

# CSE1500: Web - Lecture 4 - REST & RESTful APIs


## Lecture Summary

The WDT Team

Ujwal Gadiraju, Sagar Chethan Kumar, Ciprian Ionescu

## Contents

<b>1</b>	<b>REST &amp; RESTful APIs</b>	<b>2</b>
1.1	Uniform Interface . . . . .	2
1.2	Stateless communication . . . . .	3
1.3	Client-server Architecture . . . . .	3
1.4	Cacheability . . . . .	4
1.5	Layered System . . . . .	4
1.6	Code-on-demand . . . . .	5
<b>2</b>	<b>Authentication</b>	<b>5</b>
<b>3</b>	<b>References and Further Reading</b>	<b>5</b>

 These summaries are not extensive overviews of lecture content and **DO NOT** cover all possible exam material. Rather, these serve as a refresher and capture **main points** for a given lecture.

## Learning Goals

You should be able to do the following after following the **lecture** and **assignment**:

- **Develop server-side code for authorization, cookies, sessions, and database access. (LO5)**

# 1 REST & RESTful APIs

**REST**, also known as **RE**presentational **S**tate **T**ransfer, is an architectural style for building and structuring Application Programming Interfaces (APIs) – the building blocks that enable software applications to communicate with each other over the web. Systems employing REST principles, often called RESTful systems, are characterized by their *statelessness*, *separation of client and server* and *uniformity*.

## 1.1 Uniform Interface

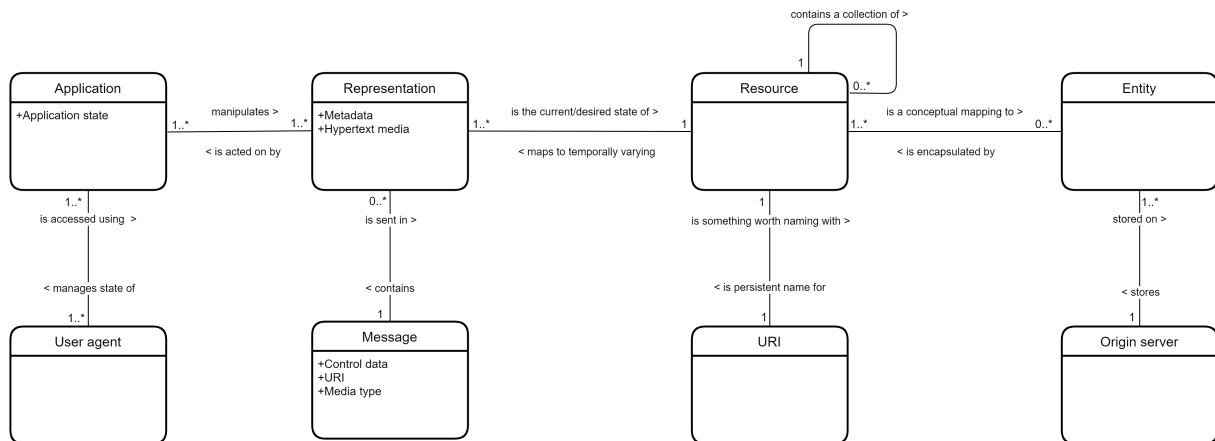


Figure 1: An entity-relationship model of the concepts expressed in the REST architectural style.

Kmcnamee, [CC BY-SA 4.0](#), via Wikimedia Commons

Perhaps the most important principle, REST's *uniform interface principle* is a fundamental principle of REST that ensures that all resources in a RESTful API can be accessed and manipulated using the same set of conventions and methods. The uniform interface is based on a few key characteristics:

### Resource Identification

Each resource in a RESTful API is identified by a unique URI (Uniform Resource Identifier). This URI serves as a global address for the resource and can be used to access and manipulate it. This URI *should not* contain information about what actions or modifications the resource(s) should suffer, or any state about the resource.

### Examples

- `/movies/tt123456` (URI clearly identifies a unique resource)
- `/movies` (URI clearly identifies a collection of resources)
- `/movies/tt123456`, `/lists/2`, `/users/cionescu` (URI is consistent and predictable)
- `/movies?search=Batman&sortBy=date` (Information that identifies a unique resource is separated from information that doesn't, usually by using query parameters for non-identifiers.)
- `/lists/2/movies` (URI uses nesting when appropriate)

## Counter-examples

- `/movies/createMovie` (*URI uses verbs or otherwise contains information about an action*)
- `/movies?id=tt123456&detailed=true` (*Non-identifying parameters are not clearly separate from identifying data*)
- `/users/cionescu?gender=male` (*Request uses the URI to represent data that belongs to a resource's state*)

## Resource Manipulation through Representations

Resources are manipulated using standard HTTP methods. For CRUD (Create, Read, Update, Delete) operations, `GET` is used to retrieve a representation of a resource, `POST` is used to create a new resource, `PUT` is used to update an existing resource, and `DELETE` is used to delete a resource.

