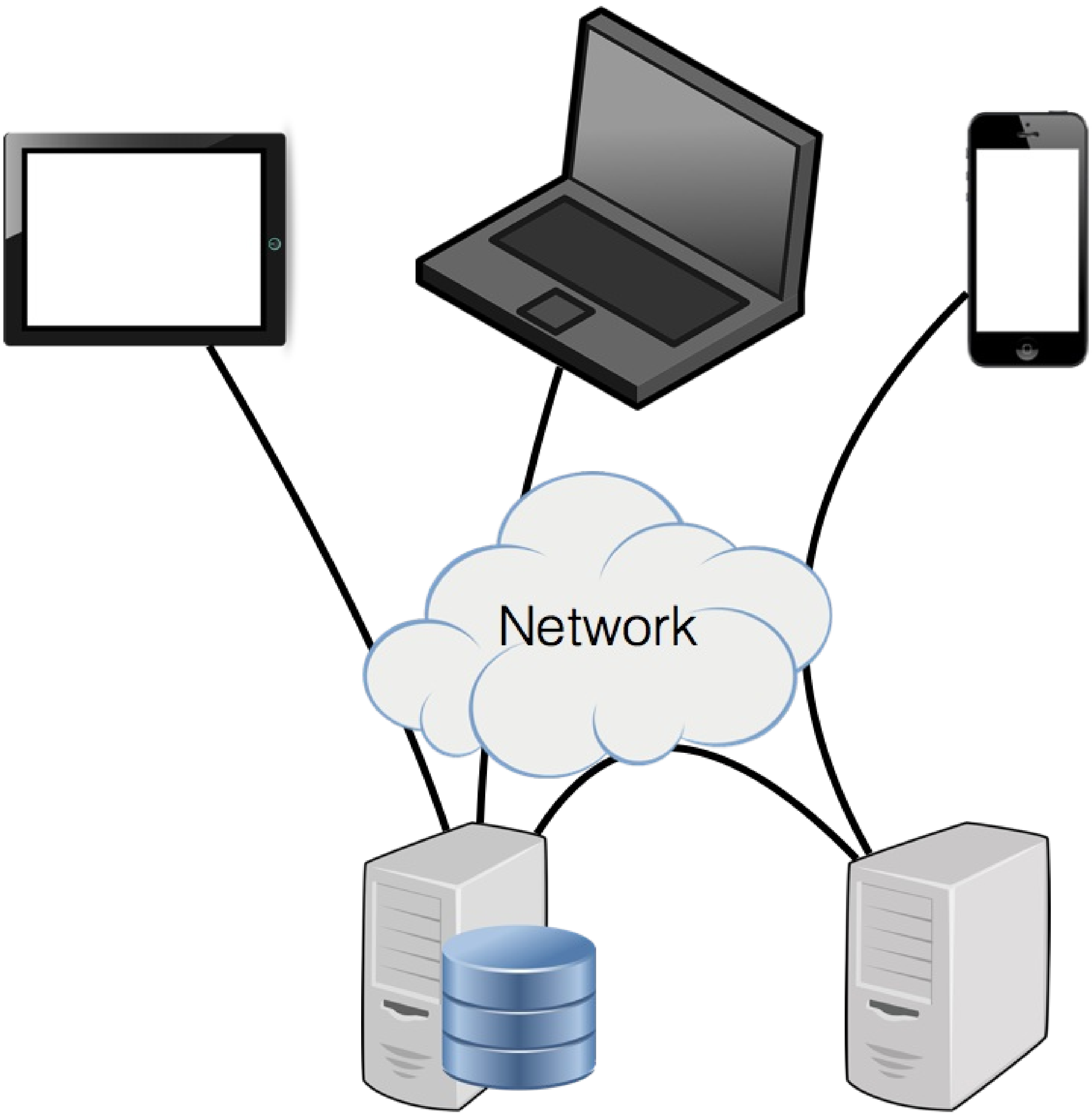
REST

REST is an architectural style heavily used in web-based systems for providing services between different systems (and interfaces). REST defines a means of connecting systems together that focuses on their interfaces rather than their implementations using self-descriptive messages. In this way, REST provides a language to define a specific kind of APIs for communicating across a network. REST services are transported over the network using the HTTP protocol.



REST services define their resources using uniform resource identifiers (URIs). URIs differ from URLs in that a URI does not define its network location and access mechanism (e.g., `/dataset` is a URI while `http://foo.com/dataset` is a URL).

Additional data can be sent to a REST endpoint in the form of HTTP headers or in the HTTP body. It is idiomatic for headers to not contain any information about the location of the resource (this should be in the URI) or to carry any data about the request (this should be in the body); headers are mainly for sending metadata (e.g., to request the response data format or encoding or to provide authorization credentials).

