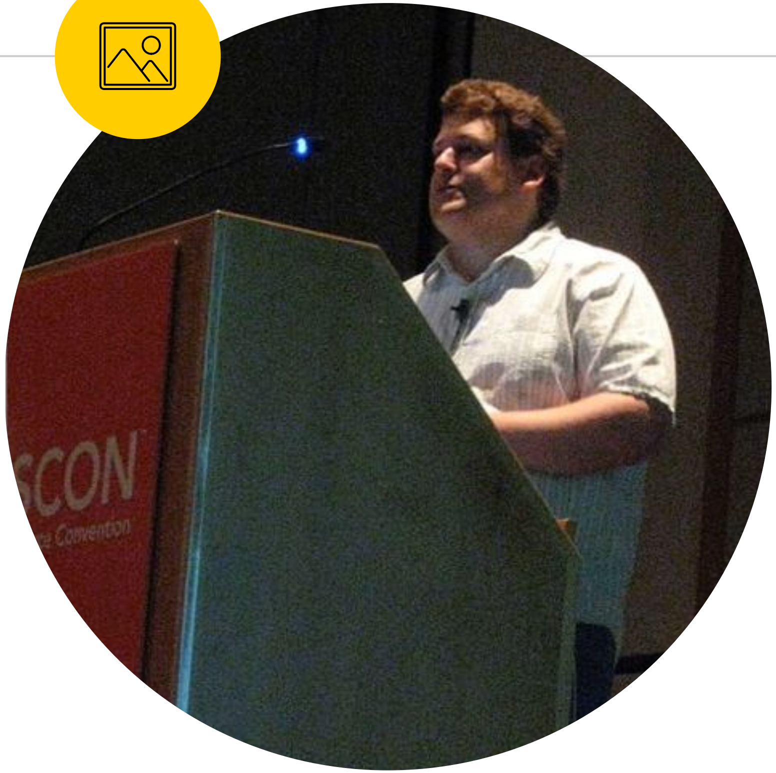
SENG 365 Week 4

REST, APIs, GraphQL

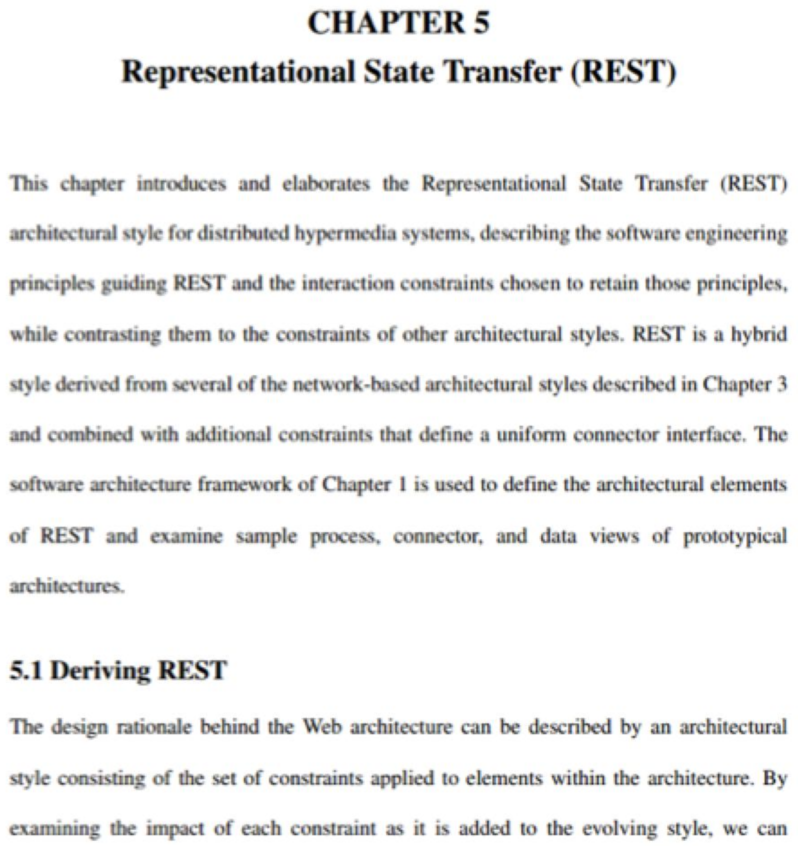
This week

- RESTful APIs
- Managing state in web applications
- API versioning



REST:
Representational State Transfer

Developed by Roy Fielding



CHAPTER 5
Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can

https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (2000)

REST and HTTP Methods

REST asks developers to use HTTP methods explicitly and consistently with the HTTP protocol definition.

