

REST API Design Patterns

Michael DeHaan
@laserllama

Remote Comm. History

- Custom TCP/IP or UDP stuff
- CORBA - 1991
- Java RMI - 1997
- XMLRPC - 1998
- SOAP - 1998 - “Enterprisey”
- REST - 2000
- Thrift - 2007
- Protocol Buffers - 2008

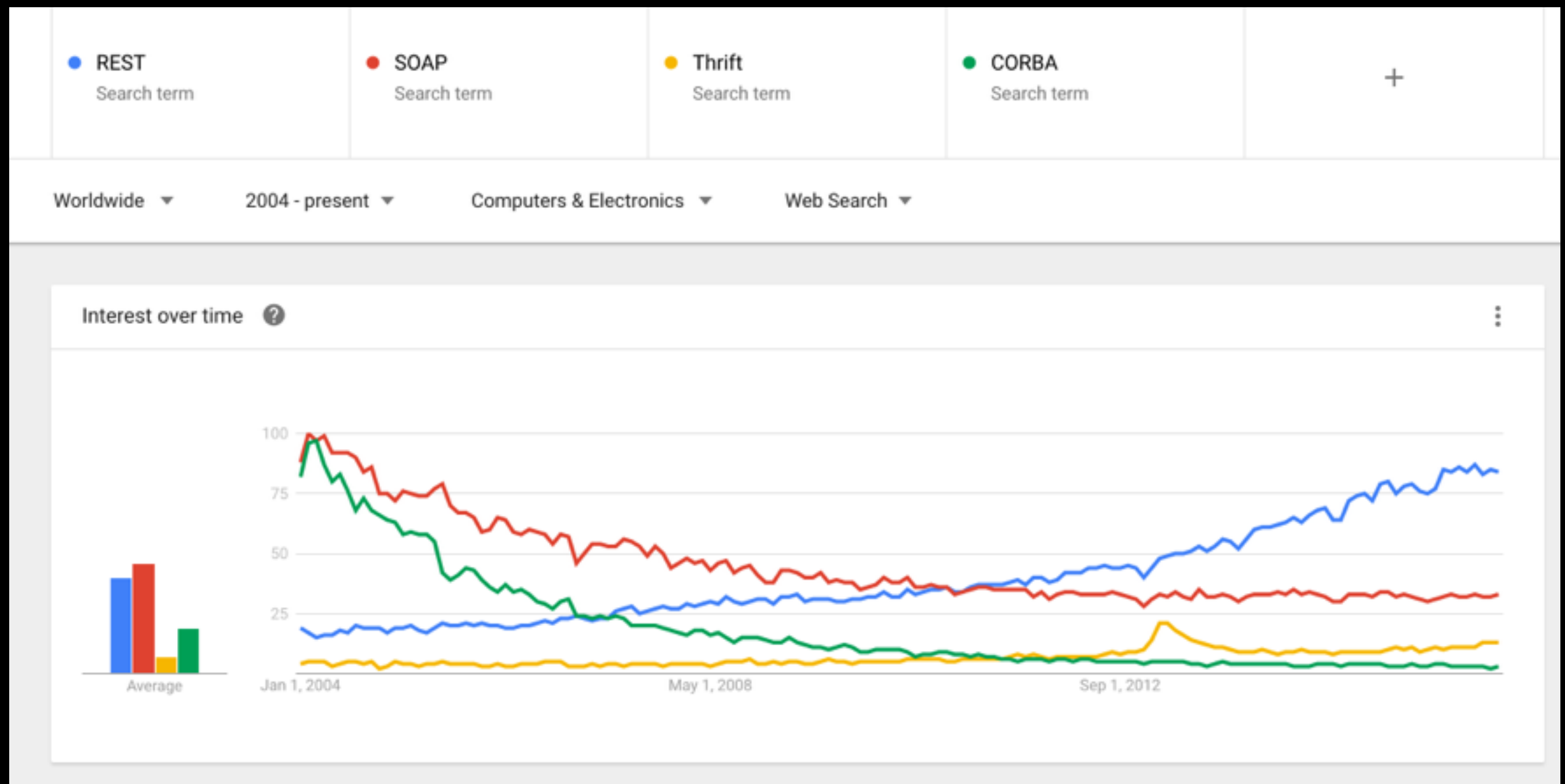
Binary Interchange

- Custom
- CORBA (Object Management Group)
- RMI (Sun)
- Thrift (Facebook)
- Protocol Buffers (Google)

