

API DESIGN, AND IMPLEMENTATION

1. Review
2. Request/response roundtrip
3. RESTful APIs
4. Designing a REST API
5. Assignment 1 (and 2)
6. Workshop

Types of web service

Message based e.g. SOAP

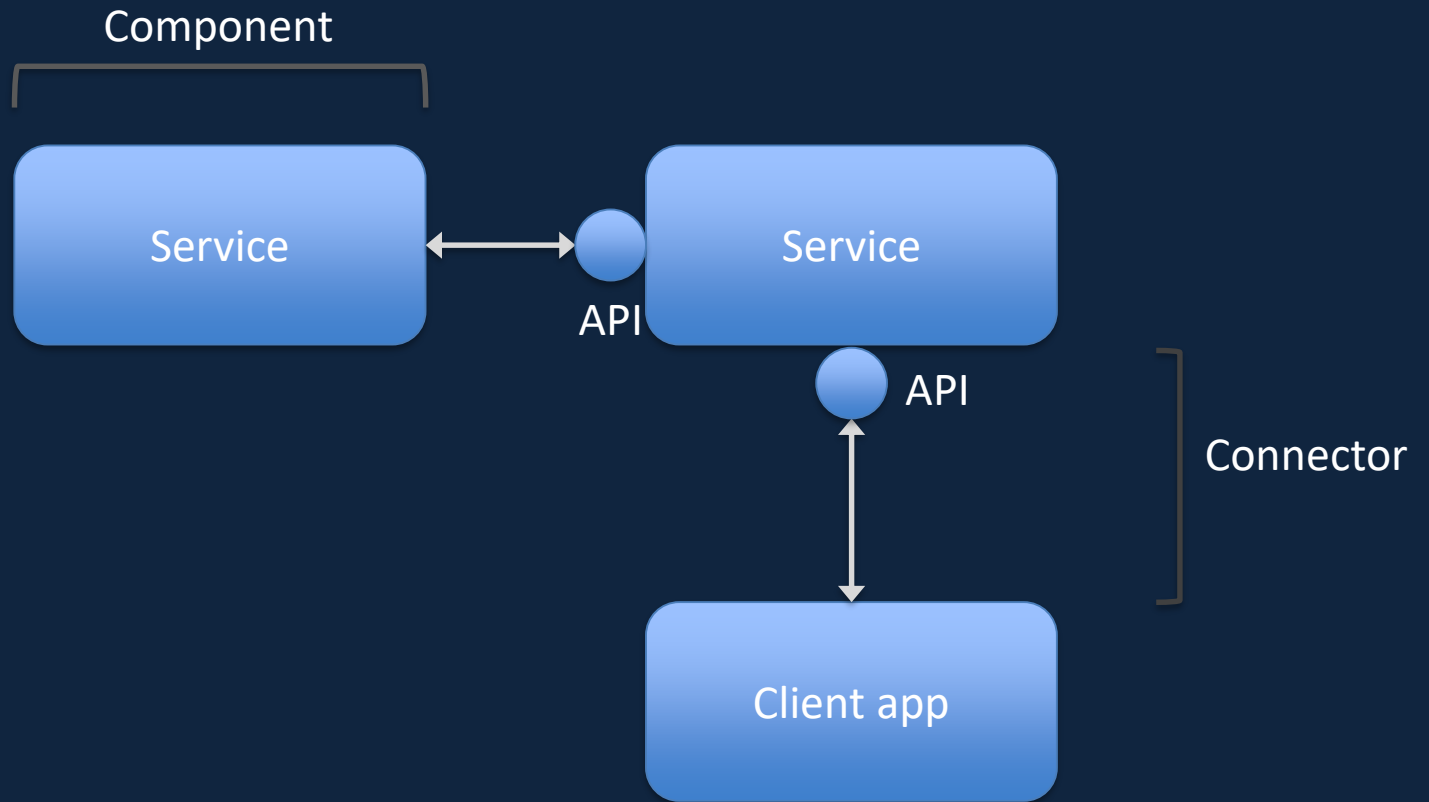
- Messages independent of method signatures
- Represent logical tasks, not implementation
- Service inspects message, chooses appropriate handler