HTTP

- HTTP is a **stateless protocol** originally designed for document retrieval
  - HTTP requests and responses are self-contained
  - Not dependent on previous requests and responses
- HTTP **common commands**
  - GET – retrieve a named resource (shouldn’t alter visible server state)
  - HEAD – like GET but only gets headers
  - POST – submits data to the server, usually from an HTML form
- Some **less common** commands
  - OPTIONS – get supported methods for given resource
  - POST – create a new resource
  - PUT – submit (changes to) a specified resource
  - DELETE – delete a specified resource
  - PATCH – modify an existing resource

Idempotence and safe methods

- **Safe** methods are HTTP methods that do not modify resources
  - Like read-only methods
- **Idempotent** methods: the *intended state* (e.g. data changed) is the same whether the method is called once or many times
  - Depends on the current state of the system
- Clarification
  - This is about best practice & standards
  - You can break these standards, but shouldn't
  - Concerns the server, not the client

HTTP method	Should be safe	Should be Idempotent
GET	Yes	Yes
HEAD	Yes	Yes
PUT (GET first)	No	Yes
POST	No	No
DELETE	No	Yes
PATCH	No	No

Why does idempotency matter? Side-effects

"Idempotency and safety (nullipotentcy) are **guarantees** that **server applications make to their clients** … An operation **doesn't automatically become idempotent** or safe just because it is invoked using the GET method, if it isn't **implemented** in an idempotent manner.

A poorly written server application might use GET methods to update a record in the database or to send a message to a friend … **This is a really, really bad design.**

Adhering to the idempotency and safety contract helps make an API fault-tolerant and robust."

https://codeahoy.com/2016/06/30/idempotent-and-safe-http-methods-why-do-they-matter/

CRUD and REST

- To **Create** a resource on the server, you should use POST
- To **Retrieve** a resource, you should use GET
- To change the state of a resource or to **Update** it, … use PUT
- To remove or **Delete** a resource, you should use DELETE

Bad practice

- You don’t have to map CRUD operations to HTTP methods.
- Here’s a bad practice:

```
GET /path/user?userid=1&action=delete
```

- What’s this HTTP request doing?

How good are these examples?

1

```
GET /adduser?name=Robert HTTP/1.1
```

2

```
POST /users HTTP/1.1
Host: myserver
Content-Type:
application/xml
<?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

3

