Distributed Systems

Tutorial #2

RESTful APIs: Representational State Transfer Architecture Style

Yaron Hay
Largely based on the slides of
Amir Dachbash and Dolev Adas

Technion
Israel Institute of Technology
Faculty of Computer Science

Fall 2024/25
November 20, 2024

v1.0 20/11/24

TECHNION | The Henry and Marilyn Taub Faculty of Computer Science

Motivation

- Communication between two components in a DS is largely a solved problem
  - TCP for basic data transfer
  - RPC or HTTP as programming models
  - Algorithms that deal with consistency of state
- Software, on the other hand, has unsolved problems
  - Almost all software is extremely difficult to change
  - Software systems are difficult to integrate

REST: Representational State Transfer

What **isn't** REST?

So, what exactly is REST?

- Is it a protocol? No.
- Is it a communication system? No.
- Is it a service definition? No.

REST is an architecture style for how the architecture of an Internet-scale distributed (hypermedia) system should behave.

- Defines a set of constraints to be used for creating web services.
- It's a spectrum: a system may be hang between not being a RESTful API to being a fully RESTful API

Architecture Concept

Representational State Transfer (REST) is an architectural style which defines how a client should interact with a web service via messages.
REST encapsulates the system's entities by the using the abstraction of resources (nouns).

- Each resources is accessible to the clients via a Uniform Resource Identifier (URI).
- Servers respond to clients with a representation of the resource.

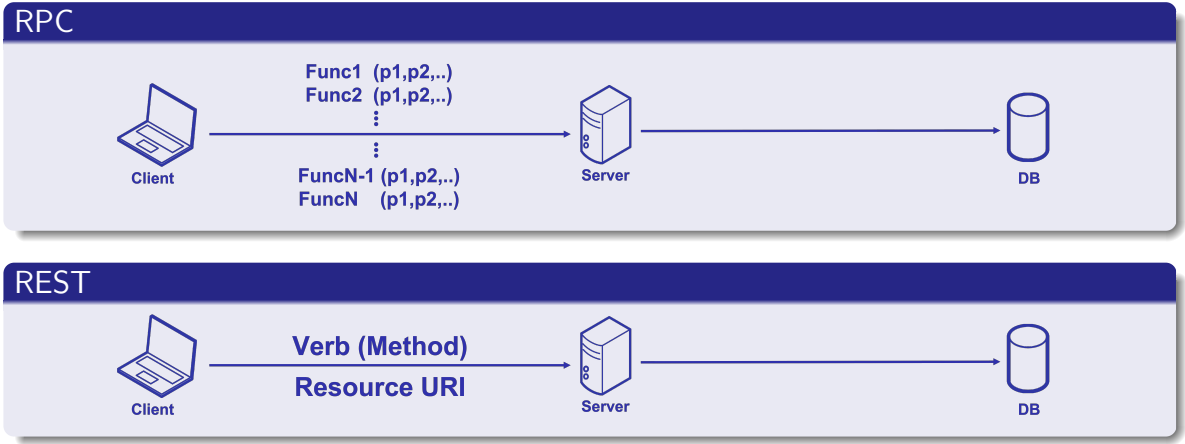
Resources are either accessed by or manipulated using one of the verbs (methods).

- The is one predefined and uniform set of verbs
- Verbs are stateless

An API following all 6 guiding constraints of REST, can be called a RESTful API.

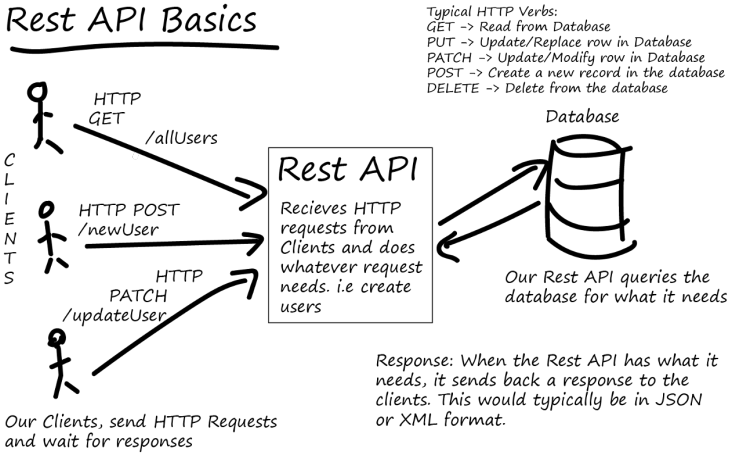
REST: Representational State Transfer

REST vs and RPC



RPC is a protocol. REST is an architecture style, NOT a protocol.