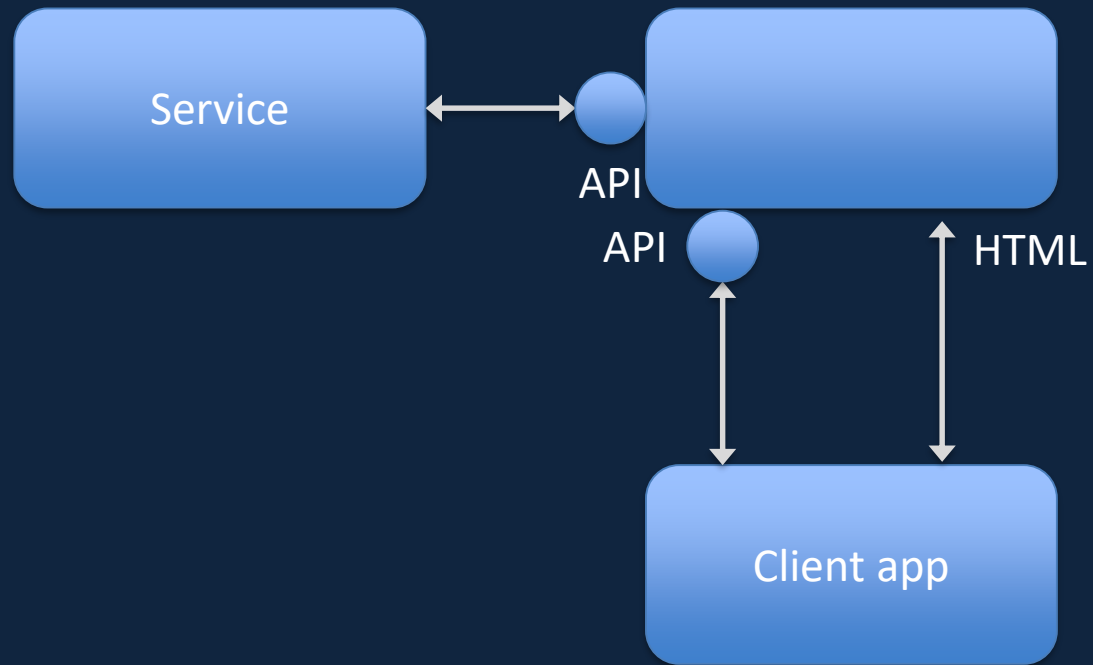
Resource based e.g. RESTFUL

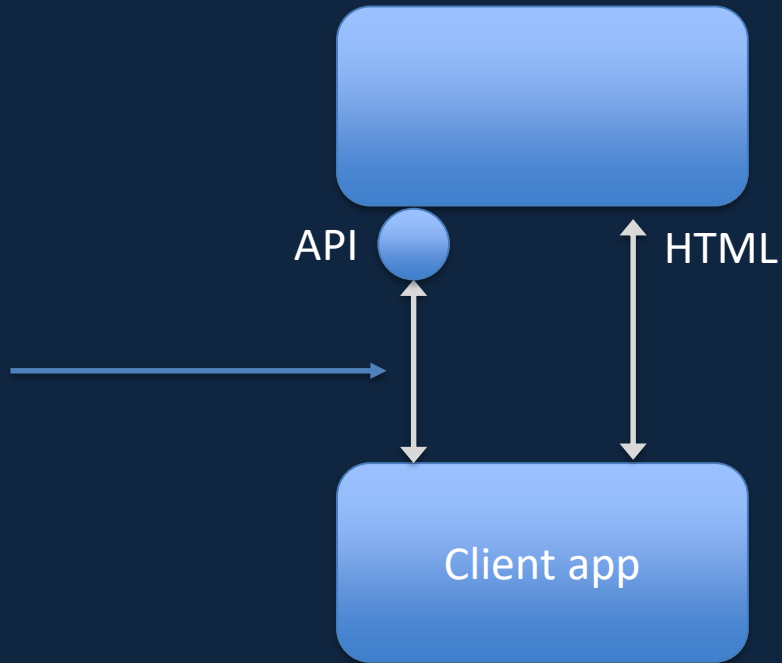
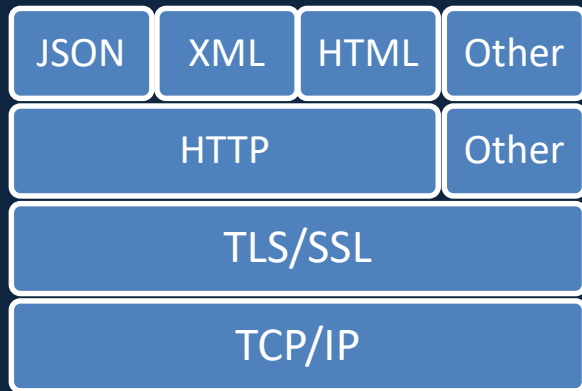
- Typically uses HTTP & standard HTTP methods GET, PUT, POST, DELETE
- Represent and manipulate data identified by URI
- Employs representational state transfer (REST)

Procedure based (RPC = Remote Procedure Call)

- Expose selected methods as service
- Messages according to method signatures
- Method call







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 are:

- 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, but sometimes used in RESTful web services

- OPTIONS – get supported methods for given resource
- PUT – submit (changes to) a specified resource
- DELETE – delete a specified resource
- PATCH – modify an existing resource

Safe vs Idempotent ($n > 0$ requests same as $n = 1$)

