web service must contain all information necessary to understand the request
- It cannot take advantage of any stored context on the web service
- **Benefits**
 - Visibility, reliability and scalability
- **Consequences**
 - Decreases network performance (increasing repetitive data)
 - Reduction of the web service's control over consistent application behavior



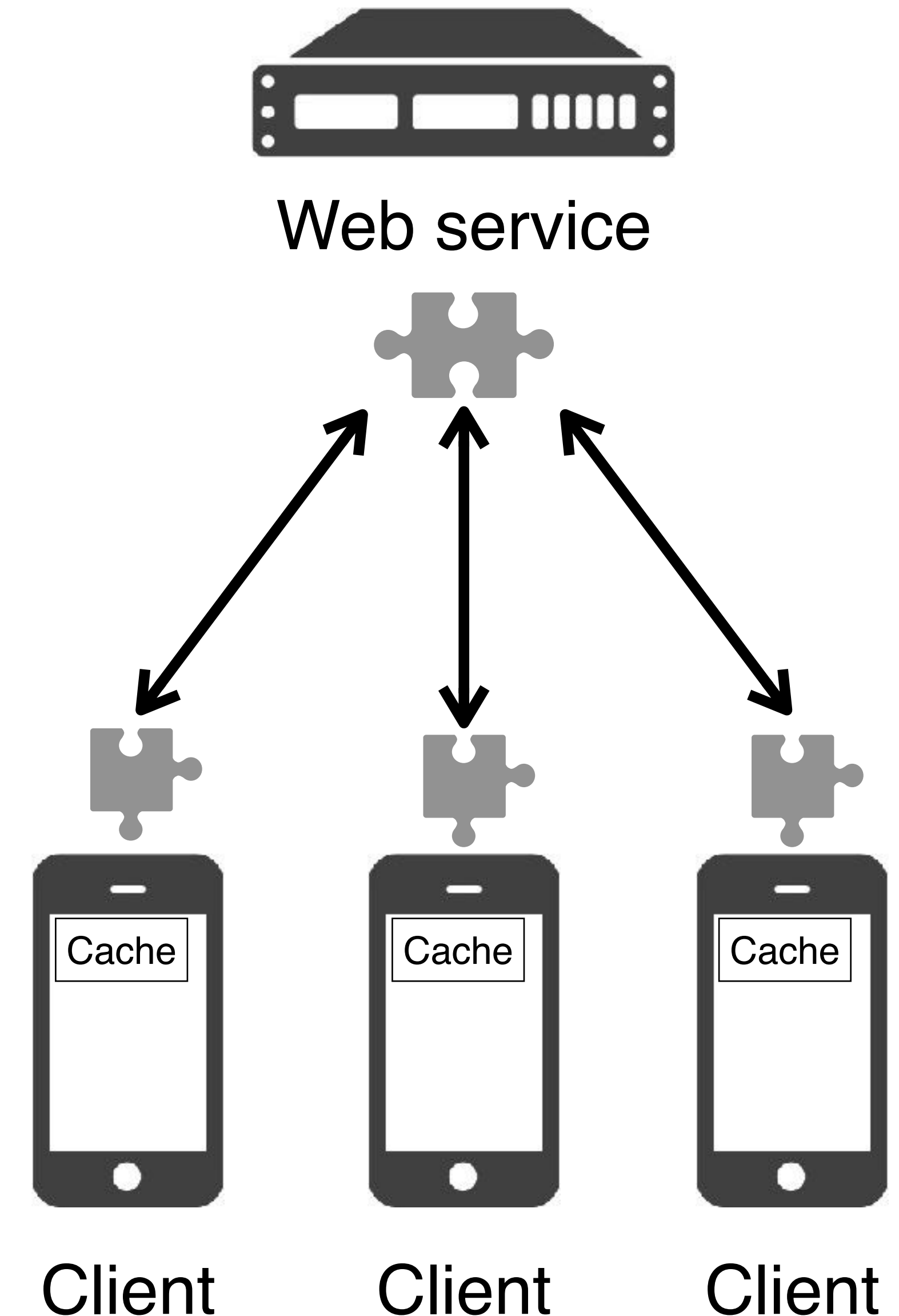
Cacheable (C\$SS)

- Data within a response to a request must be implicitly or explicitly labeled as **cacheable** or **non-cacheable**
- **Benefits**
 - Improves network efficiency
 - Eliminates some interactions
 - Scalability
 - Improved user perceived performance
- **Consequences**
 - Decrease of reliability by stale data within the cache



Uniform Interface (U)

- Introduction of a **uniform interface** between components
- Interface for
 - Identification and manipulation of resources through representations
 - Links between resources that serve as **hypermedia as the engine of application state (HATEOAS)**
- **Benefits**
 - Decoupling of Implementations from the services they provide
- **Consequences**
 - Reduction of efficiency, since information is transferred in a standardized form

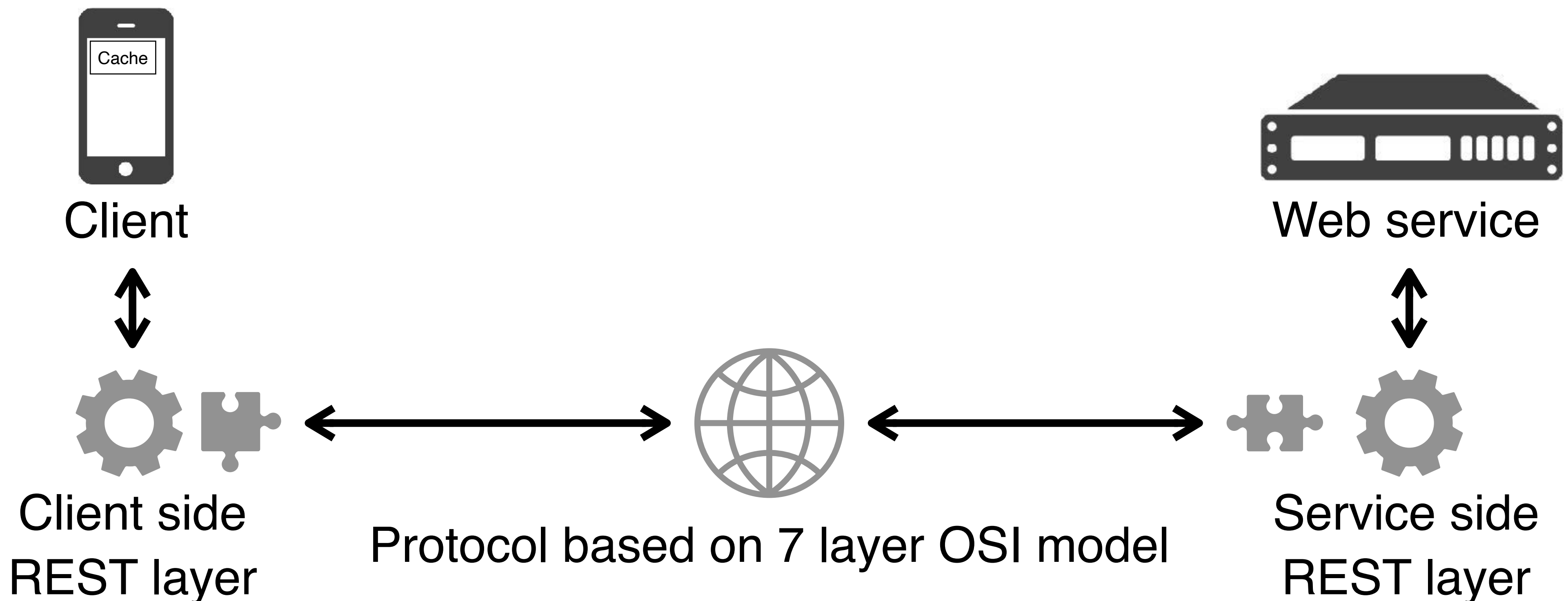


Layered system (LS)

Layered architectural style

The REST architecture should be composed of hierarchical layers

- **Benefits**
 - Reduce overall system complexity
 - Encapsulate legacy services
- **Consequences**
 - Additional overhead and latency



Code on demand (COD)