TECHNION | The Henry and Marilyn Taub Faculty of Computer Science


Rest API Basics
Typical HTTP Verbs:
GET -> Read from Database
PUT -> Update/Replace row in Database
PATCH -> Update/Modify row in Database
POST -> Create a new record in the database
DELETE -> Delete from the database
CLIENTS
HTTP GET
/allUsers
HTTP POST
/newUser
HTTP PATCH
/updateUser
Rest API
Recieves HTTP requests from Clients and does whatever request needs. i.e create users
Database
Our Rest API queries the database for what it needs
Our Clients, send HTTP Requests and wait for responses
Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.

REST $\neq$? **HTTP**.

- REST and HTTP both intend to make the web standardized and more streamline. This causes confusion. Don't make this mistake.
- An interface is not RESTful, if it does not respect the guiding principles of REST.
- REST can be implemented using many different technologies.

REST was introduced in 2000 by Roy Fielding in his doctoral dissertation.

> "I am getting frustrated by the number of people calling any HTTP-based interface a REST API."
>
> Roy Fielding, 2008

Still relevant in 2021-2022!

RESTful APIs using HTTP

Resource Naming

Use the the HTTP path as the resource's URI.
Use HTTP methods as the verbs (**GET**, **POST**, **PUT**, **DELETE**, etc.).

- Get all students
  - [GET] http://api.college.com/students
- Get the student with id {student-id}:
  - [GET] http://api.college.com/students/{student-id}
- Create a new student:
  - [POST] http://api.college.com/students
- Delete the student with id {student-id}:
  - [DELETE] http://api.college.com/students/{student-id}
- Update the student with id {student-id} (creates one if doesn't exist):
  - [PUT] http://api.college.com/students/{student-id}

When designing URIs for different types of resources, you might encounter some resources that depend on each other.

Assume there are messages and comments.
Getting all comments for a some message could be:

[GET] http://example.com/messages/3248234/comments

Now we should ask ourselves, what should the URI for comments be?

- Should it be /comments/{comment-id}?
- Or should it be /messages/{message-id}/comments/{comment-id}?

- Filtering
  - [GET] http://api.college.com/students?birth-city=haifa
- Sorting
  - [GET] http://api.college.com/students?sort=age
- Filtering and Sorting
  - [GET] http://api.college.com/students?birth-city=haifa&sort=age

Move from Verbs to Nouns.

- Identification of involved resources (students , courses etc)
- Just the noun not the action (students not getStudents or fetchStudents)
- Use plural (students not student , courses not course)

Poor URI

[GET] /getStudents.do?id=10

Good URI

[GET] /students/10

Then, make it generic:

[GET] /students/{student-id}

- REST guidelines suggest using a specific HTTP method on a particular type of call made to the server
- HTTP Provides some basic methods for Create / Read / Update / Delete (CRUD) operations
  - Get
  - Delete
  - Post
  - Put
  - Patch

- GET
  - Retrieves a representation of a resource
  - Idempotent
  - Read-only method
- DELETE
  - Removes a resource
  - Idempotent
    Although, calling DELETE on a resource a twice will result with a NOT FOUND Error (404).
- POST
  - Used to **create** a new resource
  - Not an Idempotent
  - Parameters are found within the request body (not in the URI)

- PUT
  - Replaces an entire resource
  - Used to update a resource
  - Idempotent
  - If the resource doesn't exist, you should create it and return an 201 (created) status code.
- POST and PUT create a lot of confusion.
  A great comparison for clarifying the difference: here
- PATCH
  - Used to make a **partial** update on a resource
  - Not idempotent

Status Range	Description	Examples
1xx	Informational	100 Continue
2xx	Successful Requests	200 OK 201 Created 204 No Content
3xx	Redirection	302 Found 307 Temporary Redirect 304 Not Modified
4xx	Client Error	400 Bad Request 401 Unauthorized 404 Not Found
5xx	Server Error	500 Internal Server Error