Verbs

REST describes how HTTP verbs should be used when communicating with REST services. REST services respond with payloads (typically JSON or XML) and an integer-based response code. Each REST request uses only one verb and receives only one response.

- `GET` Used to read a resource. Get requests should never modify data on a server, meaning that calling them repeatedly should have no visible effects on the server (that is, the request is idempotent). GET requests can have a body but this is not idiomatic; requested resources are defined by the URI alone. Common response codes are 200 (ok) and 404 (not found).

- `PUT` Used to update a resource at a known URI. That said, PUT can also be used to create a resource if the client is allowed to define a new URI that does not already exist. Response bodies are not typically returned for these requests. PUT modifies server resources but in the same way for every request and can be considered idempotent. 200 (ok), 204 (no content), and 404 (not found) are common response codes.

- `POST` Used to create a new resource. The response body should contain a URI to the newly-created resource. POST modifies server resources and is not idempotent. Typical response codes are 201 (created), 404 (not found), and 409 (conflict).

- `DELETE` Removes a resource from the server. Response body is typically not emitted. Although response codes will differ on subsequent requests, the resources on the server will be unchanged so this verb is idempotent. Typical response code is 200 (ok) or 404 (not found).

- `PATCH` Used to modify (not completely replace or delete) a resource. Not idempotent. This is the least-frequently used verb. Typical response codes are 200 (ok), 204 (no content), 404 (not found).

A simple set of quick tips and http status codes can be helpful for your reference.

REST services almost universally respond to requests with XML or JSON requests. XML documents are more verbose, harder to read and write by hand, and are more difficult to parse.

```xml
<?xml version="1.0" encoding="utf-8"?>
<response>
    <code>200</code>
</response>
```

In contrast, JSON documents are easier to read and write, are less verbose, and are easier to parse but are not amenable to comments which can make them harder to document (in a way that is still a valid

JSON document).

```
{
    code: 200
}
```

The pushback against XML can be more clearly understood when looking at a SOAP envelope, one of the predominant forms of communicating between network services when JSON was developed:

```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    <soap:Header>
        <origin>remote</origin>
    </soap:Header>
    <soap:Body>
        <response code="200"></response>
    </soap:Body>
</soap:Envelope>
```

Statelessness

REST services are often viewed as highly scalable. The main reason for this is that there is an expectation that the client maintains state instead of the server. This means that a client can interact with several different servers for a single set of requests (e.g., if one server gets overloaded requests can be transparently migrated).

As a concrete example, if a client is trying to undertake a complex multi-step operation, it is the client's responsibility to keep track of where they are, not the server's responsibility.

