Vineet Sagar Self Prep System D... Latest commit b89d37c on Apr 22, 2020 History

0 contributors

308 lines (254 sloc) 26.3 KB Raw Blame

- REST best practices
  - Consistency
    - Endpoint naming conventions
    - HTTP verbs and CRUD consistency
    - Versioning
    - Data transfer format
    - HTTP status codes and error handling
    - Paging
      - Scaling REST web services
        - Keeping service machine stateless
          - Benefits
          - Common use cases needing share state
        - Caching service responses
          - Cache-Control header
          - Expires
          - Last-Modified/If-Modified-Since/Max-age
          - ETag
          - Vary: Authorization

        - Functional partitioning
    - Security
        - Throttling
        - Use OAuth2 with HTTPS for authorization, authentication and confidentiality.
            - ??? API authentication
    - Documentation
    - Others

REST best practices

- Could look at industrial level api design example by Github

Consistency

Endpoint naming conventions

- Use all lowercase, hyphenated endpoints such as /api/verification-tokens. This increases URL "hackability", which is the ability to manually go in and modify the URL by hand. You can pick any naming scheme you like, as long as you're consistent about it.
- Use a noun or two to describe the resource, such as users, products, or verification-tokens.
- Always describe resources in plural: /api/users rather than /api/user. This makes the API more semantic.
    - Collection resource: /users
    - Instance resource: /users/007

HTTP verbs and CRUD consistency

- Use HTTP verbs for CRUD operations (Create/Read/Update/Delete).
    - Updates & creation should return a resource representation
        - A PUT, POST or PATCH call may make modifications to fields of the underlying resource that weren't part of the provided parameters (for example: created_at or updated_at timestamps). To prevent an API consumer from having to hit the API again for

an updated representation, have the API return the updated (or created) representation as part of the response.
    - In case of a POST that resulted in a creation, use a HTTP 201 status code and include a Location header that points to the URL of the new resource.

Verb	Endpoint	Description
GET	/products	Gets a list of products
GET	/products/:id	Gets a single product by ID
GET	/products/:id/parts	Gets a list of parts in a single product
PUT	/products/:id/parts	Inserts a new part for a particular product
DELETE	/products/:id	Deletes a single product by ID
PUT	/products	Inserts a new product
HEAD	/products/:id	Returns whether the product exists through a status code of 200 or 404
PATCH	/products/:id	Edits an existing product by ID
POST	/authentication/login	Most other API methods should use POST requests

Versioning

- **What is versioning?** In traditional API scenarios, versioning is useful because it allows you to commit breaking changes to your service without demolishing the interaction with existing consumers.
- **Whether you need versioning?** Unless your team and your application are small enough that both live in the same repository and developers touch on both indistinctly, go for the safe bet and use versions in your API.
    - Is the API public facing as well? In this case, versioning is necessary, baking a bit more predictability into your service's behavior.
    - Is teh API used by several applications? Are the API and the front end developed by separated teams? Is there a drawn-out process to

change an API point? If any of these cases apply, you're probably better off versioning your API.

- **How to implement versioning?** There are two popular ways to do it:
  - The API version should be set in HTTP headers, and that if a version isn't specified in the request, you should get a response from the latest version of the API. But it can lead to breaking changes inadvertently.
  - The API version should be embedded into the URL. This identifies right away which version of the API your application wants by looking at the requested endpoint. An API version should be included in the URL to ensure browser explorability.