Table: HTTP Status Code Categories

RESTful APIs using HTTP

Example POST

Request

```
POST http://fashionboutique.com/customers
Accept: application/json
Body:
{
  "customer": {
   "name": "John Doe",
   "email": "j.d@236351.org"
  }
}
```

Response

```
201 (CREATED)
Content-type: application/json
Body:
{
  customer": {
    "id": 123,
    "name": "John Doe",
    "email": "j.d@236351.org"
 }
}
```

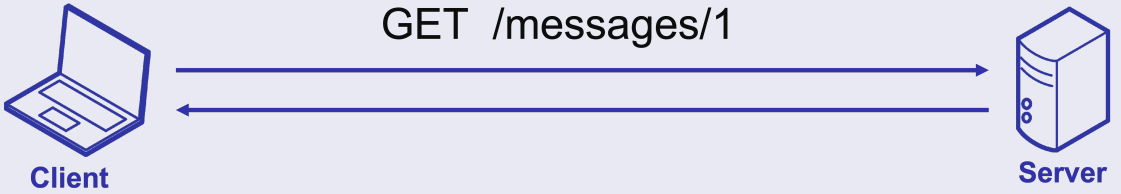
- We have designed a service ready to serve clients.
- How do clients know what our resources addressing scheme was?
- How do clients know how to navigate in our system?
- Options?
  - Documentation? Not a great option.
  - Hopefully, you don't go and read the site's documentation
  - How do you navigate when visiting a website ?

- REST has no guidelines for service definition specification.
- When was the last time you looked up any documentation to use a website?
- The answer is Hyperlinking!


```
{
  "id": "01",
  "content": "Hello World!",
  "author": "dolev",
  "postedDate": "01-02-2017"
}
```

GET /messages/1


```
{
  "id": "01",
  "content": "Hello World!",
  "author": "dolev",
  "postedDate": "01-02-2017",
  "href": "/messages/1",
  "comments-href": "/messages/1/comments",
  "likes-href": "/messages/1/likes",
  "shares-href": "/messages/1/shares",
  "profile-href": "/profiles/dolev ",
}
```

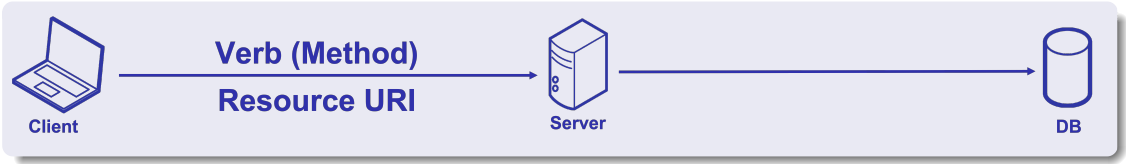



```
{
 "id": "01",
 "content": "Hello World!",
 "author": "dolev",
 "postedDate": "01-02-2017",

 "links" : [
```

```
    { "href": "/messages/1", "rel": "self" },
    { "href": "/messages/1/comments", "rel": "comment
    { "href": "/messages/1/likes", "rel": "likes" },
    { "href": "/messages/1/shares", "rel": "shares" }
    { "href": "/profiles/dolev", "rel": "author" }
 ]
}
```

The Architectural Constraints of Rest

- So, we built a web server which has an API composed from Verbss and Resources.
- Can we call it a RESTful API? No.



- REST defines 6 architectural constraints in which
- Following all 6 constraints make your API truly RESTful.

Client-Server Architecture

- Client application and server application MUST be able to evolve separately without any dependency on each other.
- A client should know only resource URIs, and that's all.

Why?

- Improve the portability of the user interface across multiple platforms.
- Improve scalability by simplifying the server components.

Statelessness

- Each request from client to server must contain all the information necessary to understand the request.
- Do not take advantage of any stored context on the server.
- Session state is therefore kept entirely on the client.

Why?

- Scalability
- Reliability
- Visibility

Cacheability

- Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
- If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Why?

