REST Web Services-2

Chantri Polprasert

Readings

- Mark Masse ‘REST API Design Rulebook’
- Richardson, L., and Ruby, S., RESTful Web Services, O’Reilly, 2007.
- Daigneau, R., Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2011.
- Fielding, R.T., Taylor, R.N., Erenkrantz, J.R., Gorlick, M.M., Whitehead, J., Khare, R., and Oreizy, P., Reflections on the REST architectural style and “principled design of the modern web architecture” (impact paper award). In Foundations of Software Engineering (FSE), ACM, 2017.

Outline

- Introduction
- Resource-Oriented Architecture
- Resource-Oriented Analysis and Design
- REST and ROA best practices
- SOAP vs REST

Resource-Oriented Analysis and Design: Introduction: REST vs. RPC

- Designing REST services is different from designing RPC services.
- It is more like Object-oriented Analysis and Design (OOAD) where we break the world into its nouns (data/objects).
    - Each noun in the domain typically becomes a class in OOAD.
    - Each class has behaviors for interaction.
- RPC designs break the world into verbs (function/action e.g. what can you do?).
- Resource-oriented designs look like extreme object-oriented designs, because every resource has the same 6 methods.

Resource-Oriented Analysis and Design: Introduction: REST vs. RPC example

- An RPC design might contain a subscribe method.
- A resource-based design would have a subscription resource associating users with channels.
- REST tends to use URL path parameters to identify specific resources
- RPC tends to use query parameters for function inputs

Operation	RPC (operation)	REST (resource)
Signup	POST /signup	POST /persons
Resign	POST /resign	DELETE /persons/1234
Read a person	GET /readPerson?personid=1234	GET /persons/1234
Read a person's items list	GET /readUsersItemsList?userid=1234	GET /persons/1234/items
Add an item to a person's list	POST /addItemToUsersItemsList	POST /persons/1234/items
Update an item	POST /modifyItem	PUT /items/456
Delete an item	POST /removeItem?itemId=456	DELETE /items/456

https://apihandyman.io/do-you-really-know-why-you-prefer-rest-over-rpc/#examples

Resource-Oriented Analysis and Design:

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

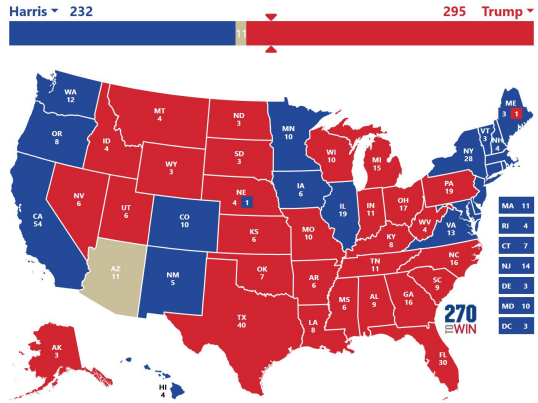
Resource-Oriented Analysis and Design: Design Procedure

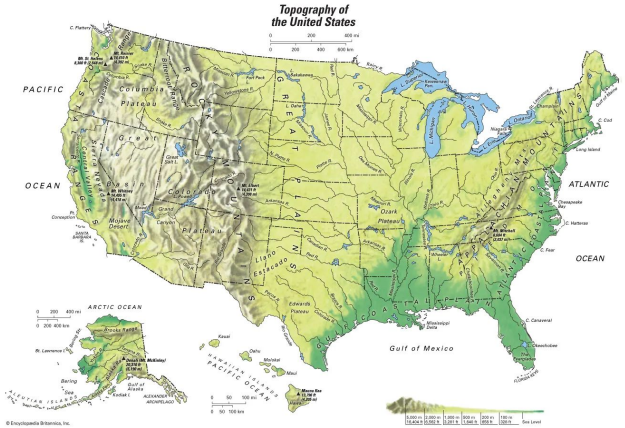
A general procedure in Richardson and Ruby's map service example

1. Figure out what the data set is.
2. Split the data set into resources.
3. Then for each resource, name it with a URI.
4. Expose a subset of the HTTP methods on the URI.
5. Design the representations of data coming from the client (e.g. JavaScript).
6. Design the representations of data sent to the client.
7. Integrate resources with hyperlinks and forms.
8. Consider the sequence of events.
9. Consider error conditions.

Resource-Oriented Analysis and Design Map service example: background

- Background information for our fictitious map service:
  - We want to present maps of any planet (Earch, Moon, Venus…) in any projection.
  - The maps could be political maps, road maps, topographic maps, etc.
  - The maps must be addressable by latitude and longitude.





Resource-Oriented Analysis and Design Map service example: data set

How to think of the data?

What information do you want to make available, and what questions do you expect clients to ask?

Resource-Oriented Analysis and Design Map service example: data set

How to think of the data?

What information do you want to make available, and what questions do you expect clients to ask?

- Maps (images)
- Points on maps (address)
- Planets
- Points on planets (represented by latitude and longitude)
- Place names with corresponding points (“Bangkok”, “Phangan Island”) (real GIS systems use sets of points to form lines and polygons) (AIT)
- Place types (city, mountain, etc.)