Data transfer format

- **Request**: You should decide on a consistent data-transfer strategy to upload the data to the server when making PUT, PATCH, or POST requests that modify a resource in the server. Nowadays, JSON is used almost ubiquitously as the data transport of choice due to its simplicity, the fact that it's native to browsers, and the high availability of JSON parsing libraries across server-side languages.
- **Response**:
  - Responses should conform to a consistent data-transfer format, so you have no surprises when parsing the response. Even when an error occurs on the server side, the response is still expected to be valid according to the chosen transport; For example, if your API is built using JSON, then all the responses produced by our API should be valid JSON.
  - You should figure out the envelope in which you'll wrap your responses. An envelope, or message wrapper, is crucial for providing a consistent experience across all your API endpoints, allowing consumers to make certain assumptions about the responses the API provides. A useful starting point may be an object with a single field, named data, that contains the body of your response.

```
{
        "data" : {}      // actual response
}
```

HTTP status codes and error handling

- Choose the right status codes for the problems your server is encountering so that the client knows what to do, but even more important is to make sure the error messages that are coming back are clear.
    - An authentication error can happen because the wrong keys are used, because the signature is generated incorrectly, or because it's passed to the server in the wrong way. The more information you can give to developers about how and why the command failed, the more likely they'll be able to figure out how to solve the problem.
- When you respond with status codes in the 2XX Success class, the response body should contain all of the relevant data that was requested. Here's an example showing the response to a request on a product that could be found, alongside with the HTTP version and status code:

```
HTTP/1.1 200 OK
{
        "data": {
                "id" : "baeb-b001",
                "name" : "Angry Pirate Plush Toy",
                "description" : "Batteries not included",
                "price" : "$39.99",
                "categories": ["plushies", "kids"]
        }
}
```

- If the request is most likely failed due to an error made by the client side (the user wasn't properly authenticated, for instance), you should use 4XX Client Error codes. If the request is most likely failed due to a server side error, then you should use 5XX error codes. In these cases, you should use the error field to describe why the request was faulty.

```
// if input validation fails on a form while attempting to create a
HTTP/1.1 400 Bad Request
{
        "error": {
                "code": "bf-400",
                "message": "Some required fields were invalid.",
                "context": {
                        "validation": [
```

```
                                "The product name must be 6-20 alp
                                "The price cann't be negative",
                                "At least one product category sho
                        ]
                }
        }
}

// server side error
{
        "error": {
                "code": "bf-500",
                "message": "An unexpected error occurred while acc
                "context": {
                        "id": "baeb-b001"
                }
        }
}
```

Paging

- Suppose a user makes a query to your API for /api/products. How many products should that end point return? You could set a default pagination limit across the API and have the ability to override that default for each individual endpoint. Within a reasonable range, the consumer should have the ability to pass in a query string parameter and choose a different limit.
    - Using Github paging API as an example, requests that return multiple items will be paginated to 30 items by default. You can specify further pages with the ?page parameter. For some resources, you can also set a custom page size up to 100 with the ?per_page parameter. Note that for technical reasons not all endpoints respect the ?per_page parameter, see events for example. Note that page numbering is 1-based and that omitting the ?page parameter will return the first page.

```
curl 'https://api.github.com/user/repos?page=2&per_page=100'
```

- Common parameters
    - page and per_page. Intuitive for many use cases. Links to "page 2" may not always contain the same data.

    - offset and limit. This standard comes from the SQL database world, and is a good option when you need stable permalinks to result sets.
    - since and limit. Get everything "since" some ID or timestamp. Useful when it's a priority to let clients efficiently stay "in sync" with data. Generally requires result set order to be very stable.

- Metadata
    - Include enough metadata so that clients can calculate how much data there is, and how and whether to fetch the next set of results. Examples of how that might be implemented:

```
{
  "results": [ ... actual results ... ],
  "pagination": {
    "count": 2340,
    "page": 4,
    "per_page": 20
  }
}
```

- Link header
    - The pagination info is included in the Link header. It is important to follow these Link header values instead of constructing your own URLs. In some instances, such as in the Commits API, pagination is based on SHA1 and not on page number.