- Latency
- Bandwidth
- Scalability: The system can handle more clients

Interface Uniformity

Defines the interface between clients and servers:

- Identification of resources
- Actions on Resources Through Representations
- Self-descriptive Messages
- Hypermedia as the Engine of Application State (HATEOAS)

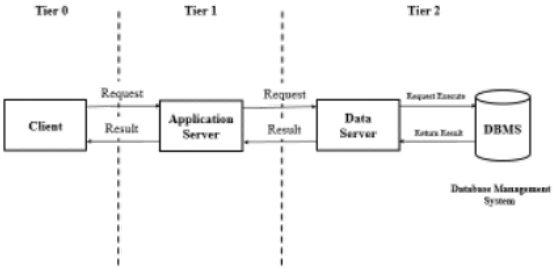
Why?

- Simplicity - Use the power of standards
- Usability – i.e navigation

Layered System

The web service should have a layered design.

- This constraint allows the architect to **inject layers** of service between the server and the client while the layering remains **transparent to the client**.
- Each layer in the system can only talk to the layer adjacent to it.



Code-on-Demand

- REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.
- This simplifies clients by reducing the number of features required to be pre-implemented.

- Common Use: scripts and applets.
- Does this remind you something?
- Why is it optional?
  - It increases the coupling between the client and the server.
  - However, it does enable the application to be more extensible.

- Spring is a lightweight framework.
- It can be thought of as a framework of frameworks because it provides support to various frameworks .
- The framework can be defined as a structure where we find solution of various technical problems.

Demo

Implementing a RESTful Service Using Spring

Using Spring, we can implement a RESTful service for managing employees in a company.

- The application will expose data via a REST API
- Jackson will be used for converting Object to JSON and vice-verse.
- We'll use an embedded H2 database as our data source.
- We'll use it along with JPA and Hibernate to access the data from the database.

A user can **create**, **retrieve**, **update** and **delete** an employee using this application.
An employee has an **id**, **username**, **first name**, **last name**, **email**, and a **date of birth**.

Tutorial: https://www.expatdev.com/posts/build-rest-api-spring-boot-kotlin/
Source Code: https://github.com/anirban99/spring-boot-examples/tree/main/spring-boot-boilerplate

Implementing a RESTful Service Using Spring

Domain Model - Kotlin Data class

```
@Entity
@Table(name = "employee")
data class Employee (
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY) val id: Long?,
  @Column(name = "user_name", unique = true, nullable = false)
    val userName: String,
  @Column(name = "first_name", nullable = false) val firstName: String,
  @Column(name = "middle_name", nullable = true) val middleName: String?,
  @Column(name = "last_name", nullable = false) val lastName: String,
  @Column(name = "email_address", nullable = false) val emailId: String,
  @Column(name = "day_of_birth", nullable = false)
    @JsonProperty("day_of_birth") val dayOfBirth: LocalDate
)
```

We're going to create a repository interface to interact with the database. Moreover, the EmployeeRepository interface will extend from the JpaRepository interface. This ensures that all the CRUD methods on the Employee entity are available. The @Repository annotation specifies that the class is a repository and represents the data access layer in our application.

```
@Repository
interface EmployeeRepository : JpaRepository<Employee, Long>
```

Implementing a RESTful Service Using Spring

Service

```
@Service class EmployeeService(private val employeeRepository: EmployeeRepository) {
  fun getAllEmployees(): List<Employee> = employeeRepository.findAll()
  fun getEmployeesById(employeeId: Long): Employee = employeeRepository.findById(
      employeeId)
    .orElseThrow { EmployeeNotFoundException(HttpStatus.NOT_FOUND, "No matching employee was found") }
  fun createEmployee(employee: Employee): Employee = employeeRepository.save(
    employee)
  fun updateEmployeeById(employeeId: Long, employee: Employee): Employee {
    return if (employeeRepository.existsById(employeeId)) {
      employeeRepository.save(Employee(id = employee.id, ... dayOfBirth = employee.dayOfBirth))
    } else throw EmployeeNotFoundException(HttpStatus.NOT_FOUND, "No matching employee was found")
  }
  fun deleteEmployeesById(employeeId: Long) {
    return if (employeeRepository.existsById(employeeId)) { employeeRepository.deleteById(employeeId) }
    else throw EmployeeNotFoundException(HttpStatus.NOT_FOUND, "No matching employee was found")
  }
}
```