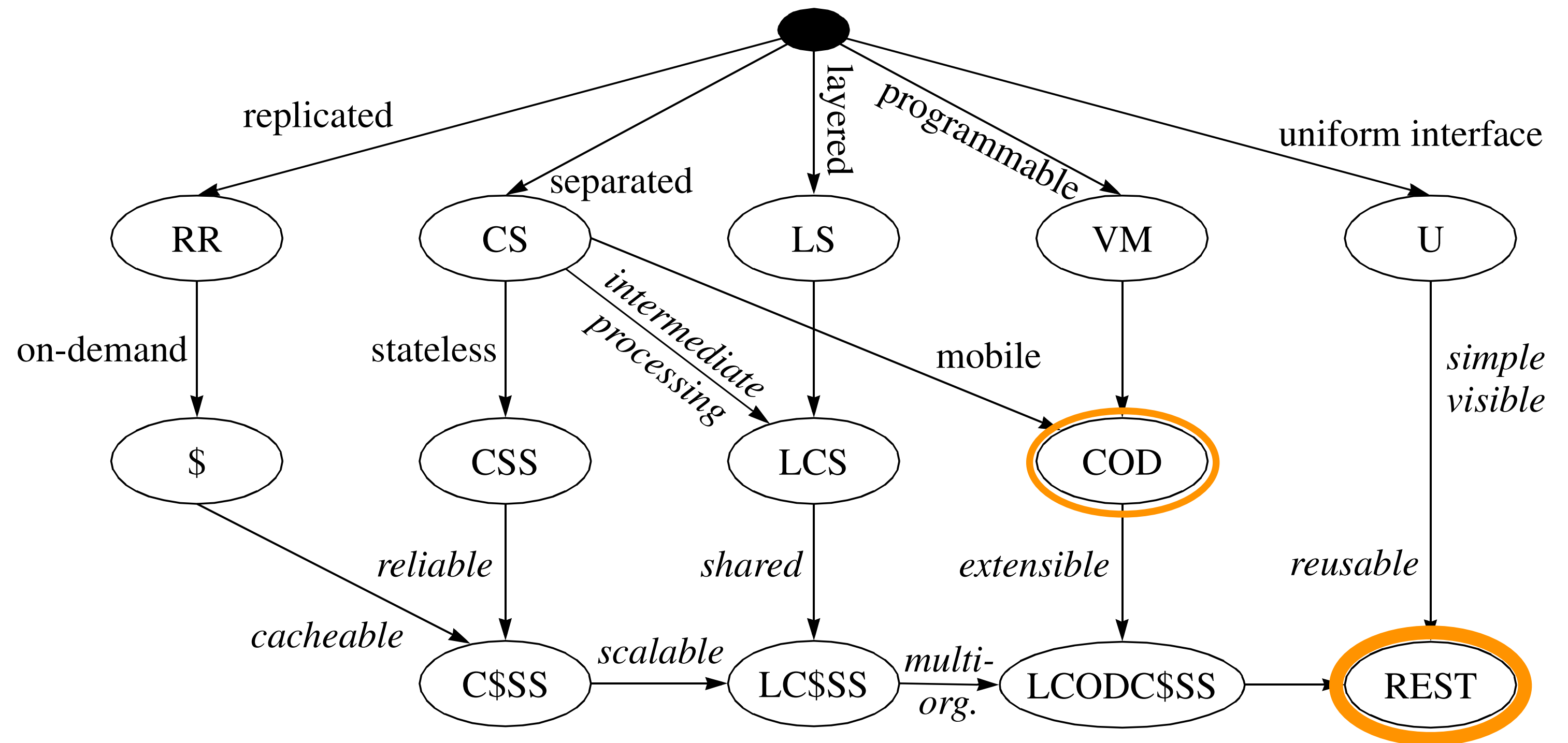


Optional requirement

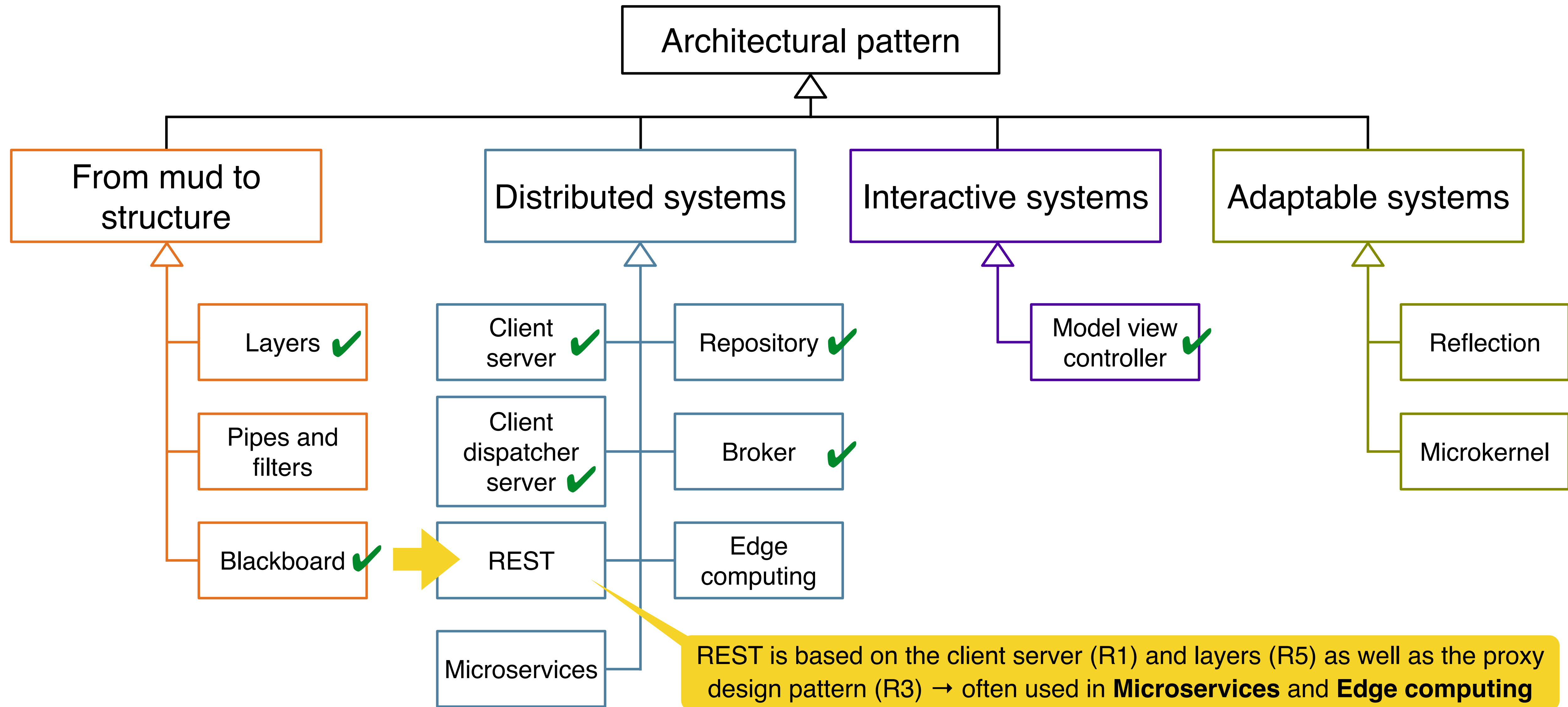
- Allows client functionality to be extended by downloading and executing code
- Benefits
 - Reduce the number of features required to be pre-implemented
- Consequences
 - Reduced visibility

REST requirements

1. Client server (CS)
2. Stateless (CSS)
3. Cacheable (C\$SS)
4. Uniform interface (U)
5. Layered system (LS)
6. Code on demand (COD)