Example HTTP requests

HTTP requests are of form:

```
Method SP Request-URI SP HTTP-Version CRLF
*(Header CRLF)
CRLF
Request Body
```

Typical GET (no body):

```
GET /pub/blah.html HTTP/1.1
Host: www.w3.org
```

Typical POST:

```
POST /pub/blah2.php HTTP/1.1
Host: www.w3.org
```

Body of post (e.g. form fields)

KEY:

SP = space
CRLF = carriage return,
line feed (\r\n)

Example HTTP responses

HTTP responses are of form

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
* (Header CRLF)
CRLF
Response Body
```

Typical successful response (GET or POST):

```
HTTP/1.1 200 OK
Date: Mon, 04 Jul 2011 06:00:01 GMT
Server: Apache
Accept-Ranges: bytes
Content-Length: 1240
Connection: close
Content-Type: text/html; charset=UTF-8

<Actual HTML>
```

Response codes

1xx, informational (rare)

- e.g. 100 continue

2xx, success

- e.g. 200 OK, 204 No Content

3xx, redirections

- e.g. 303 See Other, 304 Not Modified

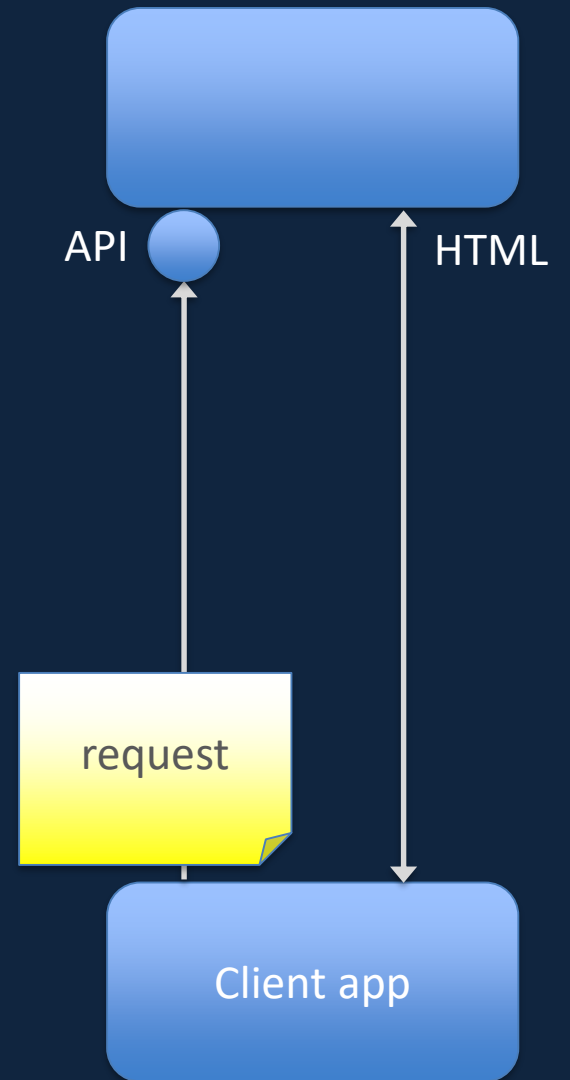
4xx, client error (lots of these)

- e.g. 400 Bad Request, 404 Not Found

5xx, server error

- e.g. 500 Internal Server Error, 501 Not Implemented

HTTP/1.1 and /2: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
<http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>



1. URI (Uniform Resource Identifier)

String of characters to identify (name, or name and location) resource

2. URL (Uniform Resource Locator)

- A URI that also specifies means of acting upon, or obtaining representation
- That is, URI with access mechanism and location

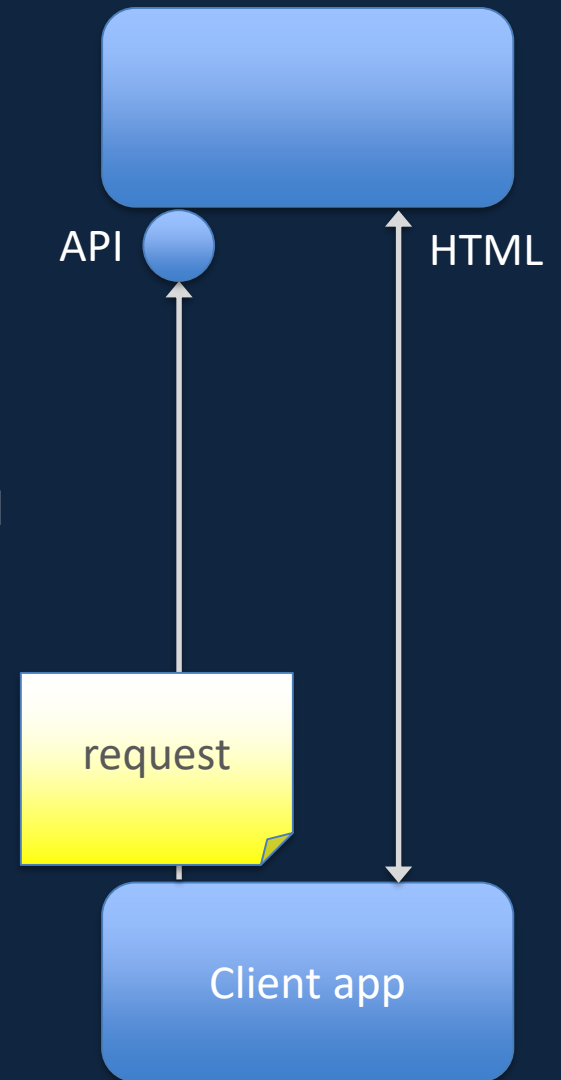
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]

80 is default, 443 for https

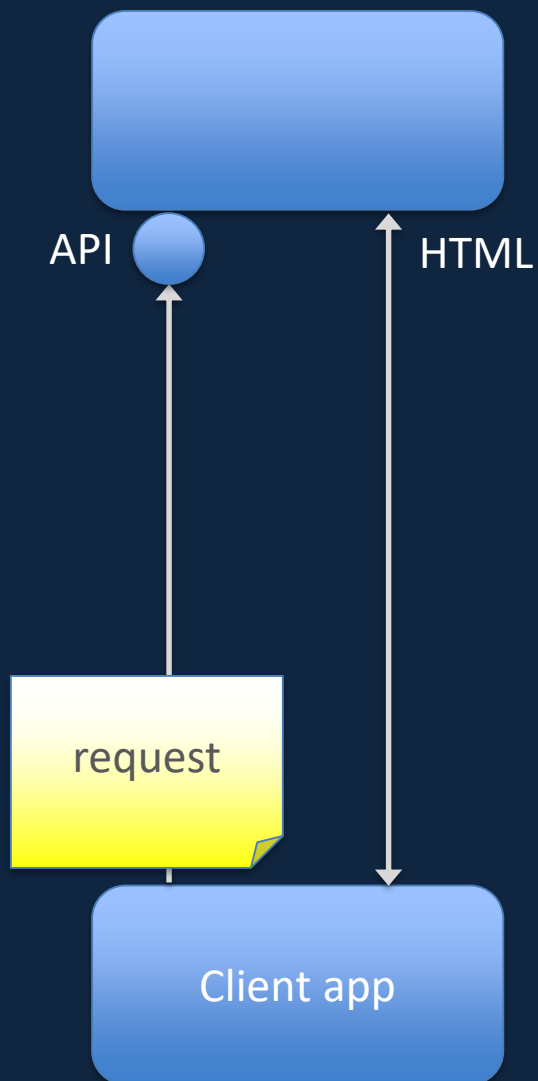
scheme://[user[:password]@]host[:port]][/path][?query][#fragment]

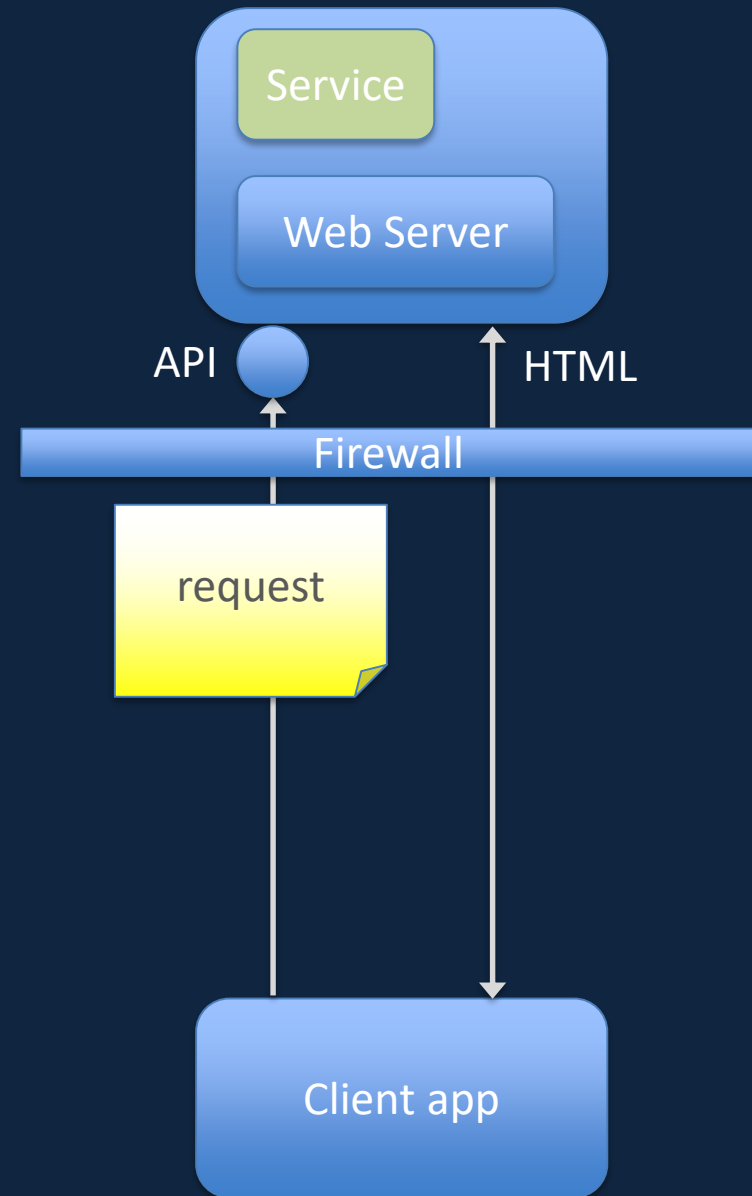
http or https

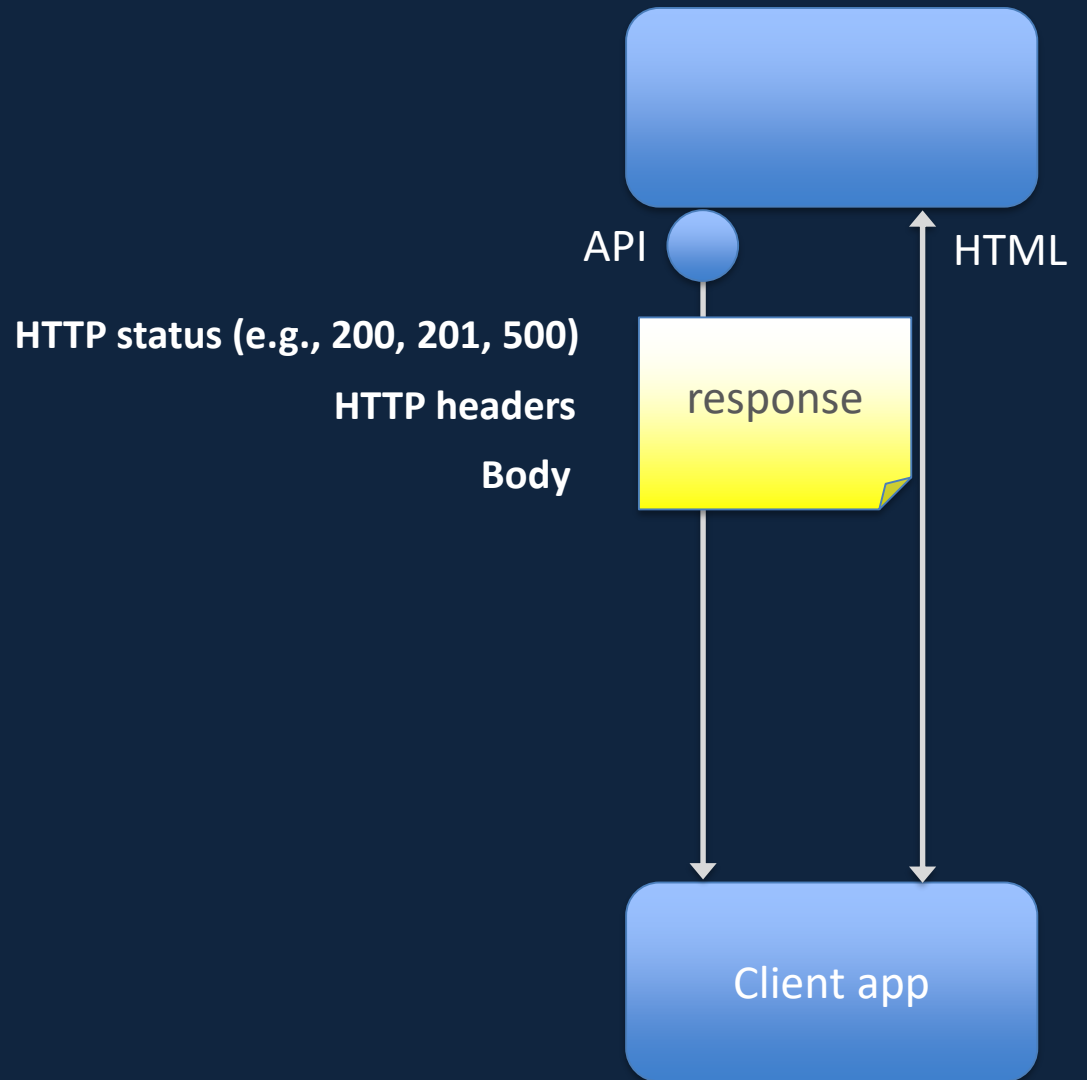
/basepath/endpoint