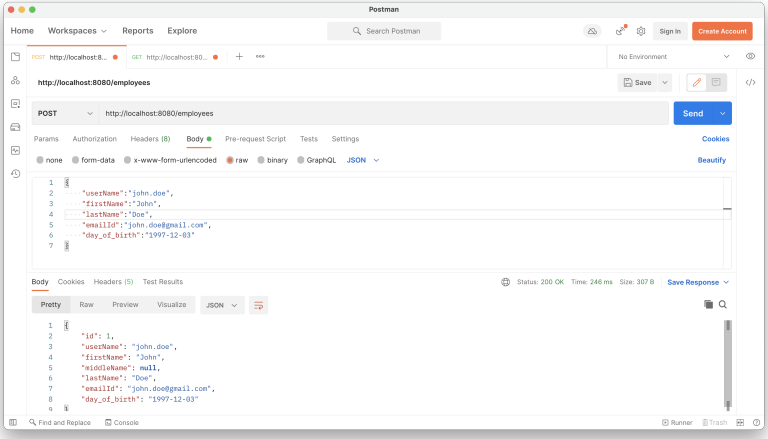
Implementing a RESTful Service Using Spring

Controller

```
@RestController class EmployeeController(private val employeeService: EmployeeService) {
  @GetMapping("/employees") fun getAllEmployees(): List<Employee> = employeeService.getAllEmployees()

  @GetMapping("/employees/{id}") fun getEmployeesById(@PathVariable("id") employeeId: Long): Employee =
    employeeService.getEmployeesById(employeeId)

  @PostMapping("/employees")
  fun createEmployee(@RequestBody payload: Employee): Employee = employeeService.createEmployee(payload)

  @PutMapping("/employees/{id}")
  fun updateEmployeeById(@PathVariable("id") employeeId: Long, @RequestBody payload: Employee):
    Employee = employeeService.updateEmployeeById(employeeId, payload)

  @DeleteMapping("/employees/{id}")
  fun deleteEmployeesById(@PathVariable("id") employeeId: Long): Unit =
            employeeService.deleteEmployeesById(employeeId)
}
```

Testing the RESTful Service Using Spring

Postman



- The application will expose data via a REST API
- Struct tags will be used for converting Object to JSON and vice-verse.
- We'll use gin to handle HTTP requests.

A user can **create**, **retrieve**, **update** and **delete** an employee using this application.
An employee has an **id**, **username**, **first name**, **last name**, **email**, and a **date of birth**.

Tutorial: https://go.dev/doc/tutorial/web-service-gin
Source Code: https://go.dev/doc/tutorial/web-service-gin#completed_code

Struct tags such as json:"artist" specify what a field's name should be when the struct's contents are serialized into JSON. Without them, the JSON would use the struct's capitalized field names – a style not as common in JSON.

```
// album represents data about a record album.
type album struct {
    ID     string  `json:"id"`
    Title  string  `json:"title"`
    Artist string  `json:"artist"`
    Price  float64 `json:"price"`
}
```

Implementing a RESTful Service Using Gin

Gin Server

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.GET("/albums/:id", getAlbumByID)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}
```

Implementing a RESTful Service Using Go

Write a GET Handler

```go
// getAlbums responds with the list of all albums as JSON.
func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, albums)
}
```

Implementing a RESTful Service Using Go

Write a POST Handler

```go
// postAlbums adds an album from JSON received in the request body.
func postAlbums(c *gin.Context) {
    var newAlbum album

    // Call BindJSON to bind the received JSON to
    // newAlbum.
    if err := c.BindJSON(&newAlbum); err != nil {
        return
    }

    // Add the new album to the slice.
    albums = append(albums, newAlbum)
    c.IndentedJSON(http.StatusCreated, newAlbum)
}
```

Implementing a RESTful Service Using Go

Write a GET Handler with parameters

```go
// getAlbumByID locates the album whose ID value matches the id
// parameter sent by the client, then returns that album as a response.
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    // Loop over the list of albums, looking for
    // an album whose ID value matches the parameter.
    for _, a := range albums {
        if a.ID == id {
            c.IndentedJSON(http.StatusOK, a)
            return
        }
    }
    c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})
}
```