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; re
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="
```

- Rel attribute
    - describes the relationship between the requested page and the linked page

Name	Description
next	The link relation for the immediate next page of results.
last	The link relation for the last page of results.

first	The link relation for the first page of results.
prev	The link relation for the immediate previous page of results.

- Cases exist where data flows too rapidly for traditional paging methods to behave as expected. For instance, if a few records make their way into the database between requests for the first page and the second one, the second page results in duplicates of items that were on page one but were pushed to the second page as a result of the inserts. This issue has two solutions:
  - The first is to use identifiers instead of page numbers. This allows the API to figure out where you left off, and even if new records get inserted, you'll still get the next page in the context of the last range of identifiers that the API gave you.
  - The second is to give tokens to the consumer that allow the API to track the position they arrived at after the last request and what the next page should look like.

Scaling REST web services

Keeping service machine stateless

Benefits

- You can distribute traffic among your web service machines on a per-request basis. You can deploy a load balancer between your web services and their clients, and each request can be sent to any of the available web service machines. Being able to distribute requests in a round-robin fashion allows for better load distributionn and more flexibility.
- Since each web service request can be served by any of the web service machines, you can take service machines out of the load balancer pool as soon as they crash. Most of the modern load balancers support heartbeat checks to make sure that web services machines serving the traffic are available. As soon as a machine crashes or experiences some other type of failure, the load balancer will remove that host from the load-balancing pool, reducing the capacity of the cluster, but preventing clients from timing out or failing to get responses.
- You can restart and decommission servers at any point in time without worrying about affecting your clients. For example, if you want to shut

down a server for maintenance, you need to take that machine out of the load balancer pool. Most load balancers support graceful removal of hosts, so new connections from clients are not sent to that server any more, but existing connections are not terminated to prevent client-side errors. After removing the host from the pool, you need to wait for all of your open connections to be closed by your clients, which can take a minute or two, and then you can safely shut down the machine without affecting even a single web service request.

- You will be able to perform zero-downtime updates of your web services. You can roll out your changes to one server at a time by taking it out of rotation, upgrading, and then putting it back into rotation. If your software does not allow you to run two different versions at the same time, you can deploy to an alternative stack and switch all of the traffic at once on the load balancer level.
- By removing all of the application state from your web services, you will be able to scale your web services layer by simply adding more clones. All you need to do is adding more machines to the load balancer pool to be able to support more concurrent connections, perform more network I/O, and compute more responses.

Common use cases needing share state

- The first use case is related to security, as your web service is likely going to require clients to pass some authentication token with each web service request. The token will have to be validated on the web service side, and client permissions will have to be evaluated in some way to make sure that the user has access to the operation they are attempting to perform. You could cache authentication and authorization details directly on your web service machines, but that could cause problems when changing permissions or blocking accounts, as these objects would need to expire before new permissions could take effect. A better approach is to use a shared in-memory object cache and have each web service machine reach out for the data needed at request time. If not present, data could be fetched from the original data store and placed in the object cache. By having a single central copy of each cached object, you will be able to easily invalidate it when users' permissions change.
- Another common problem when dealing with stateless web services is how to support resource locking. You can use distributed lock systems like

Zookeeper or even build your own simple lock service using a data store of your choice. To make sure your web services scale well, you should avoid resource locks for as long as possible and look for alternative ways to synchronize parallel processes.
    - Distributed locking creates an opportunity for your service to stall or fail. This, in turn, increases your latency and reduces the number of parallel clients that your web service can serve. Instead of resource locks, you can sometimes use optimistic concurrency control where you check the state before the final update rather than acquiring locks. You can also consider message queues as a way to decouple components and remove the need for resource locking in the first place.
