

EDAF75
Database Technology

Lecture 6: REST API

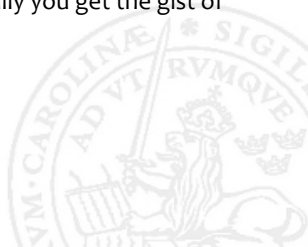
Christian.Soderberg@cs.lth.se

February 5, 2026



The use of REST in this course

- ▶ This course is not about REST API's, we only use them to make your coding more focused on what really matters in the course, i.e., the databases
- ▶ The only prerequisite for this course is a first programming course, so we will make some simplifications to make things easier – hopefully you get the gist of REST APIs anyway



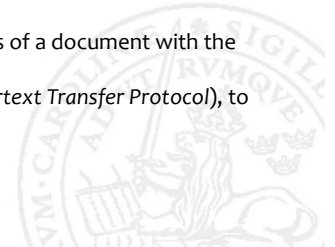
Background

- ▶ Programs today are often split up in a *frontend* and a *backend*, where the frontend is typically written in JavaScript or WebAssembly, and the backend can be written in a number of different languages (such as Java or Python)
- ▶ The frontend and backend need a way to communicate – one common technique is to use what's called a REST API



Background

- ▶ What happens when we enter the url
`http://www.example.com/index.html`
in the address bar of our browser, and press 'Return'?
- ▶ We have one *server* and one *client*:
 - ▶ A *Web Server* is running on the computer named `www.example.com`, it waits for *requests* for its *resources*
 - ▶ Our browser (the *client*) sends a request for the contents of a document with the path `/index.html`
 - ▶ The client and server use a specific protocol, HTTP (*Hypertext Transfer Protocol*), to communicate



HTTP requests

- ▶ There are several kinds of HTTP requests, amongst them:
 - ▶ GET: ask the server for some resource
 - ▶ POST: ask the server to accept some data as a new resource
 - ▶ PUT: ask the server to replace some resource with some data
 - ▶ DELETE: ask the server to remove some resource
- ▶ The requests are plain text sent between client and server

HTTP requests

- ▶ Example of an HTTP request:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: curl/7.67.0
Accept: */*
```

- ▶ Each HTTP request has three parts:

- ▶ A *request line*: tells what the client wants to do (i.e., GET/POST/...), and with what resource it wants to do it
- ▶ Some *header lines*: each line in the header is a key/value-pair specifying the request – the only required header line is Host :
- ▶ An optional *body*: it can contain data for a POST or PUT request (GET requests normally have empty bodies)

HTTP responses

- ▶ Example of an HTTP response:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 236609
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Thu, 01 Feb 2024 10:48:42 GMT
Etag: "3147526947"
Expires: Thu, 08 Feb 2024 10:48:42 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Content-Length: 1256
...
```

- ▶ A HTTP response has three parts:

- ▶ The *status line*: contains the name of the protocol (typically HTTP/1.1), a numeric return code, and short message
- ▶ Some *headers*: various key/value pairs, describing date, content length, content type, etc
- ▶ An optional *body*: This can be arbitrary data, and it is where html-code is returned

HTTP return codes

- ▶ Some of the most important HTTP return codes are:

- ▶ 200: OK
- ▶ 201: Created
- ▶ 202: Accepted
- ▶ 400: Bad Request
- ▶ 404: Not Found

- ▶ The (probably) least important return code:

- ▶ 418: I'm a teapot

Example

- ▶ Our browser sends an HTTP request (text) to the web server
- ▶ The server processes our request, and sends an HTTP response to our browser
- ▶ Our browser reads the response, and if the status is 200 (OK), and its Content-Type is text/html, it renders the body of the message as an html-page
- ▶ If the response status is anything else, the browser displays an appropriate error message

Using curl

- ▶ Instead of using our browser, we can fetch web pages on the command line using the curl command:
`curl -X GET http://www.example.com/index.html`
- ▶ It's not very useful for regular web pages, but will come in handy later today

REST services

- ▶ A REST server (or REST service) is a server which lets clients access and manipulate textual representations of web resources using stateless operations
- ▶ A REST server typically uses HTTP (GET/POST/PUT/...), to communicate with its clients
- ▶ Each resource normally has two URLs:
 - ▶ /students: represents the collection of all students
 - ▶ /students/123: represents the student with id 123

REST services

- ▶ To get information regarding a specific resource, the client makes a GET request with the URL of the resource
- ▶ To add a new resource, the client makes a POST to the corresponding collection
- ▶ REST services typically use JSON for data representation

JSON

- ▶ JSON is short for *JavaScript Object Notation*, and it's a human-readable text format for transmitting data
- ▶ JSON's basic data types are: *number*, *string*, *boolean*, *array*, *object*, and *null*.
- ▶ We can define a student as an object:

```
{
  "id": 123,
  "name": "Amy",
  "gpa": 3.9,
}
```