REST in the architectural pattern taxonomy



Outline

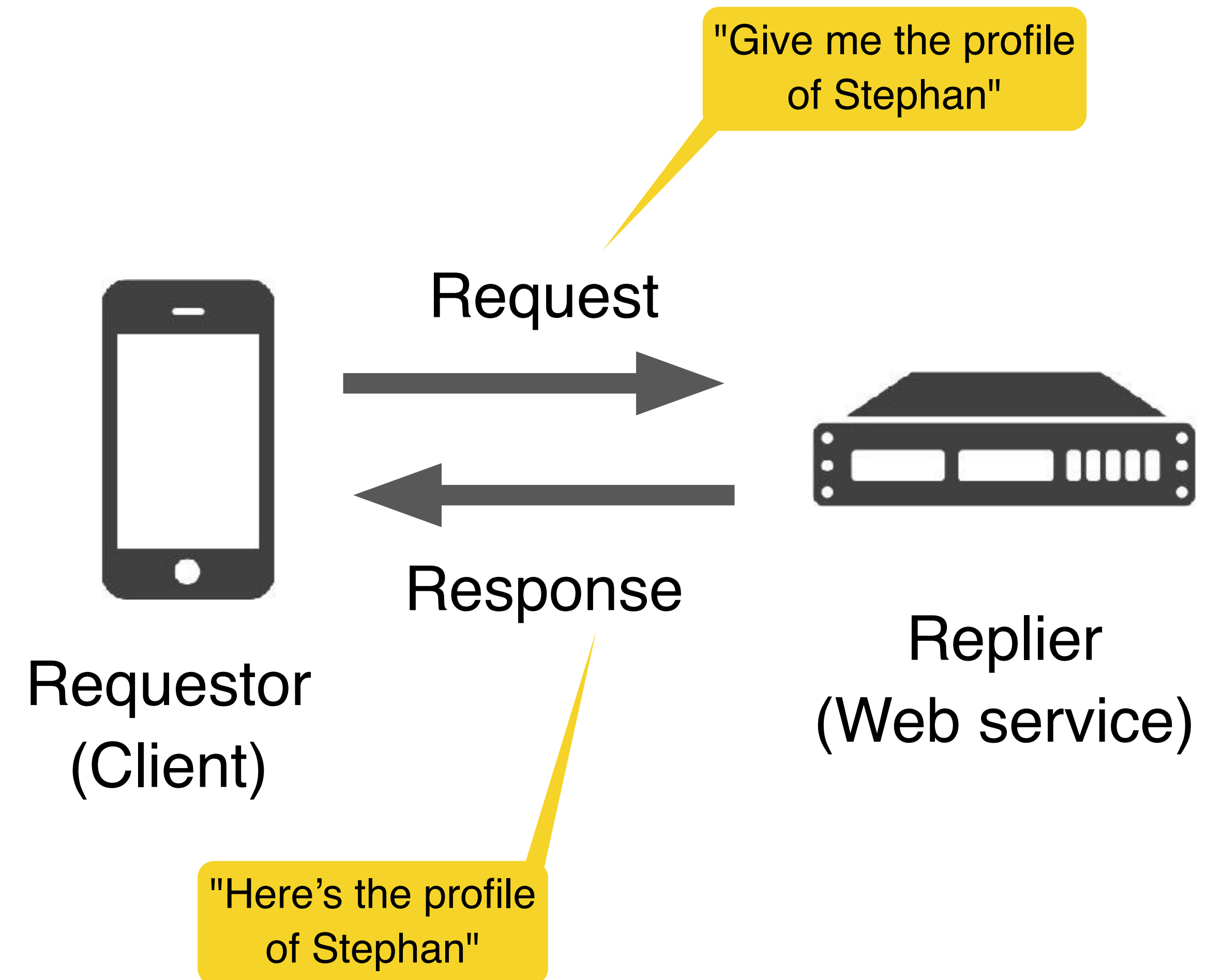
- Broker
- REST as architectural style

REST resources

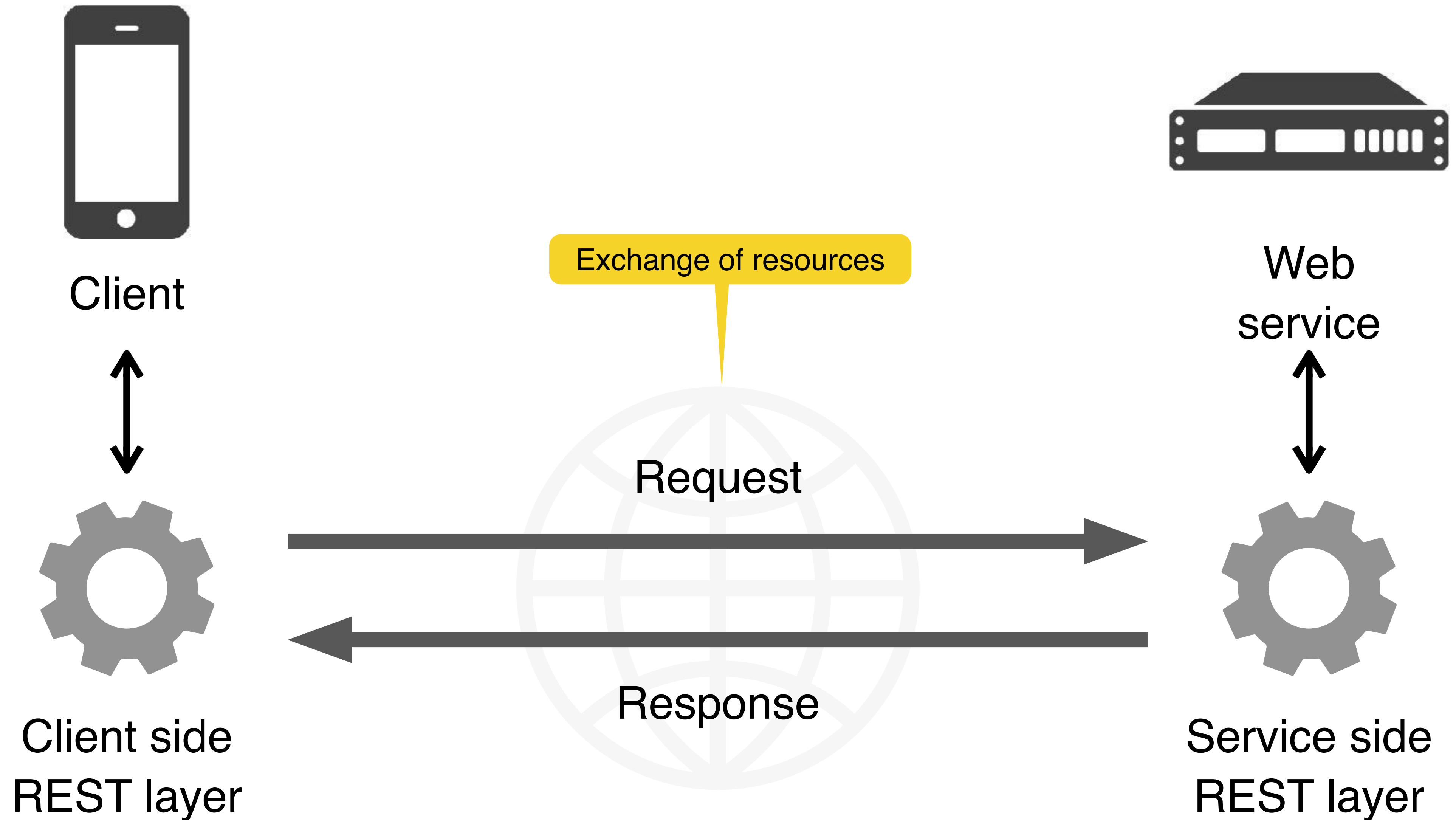
- RESTful APIs: HTTP and HATEOAS
- GraphQL
- gRPC

Request response

- **Request response** is a stateless message exchange protocol
 - Requestor sends a message to Replier
 - Replier receives and processes the request
 - Replier returns a message in response
- **Stateless**
 - The Replier does not keep a history of old requests



REST layered architecture



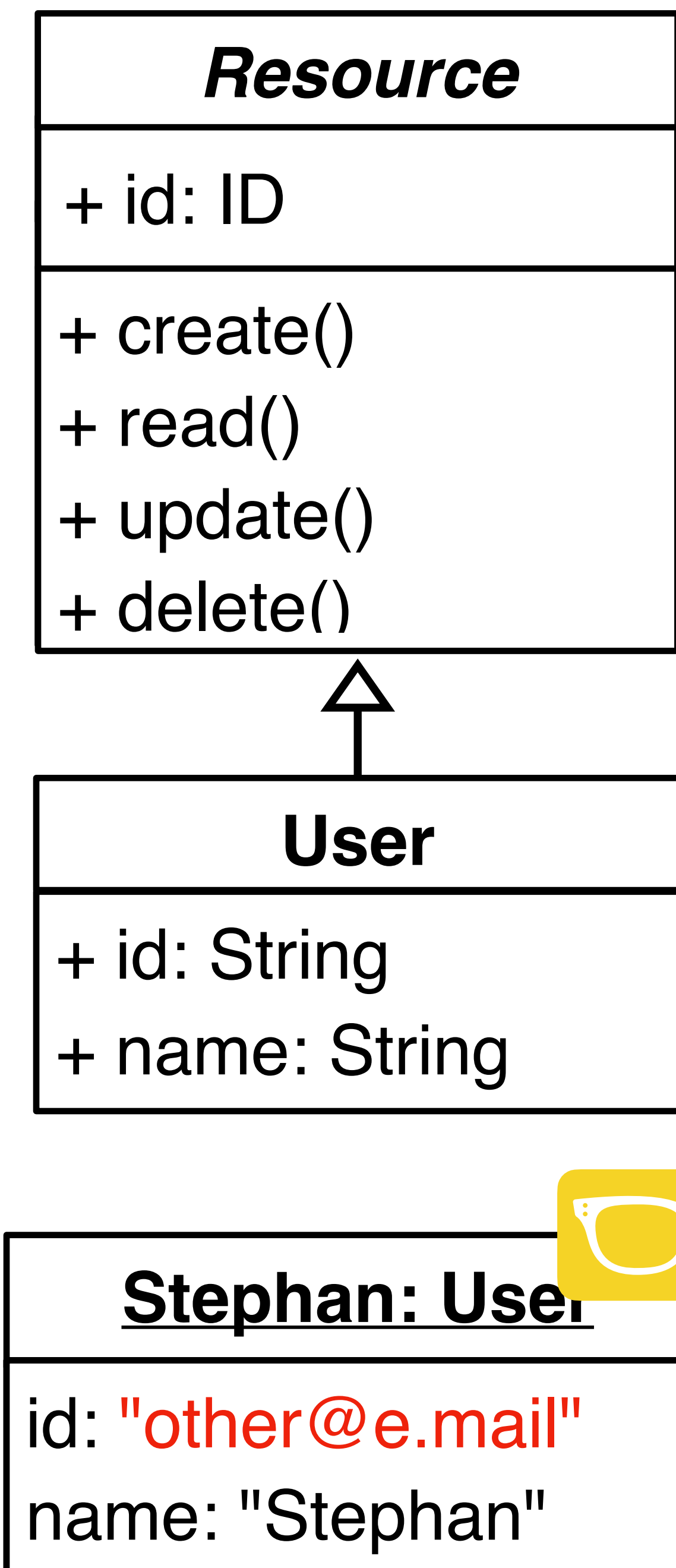
- **Resources** is the REST term for objects (which are instances of classes)
- Examples for classes
 - A **user** with name and ID (e.g email)
 - **Comment** that can be added to an **article**
- Examples for objects
 - **Stephan** is an instance of the class **User**
 - "PSE Announcement ..." is an instance of the class **Article**
- Resources should always be nouns (things), never verbs (actions)
- Resources have to be **identifiable** (e.g. with an URI)

Performing methods on resources

- To realize the uniform interface (U) requirement, REST uses four basic methods performed on any resource
- Example resource **Stephan**

```
{ id = "an@e.mail", name = "Stephan" }
```

 - **Create** — Create a new user named Stephan
 - **Read** — Give me all information about Stephan
 - **Update** — Update the email of Stephan
 - **Delete** — Delete the user named Stephan
- These methods are often summarized as **CRUD**





L05E02 REST Quiz

Not started yet.