Text Interchange

- XMLRPC (Dave Winer - Userland/Microsoft) - easy, discoverable, but limited data types (no None, etc)
- SOAP (Winer, Box, Atkinson, Al-Ghosein - Microsoft (W3C standard)) - very complex ecosystem
- REST - (Roy Fielding - UC Irvine) - mostly happy medium, but less standardization

REST Popularity



note: some data noise in graph (Thrift = 2007)

Binary Formats

- “Fast”
- Limited or no language support
- Difficult external API integration
- Poor discoverability

Textual Formats

- Technically slower
- Unlimited language integration
- Complexity of implementation varies

Text Formats

- XML
- JSON
- YAML

XML

- SAX vs DOM Parsers, XPath
- Schemas - DTD, XSD
- Unclear data type encodings (hashes vs lists)
- Elements vs Attributes

JSON

- Numbers, Strings, Hashes, Lists are most all of what you need
- Faster Parsing/Interpretation Than XML
- Simplifies Coding to Extreme Levels compared with XML Toolchains

Aside: YAML

- “YAML Ain’t Markup Language”
- Not really used for remote communication at all, but more “better JSON”, but exceptionally powerful in config files and you may like it
- Also very wide language support
- <http://yaml.org>

Ok, so REST APIs

- Warning: REST is not really a standard
- Various dogma and “best practices”, and those often differ
- “Best practices” = “this is what I do and you should do them that way because I do them that way”
- Easy to argue on the right way when there is no right way, so much of this is just one possible way

Why REST

- HTTP Port 80 and 443
- Most APIs feel *mostly* the same
- Language support
- Discoverability - we'll get into that
- Lots of user love (and less hand holding) when you get it right

Why Not REST

- In internal software, RPC between machines will be slower
- As such REST is better suited for external APIs, but you could still use it and many things do
- Also: less type checking than possible with binary stub systems
- A minefield of arguments about how to do it right

Collections Vs Items

- Collections end in slash - `/api/v0/bands/`
- Items do not - `/api/v0/bands/42`
- Item IDs should **usually** be numeric and never reusable. Database serial IDs are easiest.

REST MODELLING

- Your API is defined around what NOUNS are surfaced. You get state and you tell the system what state things should be in.
- The list of verbs is very limited
- This is the exact opposite of an RPC interface, and often, the way things exist in the real world. It takes some adjustment.

Common HTTP Verbs

- GET - grab lists or items
- PUT - update something
- POST - add something new
- DELETE - remove something*
- PATCH? - seldom used - not recommended IMHO
- OPTIONS? - seldom used, nice for documentation

GET

- GET /api/v0/bands/ — get “all” bands
- GET /api/v0/users/42 — get a specific band
- Technically REST does not imply JSON, but usually does/should.
- Content-type: application/json

PUT

GET /api/v0/bands/3000 ->

```
{  
  "id" : 3000,  
  "name" : "VanHalen",  
  ...  
}
```

PUT /api/v0/bands/3000 <-

```
{  
  "name" : "Van Halen"  
}
```

important - for security, id is read only

POST

POST /api/v0/bands/

```
{  
  "name" : "Spinal Tap",  
  "genre" : "metal",  
  "description" : "none more black"  
}
```

note: database id must be ignored if sent, etc

DELETE

- DELETE /api/v0/bands/12345
- Good backends won't actually delete in many cases, but will mark something is_deleted=True and all GETs will be coded to hide these rows

HTTP Error Codes

- Ok - 200
- Created - 201
- No Content - 204 (used in DELETE response)
- 30* - redirects, seldom used
- 40* - client errors.
- 401 - unauthorized (no login provided)
- 403 - forbidden (your login is not good enough)
- 404 - not found
- 409 - conflict (object is not internally consistent?)
- 418 - I'm a teapot
- 50* - server errors. 500 - Internal Server Error is most famous. DO NOT SHOW THE STACK TRACE IN PRODUCTION! LOG IT. Also use log aggregation services.