Resource-Oriented Analysis and Design Map service example: data set

The actions we are interested in would be to: (list the noun)

- Find a place near me and show them on a map using a road layer. E.g. Where is AIT?
- What to eat at AIT?
- Find nearby places. (Gas stations, hotels, cafe)

You might have to iterate several times to get the design right

Resource-Oriented Analysis and Design Map service example: data set

Category	Examples	Typical client questions
Celestial bodies	Earth, Mars, Venus	“What planets are available?”
Places / landmarks	Bangkok, Phuket	“What is this location’s name, type, and coordinates?”
Layers (datasets)	Terrain, temperature, atmosphere	“What datasets exist for this planet?”
Features (geometries)	A crater polygon, a volcano point	“What features lie inside this bounding box?”
Maps (compositions)	A user’s saved map view	“Show me my saved ‘Mars terrain’ map.”
Algorithms / computations	Geocode, static render	“Where is AIT?” or “Render an image for this area.”
System metadata	Version, health, time	“Is the service running? Which API version?”

Resource-Oriented Analysis and Design Map service example: split into resources

A resource is anything interesting enough to be the target of a hyperlink.

Three types:

- Predefined, “one-off” resources for important fixed aspects of the system.
  - **Examples**: site home page, directory listing, S3 root homepage(https://s3.amazonaws.com), capabilities
  - Cannot be deleted or directly modified
- Resources for every object exposed.
  - **Examples**: every S3 bucket and objects exposed as a resource (e.g. places, layers, maps)
- Resources representing results of algorithms applied to the data set, especially query results.
  - Dynamic
  - Produce different results at different time (e.g. routes, geocode)

Three types of resources in RESTFUL web services

Type	Description	Typical use	In your map service
Predefined one-off resources	Unique, singleton data that exists once per service.	Configuration, metadata, status, version, root discovery.	`/v1/status`, `/v1/capabilities`, maybe `/v1/earth` if treated as a singleton.
A resource for every object exposed	A collection where each object (planet, layer, feature, map) has its own identity and URI.	Main entities — retrievable, listable, addressable.	`/v1/layers`, `/v1/features`, `/v1/maps`, `/v1/places`.
Resources for results of algorithms	Computed or query-driven results, not persistent; output depends on input parameters.	Search, rendering, routing, geocoding.	`/v1/geocode`, `/v1/static-maps`, `/v1/routes`.

Resource-Oriented Analysis and Design Map service example: split into resources

Through some brainstorming on the map application, we might come up with:

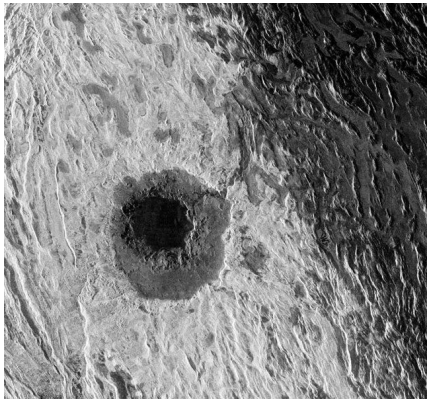
- The list of planets.
- A place on a planet identified by its name.
- A point on a planet identified by latitude and longitude.
- A list of places matching search criteria (possible attributes: type, lat, long and associated data)
- A map of a planet, centered on one point.

Note that the data and algorithms operating on those data are exposed through resources. Usually they form a hierarchical structure.

Resource-Oriented Analysis and Design Map service example: name the resources

- Remember that in REST, the URI contains all the scoping information.
- Some simple tips (RFC 3986):
  - Use the slash (/) for hierarchy (e.g. people in department)
  - Use a comma (,) or semicolon (;) to separate items in a list. Commas can be used for ordered lists; semicolons can be used for sets.
  - Use query variables (e.g., search?q=blackpink&start=20) for inputs to algorithm resources (for filtering or searching)

Resource-Oriented Analysis and Design Map service example: name the resources



Encode hierarchy into path variables

- Planets are obviously at the top of a hierarchy for us:
  - http://maps.example.com/Venus
  - http://maps.example.com/Earth
  - http://maps.example.com/Mars
- Places identified by names should be the next level of hierarchy:
  - http://maps.example.com/Venus/Cleopatra (a crater)
  - http://maps.example.com/Earth/Paris,%20France
  - http://maps.example.com/Earth/Asia/Thailand/Pathumthani/AIT

Paris in France

Resource-Oriented Analysis and Design
Map service example: name the resources

For points (latitude and longitude) hierarchy is not appropriate. Use commas or semicolons:

- http://maps.example.com/Earth/24.9195,17.821
- http://maps.example.com/Venus/3,-80

Why comma not /?

URIs for maps can be simple:

- http://maps.example.com/radar/Venus
- http://maps.example.com/radar/Venus/65.9,7.00

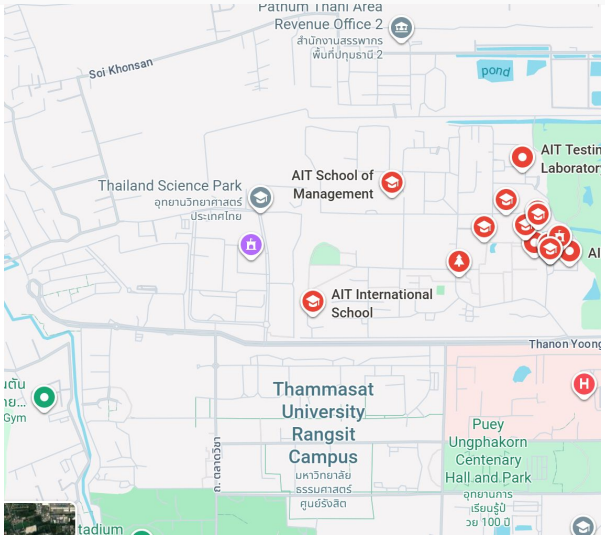
Resource-Oriented Analysis and Design
Map service example: Scaling

In a read-only map or dataset service, you might need to deliver different resolutions of the same resource:

- A coarse representation (overview or thumbnail).
- A fine representation (high detail).
- A tile in a multi-scale tiling system.