/basepath/resource

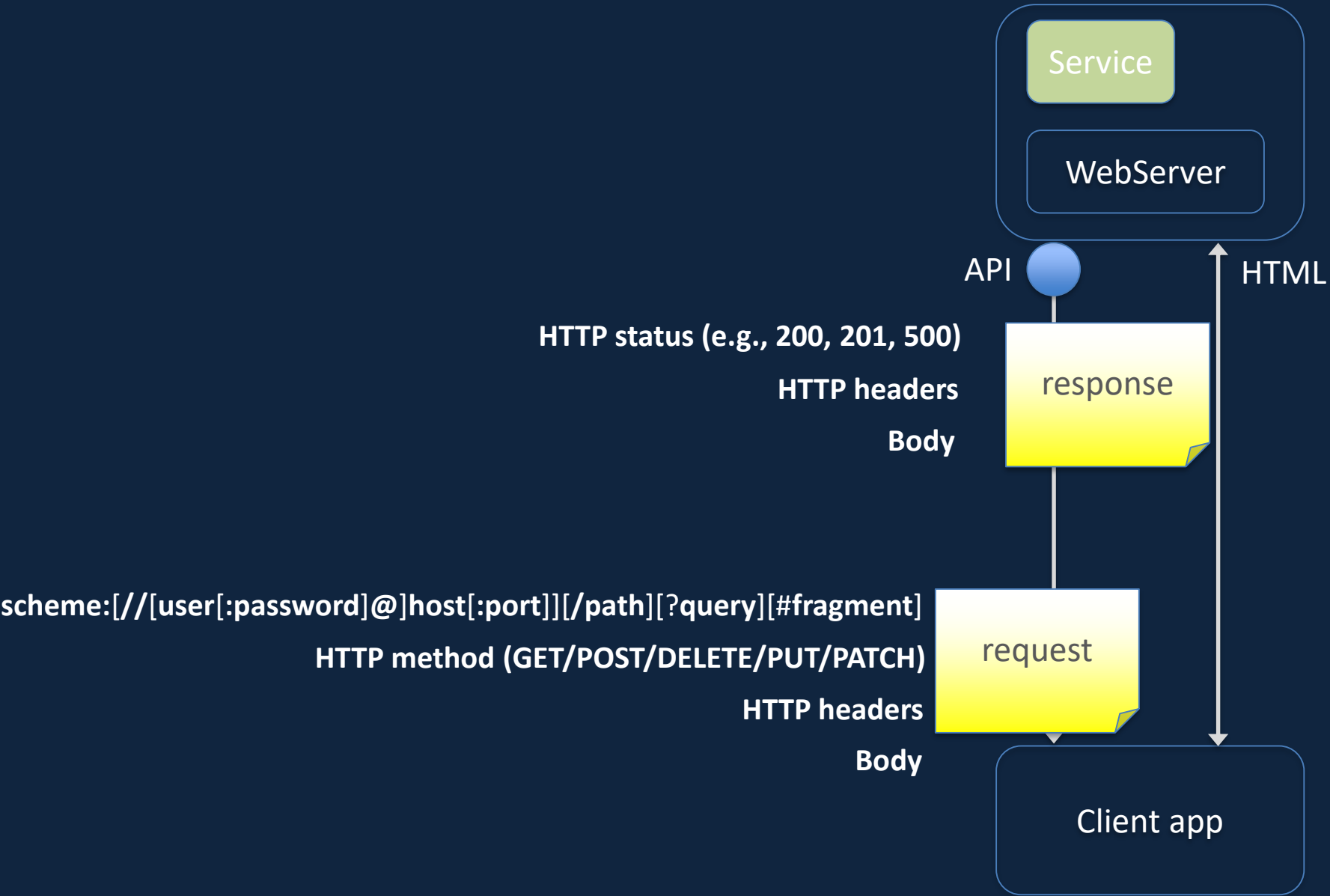


scheme://[user[:password]@]host[:port]][/path][?query][#fragment]
HTTP method (GET/POST/DELETE/PUT/PATCH)
HTTP headers
Body









REST

REPRESENTATIONAL STATE TRANSFER



https://commons.wikimedia.org/wiki/File:Roy_Fielding.jpg

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

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

(*REST \neq HTTP)

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

Resources

Nouns not verbs

Instances or Collections

Resource instance typically identified by :id

- Integer
- UUID
- see HATEOAS

CRUD and REST

To **C**reate a resource on the server, use POST

To **R**etrieve a resource, use GET

To change the state of a resource or to **U**update it, use PUT

To remove or **D**delete a resource, use DELETE

CRUD and REST

To Create a resource on the server, use POST

To Retrieve a resource, use GET

To change the state of a whole resource or to Update it, use PUT

For a partial change, use PATCH (like diff)

- Include only elements that are changing
- Null to delete an element
- <http://51elliot.blogspot.co.nz/2014/05/rest-api-best-practices-3-partial.html> and
- <https://tools.ietf.org/html/rfc7386>

To remove or Delete a resource, use DELETE

REST and HTTP methods

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

Here's an example of bad practice:

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

Why is this bad practice?

Bad, bad, baaaad ☹️

GET /adduser? name=Robert HTTP/1.1

Good-ish 😊

POST /users HTTP/1.1

Host: myserver

Content-Type: application/xml

<?xml version="1.0"?>

<user>

 <name>Robert</name>

</user>

Good 😊

POST /users HTTP/1.1

Host: myserver

Content-Type: application/json

{

 "name": "Robert"

}

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)

- For security, see next week