```
POST /users HTTP/1.1
Host: myserver
Content-Type:
application/json
{
        “name”: “Robert”
}
```

CRUD and REST cont.

- For a partial change, use PATCH (like diff)
- Include only elements that are changing
- Null (value) to delete an element
- See also “best practices”
  - http://51elliot.blogspot.co.nz/2014/05/rest-api-best-practices-3-partial.html
- JSON merge patch
  - https://tools.ietf.org/html/rfc7386

A REST service is

- Platform-independent
  - Server is Unix, client is a Mac…
- Language-independent
  - C# can talk to Java, etc.
- Standards-based (runs *on top of* HTTP), and
- Can easily be used in the presence of firewalls

A REST service is

- RESTful systems typically, but not always:
  - communicate over the Hypertext Transfer Protocol (HTTP)
  - with the same HTTP verbs (`GET`, `POST`, `PUT`, `DELETE`, etc.) used by web browsers
  - to retrieve web pages and send data to remote servers.
- RESTful apps are (a subset of) web apps

(HTTP | REST) resources

- Nouns not verbs (because resources are things not actions)
- Instances or Collections
- Resource instance typically identified by :id
  - Integer
    - sequential numbering issues, reuse of ids (generally bad idea)
    - some systems have troubles with long integers (>53 bits in JavaScript)
  - UUID
- Hypertext As The Engine Of Application State (HATEOS)
  - http://restcookbook.com/Basics/hateoas/

REST features

- REST offers **no built-in security features**, encryption, session management, QoS guarantees, etc.
- These can be added by building on top of HTTP
- For encryption, REST can be used on top of HTTPS (secure sockets)

REST vs SOAP

- REST displaced SOAP, because…
- … REST is considerably easier to use
  - e.g. SOAP has a ‘heavy’ infrastructure
- … works nicely with AJAX / XHR
  - e.g. XML is verbose; JSON is more concise
- … has some network advantages
  - accepted through firewalls

Stateless requests

- A complete, independent request doesn't require the server to retrieve any kind of application context or state.
- A RESTful Web service application includes within the HTTP headers and body of a request: all of the parameters, context, and data needed by the server-side component to generate a response.
- The entire resource is returned, not a part of it.
- Statelessness:
  - Improves Web service performance
  - Simplifies the design and implementation of server-side components…
  - because the absence of state on the server removes the need to synchronize session data with an external application.

RESTful URLs

- **Hide the server-side scripting technology** file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs
  - Hide all implementation details!
- **Be consistent** in the singularity / plurality of resource names
  - Use **singular** or **plural**, but don't mix them
  - /student & /course; not /student and /courses
- Keep everything **lowercase**
- Substitute **spaces for hyphens**
- Instead of using the 404/Not Found code if the request URI is for a partial path, always provide a **default page or resource** as a response

Issues with REST

- Tightly coupled to HTTP
- Request-response
  - Can’t push/alert or broadcast
- Multiple request-responses needed
  - Implied tree-structure
  - Underfetching and overfetching
  - Latency increases for full set of request-response

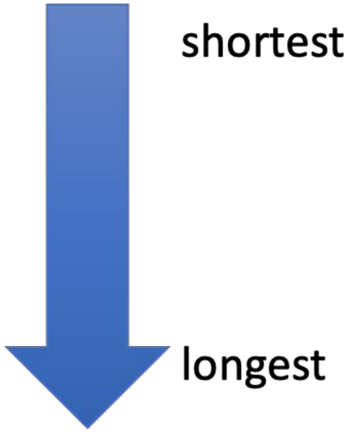
State and Statelessness

Mini-overview

- State timescales
- Session (state) information
- GET parameters e.g. after the ?
- HTTP bodies & cookies
  - Types
  - Sequence
  - Limitations
  - Session IDs

State timescales

- Individual HTTP request (stateless)
- Business transaction
- Session
- Preferences
- Record state



Session state information

- For web applications (in contrast to public web pages) there is a need to **maintain stateful information** about the client
- The application (client & server) needs to maintain consistency on **which client** (authentication) and **what actions** (authorisation)
- Longer-lived transactions
  - Anything that's built up of **multiple HTTP calls** (stateless)
  - May be stored on server, or on client (or synchronized)

GET ? parameters

- Maintain some kind of **session variable** in the **parameter** to the HTTP request
- Variable does not contain the username and password, but a unique (‘random’) identifier
- Include variable as a parameter in each network request e.g.,

```
GET www.example.com?sessionid=<var>
```

- Why is this ‘bad practice’?
- Why may something like this be needed, at times?

Cookies

- Use cookies to maintain session information
- The server issues a unique (‘random’) identifier in the cookie to the client, for that username & password
- Client sends back the cookie with **each** network request to the server
- e.g. in the POST data
- Note that the username and password are not sent (once the user is logged on)

Cookies

- A **small piece of data** initially sent by the server to the client.
  - Comprises **name-value pairs**
  - Also has **attributes** (that are not sent back to the server)
  - **Expiry 'date'** (duration)
- Used to maintain state information
  - e.g. items in a shopping basket
    (although this example may be better kept on the server)
  - e.g. browser activity such as a 'path' through a registration process

Types of Cookies

First-party cookie	A cookie set by the server to which the browser primarily connects.
Session cookie	Exists only for the duration of that browser session, and the browser typically deletes the cookie
Persistent cookie (aka tracking cookie)	Persistent data. The cookie is not deleted when the browser closes. Can be used by advertising to track user behaviour. Can be used to store credentials e.g. log in details.
Secure cookie	A cookie that can only be transmitted over an encrypted connection, such as HTTPS.
HTTPOnly cookie	Can only be transmitted through HTTP/S, and are not accessible through non-HTTP APIs such as JavaScript.
Third-party cookie	Cookies set by third-parties that serve content to the page e.g. advertising.

Cookie sequence

- Request from browser