Scaling numbers or levels of detail — they represent different representations of the same conceptual resource, partitioned into smaller, manageable sub-resources.

```
/earth/images/1      ← low-res global view
/earth/images/2      ← higher resolution
/earth/images/10     ← detailed tile
```



https://www.google.com/maps/search/ait/@14.0781631,100.6016702,15.14z?entry=ttu&g_ep=EgoyMDI1MTEwNS4wIKXMDSoASAFQAw%3D%3D

Resource-Oriented Analysis and Design
Map service example: Why include scaling in the URI (instead of a query param)

Design choice	Pros	Cons
Scaling as part of the path (URI segment)	- Emphasizes that each scale level is a **distinct resource**. - Supports caching (CDNs, proxies) — each scale has its own URL and cache key. - Makes relationships between scales discoverable (`/images/1` → `/images/2`). - Statelessness	Slightly longer URI; must document hierarchy.
Scaling as query parameter (`?scale=2`)	- Easier to add ad-hoc filtering. - Looks simpler for one-off APIs.	-Loses resource identity -Caches may treat it as the same object with different query args -Harder to link between scales.

By putting the scaling number in the path, you make it clear that:

“The level-10 image is not a different view of the same resource — it’s a separate, finer-grained resource.”

Resource-Oriented Analysis and Design Map service example: name the resources

OK, the final list of URIs for the map service:

- The list of planets, at ./index
- A planet or a place on a planet: /{planet}[{scoping-information}/][{place-name}]
    - {scoping-information} could be a hierarchy of place name (Thailand/Bangkok) or lat/long
- A map of a planet or a point on a map: /{map-type}{scale}/{planet}[{scoping-information}/].

Resource-Oriented Analysis and Design Map service example: representation of the top-level resource

We have decided the resources to be exposed together with their URI. Now, time to decide what data to send according to the request from the clients and its format.

- The representation conveys the state of a resource.
- It also needs to link to other resources (connectedness).
- Preferably, it will also explain to the client what POST and PUT requests to that resource should be formatted.
- Map service example: the top-level “home page” will return a list of planets for which we have maps.
- Some possible representations (sent to client):
  - Plain text: simple to generate but difficult to parse.
  - JSON: JavaScript Object Notation for serialized data structures:

    [{url:"http://maps.example.com/Earth, description:"Earth"},

    {url:"http://maps.example.com/Venus, description:"Venus"}, ..]

  - XML

Resource-Oriented Analysis and Design Map service example: representation of the top-level resource

Represent the list of planets:

- HTML:

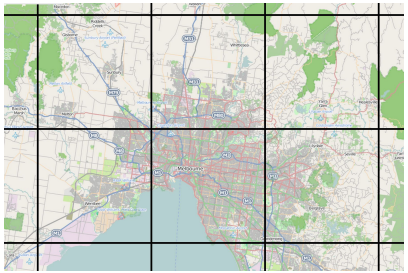
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Planets</title>
    <link rel="self" href="/v1/planets">
    <link rel="alternate" type="application/json" href="/v1/planets?format=json">
  </head>
  <body>
    <h1>Planets</h1>

    <ul class="planets">
      <li><a rel="item planet" href="/v1/planets/earth">Earth</a></li>
      <li><a rel="item planet" href="/v1/planets/mars">Mars</a></li>
      <li><a rel="item planet" href="/v1/planets/venus">Venus</a></li>
    </ul>
```

The class attribute on the ul list tag is used to provide semantic information to the client program

Resource-Oriented Analysis and Design Map service example: representation of a map

- To respond to a query to get a location of some places, it's not necessary to send a whole map image (waste of bandwidth, users only focus on that location
- Send only the part of the map together with the associated links for further information.

```
<!-- Pan links (HATEOAS) -->
<link rel="north" href="/v1/earth/map?center=19.08,100.61&zoom=6">
<link rel="south" href="/v1/earth/map?center=9.08,100.61&zoom=6">
<link rel="east"  href="/v1/earth/map?center=14.08,105.61&zoom=6">
<link rel="west"  href="/v1/earth/map?center=14.08,95.61&zoom=6">

<!-- Zoom links -->
<link rel="prev" href="/v1/earth/map?center=14.08,100.61&zoom=5">
<link rel="next" href="/v1/earth/map?center=14.08,100.61&zoom=7">

<!-- Alternate (image representation of same view) -->
<link rel="alternate" type="image/png"
      href="/v1/static-map?center=14.08,100.61&zoom=6&layers=terrain,places&size=800x600">
</head>
<body>
<h1>Earth — center 14.08, 100.61 (zoom 6)</h1>

