REST

Service-Oriented Architecture

Jeremy Gibbons

Contents

1 Representational state transfer (REST)

2 Architectural styles

3 The REST architectural style

4 Resource-oriented architecture

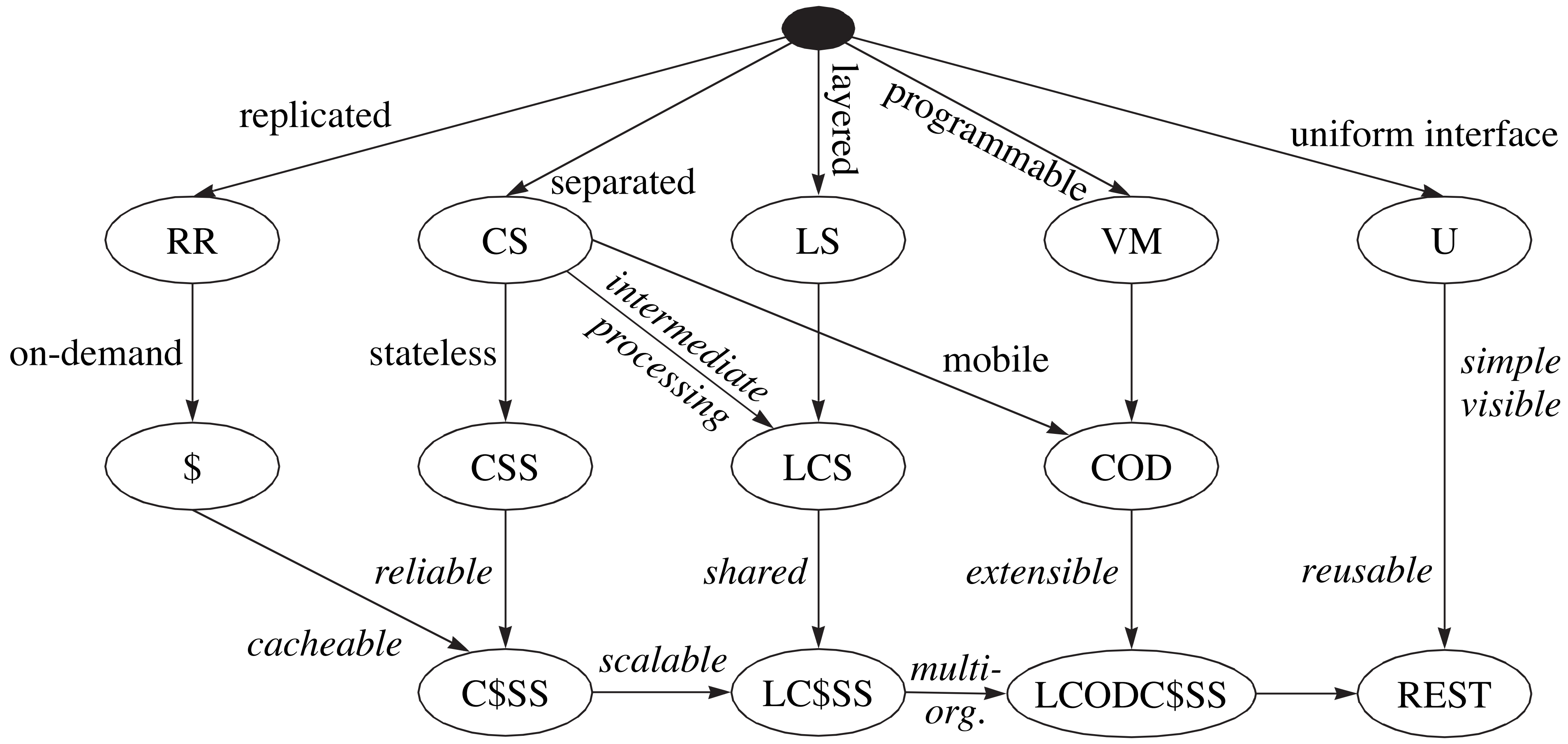
5 Example: social bookmarking

1 Representational state transfer (REST)

- Roy Fielding, a principal author of HTTP
- PhD thesis *Architectural Styles and the Design of Network-based Software Architectures* (2000)

  more about evaluation than a cookbook
- subsequent article *Principled Design of the Modern Web Architecture* (ACM TOIT 2:2, 2002)
- Richardson & Ruby, *RESTful Web Services*
- architectural patterns of the web
- taking HTTP seriously as a distributed computing protocol: fixed few verbs, emphasis on the nouns

2 Architectural styles



2.1 Client–server (CS)

- server offers services, listens for requests
- client sends request, waits for response
- transient, triggering client; persistent, reactive server
- separation of concerns: user interface from behaviour
- improves *portability* to a new user interface
- improves *scalability* by simplifying components
- improves *evolvability* by allowing independent evolution of components