```
GET /index.htm HTTP/1.1
Host: www.example.com
```

- Response from server

```
HTTP/1.1 200 OK
Content-type: text/html
Set-cookie: sessionToken=a1b2c3; Expires = [dat]
```

- Follow-up request from browser

```
GET /profile.htm HTTP/1.1
Host: www.example.com
Cookie: sessionToken=a1b2c3
```

Limitations of cookies

- Each browser maintains its own ‘cookie jar’
- A cookie does not identify a person
- A cookie identifies the combination of:
    - User account
    - Web browser
    - Device
- A cookie requires that the browser is cookie-enabled and is set to allow cookies

API Versioning

API versioning: overview

- Compatibility between API provider and API consumer
- Semantic versioning
- Specifying API versions in HTTP requests (and handling of those)
- JSON
- Publishing APIs

Interesting (though a bit dated) discussion at SO:

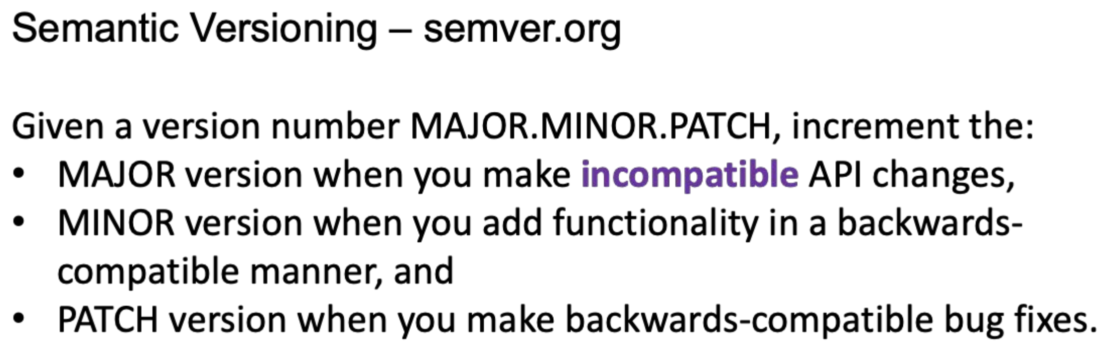
https://stackoverflow.com/questions/389169/best-practices-for-api-versioning

Why version APIs?

- Maintain compatibility of APIs with API consumers, as APIs change
- Add / amend functionality/services
- Performance improvements e.g., reduce the number of HTTP requests
- Market testing of ideas e.g., A/B testing
- System testing e.g. dev, test, prod
- Subsets of clients, e.g.,
  - B2B vs B2C
  - Geographical / political region
- Rollback of (unacceptable) API
- From client perspective:
  - API is backward compatible if client can continue through service changes
  - Forward compatible if client can be changed without needing service change

Semantic versioning: `MAJOR.MINOR.PATCH`

1. Caching - want different versions cached differently
2. Number or name?
   - If number, what format? Semver? Counter?



Semantic Versioning – semver.org

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make **incompatible** API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

Specifying API version in HTTP request

- Query parameter
  - `?v=xx.xx` or `?version=xx.xx` or `?Version=2015-10-01`
  - e.g., Amazon, NetFlix
- URI
  - `api/v1/`
  - e.g. Facebook: https://graph.facebook.com/v2.2/me/adaccounts
  - Semantically messy (implies version refers to version of object)
- Header
  - Accept header – hard to test – can't just click on link or type URL
  - Custom request header – duplicates Accept header function
  - E.g., GitHub: https://developer.github.com/v3/media/

Publishing an API

- **Documentation** – current, accurate, easy, guide/tutorial/directed (management tool generated)
- **Direct access** (no SDK required)
  - e.g., through Postman or curl (say, `curl -L http://127.0.0.1:4001/v2/keys/message-XPUT -d value="Hello world"`)
- SDKs/Samples in **developer preferred languages**
  - Any SDK is just libraries to access REST/SOAP API, nothing more. Potentially an impediment to simply making use of the straight API.
  - Straightforward install and use
- **Free/Freemium** use for developers
- **Instant API keys**
- **Simple sandbox** to try things out for developers
- Before API available, establish **API landing page** on web to discover interest and potential user types