<figure>
  <img
    src="/v1/static-map?center=14.08,100.61&zoom=6&layers=terrain,places&size=800x600"
    alt="Read-only map centered at 14.08,100.61" width="800" height="600">
  <figcaption>Terrain + places, zoom 6</figcaption>
</figure>
```

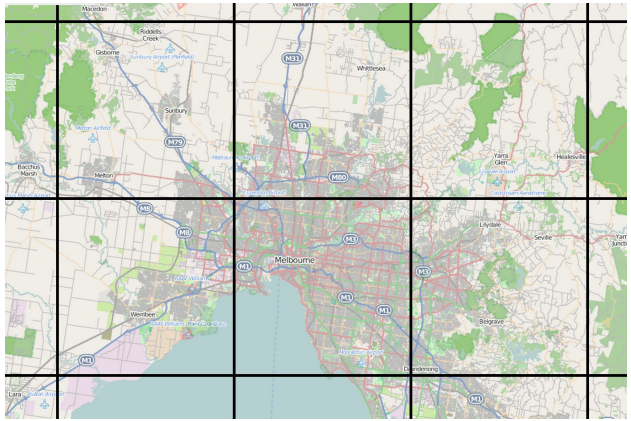


https://en.wikipedia.org/wiki/Tiled_web_map#/media/File:Tiled_web_map_Stevage.png

- The map itself is the resource.
- The center coordinates and zoom define its state.
- The links (north, south, east, west) let the client navigate to adjacent views — exactly like a user panning a map.
- The server doesn't tell the client how to pan — it gives links that the client can follow.

Resource-Oriented Analysis and Design Map service example: representation of a map

- Representations convey the state of the resource, but they don't have to expose the entire state all at once.
- In the road map example, only part of the resource's state is supplied.
- But through connectedness the client could retrieve the rest of the resource state.
- **Hypermedia as the Engine of Application State (HATEOAS)**: a constraint of the REST architectural style where clients interact with a network application solely by hypermedia links provided dynamically in server responses.
    - client learns what to do next directly from the links in the responses.
    - Make the API more self-describing and discoverable.



https://en.wikipedia.org/wiki/Tiled_web_map#/media/File:Tiled_web_map_Stevage.png

Resource-Oriented Analysis and Design Map service example: representation of a point (place resource)

- A point (like Bangkok or AIT) is another resource.
- Its representation includes coordinates and links back to maps or related features.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Asian Institute of Technology (AIT)</title>
  <link rel="self" href="/places/ait" />
  <link rel="up" href="/places" />
  <link rel="related map" href="/earth?lat=14.0792&lon=100.6049&zoom=14" />
</head>
<body>
  <h1>Asian Institute of Technology (AIT)</h1>
  <p>Coordinates: (14.0792, 100.6049)</p>
  <p><a rel="map" href="/earth?lat=14.0792&lon=100.6049&zoom=14">View on map</a></p>
</body>
```

```
  <!-- Hypermedia navigation -->
  <nav>
    <a rel="map" href="/v1/planets/mars/map?center=18.65,226.2&zoom=8">Zoom out</a> ·
    <a rel="map" href="/v1/planets/mars/map?center=18.65,226.2&zoom=10">Zoom in</a> ·
    <a rel="collection" href="/v1/planets/mars/places">All places on Mars</a> ·
    <a rel="up" href="/v1/planets/mars">Planet page</a>
  </nav>
```

The key idea:

- /places/ait is a point resource (like a feature in GeoJSON).
- It links to the map resource representing its context.
- The map representation, in turn, links to the points visible in that view.

Resource-Oriented Analysis and Design Map service example: representation of a planet

```
<body>
  <h1>Planet Earth</h1>

  <p>
    Earth is the third planet from the Sun and the only known world to support life.
    It has vast oceans, continents, and a protective atmosphere.
    This resource provides different map views that represent the planet's surface.
  </p>

  <h2>Available map types</h2>
  <ul>
    <li><a rel="alternate" href="/earth.png?style=road&zoom=4">Road Map</a> – shows human geograph
    <li><a rel="alternate" href="/earth.png?style=satellite&zoom=4">Satellite Map</a> – shows Eart
  </ul>

  <figure>
    <img src="/earth.png?style=road&zoom=4"
         alt="Earth road map view" width="800" height="400" />
    <figcaption>Default view: Road map (zoom 4)</figcaption>
  </figure>
```

For planets and other places we might provide a more structured representation pointing to each type of map:

Resource-Oriented Analysis and Design Map service example: search result representation

- A set of search results is a resource
- Example: /Earth/USA?show=Springfield (All places matched to Springfield in US)

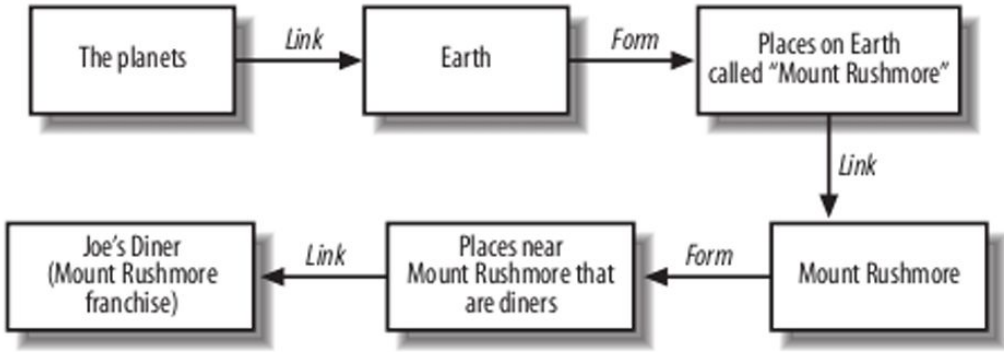
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Search results for “Springfield”</title>
  <link rel="self" href="/search?q=springfield" />
</head>
<body>
  <h1>Search results for “Springfield”</h1>