    - If none of the above techniques work for you and you need to use resource locks, it is important to strike a balance between having to acquire a lot of fine-grained locks and having coarse locks that block access to large sets of data. By having too many fine-grained locks, you increase risk for deadlocks. If you use few coarse locks, you can increase concurrency because multiple web services can be blocked waiting on the same resource lock.
- The last challenge is application-level transactions. A distributed transaction is a set of internal service steps and external web service calls that either complete together or fail entirely. It is very difficult to scale and coordinate without sacrificing high availability. The most common method of implementing distributed transactions is the 2 Phase Commit algorithm. An example of a distributed transaction would be a web service that creates an order within an online shop.
    - The first alternative to distributed transactions is to not support them at all. As long as the core of your system functionality is not compromised, your company may be fine with such a minor inconsistencies in return for the time saved developing it.
    - The second alternative to distributed transactions is to provide a mechanism of compensating transactions. A compensating transactins can be used to revert the result of an operation that was issued as part of a larger logical transaction that has failed. The benefit of this approach is that web services do not need to wait for one another; they do not need to maintain any state or resources for the duration of the overarching transaction either.

Caching service responses

- From a caching perspective, the GET method is the most important one, as GET responses can be cached.
- To be able to scale using cache, you would usually deploy reverse proxies between your clients and your web service. As your web services layer grow, you may end up with a more complex deployment where each of your web services has a reverse proxy dedicated to serve its results. Depending on the reverse proxy used, you may also have load balancers deployed between reverse proxies and web services to distribute the underlying network traffic and provide quick failure recovery.

Cache-Control header

- Setting the Cache-Control header to private bypasses intermediaries (such as nginx, other caching layers like Varnish, and all kinds of hardware in between) and only allows the end client to cache the response.
- Setting it to public allows intermediaries to store a copy of the response in their cache.

Expires

- Tells the browser that a resource should be cached and not requested again until the expiration date has elapsed.
- It's hard to define future Expires headers in API responses because if the data in the server changes, it could mean that the cleint's cache becomes stale, but it doesn't have any way of knowing that until the expiration date. A conservative alternative to Expires header in responses is using a pattern callled "conditional requests"

Last-Modified/If-Modified-Since/Max-age

- Specifying a Last-Modified header in your response. It's best to specify a max-age in the Cache-Control header, to let the browser invalidate the cache after a certain period of time even if the modification date doesn't change

> Cache-Control: private, max-age=86400 Last-Modified: Thu, 3 Jul 2014 18:31:12 GMT

- The next time the browser requests this resource, it will only ask for the contents of the resource if they're unchanged since this date, using the If-Modified-Since request header. If the resource hasn't changed since Thu, 3 Jul 2014 18:31:12 GMT, the server will return with an empty body with the 304 Not Modified status code.

> If-Modified-Since: Thu, 3 Jul 2014 18:31:12 GMT

ETag

- ETag header is usually a hash that represents the source in its current state. This allows the server to identify if the cached contents of the resource are different than the most recent versions:

> Cache-Control: private, max-age=86400 ETag: "d5jiodjiojiojo"

- On subsequent requests, the If-None-Match request header is sent with the ETag value of the last requested version for the same resource. If the current version has the same ETag value, your current version is what the client has cached and a 304 Not Modified response will be returned.

> If-None-Match: "d5jiodjiojiojo"

Vary: Authorization

- You could implement caching of authenticated REST resources by using headers like Vary: Authorization in your web service responses. Responses with such headers instruct HTTP caches to store a separate response for each value of the Authorization header.

Functional partitioning

- By functional partitioning, you group closely related functionality together. The resulting web services are loosely coupled and they can now be scaled independently.

Security

Throttling

- This kind of safeguarding is usually unnecessary when dealing with an

internal API, or an API meant only for your front end, but it's a crucial measure to make when exposing the API publicly.

- Suppose you define a rate limit of 2,000 requests per hour for unauthenticated users; the API should include the following headers in its responses, with every request shaving off a point from the remainder. The X-RateLimit-Reset header should contain a UNIX timestamp describing the moment when the limit will be reset

> X-RateLimit-Limit: 2000 X-RateLimit-Remaining: 1999 X-RateLimit-Reset: 1404429213925

- Once the request quota is drained, the API should return a 429 Too Many Request response, with a helpful error message wrapped in the usual error envelope:

