Rest API

→ Response : XML/JSON (proper structure) : instead of HTML

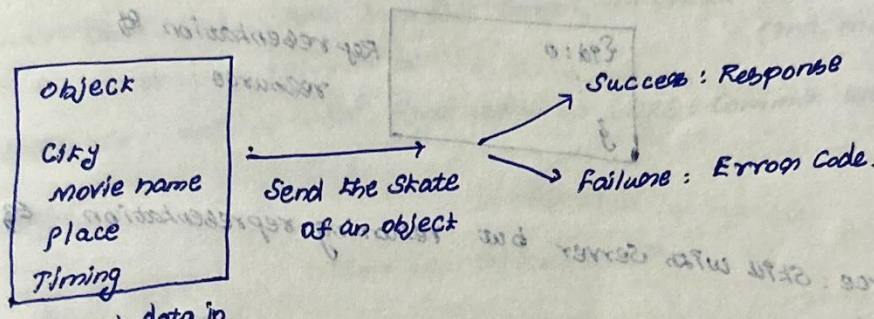
Continuous request: **Issue: lot of methods** doPost ....

↓

REST API : Avoid this scenario : Creates objects, sends response.

what ?

REpresentational State Transfer.



Just sending values (state of an object) — we can reuse object instead of creating each time & filling info

REST : architectural approach (style) : Comm purpose — Various webservices
less BW, Based on resources.

Features:
1. proper doc, error messages, simple than SOAP

Principles (6 mainly) : Dr fielding (2000)

1. Stateless :
2. Client Server
3. Uniform Interface
4. Cacheable (response header : Cacheable)
5. Layered sys
6. Code on demand

→ Stateless : Request (all info from Server)! (Preferrably body)
→ Client Server : Seperate client, Server : enhance scalability, portability
→ Uniform Interfaces → Resource identification
→ Resource manipulation using repres..
→ Self descriptive messages
→ Hypermedia as the Engine of app. state
→ Cacheable : reuse

→ Layered system: improve security, Load balance (enhance performance).

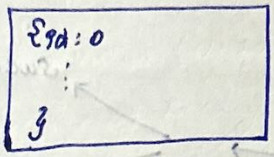
→ Code on demand: least used.

Methods

POST
GET
PUT
DELETE

Resource: Something held by server

www.apisample.com/product/0



Representation of resource

Note: Resource: Still with server but returning representation of the resource (JSON).

Resource: held by server

Representational: Something Server returns to client

REST doesn't have session at server side.

State: Resource's current state (value: dynamic)

★ REST: Transfer of state of any representation of resource!

---

6 architectural Constraint

★ Implement to get REST.

★ JSon (light weight): Preferred, HTTP protocol

SOAP: getProducts/ → verb/actions

REST: Products/1 → Noun/resources — get, post, put, DELETE — using method telling server what to do.

Demand (why?)

1. Seperate client & server (evolve seperately): Loosely coupled
   - ★ Independent of tech, platform, languages
   - ★ Flexible (data: Not tied to methods/resources)
   - ★ Scalable (doesn't store client side data)

★ No data storage – No intercomm b/w Servers
★ Not constrained to return one format (XML, JSON, YAML..)
★ Built on top of Http (use Http caching : possible)
★ Discoverable.

HATEOS: Hypermedia AS The Application of Engine State.

Server : return response + links (For client better understand how to process).

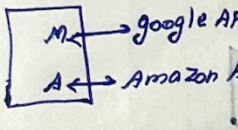
(Navigate to particular orders)

Web Services (API) : how two diff comp. interact with each other! (ind. machine, OS, ...)

web: Interconn. networks

Web Server: receive req, send response (IIS: Common web server)

distributed systems



: Comm b/w software app.

Rest/ Restful services: Build using REST archi. principle.

Web API: tech offered by microsoft : build Restful API

SOAP → Simple Object Access protocol.
→ Build using WCF (Windows Comm. Foundation)
→ only XML
→ Web Services are SOAP Services

WEB API : REST
WEB Services : SOAP Services

SOAP (vs) REST → protocol / architectural style.

Sometimes used interchang..

★ Based on app..
★ Rest (light weight) – preferred

REST

★ program: return data from DB (req (client))

Constraints : 6 archi. principles

client-server architecture:

1. Seperate client & server – evolve indep..
2. Secure – business logic / data access layer
3. Server – no knowledge of UI of client..