 Start quiz

Live

Easy

Due date in 5 minutes



5 min



3 pts



- Open the live quiz now
- Participate in the quiz
- Submit your answer within the due date

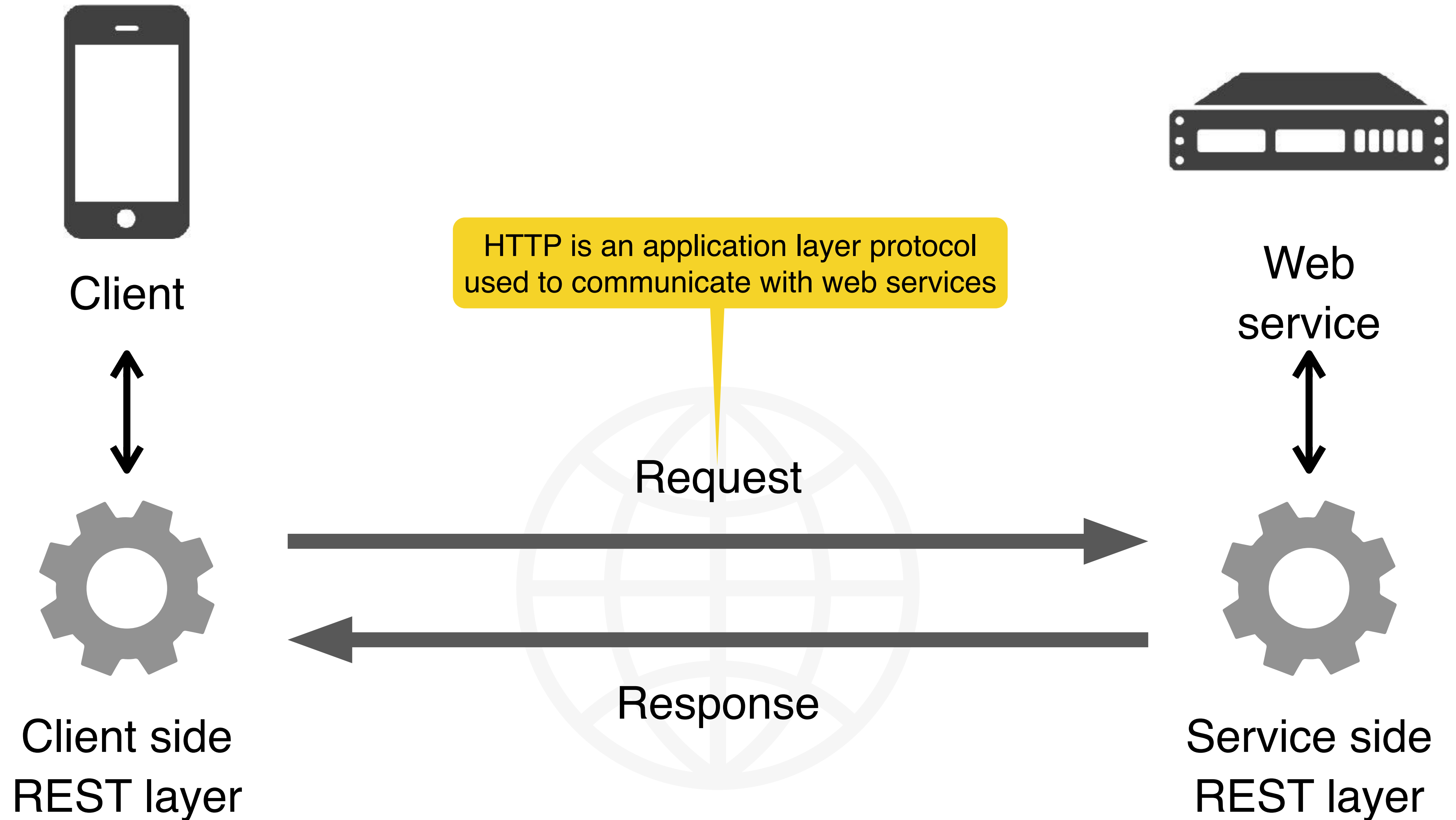
Outline

- Broker
- REST as architectural style
- REST resources

RESTful APIs: HTTP and HATEOAS

- GraphQL
- gRPC

REST layered architecture



Mapping of CRUD to HTTP

- **HTTP** is an application layer protocol used to communicate with web services
- HTTP messages consist of a **URI**, a **HTTP method**, a **header** that contains key value pairs and a message body that can contain arbitrary data

Abstraction	Description	HTTP method
Create	Create new resource	POST
Read	Retrieve existing resources	GET
Update	Update an existing resource with an identifier	PUT
Delete	Delete existing resources	DELETE

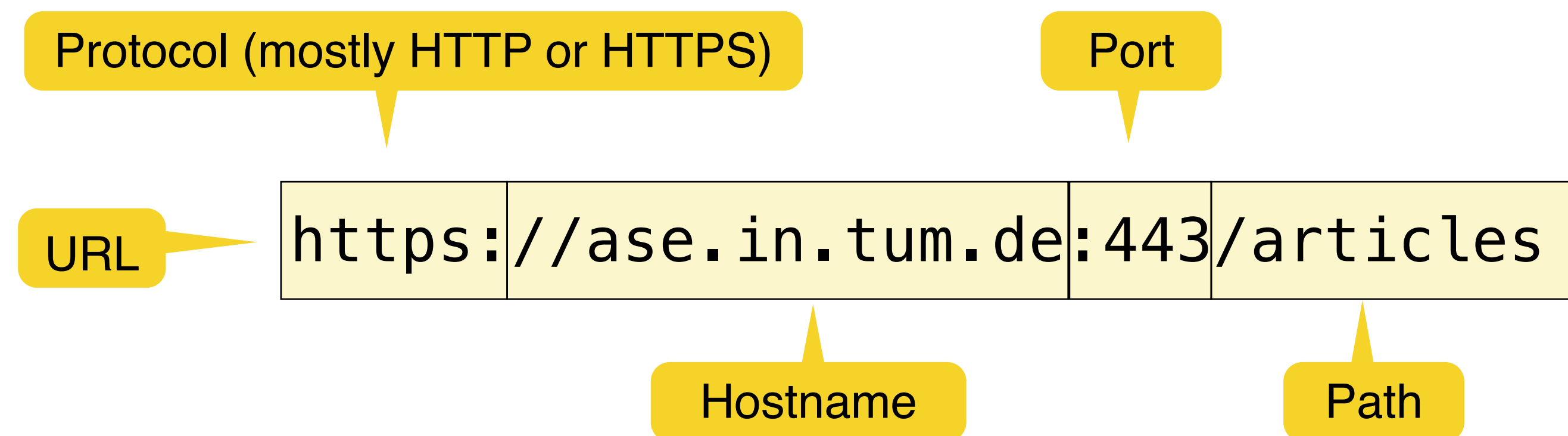
Safe and idempotent HTTP methods

- **Idempotent** methods can be applied multiple times without changing the result
- **Safe** methods do not modify objects (can be used for caching and prefetching objects)

Properties	GET	POST	PUT	DELETE
Usage	Retrieve existing resources	Create new resource	Update an existing resource with an identifier	Delete existing resources
Idempotent	✓	✗	✓	✓
Safe	✓	✗	✗	✗

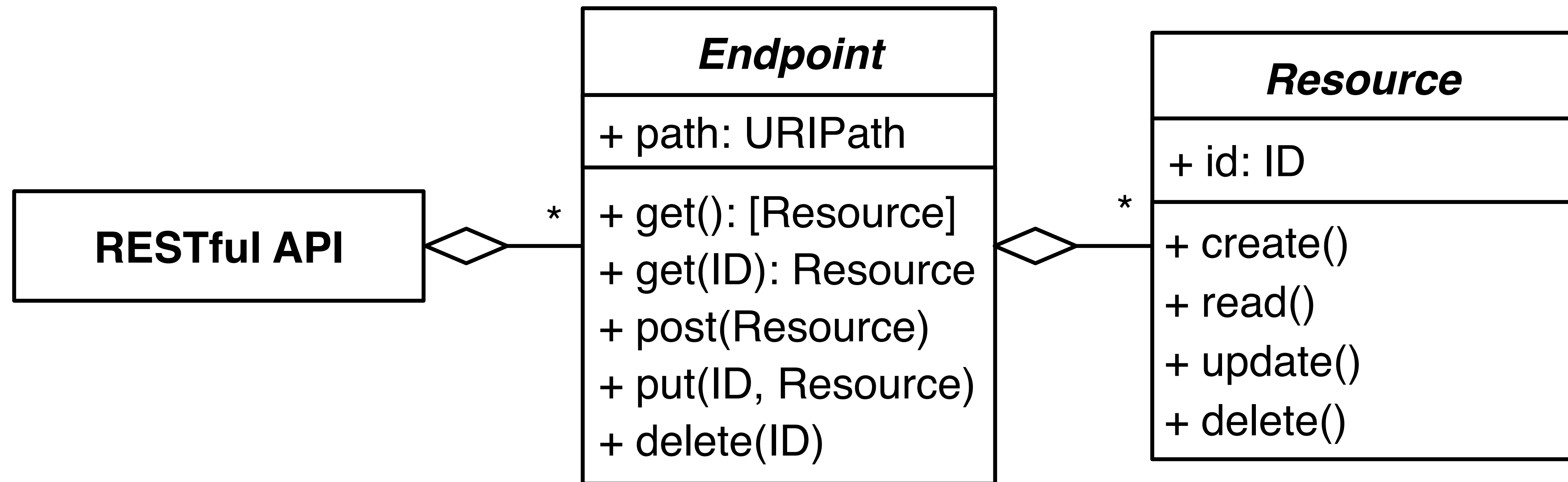
Identifying resources

- **URI = uniform resource identifier**
 - A text string used to identify a resource (e.g.: smart objects, ...)
- Most often: a URL (uniform resource locator)