  <ul>
    <li><a href="/places/springfield-il">Springfield, Illinois, USA</a></li>
    <li><a href="/places/springfield-mo">Springfield, Missouri, USA</a></li>
    <li><a href="/places/springfield-ma">Springfield, Massachusetts, USA</a></li>
    <li><a href="/places/springfield-uk">Springfield, UK</a></li>
  </ul>
```

One use of the search result representation:

- Fetch the map tile for a point.
- Fetch the surrounding map tiles.
- Stitch the tiles together with markers for the places.

Resource-Oriented Analysis and Design Map service example: search result links

- How to notify users all possible query?
- One method from the human web is a form.



Richardson and Ruby (2007), Fig. 5-4

```
<form id="searchPlace" method="get" action="">
 <p>
  Show places, features, or businesses:
  <input id="term" repeat="template" name="show" />
  <input class="submit" />
 </p>
</form></screen>
```

Resource-Oriented Analysis and Design Map service example: HTTP response

- In the HTTP response we need appropriate headers and response codes.
- The Content-Type header should be correct, e.g., application/xhtml+xml or image/png.
- Other useful headers include Last-Modified, which can be used with caching and If-Modified to implement conditional GET.
- In normal cases we return status code 200 (OK) and the response body.
- When a conditional GET fails, we would return 304 (Not Modified).
- Other conditions include 404 (Not Found), 303 (See Other), 400 (Bad Request), 503 (Service Unavailable), or 500 (Internal Server Error).

Example:

- 400 Bad Request: For invalid coordinates, missing required parameters.
- 403 Forbidden: When a non-admin user tries to access restricted actions.
- 404 Not Found: When the requested POI or route doesn't exist.
- 500 Internal Server Error: For unexpected errors.

Resource-Oriented Analysis and Design Map service example: Read/write resources

- So far we just have read-only resources. What happens if we allow users to create, update, and delete resources?
  - Who can add/update/delete places?
- Example: allowing users to add comments about existing map places and new places at arbitrary points.
- First we need resources for authentication and authorization.
- Then we need another round of resource-oriented analysis and design to design the read/write interface.

Resource-Oriented Analysis and Design Map service example: Authentication

- In REST, authentication information travels with every request. The server does not maintain a session or login state (that would break statelessness).
- Each request must be self-contained, including all credentials needed to identify the user

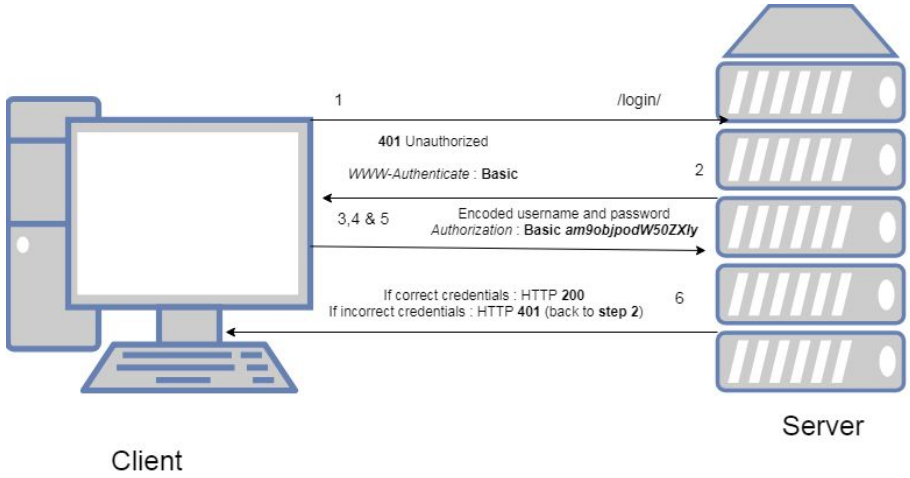
Type	How it works	Typical header
HTTP Basic Auth	Username + password encoded in base64.	`Authorization: Basic dXNlcjpwYXNzd29yZA==`
Token-based (Bearer / API key)	Client sends a token or key issued previously.	`Authorization: Bearer eyJhbGciOi...` or `Authorization: ApiKey 123456`
OAuth 2.0	Delegated access. Client obtains an access token via an authorization server, then uses it.	`Authorization: Bearer <access_token>`
HMAC-Signed Requests	Each request signed with a secret key (used by AWS, etc.). (key + Hash)	`Authorization: AWS4-HMAC-SHA256 Credential=...`

Resource-Oriented Analysis and Design Map service example: Authentication

- In REST, authentication information travels with every request. The server does not maintain a session or login state (that would break statelessness).
- Each request must be self-contained, including all credentials needed to identify the user
- In the spirit of REST, we can use HTTP's capabilities.
- A simple possibility is HTTP Basic authentication:
  - Client requests resource.
  - Server responds with 401 (Unauthorized) and the WWW-Authenticate header.
  - Client responds with a (cleartext) username and password.
- To prevent snooping the password, HTTP Basic authentication should only be used with SSL

Resource-Oriented Analysis and Design Map service example: Authentication

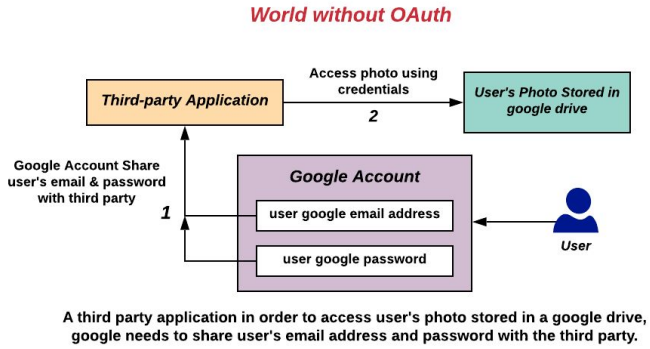
- In the spirit of REST, we can use HTTP's capabilities.
- A simple possibility is HTTP Basic authentication:
  - Client requests resource.
  - Server responds with 401 (Unauthorized) and the WWW-Authenticate header.
  - Client responds with a (cleartext) username and password.
- The credential (base64 encoding) must be attached to every request message
- To prevent snooping the password, HTTP Basic authentication should only be used with **SSL**.



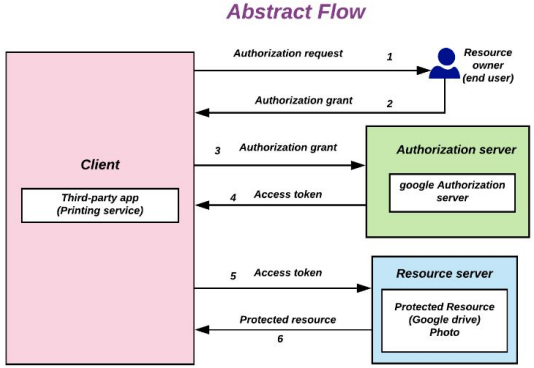
https://techmonger.github.io/41/basic-access-auth/

Resource-Oriented Analysis and Design: Alternative authentication methods (OAuth 2.0)

- An open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords
- Client stores a token that grant limited access
- Each REST API request includes the token in the Authorization header
- The API (resource server) validates the token and decides whether the request is authorized.
    - Pros: Highly secure, supports delegated access, and can work with various client types (web, mobile).
    - Cons: More complex to implement compared to other methods



OAuth 2.0 flow



Resource-Oriented Analysis and Design: Alternative authentication methods (Token-Based)

JSON Web Token (JWT)

- A compact, digitally signed token used to securely transmit information between a client and a RESTful API.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- A token-based stateless authentication mechanism
- JWT consists of 3 parts: Header (signing algorithm), Payload (User data) and Signature
- Can combine with OAuth

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

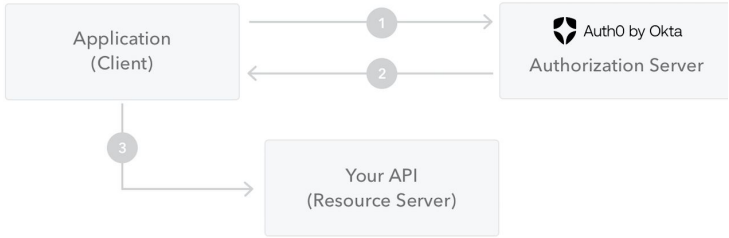
Header (algo)

