**9321 SUPERNOTE**

REST:

Browser -> server: Request line, header, (additional data)

Server -> browser: status line, header, body

HTTP status code:

1: informational

2: success

3: redirection

404, 401, 403

500: internal server error

Request method: GET, POST, HEAD, OPTION, PUT, PATCH

PUT: resource to be replaced, PATCH: existing resource to be modified.

SAFE: methods that do not modify resources, such as GET, HEAD.

IDEMPOTENT: methods that can be call multiple time without different outcomes.

|  |  |  |
| --- | --- | --- |
|  | Idempotent | safe |
| POST | NO | NO |
| PUT | YES | NO |
| PATCH | NO | NO |
| DELETE | YES | NO |

Header: Content-Type, Content-Length, Cache-Control

API allows software to communicate to each other. Basically, service request, service response

Its constraints are:

• Uniform interface: client and server interact in an uniform manner.

A resource needs to have at least 1 URI

Every URI addresses at least 1 resource.

• Cacheable: Well-managed caching partially or completely eliminates some client–server interactions, improving scalability and performance.

To scale, RESTful API must be work-shy (only generate data traffic when needed, other times use cache)

This requires ’server-client’ collaboration: • Client provide guard clauses in requests so that servers can determine easily if there’s any work to be done • If-Modified-Since, Last Modified, If-None-Match/ETag

• Statelessness:

Client server interaction should be statelessness.

Server should not store information about the request from server.

Client can be a stateful app, but each request to server should contain necessary information.

Stateless means every HTTP request happens in a complete isolation.

• Layered system: rest API allows layered architecture, server A, B, C. User do not know about the end server

de-coupling allows the components in the architecture to evolve independently

• Client-server: client app and server app evolve separately and independent of each other. Client app knows only the resource URI of server app.

• Code on demand (optional):

A resource is a thing that:

• is unique (i.e., can be identified uniquely)

• has at least one representation

• has one or more attributes beyond ID

• has a potential schema, or definition

• can provide context (state) – which can change (updated)

• is reachable within the addressable universe

• collections, relationships

Leonard Richardson: can we measure to what level your service is RESTful?

Level 0: One URI (single endpoint) exposed, requests contain operation details

Level 1: Expose resource URIs - individual URIs for each resource. Requests could still contain some operation details

Level 2: HTTP Methods - use the standard HTTP methods, status codes with the resource URIs, Level 3: HATEOAS - self-documenting responses, responses include links that the client can use

Safety and Idempotency matter because it ensures uniformity

The point about REST Uniform Interface is in the ’uniformity’: that every service uses HTTP’s interface the same way.

The server guides the client’s path by serving hypermedia: links and forms inside resource representations.

Linked resources.

HATEOAS

WEEK 6

• URI design (API end points):

/items

/items/{id}

/items?price=30&weight<30

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Resource/URI | GET | POST | PUT | PATCH | DELETE |
| /CoffeeOrder | 200  return a list | 201, return created object |  |  |  |
| /CoffeeOrder/{id} |  |  | 200,204, return updated item |  | Return nothing  200, 204 |

When design your object aim for HATEOAS:

{

result: “25”

links: [

{href:,

ref:

}

]

data: {

}

}

Use HATEOAS:

• help the clients use the API (self-describing as possible)

• navigate paging (prev, next)

• help create new/related items

• allow retrieving associations (i.e., relationships)

• hint at possible actions (update, delete)

• evolve your workflow (e.g., adding extra step in a workflow = adding a new link)

REST API and security:

• HTTP basic

Issues with HTTP Basic Auth as an API authentication scheme • The password is sent over the network in base64 encoding - which can be converted back to plain text • The password is sent repeatedly, for each request - larger attack window • HTTP Basic Auth combined with SSL could work for some simple situations … But normally this scheme is not recommended and considered not secure “enough”

• Token base

• API key + signature

• OAuth