REST has 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, while processing the request, to retrieve any kind of application context or state.

A RESTful Web service application (or client) 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.

- Design resources carefully

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 URIs

- Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs – if fact, hide all implementation details!
- Singular resource names rather than plurals...or plural rather than singular (there is some debate) – be consistent
- Keep everything lowercase
- Substitute spaces 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

REST issues

1. Tightly coupled to HTTP
2. Request-response
3. Implied tree-structure

HATEOAS

Hypermedia As The Engine Of Application State

Hay-tee-os

Hay-dee-os

Hate-O-A-S

ha-TAY-oh-ahss *

HATEOAS

Hypermedia As The Engine Of Application State

Resource identification by URIs

- Manipulation through their representations
- Hypermedia/links
 - “href” in JSON for id – link to resource
- Self-descriptive

HATEOAS

<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

<http://tools.ietf.org/html/draft-nottingham-json-home-03> - draft standard/guidelines


<http://jsonapi.org/> - semi-standard for responses, doesn't focus on states

<http://intercoolerjs.org/2016/01/18/rescuing-rest.html>

<https://keyholesoftware.com/2016/02/29/dont-hate-the-hateoas/>

[Hypermedia APIs as FSMs](#)

Stefan Tilkov's [presentation](#) to BuildStuff 2014



Responses are the
links in a FSM

[Facebook Graph API](#) – can explore responses with [Graph API Explorer](#)

[Twitter GET Users/Show](#)

[PayPal payments](#)

(Netflix – no longer public)

API documentation

Automated/discoverable vs manual

- Swagger to OpenAPI Specification (OAS)
- RAML
- API Blueprint
- Markdown

Service discovery (for your developers...!)

- What is there