```
{
  "loggedInAs": "admin",
  "iat": 1422779638
}
```

Payload (data)

```
HMAC_SHA256(
  secret,
  base64urlEncoding(header) + '.' +
  base64urlEncoding(payload)
)
```

Signature (integrity check)



Resource-Oriented Analysis and Design: Alternative authentication methods (API key)

- A unique generated value is assigned to each first time user, signifying that the user is known.
- When the user attempts to re-enter the system, their unique key is used to prove that they're the same user as before.
- **Implementation**: The API key can be sent in the request header, body, or as a query parameter, depending on the API's design.
- Pros: Easy to implement and allows for permission management per key.
- Cons: Less secure than other methods if keys are exposed

```
const url = 'https://example.com/api';
const apiKey = 'your_api_key';

fetch(url, {
    method: 'GET',
    headers: {
        'Authorization': `Api-Key ${apiKey}`
    }
})
```

Provider	How key is generated	How shared
Google Cloud / Maps / YouTube	Generated in Google Cloud Console	Shown once in the console
OpenAI API	Generated in user dashboard	Displayed once; must be copied manually
AWS	Access keys created in IAM	Downloaded as CSV once

Resource-Oriented Analysis and Design Map service example: Read/Write Oriented Services

To provide CRUD operations for user account information:

- Dataset: usernames and passwords
- Split into resources: each user account is a resource
- Name the resources: https://maps.example.com/user/{user-name}
- Expose a subset of the uniform interface: PUT/POST for a new account, DELETE for a delete, GET or HEAD to check if a username already exists.

Treating user account as a resource

- Each account has a stable URI.
- The service supports operations like GET, PUT, DELETE, etc., depending on what you expose.
- You focus on representing the state of the account (profile info, settings) rather than performing “login/logout” actions as RPC.

Resource-Oriented Analysis and Design Map service example: Read/Write Oriented Services

HTTP messages to create a user account and response.