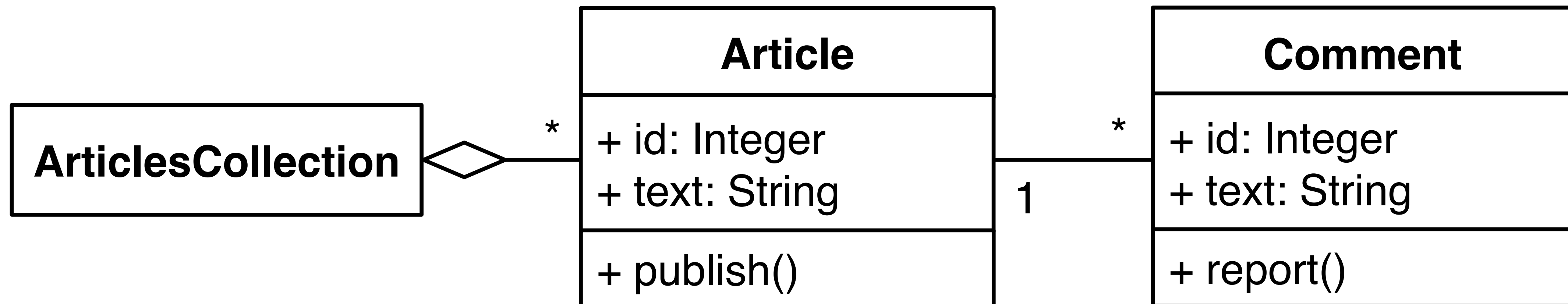
Definition RESTful API

- Collection of **endpoints**
- An endpoint is identified by an **URI path** and **HTTP method**
- It handles HTTP messages to manipulate REST resources
- UML classes, objects and association can be mapped to a RESTful API



Mapping UML constructs to a RESTful API

- UML classes, objects and association can be mapped to a RESTful API
 - Classes to **endpoints**
 - Objects to **resources** that are returned by endpoints
 - Associations to **nested endpoints**

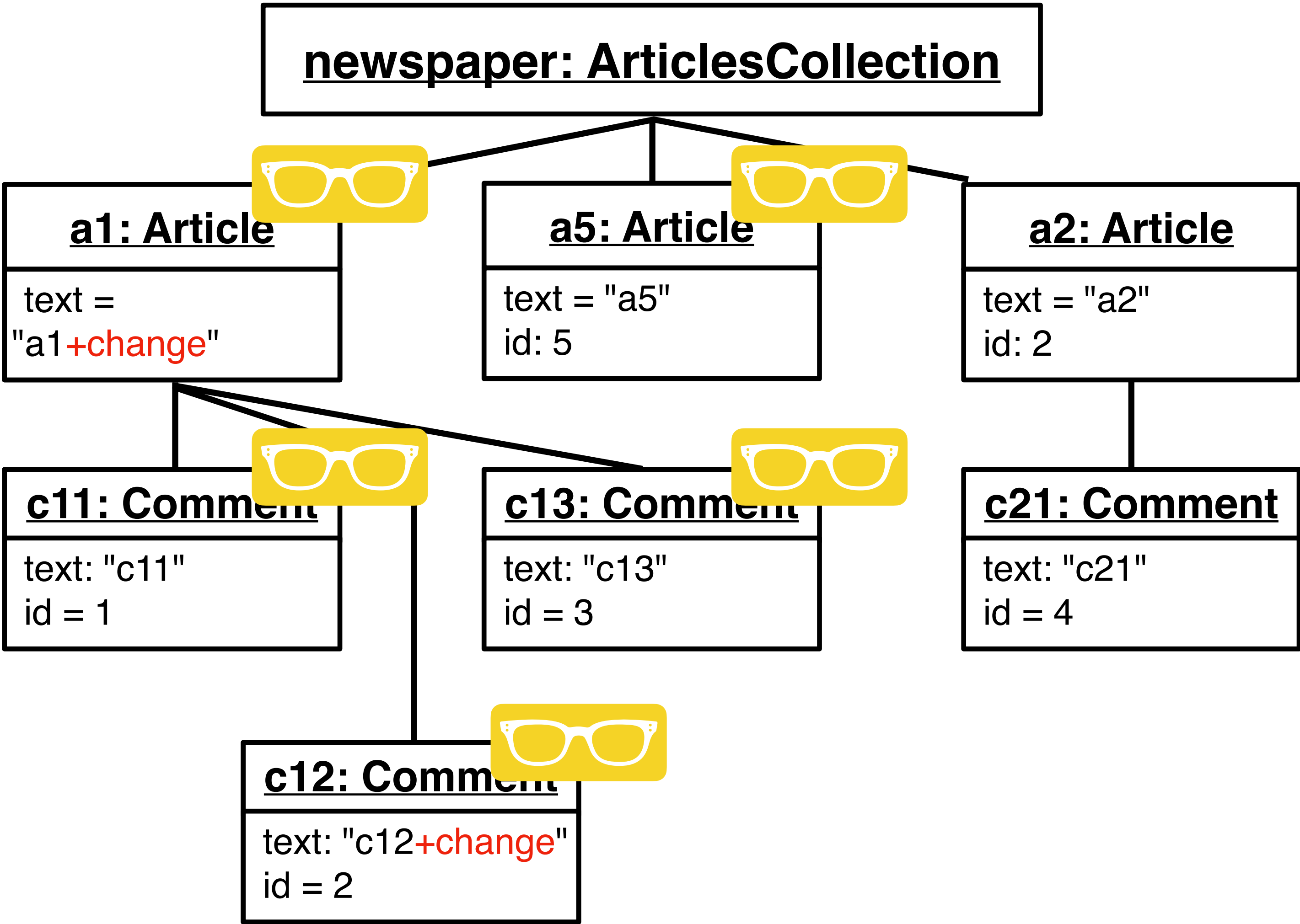


Example: mapping web requests to CRUD operations

Client web request

HTTP method	URI path component
GET	/articles
POST	/articles
PUT	/articles/1
DELETE	/articles/5
GET	/articles/1/comments
POST	/articles/2/comments
PUT	/articles/1/comments/2
DELETE	/articles/1/comments/3

Web service side state



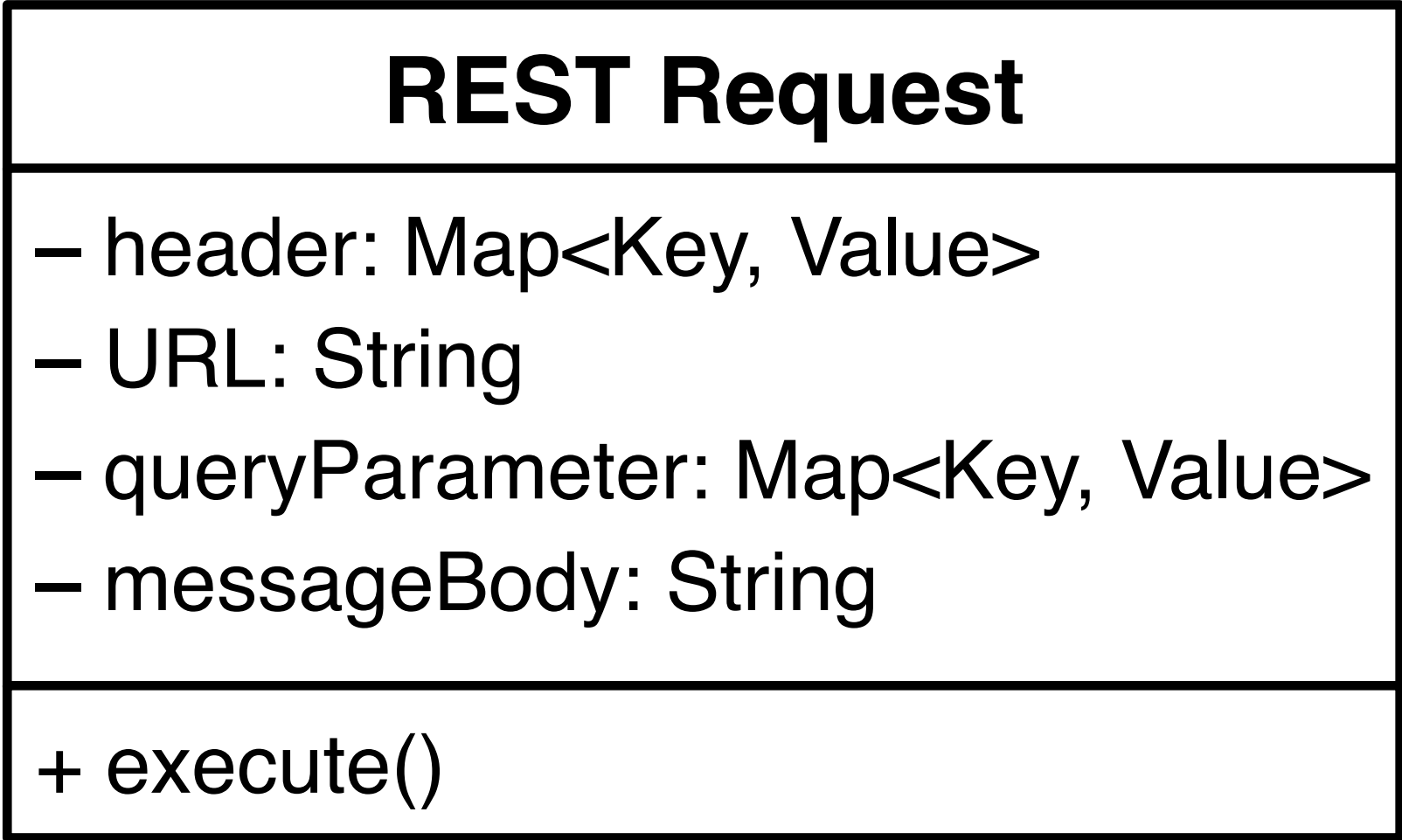
HATEOAS: hypermedia as engine of application state