Servers are responsible for sending back detailed information about the resources they are returning so that clients have the ability to make further requests of the server with the correct URIs. For example, while could might expect the GitHub `POST /user/:repo` endpoint to simply return `{success: true}` if a repository was successfully created, it instead returns a wide variety of information that can be used by the user to construct any subsequent requests:

```
Status: 201 Created
Location: https://api.github.com/repos/octocat/Hello-World
{
  "id": 1296269,
  "owner": {
    "login": "octocat",
    "id": 1,
    "avatar_url": "https://github.com/images/error/octocat_happy.g
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat",
    "html_url": "https://github.com/octocat",
    "followers_url": "https://api.github.com/users/octocat/followe
    "following_url": "https://api.github.com/users/octocat/followi
    "gists_url": "https://api.github.com/users/octocat/gists{/gist
    "starred_url": "https://api.github.com/users/octocat/starred{/
    "subscriptions_url": "https://api.github.com/users/octocat/sub
    "organizations_url": "https://api.github.com/users/octocat/org
    "repos_url": "https://api.github.com/users/octocat/repos",
    "events_url": "https://api.github.com/users/octocat/events{/pr
    "received_events_url": "https://api.github.com/users/octocat/r
    "type": "User",
    "site_admin": false
  },
  "name": "Hello-World",
  "full_name": "octocat/Hello-World",
  "description": "This your first repo!",
  "private": false,
  "fork": true,
  "url": "https://api.github.com/repos/octocat/Hello-World",
  "html_url": "https://github.com/octocat/Hello-World",
  "archive_url": "http://api.github.com/repos/octocat/Hello-World/
  "assignees_url": "http://api.github.com/repos/octocat/Hello-Worl
  "blobs_url": "http://api.github.com/repos/octocat/Hello-World/gi
  "branches_url": "http://api.github.com/repos/octocat/Hello-World
  "clone_url": "https://github.com/octocat/Hello-World.git",
  "collaborators_url": "http://api.github.com/repos/octocat/Hello-
  "comments_url": "http://api.github.com/repos/octocat/Hello-World
  "commits_url": "http://api.github.com/repos/octocat/Hello-World/
  "compare_url": "http://api.github.com/repos/octocat/Hello-World/
  "contents_url": "http://api.github.com/repos/octocat/Hello-World
  "contributors_url": "http://api.github.com/repos/octocat/Hello-W
  "deployments_url": "http://api.github.com/repos/octocat/Hello-Wo
  "downloads_url": "http://api.github.com/repos/octocat/Hello-Worl
  "events_url": "http://api.github.com/repos/octocat/Hello-World/e
  "forks_url": "http://api.github.com/repos/octocat/Hello-World/fo
  "git_commits_url": "http://api.github.com/repos/octocat/Hello-Wo
  "git_refs_url": "http://api.github.com/repos/octocat/Hello-World
  "git_tags_url": "http://api.github.com/repos/octocat/Hello-World
  "git_url": "git:github.com/octocat/Hello-World.git",
  "hooks_url": "http://api.github.com/repos/octocat/Hello-World/ho
  "issue_comment_url": "http://api.github.com/repos/octocat/Hello-
  "issue_events_url": "http://api.github.com/repos/octocat/Hello-W
  "issues_url": "http://api.github.com/repos/octocat/Hello-World/i
  "keys_url": "http://api.github.com/repos/octocat/Hello-World/key
  "labels_url": "http://api.github.com/repos/octocat/Hello-World/l
  "languages_url": "http://api.github.com/repos/octocat/Hello-Worl
  "merges_url": "http://api.github.com/repos/octocat/Hello-World/m
  "milestones_url": "http://api.github.com/repos/octocat/Hello-Wor
  "mirror_url": "git:git.example.com/octocat/Hello-World",
  "notifications_url": "http://api.github.com/repos/octocat/Hello-
  "pulls_url": "http://api.github.com/repos/octocat/Hello-World/pu
  "releases_url": "http://api.github.com/repos/octocat/Hello-World
  "ssh_url": "git@github.com:octocat/Hello-World.git",
  "stargazers_url": "http://api.github.com/repos/octocat/Hello-Wor
  "statuses_url": "http://api.github.com/repos/octocat/Hello-World
  "subscribers_url": "http://api.github.com/repos/octocat/Hello-Wo
  "subscription_url": "http://api.github.com/repos/octocat/Hello-W
  "svn_url": "https://svn.github.com/octocat/Hello-World",
  "tags_url": "http://api.github.com/repos/octocat/Hello-World/tag
  "teams_url": "http://api.github.com/repos/octocat/Hello-World/te
  "trees_url": "http://api.github.com/repos/octocat/Hello-World/gi
  "homepage": "https://github.com",
  "language": null,
  "forks_count": 9,
  "stargazers_count": 80,
  "watchers_count": 80,
  "size": 108,
  "default_branch": "master",
  "open_issues_count": 0,
  "has_issues": true,
  "has_wiki": true,
  "has_pages": false,
  "has_downloads": true,
  "pushed_at": "2011-01-26T19:06:43Z",
  "created_at": "2011-01-26T19:01:12Z",
  "updated_at": "2011-01-26T19:14:43Z",
  "permissions": {
    "admin": false,
    "push": false,
    "pull": true
  }
}
```

Another example of this is in terms of pagination. Instead of expecting there to be a REST endpoint called `GET /:query/next` which would require the server to keep track of the next result for every client, in REST-based systems a more appropriate endpoint would be `GET /:query/:rowNum` where the client would instead track which row they wanted to see next.

Design goals

There are several orthogonal ideas to keep in mind when designing REST APIs.

The first is that the API must provide a *valuable* service. Without this value, the API will not be widely used, no matter how well it is designed. Value can be derived in many ways be it through access to data (like a weather API), access to a piece of functionality (like a machine learning API), or access to an audience (like a social network API).

APIs should also be sufficiently *flexible* to work in a wide variety of circumstances. This might mean providing multiple data formats, supporting different protocols, or versions. Additionally, enabling batch or update operations can also ease common developer tasks that might otherwise be hard to perform. This often involves considering the bootstrap process and how long it takes a developer from visiting an API document to successfully making a query (often referred to as the time-to-hello-world).

Effectively *measuring* your APIs is also important for understanding how your customers are using your system. When an API is servicing hundreds of thousands of requests per day, it is important to understand how your endpoints are being used and how they are generating errors. These measurements are useful both to the API developer to monitor the API, but also for API clients so they can check on the API if they are having any difficulties. Many API platforms have status pages where you can check how the platform is performing.

Finally, effective API *documentation* is key for helping developers understand how to use any API. These documents should clearly describe what the API does, what parameters it expects, and the structure of its return document. Concrete examples showing how the API can be used are also extremely helpful in this context. Many great examples can be found online demonstrating effective REST API documentation.