```
POST /accounts HTTP/1.1
Host: example.com
Content-Type: application/xml
Accept: application/xml

<account>
  <username>alice</username>
  <email>alice@example.com</email>
  <fullName>Alice Tan</fullName>
  <password>secret</password>
</account>
```

```
<body>
  <h1>Welcome, alice!</h1>

  <p>Your account has been successfully created, and you are currently logged in.</p>

  <section>
    <h2>Account Information</h2>
    <ul>
      <li><strong>Username:</strong> alice</li>
      <li><strong>Email:</strong> alice@example.com</li>
      <li><strong>Full name:</strong> Alice Tan</li>
    </ul>
  </section>

  <section>
    <h2>Available Actions</h2>
    <ul>
      <li><a rel="edit" href="/accounts/alice">Edit Profile</a></li>
      <li><a rel="change-password" href="/accounts/alice/password">Change Password</a></li>
      <li><a rel="logout" href="/logout">Log Out</a></li>
    </ul>
  </section>
```

Resource-Oriented Analysis and Design Map service example: Read/Write Oriented Services in Ruby on Rails

Rails automatically routes standard RESTful requests when using Scaffold:

rails generate scaffold Account username:string email:string

HTTP Method	Example URI	Controller Action
GET	/accounts	accounts#index
GET	/accounts/1	accounts#show
POST	/accounts	accounts#create
PUT/PATCH	/accounts/1	accounts#update
DELETE	/accounts/1	accounts#destroy

Rails controllers automatically render structured responses:

- status: :created sets HTTP/1.1 201 Created
- location: @account sets Location: /accounts/:id
- render json: generates the representation automatically (the response body)
- So the response message (status code, body, headers) is mostly built.

```
def create
  @account = Account.new(account_params)
  if @account.save
    render json: @account, status: :created, location: @account
  else
    render json: @account.errors, status: :unprocessable_entity
  end
end
```

Resource-Oriented Analysis and Design Map service example: access to account information

- For the response codes we might use 201 (Created), 200 (OK) or 205 (Reset Content) for a legal modification, 200 (OK) for a successful DELETE request.
- Errors might include 415 (Unsupported Media Type) if the input representation is bad, 400 (Bad Request), 401 (Unauthorized), 409 (Conflict) for an attempt to create an already-existing account, 500 (Internal Server Error), or 503 (Service Unavailable).

Resource-Oriented Analysis and Design Map service example: CRUD for custom places

To implement custom place management (places submitted from users):

- URIs: /user/{username}/{planet}/{lat},{long}/{placename}
- POST for adding subordinate comments to an existing resource.
- PUT to modify places (name, location, type, description)
- Form encoding for client representation.
- Links: add forms for changing places when authenticated.

Outline

- Introduction
- Resource-Oriented Architecture
- Resource-Oriented Analysis and Design
- REST and ROA best practices
- SOAP vs REST

REST and ROA best practices: Introduction

We now consider best practices for developing RESTful services. Topics:

- Tips for the uniform interface.
- Why ROA?
- Tips for resource design.
- Building blocks for your services.

REST and ROA best practices: Tips for uniform interface

- A GET or HEAD request should be safe: a client that makes a GET or HEAD request is not requesting any changes to server state.
- A PUT or DELETE request should be *idempotent*. Making more than one PUT or DELETE request to a given URI should have the same effect as making only one request.
- POST request are neither safe nor idempotent.

REST and ROA best practices: Why ROA?

- RESTful resource oriented services are simpler, easier to use, more interoperable, and easier to combine than RPC services:
  - Addressability allows others to use your service in mashups.
  - Statelessness simplifies server-side software and allows horizontal scaling.
  - Connectedness allows discoverability and for new relationships and resources to emerge.

REST and ROA best practices: Resource design

- Relationships between resources: Given 2 resources that are have some relationships, can assign their relationship as another resource.
- Batch operations: Specify multiple resources in a set by semicolons (;) to allow batch operations e.g. DELETE http://www.example.com/users/user1;user2 to delete both user1 and user2 resources.
- Transactions like bank account transfers can be exposed as resources and executed atomically server side.

REST and ROA best practices: Building blocks

For design and documentation of APIs:

- WADL (Web Application Description Language) was an early specification for describing resources in an API (WSDL style) or can be included within an XML document.
- Swagger is a more modern API specification language based on YAML that is paired with design and client coding tools.

REST and ROA best practices: Building blocks

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <resources base="https://example.com/api/">
    <resource path="planets">
      <method name="GET">
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
      <method name="POST">
        <request>
          <representation mediaType="application/json"/>
        </request>
        <response status="201">
          <representation mediaType="application/json"/>
        </response>
      </method>
    </resource>
    <resource path="planets/{id}">
      <method name="GET">
        <response>
          <representation mediaType="application/json"/>
        </response>
```

WADL provides a machine-readable description of:

- The resources a REST service exposes.
- The HTTP methods supported by each resource.
- The input/output representations (XML, JSON, etc.).
- The parameters, headers, and possible responses.

This example declares:

- /planets supports GET (list) and POST (create).
- /planets/{id} supports GET (retrieve).