Stateless :

1. Shouldn't store session related data from client
2. client stores session info. — improves scalability. [cookie]

more bandwidth

Cache:

* If possible: cacheable (duration) — include in header
* Low latency. (no need for same data request)

Uniform Interface: (key) — many devices (same interface!) — uniform way

* Identification of resources (URL)
* Manipulation of resources through representations
* Self descriptive messages of each request
* HATEOS

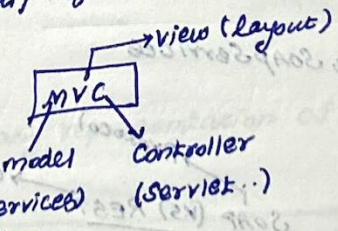
api/products/1 → JSON response

"promotes generality: end of devices"

Layered System:

multiple layers : Abstraction, Security

layer 1 doesn't know layers behind / in front of it



↓ Complexity
decoupled
Secure!

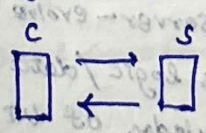
Code on demand: (optional):

* provide JS/applet code → download → execute

(Security threats : reduces visibility)

Serialization

* Converting objects to string (to any encoding format): XML & JSON



why! (.NET, JAVA — objects)

client - hardware infrastructure (0's and 1's): So need to convert it to some format!

Deserialization

* JSon to objects : from any encoding format (Json & XML)
* In Java — we work with objects! (not strings)

Richardson maturity model

* Leonard Richardson — grade API by their Restful maturity (0–3) : 4 levels

→ Level 4: Truely Rest API (other: Stepping Stones)

Level 0: plain old xml.

HTTP — only transport protocol — no use of methods / cache to full extend
- POST — for both ~~Get~~ & post

get → POST http://localhost:400/customer
post → POST http://localhost:400/customer

] No caching

Only 1 URI

Level 1:

* Seperate URI
  - http://localhost:2300/products → List
  - http://localhost:230/products/1 → Specific.
* Still "post" → to get / delete ... (retrieve / ~~create~~)

Level 2:

* Http verbs used + Http response code
* multiple URL
  - → GET http:
  - → 200 OK

Level 3:

* Support HATEOS

GET
Customer JSon + Self-doc hypermedia

→ Self-doc-hypermedia: different orders my customers might placed & link to that orders.
→ dev: Knows about orders, analyze!

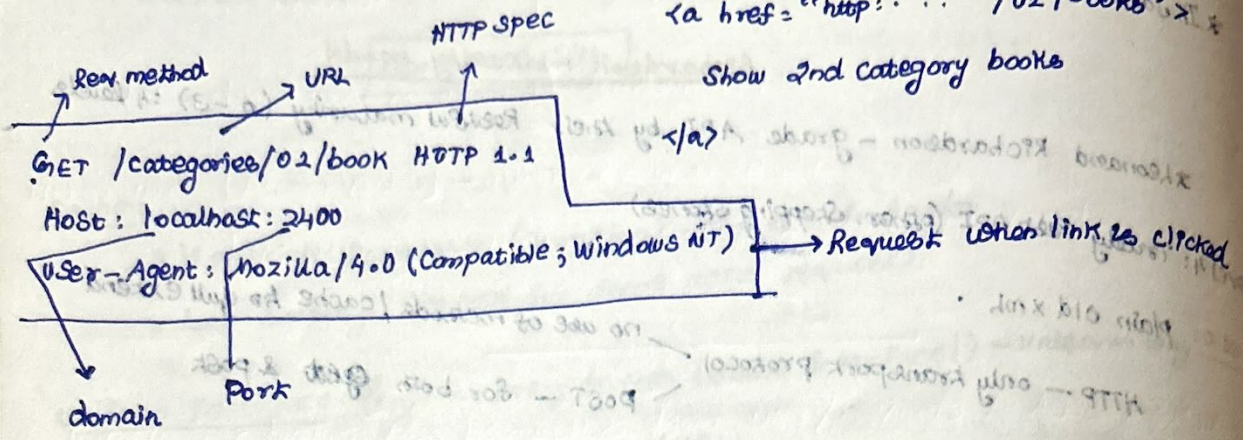
Tools

Fiddler — Inspect response / request

MySQL
VS code ...
- .NET web API

HTTP requests (GET)