- How does it work
- HumaneRegistry as used by Soundcloud

Testing

Must be automated!

Tools

- [Runscope](#)
- [Dredd](#) (CI testing of docs in API Blueprint against implementation)

Test generation

- e.g., <https://github.com/apigee-127/swagger-test-templates>

Examples

- GitLab API - <https://docs.gitlab.com/ce/api/>
- Twitter API - <https://dev.twitter.com/rest/public>
- Swagger – <https://swagger.io/>
- PayPal payments -
<https://developer.paypal.com/docs/api/hateoas-links/>
- ProgrammableWeb -
<https://www.programmableweb.com/>

Next week

- Inside the box – Node.js and ES2015

Prep for next week

- Concurrency vs Parallelism
- Research Javascript closures
 - what are they
 - when are they useful

Further reading

- Course wiki – Foundations of APIs

Assignment 1

1. API workshop today to explore/define
2. We'll consolidate, and provide Swagger
3. You implement the API

Notes are on Learn

Full Assignment will be on Learn

Forum on ... Learn

Designing a RESTful API

1. Describe use cases/stories
2. Identify the resources ("nouns") and actions ("verbs") on those resources
 - Resources become the URIs/endpoints
 - Actions become the HTTP methods
3. Create static JSON responses to confirm/validate
4. Create test suite from definition
5. Build it!
6. Generate documentation (perhaps automatic)
7. Roll it out!

1. Work in groups
2. Feel free to use laptops, phones, semaphore...
3. Read the handout
4. In your groups, work through the questions and record your answers
5. Tape ERD/UML diagram to wall, circulate
6. Document your API, hand it in

Ranges (pagination)

- HTTP range header, or
- Query string

Long-lived operations

- Response code 202/Accepted (instead of 200/OK or 201/Created) with temporary location in response header
- GET temporary location returns up-to-date status
- When completed, GET returns 303 and final location
- (Further GETs give 404)