JSON

- ▶ A course with arrays of weekly lectures and lab sessions can be defined as:

```
{
  "courseCode": "EDAF75",
  "lectures": [
    {"day": "Monday", "startTime": 13},
    {"day": "Thursday", "startTime": 13}
  ],
  "labs": [
    {"day": "Wednesday", "startTime": 10},
    {"day": "Wednesday", "startTime": 13},
    {"day": "Friday", "startTime": 8}
  ]
}
```



JSON

- ▶ Applications for a given student can be defined as:

```
{
  "id": 123,
  "name": "Amy",
  "gpa": 3.9,
  "applications": [
    {"college": "Stanford", "major": "CS"},
    {"college": "Stanford", "major": "EE"},
    {"college": "Berkeley", "major": "CS"}
  ],
}
```



REST and CRUD

- ▶ CRUD is an acronym for *Create*, *Read*, *Update*, and *Delete*
- ▶ REST services are often used for CRUD
- ▶ Create: POST
- ▶ Read: GET
- ▶ Update: PUT
- ▶ Delete: DELETE



Example

In the following few slides we'll assume we have a REST server running at port 4567 on localhost, with our college application information as resources

- ▶ To get information about all students, we make a GET request for the resource /students:

```
GET http://localhost:4567/students
```

- ▶ We can try out the request above using curl:

```
curl -X GET http://localhost:4567/students
```


or we can use the URL in a browser
- ▶ The server usually returns JSON data



Example

- ▶ If we want information about a student with a given id, we can use her full url:

```
GET http://localhost:4567/students/123
```
- ▶ To get information about any students named "Irene", we add a *query string* to the resource (URL) for all students:

```
GET http://localhost:4567/students?name=Irene
```
- ▶ We must make sure that our path and all our parameters are properly URL-encoded (our shell may or may not help us with this)



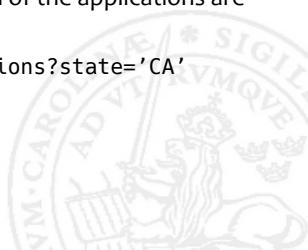
Example

- ▶ We can expand the URL for a resource to dig in deeper – to see all applications by a given student we can use:

```
GET http://localhost:4567/students/123/applications
```

- ▶ And we can combine that with a query string to see which of the applications are for colleges in California:

```
GET http://localhost:4567/students/123/applications?state='CA'
```



Example

- ▶ To add a new student, we make a POST to /students
- ▶ Normally we put the data for the new student in the body of the request, as a JSON description
- ▶ It's common practice to let the server return the id or URL of the newly created resource



What's the point?

- ▶ Asking a server to send a page, and then rendering it in the browser, is costly – the server potentially has to serve many clients, and the client/browser has to create each page from scratch every time
- ▶ Much more efficient is letting a JavaScript or WebAssembly program running in the browser fetch only the data it needs, and then update parts of the page
- ▶ Also, instead of getting a web page, we might be interested in saving the important data directly to our computer

REST services, the CliffsNotes

- ▶ Resources normally have two urls:
 - ▶ One collective: /students
 - ▶ One individual: /students/123
- ▶ We use HTTP-methods to operate on our resources:
 - ▶ GET to get data
 - ▶ POST to add data
 - ▶ PUT to update data
 - ▶ DELETE to delete data
- ▶ We use query parameters to fine tune a search
- ▶ Resources are nouns, in plural form (sometimes we also have operations which doesn't involve resources, their names can be verbs)
- ▶ We use JSON to represent data
- ▶ We use CamelCase for attribute names

Things we simplify in the course

- ▶ Resources must often be protected, there are several ways to handle authorization, but we won't deal with it
- ▶ Some requests may return lots of data, and we sometimes don't want all at once – a real REST API uses some kind of paging and continuation tokens for this, we will not
- ▶ Ideally we want to embed documentation for the service in our responses, e.g., a response could contain links to related resources – we won't do that
- ▶ Sometimes responses 'sideload' information which might be useful for the client, or embed hierarchies of objects, to avoid extra roundtrips – we will not do that

Frameworks

- ▶ Up until this year, the course had a section about how to connect to SQL from a Java program, but it involves a lot of details which has very little to do with databases, and this year I've thrown it out
- ▶ There are many different libraries/frameworks which helps us implementing REST services in Python, to make things simple, we're going use a library called Bottle

Exercises

- ▶ Add the endpoint GET /students
- ▶ Add the endpoint GET /students/<:id>
- ▶ Add the endpoint GET /students/?name\=Amy
- ▶ Add the endpoint POST /students
- ▶ Add the endpoint GET /students/<:id>/applications
- ▶ Add endpoint for GET /applications/?college\=Stanford\&major\=CS