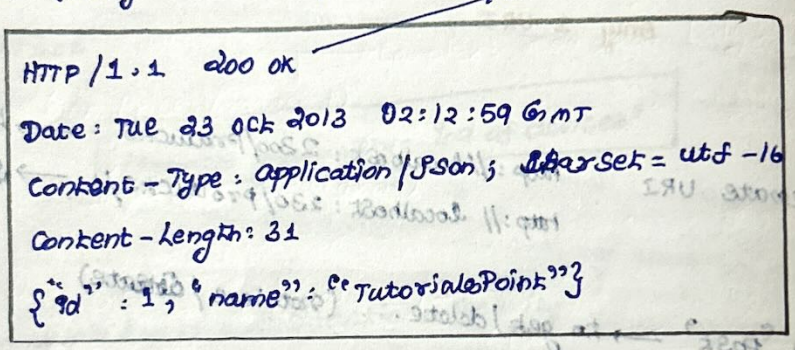
* client to server



GET POST PUT DELETE PATCH (partial update)

Http response

* Server → client (every req → response)



Design REST URLS

* http://localhost:2300/productCategory/SubCategory/Products

1. hierarchical relationships by '\'

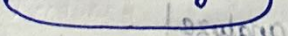
* plural nouns : http:// . . /organization/Departments/1 (plural nouns when necessary)

* Readability : http://. . . . /user-ratings

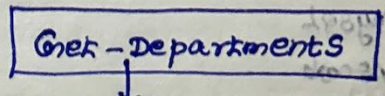
↓

Instead of

↓



* Don't use : .html, .asp [we are dealing with resources]
* For filtering : use query Parameters ?Name=HR
* Don't use CRUD within URL!



(That's why Http methods!)

Controllers & actions

→ Controller: Inherits API Controller class

URL mapping must be done! : appropriate Controller (map!)

→ Controller takes action

api / Products / 1

↳ class Products {

M1();

M2();

} — Actions-method

Controllers (classes)

URL (mapped) to controller ⟶ Actions (responds) to user request!

Routing templates

→ URI with placeholders (Similar)

api/{Controller}/{ProductCategory}/{id}

defaults: new {ProductCategory = "Books"}

→ api/Products/2 : fetch from 'books' by default. [AS ProductCategory: not given]

api/ → Controller

2 → id

default: "Books"

Controller Produ.. id

api/Products/Shoes/2

Route template can also provide Constraints

* new {id: int}

* Routed only if id is int (request ignored!)

Routing attributes

Defaults, GET request on a URL, web API route to a method name

Starts with 'Get' / "GET" (named)

Get()

GetCustomers()

Get request on a URL can invoke any action methods!

many get methods!

Alternatively: decorate action methods with any attribute

→ HttpGet

→ HttpPut

→ HttpPost

→ HttpDelete

HttpGet
HttpPut — M1(){ }
HttpPost

M1 → delete won't be allowed! (certain request alone!)

Note: use [AcceptVerbs] – allow certain methods.
[NonAction] – prevent a method from getting invoked.

Parameters in request

* parameter binding: Pass parameters from URI/requests to Action methods
* parameter (Simple type): Web API reads parameter from URI
  Complex type: Read parameter from request body using (media-type formatter)

eg:
1. Method Name (int Price, Product P)
   - Simple (URI : read from)
   - Complex : read from body!

GET api/Products/1 → parameter
Get (int ?){ }

Simple: int, float ..
Complex: our objects, ...

Force - webAPI (Simple parameter): from body of request

Entity Framework (specific)
- Method Name ([FromBody] int Price); → from body
- Method Nam ([FromURI] Product P); → from URI

Model validation

→ Model: class.

```
Class Product{
    int Id;
    String name;
    int Price;
}
```
model these parameters required

'Validation': must!

data annotation: for validating model
* [Required] – field is required

* [Range (0,10)] : I/p values in this range.

"User can't skip it"

(or)

ModelState.IsValid ⟶ check model for validation attributes

Underposting: values not specified (all)

overposting: additional values specified.