Versioning...

W3C Versioning

Consumer backward compatibility

- Consumer upgrades
- Producer downgrades - e.g., rollback

•Consumer forward compatibility

- Producer upgrades (or continue to support earlier versions)
- Support a given range of clients
- Assume don't change semantics, just add or subtract

From client perspective:

1. API is backward compatible if client can continue through service changes
2. Forward compatible if client can be changed without needing service change

2.10. Robustness Principle

TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

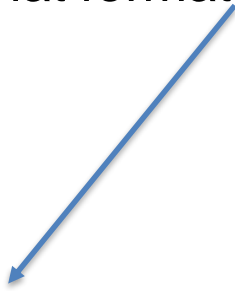
<http://tools.ietf.org/html/rfc761>

1980

MustIgnore pattern

<http://www.martinfowler.com/articles/consumerDrivenContracts.html>

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.

"Facebook's Marketing API now supports both versioning and migrations so that app builders can roll out changes over time. Read on to understand how you are affected by versions, how to use those versions in our Marketing APIs and Ads SDKs, and what migration windows are.

While Facebook's Platform has a core and extended [versioning](#) model, starting Oct 30th, 2014, Facebook's Marketing API will move to a versioning scheme to manage changes in the Marketing API. With Marketing API versioning, all breaking changes will be rolled up into a new version. Multiple versions of Marketing APIs or Ads SDKs can exist at the same time with different functionality in each version.”

<https://developers.facebook.com/docs/marketing-api/versions>

Three approaches to specifying API version in request

- Query parameter
 - ?v=xx.xx, or ?version=xx.xx or ?Version=2015-10-01
 - e.g., Amazon, Netflix
 - URI
 - /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
 - <https://developer.github.com/v3/media/>
 - <https://blog.pivotal.io/labs/labs/api-versioning>
- 

