My Rules of REST

These match most common REST implementations

- URL represents a "resource" to interact with
- HTTP method is the interaction with the resource
- HTTP Status code is interaction result

These are MY summary of the core REST concepts

- Can't Google "three rules of REST" and find this
- REST covers a lot, this is the **core**

First Rule of REST

The URL represents a "resource" to interact with

- Often a noun (the HTTP method is the verb)
- Plural if a collection
- Common Best Practice: kebab-case
    - **Good** - `/students/`
    - **Good** - `/grades/`
    - **Good** - `/locations/`
    - **Bad** - `/addStudent/`
    - **Bad** - `/updateGrade/`
    - **Bad** - `/searchLocations/`

More REST Rule 1: URL as resource (the "thing")

- Parameters: in query, body, or path
- Often different based on method
    - `GET /students`
    - `GET /students?startsWith=Am`
    - `POST /students/`
    - `PUT /students/Li/Xui/`
    - `PATCH /students/34322/`
    - `DELETE /students?billingStatus=overdue`
- **The path of the URL identifies the "thing"**
    - Params do NOT identify the resource
    - Params DO filter the resource

Second Rule of REST

HTTP method is the interaction with the resource

- The URL is the "thing"
- The method is what you "do" to it

Examples of the Second Rule of REST

The method shows the kind of interaction:

- `GET /students/` - read
- `POST /students/` - create
- `PUT /students/Naresh/Rajkumar` - overwrite
- `DELETE /students/Naresh/Rajkumar` - remove
- `PATCH /students/Naresh/Rajkumar` - partial update

These have passed params, but

- Method and the URL alone say what is happening

POST vs PUT vs PATCH

Common confusion: **Create** vs **Overwrite** vs **Update**

- POST (create)
    - No existing record; Create new one
- PUT (replace)
    - Replace existing record
    - Save nothing from existing record
- PATCH (update)
    - Replace certain fields in the record
    - Unmentioned fields stay as-is

What is passed/received?

- `POST /students/` - **create**
  - Send: (data for new student)
  - Get: (url or data to identify new record)
- `PUT /students/Naresh/Kumar` - **overwrite**
  - Send: (data to replace with)
  - Get: (usually updated record)
- `PATCH /students/Naresh/Kumar` - **partial update**
  - Send: (fields with changed values)
  - Get: (usually updated record)

Third Rule of REST

HTTP Status code is interaction result

- There are many Status codes!
    - With meaningful names
    - Use them!
    - Make sure to confirm the meaning (MDN)
- Add details in body

Status Codes

Some general "classes" of status codes

- 100-199 (1xx): Informational (very rare)
- 200-299 (**2xx**): Successful
- 300-399 (**3xx**): Redirection
- 400-499 (**4xx**): Error (client-caused)
- 500-599 (**5xx**): Error (server-side)

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

REST Status Code Examples

Some common scenarios

- **200 (OK)** - Means real success
- **400 (Bad Request)** - bad input
    - Provide detail in body of response
- **404 (Not Found)**
    - Common point of confusion
        - See next slide
- **500 (Internal Server Error)** - server had issue
    - Not user's fault
    - Not expected!

Common issues with some REST status codes

- **404 (Not Found)**
  - API Path wrong?
  - OR API Path right, but that data doesn't exist
  - Can return a 200 w/an empty data (`{}` or `[]`)
  - Can return a JSON body making it clear
  - Service calls should not return html pages
- **204 (No Content)**
  - Could be returned by a POST/PUT/PATCH
  - Saves on bytes sent
  - But makes parsing service results harder

REST Response

- Other than HTTP Status code
    - Not much direction given
- Common responses (Can vary!)
    - If server created a UUID/ID for new resource
        - Provide in response
            - Record or URL
    - If a record changed
        - Provide the new record
    - If an error code
        - Provide details in body
        - Details in same format as success

JSON is common

JSON is common, even from non-JS services

Pro:

- Very portable
- Very readable

Con:

- No built-in schema validation
- No comments

Basic REST Express Example

```
const cats = {};

app.get('/cats', (req, res) => {
  res.json(Object.keys(cats));
});

app.get('/cats/:name', (req, res) => {
  const name = req.params.name;
  if(cats[name]) {
    res.json(cats[name]);
    return;
  }
  res.status(404).json({ error: `Unknown cat: ${name}` });
});
```

- `:name` syntax (express) sets the `req.params.name`
  - example: `GET /cats/Jorts`
- `.json()` does `JSON.stringify()`
  - AND sets the response `content-type` header

More REST Express Example

```
app.post('/cats', express.json(), (req, res) => {
  const name = req.body.name;
  if(!name) {
    res.status(400).json({ error: "'name' required" });
  } else if(cats[name]) {
    res.status(409).json({ error: `duplicate: ${name}`});
  } else {
    cats[name] = req.body; // Poor Security!  Unsanitized!
    res.sendStatus(200);
  }
});
```

`express.json()` middleware requires request `content-type` of `application/json`, populates `req.body`

No `content-type` === no `req.body` value

A REST service in express()

- Have a route URL that matches Rule 1
- Use a method that matches Rule 2
- Use correct status codes for Rule 3
- Check for any auth requirements!
    - Same req.cookie/sid checks
- Parse incoming body data
    - `express.json()` for JSON
- Set `res.status` if not 200
- Send JSON data in response
    - `res.json()`
- No HTML, No Redirects

REST in Express Auth Details

- Check for any auth requirements!
    - On ALL requests that expect auth'ed user
    - Same `req.cookie`/`sid` checks
        - But **no redirect/login form if bad sid!**
        - Send correct status if bad sid
        - 401 (Auth Missing)
            - If no sid/bad sid
        - 403 (Forbidden)
            - If valid sid but not allowed
            - We also do this for user "dog"

REST in Express Parsing Details

- Resource identifiers from URL path in `req.params`
- Parse incoming body data
    - `express.json()` for JSON
    - Sanitize incoming data!
- Set appropriate status if data has problems
    - 400 (Bad Request)
        - General "user sent bad data" response
        - Provide details in response body
    - 409 (Conflict)
        - User request conflicts with existing data
    - Send services data as JSON in the body

REST in Express Sending Response Details

- If status is not 200 (Success)
    - Set `res.status` BEFORE `.send()`/`.json()`
    - Can't change/set status after response sent!
- Send JSON data in response
    - `res.json()`
        - Stringifies JSON Body
        - Sets content-type header for response
    - I recommend JSON for both errors/success
        - Makes parsing in client easier
- No HTML, No Redirects
    - Service response is not a page response

Writing a REST Service

- Service is entirely state/data changes!
    - No presentation! No HTML View!
    - Server MVC pattern still valid
        - "View" now subset of state to return
            - Often not exact actual state
            - Only the data the client needs
- Server state and Client state NOT the same
    - Often similar, not the same