Entity frameworks: domain names

1. create models (object)
2. Controllers
3. Context classes
4. Seed DB
5. Add Migrations
6. Add DTOs

3 approaches to EF

EF

a) Code first (generates DB)
b) DB first (generates DB & model)
c) model first. (both code & DB for us.)

Entity Framework: object Related mapper (enables .NET dev to work with relational data using domain spec object) : (No need to write too much)

Models

Employee — Id (required)
— name

office — Id (req)
— location
— Employees.

Controller

handles req & response

"redirection"

handle req!

Context classes

* database — perform CRUD

Seed DB

Get Started: Add data.

Migration

Changes (from app) ⟶ database

Code first migration: Allows define model classes (employee) as per requirement!

: Complete Control.

↓

perform CRUD based on models.

DTO : Data Transfer object

→ Public properties

Use: Transfer data (Serialized/Deserialized)

Client & Server: Independent

Paging: Can't share data in one go.

Slice: share as pieces : paging

origin: (Same origin)

2 URLs : Same origin ⟶ identical scheme, host, Port

scheme: protocol (http, https.)

host: (domain)

Sample.com
Sample.net
] different

Port: app listens to this place.

(unless request is from same origin : request will be rejected by browser)

http://localhost:500
http://localhost:500
] Same origin.

solution: CORS (Cross-origin request sharing)

default: browsers prevent AJAX request from another origin

CORS: Relax this standard (Same-origin policy)

Selectively allow cross origin req, reject others

enable CORS

Config.EnableCors in WebAPIConfig

Add [EnableCors] attribute to Controllers class.

Deferred Execution

→ Queue is used (deferred execution): only executed for-each loop (iteration)

Immediate: executed at the point of declaration

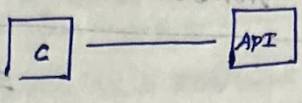
force immediate execution : use methods returning single value.

Caching: (↑ performance) : reduce no. of requests

expiration mechanism

Full response : no need (↓ bandwidth) — Expiration, validation — 2 models.

* Cache sits b/w client app & api



Private/ Client cache — 1. within browser

Gateway/ Server cache — 2. Server side

Proxy cache. — 3. or b/w client & server

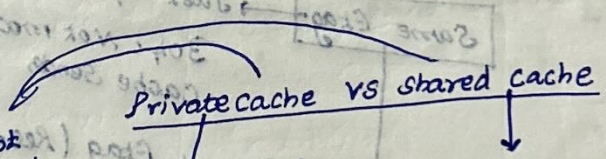
Expiration model:

* Expires header: Expires, Mon, 23 July, 2017.
* Cache-Control Header: public, max-age = 2500 → how many seconds

→ This header preferred

Expires header has clock sync issue, (Time Zone !)

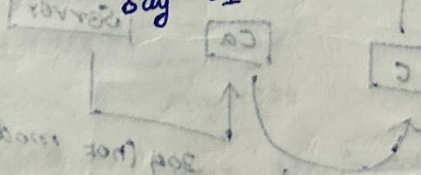
Private cache → dont hit server → fetch cache response



Private cache vs shared cache

Request no. doesn't reduces. — Client Side (browser) (BW ↓)

Say C1 uses cache, others don't : Server need to retrieve more



Cache at server! / proxy.

hit proxy : reduces load to API

Validation model:

↳ * validate freshness of model (cache) — check with server (expired?)

* validate against validators

Types — Strong validators (byte to byte comparison) → Preferred.

Weak (Semantic)

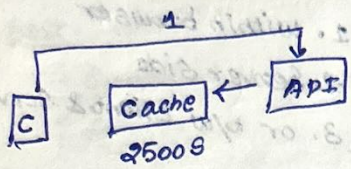
* Each request, server responds with Etag : (cached)

Etag: "3424"

Say: data expired: Cache: Send Etag to resource: update

match (already found) & expired: update

Not found: Not modified (304)



within 2500s — cache used

2500 over

* Client prepares fresh one to API
* Client download entire data

Cache again!

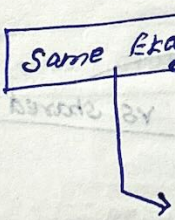
---

even though: data not changed: download again

expiration model

Solution: Validation model

checks Etag (if something changed: Etag updated)!