Clients have the ability to modify or delete resources by sending specific HTTP requests along with a representation of the desired state of the resource. Because a representation of the data received from the server contains all the necessary information for clients to understand the resource's current state, a client has all the necessary information to also perform any mutations to the state.

## Examples

In the example in [Figure 2](#), we've modeled an extremely simple contest management system. Assuming a contest resource for the Delft Algorithm Programming Contest already exists in the system, we'd like to start the contest. Instead of abusing the URI of the resource to encode this action, such as by doing a `POST /contests/dapc/start`, we retrieve the state of the contest at the current moment. This gives us all the information we need to update the state, which we'll do by means of a `PUT` request.

Notice how we represent the entire resource's state in the update, and how the system confirms the mutation by responding with the representation of the new state. While this is how this interaction is modelled in idiomatic REST, in the real world we might see the `PATCH` HTTP method used for updating, alongside `PUT`. The difference usually lies in the implementation: a `PATCH` does not need to receive a complete representation of the resource, only the mutated fields.

## 1.2 Stateless communication

Interactions between systems that follow the REST paradigm are stateless. This does not mean that the entire system is stateless! Rather, it means that the server does not need to know nor keep track of what state its clients are in.

The client must convey its session data explicitly to the server in a self-contained manner, ensuring that each data packet can be deciphered *independently*, without relying on contextual cues from prior packets in the communication flow.

Through this, both the server and the client can understand any message received in isolation, irrespective of knowing about previous messages. This constraint of statelessness is enforced through the use of resources, rather than commands. Resources are the nouns of the Web – they describe any object, document, or thing that you may need to store or send to other services.

## 1.3 Client-server Architecture

Perhaps the most self-explanatory principle, it prescribes a clear separation between a server and a client. A server acts as an owner and manager of data, while a client acts as a consumer