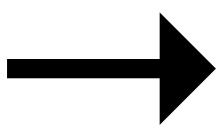
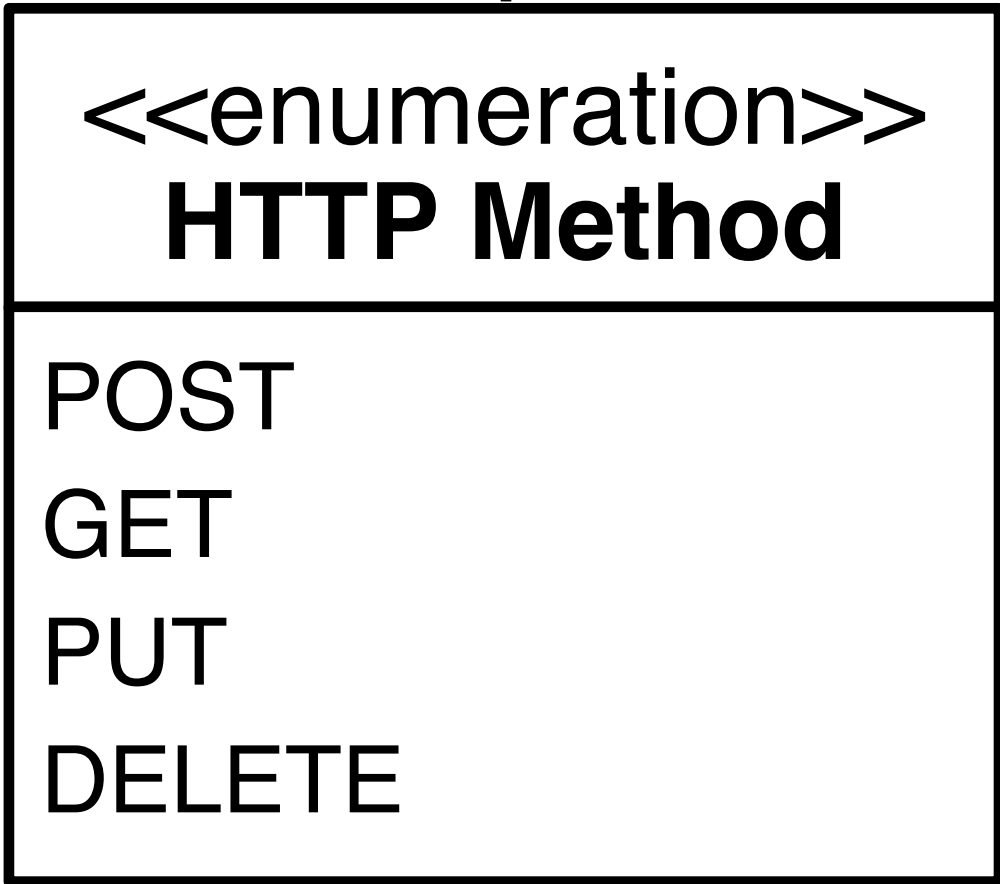
- Part of the uniform interface constraint of the REST application architecture
- **Hypermedia** refers to links to other content such as resources
- HATEOAS does not need a formal interface description language (IDL) to describe the API
 - IDLs such as **OpenAPI** help to describe RESTful APIs
- HATEOAS allows to traverse the web API by following links, similar to how you visit a website
- The RESTful architectural style officially requires responses to contain **links to related resources**

A web service can contain a root endpoint to provide links to explore the web service

Implementation of REST requests using HTTP



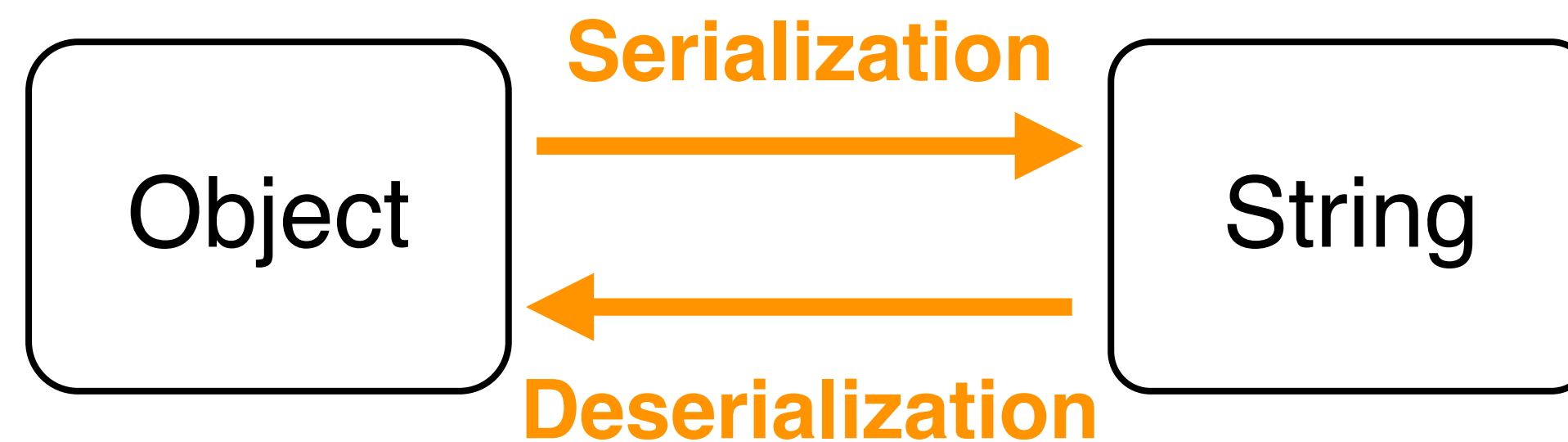
method



Attribute	Description / example
Header	Key value information to configure the request <i>Accept: */*</i>
Method	<i>GET</i>
URL	<i>https://ase.in.tum.de/articles</i>
Query parameter	Additional query parameter in the URL <i>https://ase.in.tum.de/articles?name=test</i>
Message body	Message to the web service <div>No body is sent to the server in GET requests</div>

Serialization

- The message body in requests and responses is arbitrary data
- There are several ways of representing objects as data, e.g. using **UTF-8 strings** and **JSON**



JSON: JavaScript object notation

- Open standard format
- Uses human readable text to transmit objects consisting of key value pairs
- Language independent data format

Movie
<ul style="list-style-type: none">– imdbID: String– title: String– simplePlot: String– genres: [String]– year: Int– rating: String

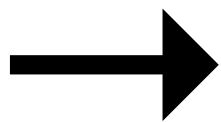
<u>theGodfather: Movie</u>
imdbID = "tt0068646" title = "The Godfather" simplePlot = "The ..." genres = [Crime, Drama] year = 1972 rating = "9.2"

```
{  
    "imdbID" : "tt0068646",  
    "title" : "The Godfather",  
    "simplePlot" : "The aging ...",  
    "genres" : [ "Crime", "Drama"],  
    "year" : 1972,  
    "rating" : "9.2"  
}
```

Implementation of REST responses using HTTP

REST Response

- statusLine: String
- header: Map<Key, Value>
- messageBody: String



Attribute	Description / example
Status line	Information about the result of the request, e.g. <i>HTTP/1.1 200 OK</i> HTTP Status Code
Header	Key value information about the response, e.g. <i>Content-Type: application/json; charset=utf-8</i> <i>Date: Mon, 18 Dec 2017 10:53:26 GMT</i> <i>Content-Encoding: gzip</i>
Message body	Message to the client, e.g. in JSON <pre>{ "title": "PSE 2022/23", "text": "I love this lecture!", "_links": { "author": "/author/42" } }</pre> HATEOAS

HTTP status codes

- RESTful APIs use HTTP status codes to classify responses

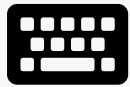
No error occurred	An error occurred
2xx Success 200 OK 201 Created (POST) 202 Accepted 3xx Redirection	4xx Client error 400 Bad request 401 Unauthorized 403 Forbidden 404 Not found 405 Method not allowed 5xx Web service error 500 Internal server error 501 Not implemented

RESTful APIs: HTTP and HATEOAS



- Communication using a layered architecture
 - Used in e.g.: **client - web service**, **Microservices**, and **Edge computing** architectures
- Provides access to **resources** with request response protocol
- RESTful APIs can be implemented using **HTTP**
 - **URIs** identify resources: **/articles/1/comments**
 - **HTTP methods** allow operations: GET, POST, PUT, DELETE
 - **Serialization** and **deserialization** transfer objects (e.g. JSON)
 - **HATEOAS** allows to explore and document the web API