→ Just update time

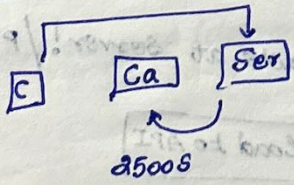
304: Not modified

Cache sends 200 OK.

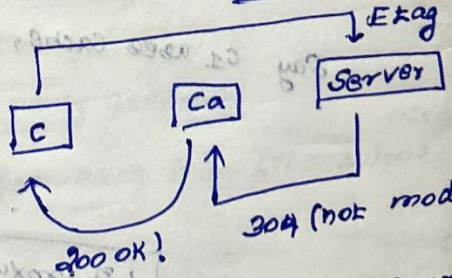
different Etag (Resource changed)

download again, cache again

Validation model

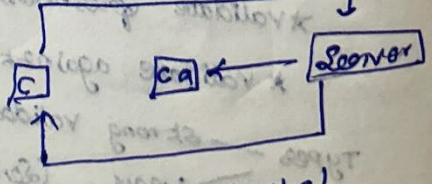


After 2500s



Same resource

different Etag



helps BW, load!

Cache-control directives

Server adds Cache Control directives

freshness: max-age, s-max-age → Shared cache (mostly: overwrites max-age too)

other: no-store, no-cache → each time check with server

Cache-type: public, private. → Banking: never be cached!

'Client can override while requesting resource'

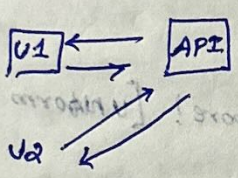
no-cache: Cache allowed - but each request (Contact with Server) — Is updated?

no-store: Cache disabled

private: local users, Single user

Concurrency

★ Concurrency Strategy (using Etag)!



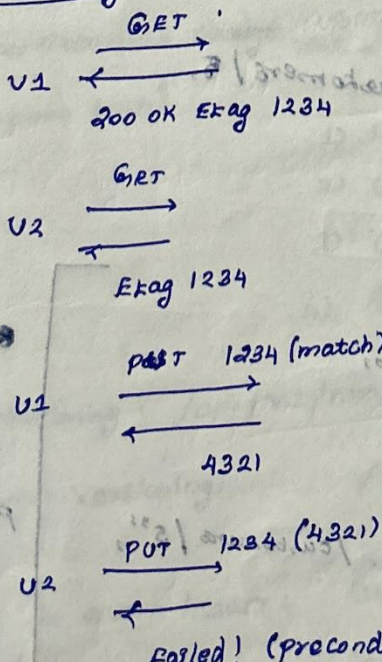
→ Requests Same resource

→ U1 changes resource

→ U2 changes same resource [changes of U1 lost]

Since U2 - updates!

Rest solves this Etag:



JSON Serialization

★ application/json : media type

★ MediaTypeFormatter (deserialization) JSON ⟶ String/object

- CSV (our own MediaTypeFormatter)

JSON

★ JsonMediaTypeFormatter (uses Json library)

★ Json.Library (public prop & fields)

[JsonIgnore] ⟶ to avoid that property while Json Serialization.

Control the process : which one to serialize

read-only : Serialized by default

HATEOS

★ api (self doc/discoverable) : dev can know more! [Uniform Interface]

HATEOS

★ dev: need not know everything

★ HATEOS allow api : evolve independently by consuming app.

★ ↑ discoverability.

Links for API : hypermedia

http://localhost:2300/customers/5.

```
{"id": 5, "name"...
 "Links": [
    {"rel": "self"
     "href": ———— /5"
     "action": PUT },
    {"rel": "self"
     "href": "http:loca.../customers/5",
     "action": "DELETE"
    }
 ]
}
```

explaining endpoints for PUT, DELETE :

Approaches to return HATEOS

1. HAL
2. Collection + JSON

] based on Custom Content type

Content type: for formatting

profile media type: Structure of data.

*

HAL: Hypermedia App. Language (brief std)

app.. / hal+JSON

hal + XML

* used with profile media type

- app.. /hal+JSON ; profile = http://... :2300/products

Collection + JSON: std read/write collection :

* more meta data than HAL
* uses profile media type
* Include UI info.

app.. /vnd.Collection+JSON : profile = http..../customers

API versioning

* Requirements change : Users rely on API not changing.
* Support both!