2.2 Replication (RR) and caching ($)

- *replicated repository*: multiple servers provide same service
- present the illusion of a single, centralized service
- improves *performance*: latency, redundancy
- *maintaining consistency* the primary challenge

- a variation: *caching* responses for later reuse
- effectively a replication of a fragment (typically, potential data set is huge or infinite)
- responses explicitly or implicitly labelled cacheable or not
- lazy or active replication
- less effective than full replication, but cheaper and simpler

2.3 Stateless (S)

- each request from client must carry all necessary context
- no *session state* stored on server — kept entirely on client
- (*resource state* is a different matter)
- improves *visibility* for monitoring
- improves *reliability* by simplifying recovery from partial failure
- improves *scalability* by allowing server to free resources quickly
- improves *evolvability* by simplifying server, cache
- decreases *performance* by increasing overhead

2.4 Layered systems (LS)

- hierarchical arrangement
- layer provides services to layer above, uses services from layer below
- improves *evolvability* and *reusability* through abstraction
- decreases *performance* through overhead, latency

- layered-client-server (LCS) adds proxy and gateway components to CS
- *proxy* acts as shared server for one or more clients, forwarding (maybe translated) requests
- *gateway* appears as normal server, but forwards (maybe translated) requests to lower layers: load balancing, security

2.5 Uniform interface (U)

- improves *simplicity* and *visibility*
- decreases *efficiency* through possible data translations

- for REST, optimized for large-grain hypermedia data transfer
- identification of *resources*
- manipulation of resources through *representations*
- *self-descriptive* messages
- hypermedia as the engine of application *state*

2.6 Virtual machine (VM) and code-on-demand (COD)

- mobile code
- dynamically relocate processing between data source and destination
- improves *performance* by relocating code near data
- data element must be transformed into component

- extend client functionality by downloading applets/scripts
- virtual machine to provide controlled environment
- improves *simplicity* and *extensibility* of client
- reduces *visibility*
- not a big part of REST-based SOA (yet: cf AJAX)

3 The REST architectural style

- ‘uniform, layered, cached-client, stateless-server, with code-on-demand’
- data elements
- connectors
- components
- some good reading at `http://www.prescod.net/rest/`

3.1 Resources

- key idea: every data element is a *resource*
  - named document
  - temporal service (‘today’s lunch menu at Rewley’)
  - collection of other resources
  - non-virtual object (‘Marilyn Monroe’)
  - anything you might want reference, annotate, or act upon
  - abstractly, just a time-dependent set of values
- resources referenced by URIs
- generality
- late binding of reference to representation
- insulation of reference from representation
- quality of identifier proportional to effort spent maintaining validity

3.2 Representations

- state of resource captured and transferred between components
- might be current or desired future state
- represented as data plus metadata (name–value pairs)
- think document+headers, or HTTP message
- metadata includes control data, media type
- one resource might have several representations
- selected via separate URIs, or via *content negotiation*

3.3 Interaction

- abstract interface for component communication
- stateless interactions:
  - connectors need not retain application state between requests
  - interactions can be processed in parallel, naively
  - intermediary may view and understand request in isolation
  - reusability of cached response can be determined from response itself
- request parameters: control data, target URI, optional representation
- response parameters: control data, optional resource metadata, optional representation

3.4 Connectors

- initiating *client*
- reactive *server*
- *cache*, either at client or at server, maybe shared
- *resolver* translates URIs into locations (eg DNS, DOI)
- *tunnel*, relaying communication across a connection boundary