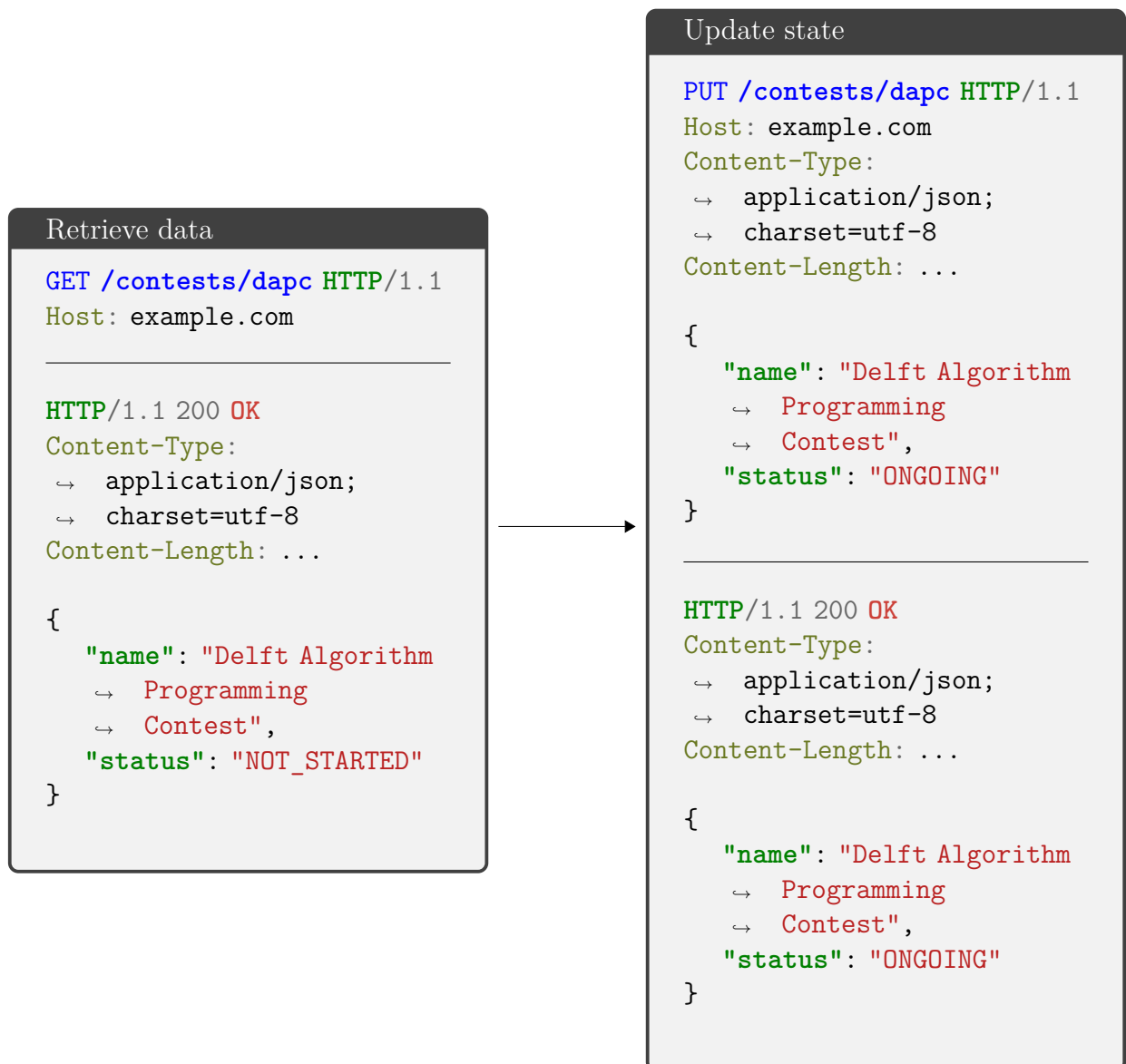


Figure 2: Example interaction of starting a contest in a (fictional) contest management system.

for the server's data. A server is mainly concerned with storing data, while a client is mainly concerned with presenting the data to a user, or to a dependant system.

## 1.4 Cacheability

Responses from a RESTful API should explicitly indicate whether they can be cached and, if so, for how long. This information is typically communicated through HTTP headers, such as the `Cache-Control` header. By providing clear caching directives, RESTful services enable intermediaries like proxies and browsers to store copies of responses locally. This enhances performance by reducing the need for repeated requests to the server for the same resource.

## 1.5 Layered System

REST inherently supports a multi-layered architecture as it's built on top of HTTP. This layered system allows for the introduction of intermediaries, such as proxies and gateways, which can

enhance system functionality without impacting the core components. REST mandates that clients and servers must not break HTTP's layering and proxying capabilities.

## 1.6 Code-on-demand

While this principle in practice is not really taken into account today when designing REST APIs, it can be argued that web apps which make asynchronous JavaScript (AJAX/fetch) requests are a form of code-on-demand that consumes and transforms data from REST APIs.

## 2 Authentication

Authentication is usually seen by users as a stateful operation: they log in, they perform whatever actions they need, then they log out. However, as established earlier, REST has no notion of state, which raises an issue: how do we perform authentication with a REST API?

A naive solution would be to just store the user's password client-side, and transmit it for every request. This might seem safe enough, but it breaks a crucial feature: session expiration. If an attacker somehow compromises the client's state, they would have a persistent means of authentication to the server, and any other service where the user would reuse the password.

A different solution might be to assign each user a random session token when logging in, which can be sent in place of a password for each request. While this addresses the problem of expiring sessions, in today's world it can introduce a big downside: scalability. Because the server would have to store a session token  $\mapsto$  user mapping for every open session, the component responsible for handling this data can very quickly become overloaded, as every other component would depend on it. A failure of that component would mean a complete failure of the service.

Enter the *JSON Web Token (JWT)*. Unlike session tokens, JWTs are self-contained: they carry all necessary information to verify their validity and identify the user without requiring server-side storage. When a user logs in, the server creates a signed token that encodes information such as the user ID and any relevant claims. The client then includes this token with each request, allowing the server to verify and authenticate the request purely based on the token content.

JWTs offer several benefits, such as statelessness and scalability, but must be handled with care. For instance, signing keys must remain secure to prevent token tampering, and tokens should be given an appropriate expiration time to minimize the impact of a compromised token.

## 3 References and Further Reading

Here are some useful resources for learning more about **Lecture 4**:

- [REST Intro](#)
- [AWS RESTful API](#)
- [Richardson Maturity Model](#)