1) URI path versioning
2) URL Parameter versioning
3) Content negotiation
4) Request header

1) URI path versioning : /api/v1/Customers/1
2) URI parameter versioning: /api/Categories?v=1.1
3) Content type in accept header: http://.../Categories/1

Accept: application/app.v1.Categories

4) Custom header : GET .... /1

X-App-Version: 1.3

5. use date instead of version – number

X-App-Version: 2017-08-12.

Server:

★ Service — data, email, web (Retrieve website) } Separate (large org)

★ desktop: don't handle payload – concurrent connections.

★ desktop CPU
97 Core
(Single processor)

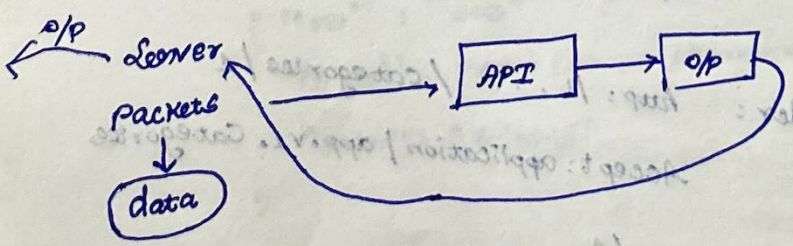
Server CPU
XEON → multiprocessing environments (other Xeon..)
→ ECC ram (error code correction) – memory errors
→ Large RAM, Core count, Cache
→ hot swappable hard drives (RAID)
★ Replace harddrive (no shutdown)
★ Rebuilt data (as in multiple discs)
→ 24/7 power
→ OS: Linux/mac/windows
Concurrent connections

★ webserver: host websites
★ email server: facilitates send & receive
★
webserver: Request data will be given (no processing)
app. server: process & return o/p

API:
http request → give data to app logic → o/p → Response.

Contract/language: both can understand:
★ Data exp? form? o/p: ?

★ API → Contract
★ web server → mechanism to transfer.



API : Takes request, returns response!

eg: Menu ⟶ API ⟶ Kitchen

doorway

Threadlock

Thread 1

```
Synchronized (resource 1){
    ____
    sleep (100)
    Sync (resource 2){
    ____
    }
}
```

Thread 2

```
Sync (res 2) {
    ____
    sleep(100)
    Sync (res 1){
    }
}
```

lock.

why? * res 1 locked by Thread 1

↓

Thread 1 sleep

↓

Enter thread 2 (res 2 locked by Thread 2)

↓

sleep (return to Thread 1)

↓

resource 2 - lock acquired by thread 2

(Thread 1 - dead end)

resource 1 - by thread 1

(Thread 2 - dead end).

Solve:

Thread 1

```
Sync (res 1) {
    ____
    sleep
    sync (res 2) {
    }
}
```

Thread 2

```
Sync (res 1) {
    ____
    sleep
    Sync (res 2) {
    }
}
```

Now: Thread 1 acquires res 1

↓

Sleep

↓

Thread 2 – Can't acquire res 1

↓

Sleep loop is over: Thread 1 acquires res 2

↓

Job done – unlock resources

↓

Thread 2 – acquires res 1

↓

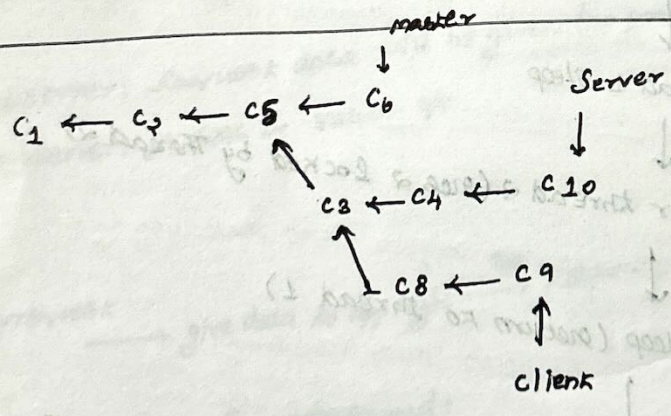
Sleep

↓

Th 2 – acquires res 2

↓

Job done!



merge client but not server:

* Take changes in client (not on server)