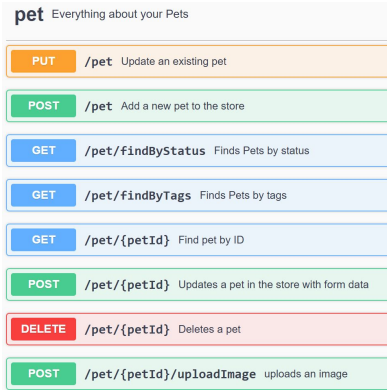
REST and ROA best practices: Building blocks

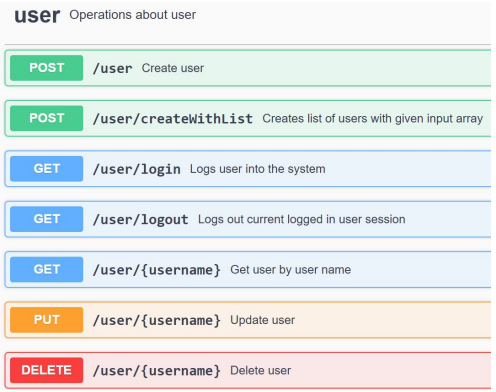
- WADL was never adopted very broadly (XML heavy).
- In 2020, Swagger (http://swagger.io) is probably the most popular tool for designing and documenting REST APIs.
- APIs are simply defined with YAML. Example from Swagger for a “pet store” API:

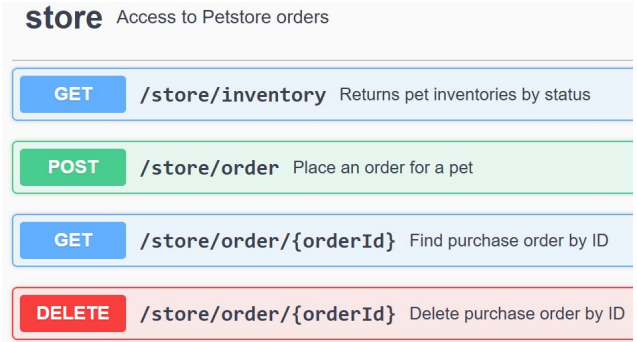
```yaml
openapi: 3.0.2
servers:
  - url: /v3
info:
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about
    Swagger at [http://swagger.io](http://swagger.io). In the third iteration of the pet store, we've switched to the desi
    You can now help us improve the API whether it's by making changes to the definition itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new features in OAS3.
```

REST and ROA best practices: Building blocks

```
  title: Swagger Petstore - OpenAPI 3.0
  termsOfService: 'http://swagger.io/terms/'
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: 'http://swagger.io'
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: 'http://swagger.io'
  - name: user
    description: Operations about user
```

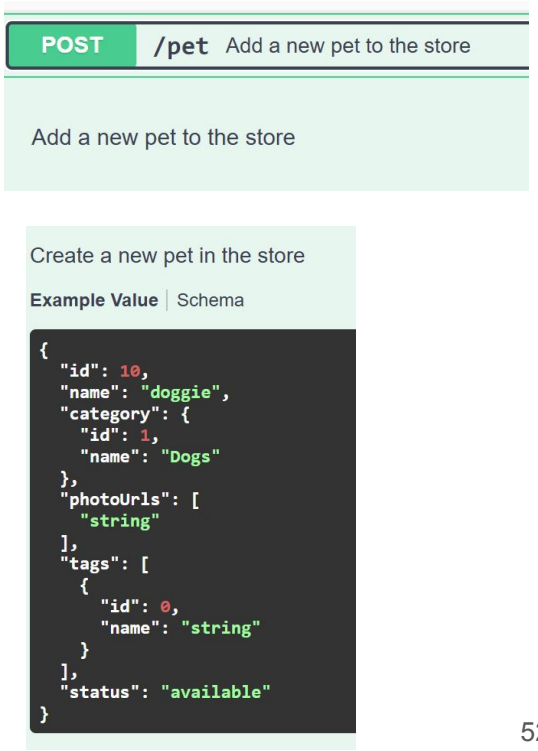






REST and ROA best practices: Building blocks

```yaml
paths:
  /pet:
    post:
      tags:
        - pet
      summary: Add a new pet to the store
      description: Add a new pet to the store
      operationId: addPet
      responses:
        '200':
          description: Successful operation
          content:
            application/xml:
              schema:
                $ref: '#/components/schemas/Pet'
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        '405':
          description: Invalid input
```



Outline

- Introduction
- Resource-Oriented Architecture
- Resource-Oriented Analysis and Design
- REST and ROA best practices
- SOAP vs REST

SOAP vs REST

SOAP

- SOAP by itself is simple and was originally designed as an RPC protocol.
- With WS-* it aimed to solve big, process-oriented, brokered distributed system problems.
- In these systems, transactions are crucial. E.g. book a flight, hotel, rental car atomically.
- Web Service Description Language (WSDL) adds strong typing to SOAP, providing port and binding abstractions that tie us down to the RPC model.
- REST people prefer simpler more flexible mechanisms like a human readable document or a Swagger specification that do not abstract away the Web.

SOAP vs REST: Security

- WS-Security provides many authentication, signature, and federation schemes.
- REST APIs typically use HTTP authentication over SSL, OAuth, or a custom token-based authentication mechanism e.g. JWT.
- Beware of security threat in many token-based authentication system e.g. key bruteforce, self-signed JWT etc.. (link)

Conclusions

REST API

- Statelessness: Each client request contains all necessary information for the server to process it, allowing scalability without server-side session management 14.
- Cacheability: Responses from the server can be cached by clients to improve performance and reduce server load, as repeated requests for the same resource can be served from cache.
- Client-Server Architecture: This separation of concerns allows clients to focus on the user interface while servers handle data processing and storage, enhancing flexibility and scalability.
- Uniform Interface: REST APIs use standard HTTP methods (GET, POST, PUT, DELETE) and clear resource naming conventions, simplifying interactions 410.
- Idempotency

References

- https://en.wikipedia.org/wiki/OAuth
- https://portswigger.net/web-security/jwt
- https://jwt.io/
- https://swagger.io/
- https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design