Idioms/Patterns

API Versioning

curl <https://server.example.com/api>

```
{  
  "/api/v0" : "v0"  
}
```

API Versioning (2)

- The '/api/v0' is usually about appeasing customer fears and is usually not utilized in practice for the same reason.
- Usually web software will not (due to pragmatism) support multiple versions of an API, and sometimes will not make new API versions between minor releases
- Adding URLs is always ok, changing URLs is always ok (we'll get to why), adding fields is usually ok
- Major incompatible overhauls (equivalent to rewrites) should replace /api/v0 with /api/v1, but it unlikely the application will support multiple API versions at once.

Filtering And Search By Query String

- `/api/v0/bands/?name="Van Halen"`
- `/api/v0/bands?
name__like="Iron"&genre="metal"`
- `/api/v0/bands?
genre="shoegaze"&order_by="found_date"`

Authentication

- Either HTTP username password or...
- User equivalency for login
 - X-API-Key: “your key”
- Send every time, may support a /login URL that returns a session (equivalent to X-API-Key) that can be used in headers. Not required, but useful.

Pagination

- GET /api/v0/bands
- GET /api/v0/bands?page=2&page_size=50
- (some APIs use headers)

Query Strings Vs Headers

- Query strings show up in server logs
- This is usually good
- Browser usage is usually much easier

Pagination

GET /api/v0/bands/

```
{
  "page" : 2,
  "count" : 10000,
  "total_pages" : 100,
  "page_size" : 100,
  "next_page" : "http://server.example.com/  
api/v0/users/?page=2"
  "prev_page" : None,
  "items" : [
    { ... },
    { ... }
  ]
}
```

Discoverability

- Your UI should hard code no other URL than “/api/v0/”
- Every object provides links to related objects (continued...)

Discoverability

```
{  
  "id" : 5000,  
  "href" : "/api/v0/bands/5000".  
  "name" : "Parts & Labor".  
  "members" : "/api/v0/bands/5000/members/"  
}
```

Filtering

- Collections: /api/v0/users/ may only show YOU unless you are an admin
- Collections and items:
 - Certain fields could be write-once
 - Certain fields should be read-only (id)
 - Certain fields should be write-only (passwords)
- A good REST framework will allow defining serializers that help map your model into view space more easily.

Async Actions

- No REST calls should return quickly, and should block
- Hence it makes sense to develop a “jobs” construct for long running operations
- This is also not-standardized or consistent

Idioms: Async Actions

- GET “/api/v0/job_definitions”
- POST “/api/v0/jobs” <- job definition JSON ->
job
- GET “/api/v0/jobs/42” for progress

Idioms: Sub Collections

- Examples:
 - Listing favorite bands
 - Adding a favorite band
 - Unliking a band?

Listing Favorite Bands

- GET /api/v0/users/1/favorite_bands/

Adding A Relationship

- GET /api/v0/bands/?name="Van Halen" -> JSON
- POST /api/v0/users/42/favorite_bands/ <- JSON

Disassociation From A Subcollection

- Model the relationships?
 - `/api/v0/favorite_band_mappings/`
- POST with alternative metadata?
 - `POST /api/v0/users/42/favorite_bands <- { "id": 5150, disassociate: True }`
- DELETE with BODY?
 - `DELETE /api/v0/users/42/favorite_bands <- { "id": 5150 }`

Things That Make Less Sense On A Subcollection

- PUT - just PUT on one of the related objects

Rate Limiting

- You may encounter this if an API is public
- HTTP 429 - too many requests
- X-Rate-Limit-Limit
- X-Rate-Limit-Remaining
- X-Rate-Limit-Reset
- Do not write naive `time.sleep()` based logic if this is available
- In house apps usually do not implement rate limiting.

You Should Probably Still Release A Client Lib

- Suppose you have a REST API
- It's not documented much
- Consider OPTIONS but...
- Having a sample program that uses it helps users tons
- Example: python or Ruby library (lib-myservice)

Tool Recommendations

- curl
- Python
 - <http://www.django-rest-framework.org/>
 - <http://docs.python-requests.org/en/master/>
- Charles Proxy (Mac), Chrome Dev Tools, Firebug
 - <https://www.charlesproxy.com/>

Summary

- REST is great for external public APIs and moderate-performance internal APIs between systems
- Good REST Is About Consistency
- Verbs and Error Codes
- Discoverability Eliminates User Questions and Allows URL changes
- Query Strings for Search and Ordering
- Pagination Is Required
- Filtering and Read-Only Fields for Security
- Debugging - Browser, Proxy
- If using Python, Django REST Framework is Gold. Check out the API Browser.