```
X-RateLimit-Limit: 2000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1404429213925
{
        "error": {
                "code": "bf-429",
                "message": "Request quota exceeded. Wait 3
minutes and try again.",
                "context": {
                        "renewal": 1404429213925
                }
        }
}
```

- However, it can be very useful to notify the consumer of their limits before they actually hit it. This is an area that currently lacks standards but has a number of popular conventions using HTTP response headers.

Use OAuth2 with HTTPS for authorization, authentication and confidentiality.

??? API authentication

- https://jobandtalent.engineering/api-authentication-strategies-in-a-microservices-architecture-dc84cc61c5cc

Documentation

- Good documentation should
  - Explain how the response envelope works
  - Demonstrate how error reporting works
  - Show how authentication, paging, throttling, and caching work on a high level
  - Detail every single endpoint, explain the HTTP verbs used to query those endpoints, and describe each piece of data that should be in the request and the fields that may appear in the response
- Test cases can sometimes help as documentation by providing up-to-date working examples that also indicate best practices in accessing an API. The docs should show examples of complete request/response cycles. Preferably, the requests should be pastable examples - either links that can be pasted into a browser or curl examples that can be pasted into a terminal. GitHub and Stripe do a great job with this.
  - CURL: always illustrating your API call documentation by cURL examples. Readers can simply cut-and-paste them, and they remove any ambiguity regarding call details.
- Another desired component in API documentation is a changelog that briefly details the changes that occur from one version to the next. The documentation must include any deprecation schedules and details surrounding externally visible API updates. Updates should be delivered via a blog (i.e. a changelog) or a mailing list (preferably both!).

Others

- Provide filtering, sorting, field selection and paging for collections
  - Filtering: Use a unique query parameter for all fields or a query language for filtering.
    - GET /cars?color=red Returns a list of red cars
    - GET /cars?seats<=2 Returns a list of cars with a maximum of 2 seats
  - Sorting: Allow ascending and descending sorting over multiple fields
    - GET /cars?sort=-manufactorer,+model. This returns a list of cars sorted by descending manufacturers and ascending models.
  - Field selection: Mobile clients display just a few attributes in a list.

They don’t need all attributes of a resource. Give the API consumer the ability to choose returned fields. This will also reduce the network traffic and speed up the usage of the API.
        - GET /cars?fields=manufacturer,model,id,color
    - Paging:
        - Use limit and offset. It is flexible for the user and common in leading databases. The default should be limit=20 and offset=0. GET /cars?offset=10&limit=5.
        - To send the total entries back to the user use the custom HTTP header: X-Total-Count.
        - Links to the next or previous page should be provided in the HTTP header link as well. It is important to follow this link header values instead of constructing your own URLs.
- Content negotiation
    - Content-type defines the request format.
    - Accept defines a list of acceptable response formats. If a client requires you to return application/xml and the server could only return application/json, then you'd better return status code 406.
    - We recommend handling several content distribution formats. We can use the HTTP Header dedicated to this purpose: “Accept”.
    - By default, the API will share resources in the JSON format, but if the request begins with “Accept: application/xml”, resources should be sent in the XML format.
    - It is recommended to manage at least 2 formats: JSON and XML. The order of the formats queried by the header “Accept” must be observed to define the response format.
    - In cases where it is not possible to supply the required format, a 406 HTTP Error Code is sent (cf. Errors — Status Codes).
- Pretty print by default and ensure gzip is supported
    - An API that provides white-space compressed output isn't very fun to look at from a browser. Although some sort of query parameter (like ?pretty=true) could be provided to enable pretty printing, an API that pretty prints by default is much more approachable.
- HATEOAS: Hypertext As The Engine of Application State
    - There should be a single endpoint for the resource, and all of the other actions you’d need to undertake should be able to be discovered by

inspecting that resource.
    - People are not doing this because the tooling just isn't there.