```
curl https://api.github.com/users/technoweenie -I \ -H "Accept: application/vnd.github.v3.full+json"
```

Credibility

“If a developer doesn’t believe you, then good luck getting them to use your product.”

Support

“Developers are looking for signs of support. Something at some point is going to go wrong, so developers want to know they can solve their problems quickly.”

Success

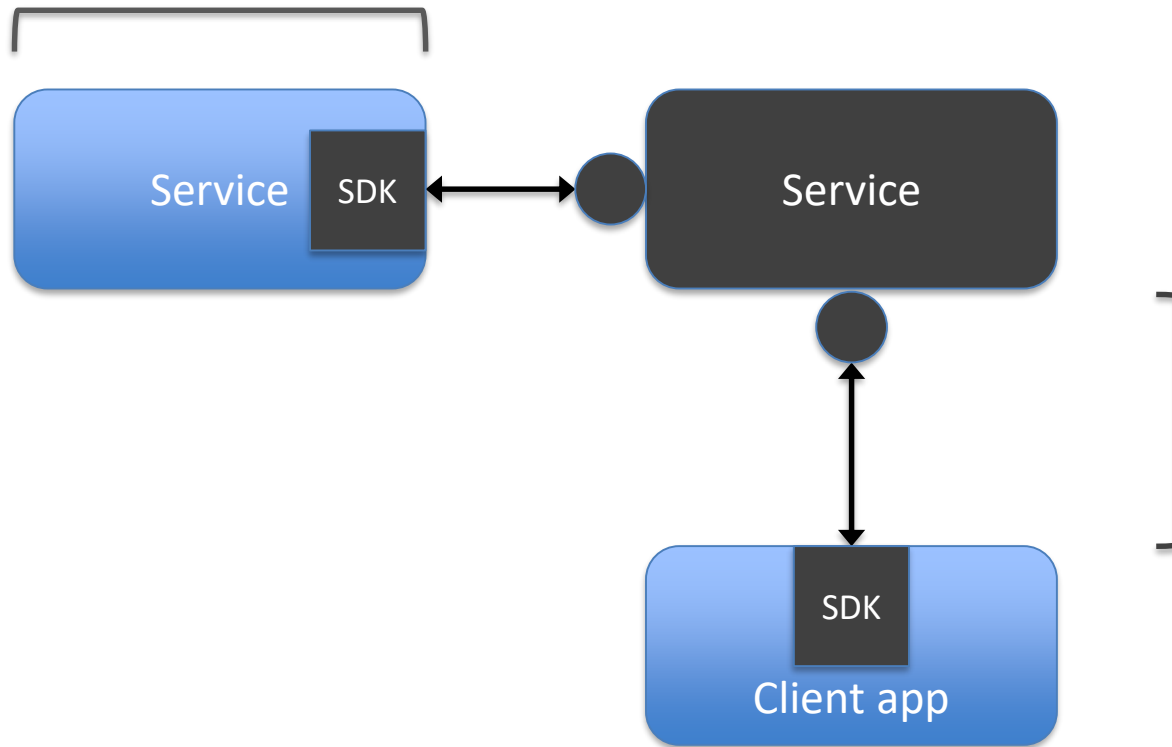
“Look to share details about how your API can help them achieve something. A lot of companies don’t have that understanding that providing an API is going to create a win for both of you.”

<http://www.programmableweb.com/news/how-to-maximize-developer-adoption/analysis/2014/11/17>

- 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

<http://www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/apmu1306.html>

- [Usage limits](#)
- dev.abc.com or developer.abc.com



**SDK – Software development kit
aka client library**