L05E03 RESTful Client

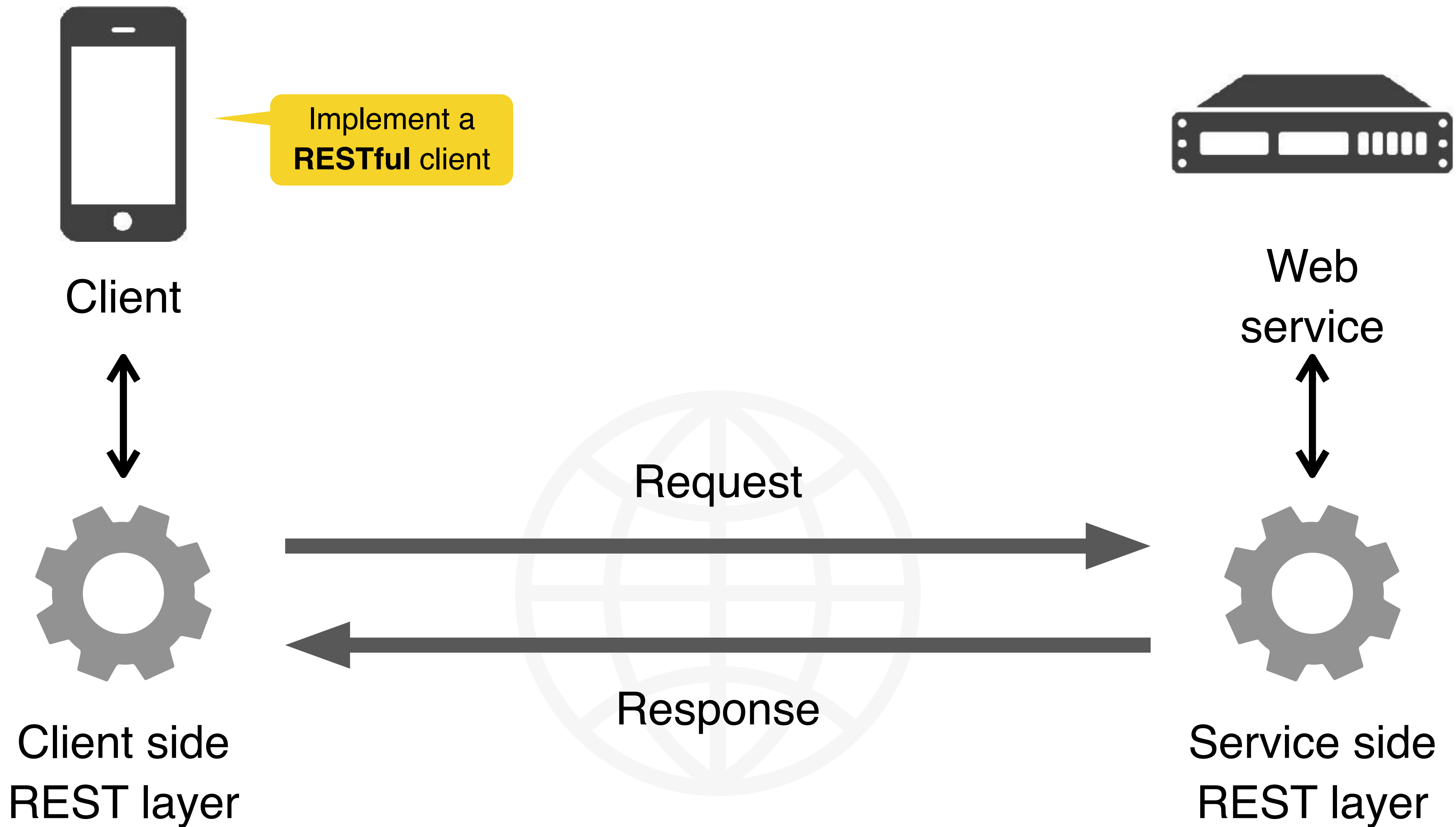
[Start exercise](#)

Medium

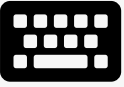
Not started yet.
Due Date: in 7 days

 20 min

 10 pts








L05E04 RESTful Web Service


[Start exercise](#)

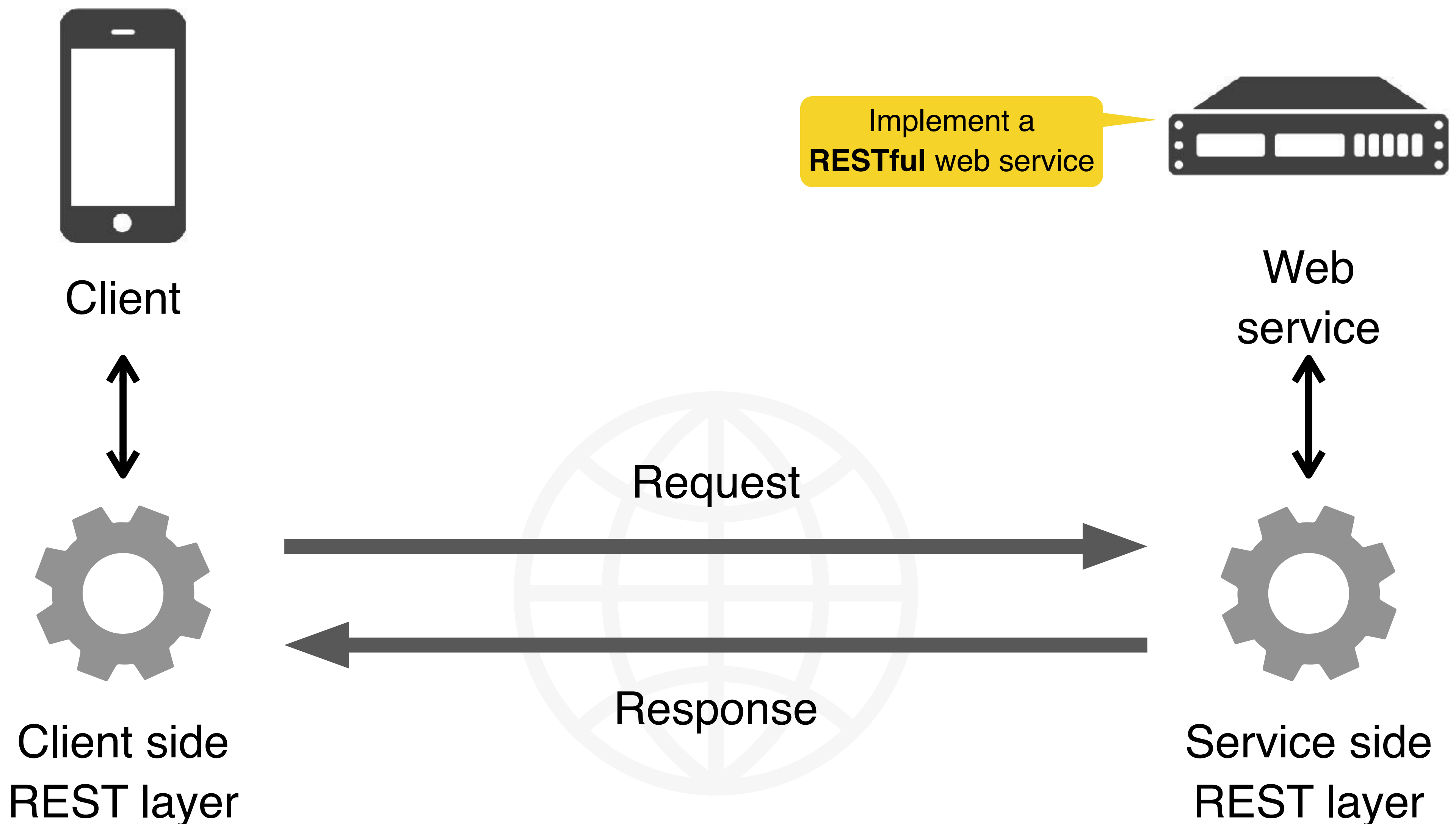
Hard

Not started yet.

Due Date: in 7 days

 20 min

 14 pts



Outline

- Broker
- REST as architectural style
- REST resources
- RESTful APIs: HTTP and HATEOAS

GraphQL

- gRPC

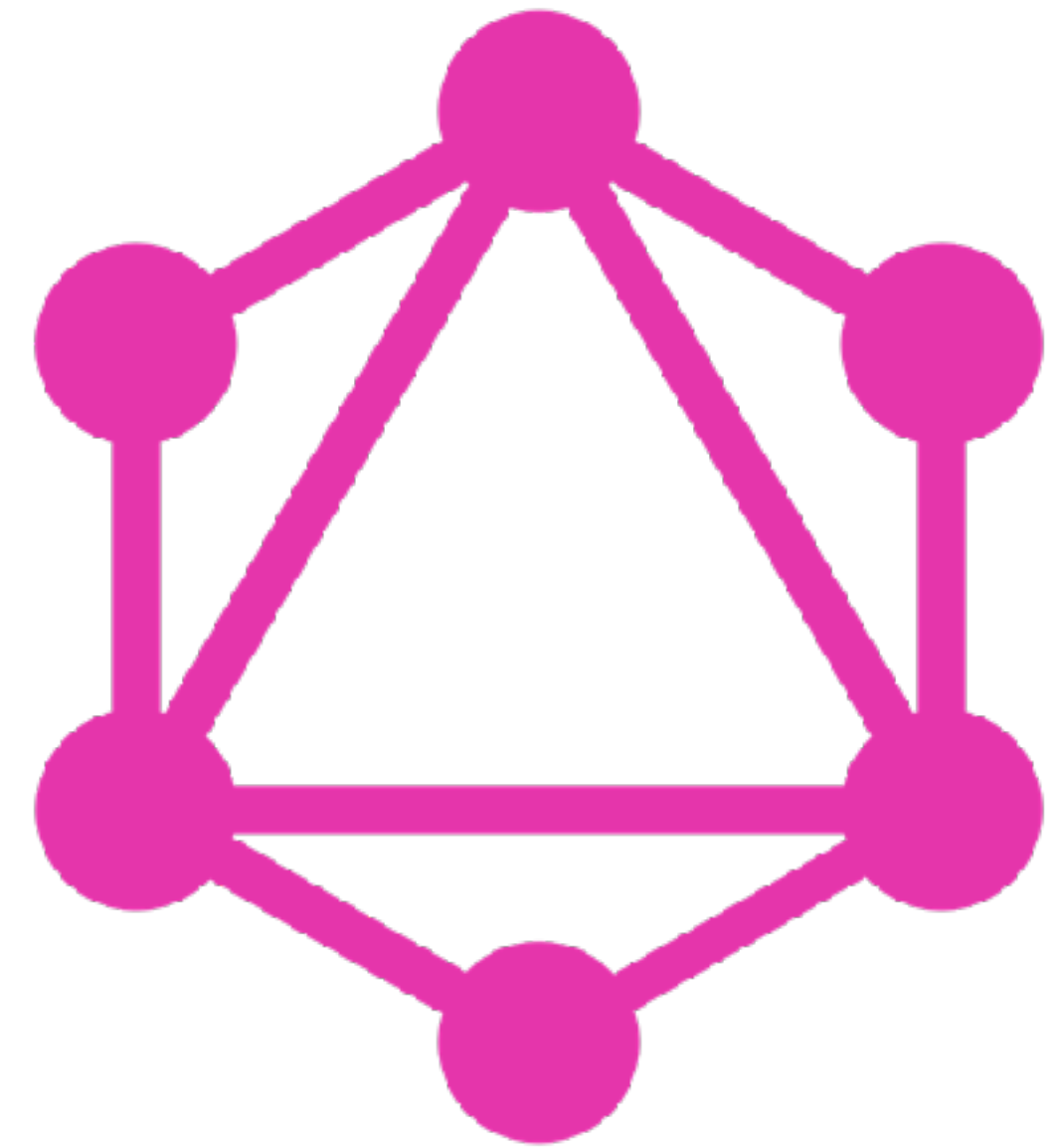
Weaknesses of REST



- Decisions where to locate functionality (in the client or in the web service) are often **complex** and **costly** to change after a system has been built
- Larger, nested web resource exchanges often **require multiple independent requests**
- The stateless nature prohibits e.g. publisher subscriber use cases

GraphQL as **refactored solution**

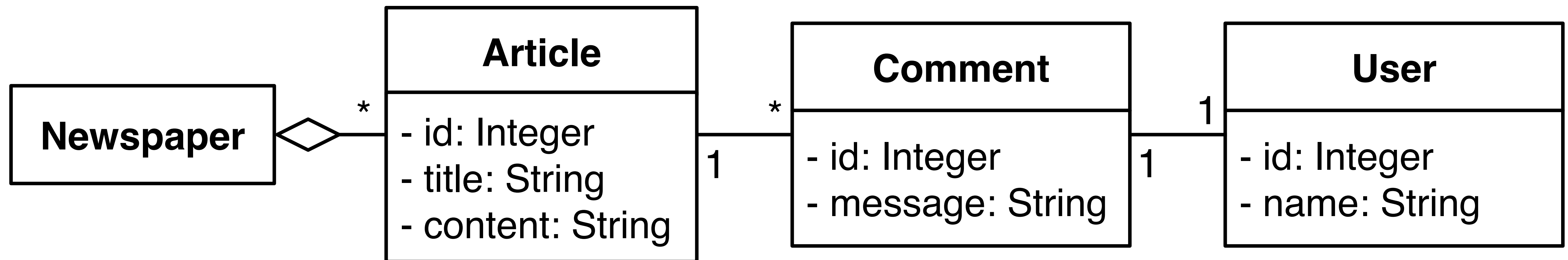
- Dynamic queries offer more flexible APIs
- Queries can traverse related objects and their fields
- Subscriptions allow to push data from the web service to the clients that listen to real time messages
- Also uses HTTP messages and encodes queries and responses in the HTTP body



GraphQL: fields

- In contrast to web resources in REST, GraphQL is using the term **object**
- At its core, GraphQL is offering the functionality to ask for specific **fields** (i.e. attributes) on objects
- Queries can **traverse related objects** and their **fields**
 - It is possible to sub select fields of related objects
 - This allows clients to fetch lots of related data in one request

Would require several roundtrips with a RESTful API



GraphQL: fields

```
{  
  articles {  
    title  
    content  
    comments {  
      message  
    }  
  }  
}
```

GraphQL query



```
{  
  "data": {  
    "articles": [  
      {  
        "title": "PSE Announcement",  
        "content": "...",  
        "comments": [  
          {  
            "message": "Great! 🎉"  
          }  
        ]  
      },  
      {  
        "title": "iPraktikum",  
        "content": "...",  
        "comments": [  
          {  
            "message": "Apple"  
          }  
        ]  
      }  
    ]  
  }  
}
```

GraphQL schema language

- Used to describe the API of GraphQL web services **independent** of the programming language
- The basic components are **object types**
- Every GraphQL service has a **query type** (might have a mutation type)
- Query types define the entry point of every GraphQL query

```
type Article {  
  id: ID!  
  title: String!  
  content: String!  
}
```

Example of a type definition in the
GraphQL schema language

Outline

- Broker
- REST as architectural style
- REST resources
- RESTful APIs: HTTP and HATEOAS
- GraphQL

 **gRPC**

- RPC framework with protocol buffers as the IDL
- Efficient connection between services across a system (e.g. Microservices), mobile devices, and browsers
- Generation of client and web service libraries in various languages
- Highly efficient on wire due to default binary encoding
- An API is structured in services that each offer service methods
- Bidirectional streaming with HTTP/2 based transport
- Used by Google, Apple, Dropbox, Netflix, ...



gRPC IDL: protocol buffer



- **Language neutral**, **platform neutral**, **extensible** way of serializing structured data
- Used in communications protocols, data storage, and more
- Advantages over XML/JSON for serializing structured data
 - 3 to 10 times smaller
 - 20 to 100 times faster
 - Data access classes are easier to use programmatically
- Define protocol buffer message types in **.proto files**
 - Each protocol buffer message is a small logical record of information, containing a series of **name value pairs**
 - Each message type has one or more **uniquely numbered fields** which enable an easy rollout of protocol changes

gRPC IDL: protocol buffer

```
message Person {  
  string name = 1;  
  int32 id = 2;  
  string email = 3;
```

Name independent identifier of a **field**, e.g. a name change in the **.proto** file won't break the API for older clients

```
  enum PhoneType {  
    MOBILE = 0;  
    HOME = 1;  
    WORK = 2;  
  }
```

Nested message definition

```
  message PhoneNumber {  
    string number = 1;  
    PhoneType type = 2;  
  }
```

```
  repeated PhoneNumber phone = 4;
```

Collection of multiple elements

```
}
```

gRPC services: method types

- **Unary RPCs**: client sends a single request to the web service and gets a single response back

```
message CreatConfirmation {  
    bool created = 1;  
}
```

Service Definition in a **.proto** file

```
service PersonService {  
    rpc CreatePerson (Person) returns (CreatConfirmation);  
}
```

Parameter

Return Type

Using gRPC in a Java project

- Message types and services are defined in a .proto file
- Definitions can be converted to many programming languages, e.g. Java
 - Generate Java types that match the message types
 - Generate a client and web service interface for services
 - You can use the **protobuf-gradle-plugin** to generate Java code

```
service PersonService {  
    rpc CreatePerson (Person) returns (CreatConfirmation);  
    rpc GetPersons (RequestPersons) returns (stream Person);  
    rpc CreateMultiplePersons(stream Person) returns (PersonsSaved);  
    rpc Greetings(stream Person) returns (stream Greeting);  
}
```


- **Broker**: systems interact via remote service invocations
- **REST** is an architectural style based on the layered architecture pattern
- REST typically use **HTTP** and **HATEOAS**
- REST clients and web services are easy to prototype
 - However: there are **tricky details** when the system becomes more complex
- Avoid REST bad practices
 - Upcoming lectures about anti-patterns and code smells
- **GraphQL** addresses some of the problems and enables querying across multiple resources using a type based schema definition language
- **gRPC** can include code generators and modern and efficient communication mechanisms

- F. Buschmann, K. Henney, D. C. Schmidt: Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, John Wiley & Sons, 2007
- Sun Microsystems, Inc.: Sun OS Documentation Tools, Formatting Documents, March 1990
- Corba: <http://www.corba.org>
- Roy Thomas Fielding: Architectural Styles and the Design of Network-based Software Architectures <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- Martin Fowler: Enterprise Integration Using REST: <http://martinfowler.com/articles/enterpriseREST.html>
- Martin Fowler: Richardsons Maturity Model: <http://martinfowler.com/articles/richardsonMaturityModel.html>
- Jim Webber: REST in Practice: Hypermedia and Systems Architecture, O'Reilly, 2010
- Robert Daigneau: Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison Wesley, 2011
- Phil Sturgeon: APIs you won't hate: <https://apisyouwonthate.com/blog>
- HTTP standard: <https://www.w3.org/Protocols>