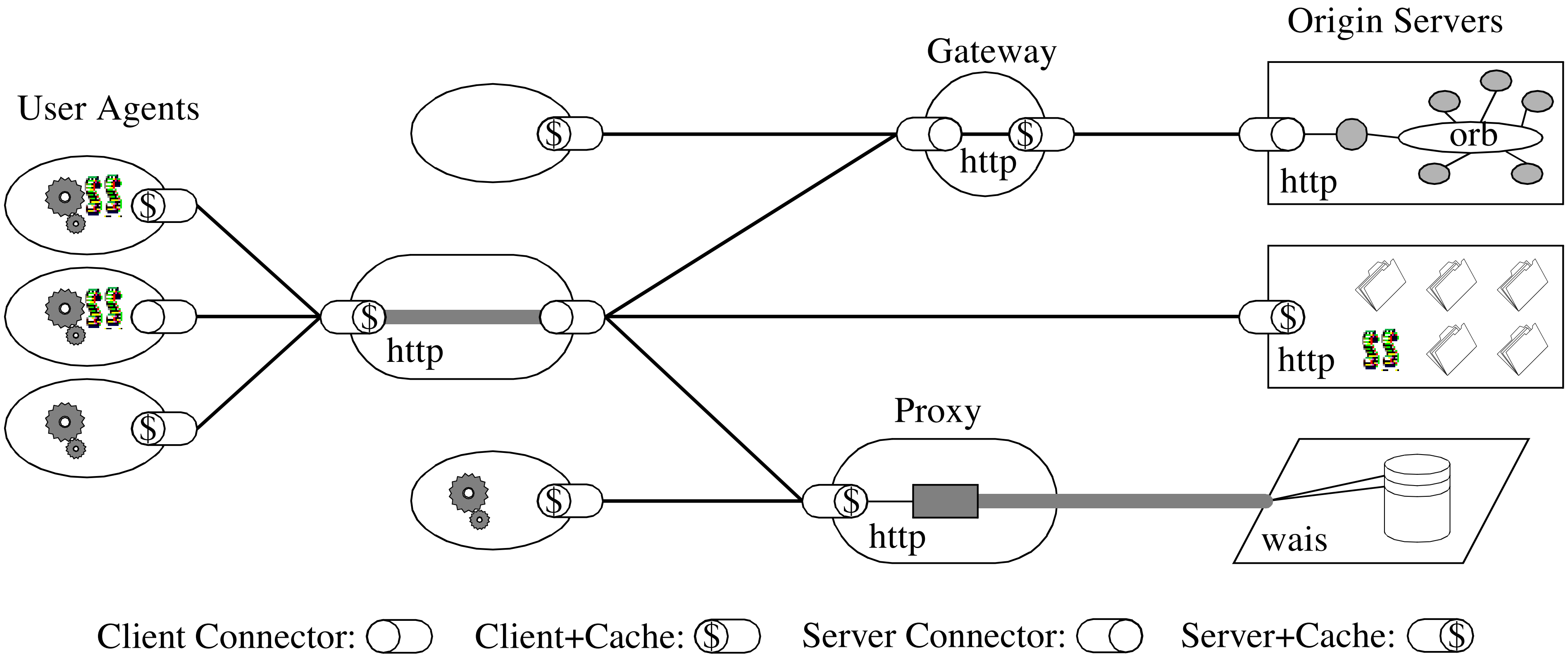
3.5 Components

- *user agent* uses client connector to initiate response (eg browser)
- *origin server* uses server connector to govern namespace for requested resource
- *proxy* and *gateway* act as both client and server

3.6 State transitions

- control state concentrated into resource representations
- embedded resource identifiers indicate possible next states
- allows user to directly manipulate state (eg through browser history, context switching)
- model application is therefore *an engine that moves from state to state by selecting from alternative transitions in current set of representations*

3.7 REST-based architecture



4 Resource-oriented architecture

- after Richardson & Ruby, *RESTful WS*
- action identified in HTTP method, not in payload
- scoping information in URI

```
GET reports/open-bugs HTTP/1.1
```

- in contrast to RPC-style interaction

```
POST /rpc HTTP/1.1
Host: www.upcdatabase.com

<?xml version="1.0">
<methodCall>
  <methodName>lookupUPC</methodName> ...
</methodCall>
```

- . . . or hybrid

```
http://www.flickr.com/services/rest?method=search&tags=cat
```

4.1 RESTful operations

CRUD verb	*HTTP method*	*arguments*
create	PUT	fresh representation
retrieve	GET	
update	PUT	revised representation
delete	DELETE	

4.2 PUT vs POST

- actually, creation by either PUT to new URI or POST to existing URI
- typically, create a subordinate resource with a POST to its parent
- use PUT when client chooses URI; use POST when server chooses
- successful POST returns code 201 ‘Created’ with `Location` header
- (POST also sometimes used for form submission, but this can be non-uniform)

4.3 Designing a ROA (Richardson & Ruby)

- figure out dataset
- split dataset into resources

and for each resource:

- name the resource with URI(s)
- expose (a subset of) the uniform interface
- design representations accepted from client
- design representations served to client
- integrate with other resources, using links
- consider typical course of events
- consider exceptional conditions

4.4 Asynchronous operations

- HTTP is synchronous: request–response
- what about long-running requests? *deferred synchronous* interaction
- client POSTs request (because not idempotent)

  ```
  POST /queue HTTP/1.1
  Host: jobservice.com

  Please tell me whether 2ˆ43,112,609 - 1 is prime.
  ```

- server queues task, returns code 202 'Accepted' with URI:

  ```
  202 Accepted
  Location: http://jobservice.com/queue/job11a4f9
  ```

- client polls resource:

  ```
  GET /queue/job11a4f9 HTTP/1.1
  ```

  getting either status report or result

4.5 URI design

- URIs should be meaningful and well-structured
- client should be able to construct URI to access a resource (increases *surface area*)
- use paths to separate elements of hierarchy, general to specific
- use punctuation to separate items at same hierarchical level
- commas when order matters (eg coordinates), semicolons otherwise
- use query variables only for ‘arguments’
- URIs denote resources, not operations (unless the operation is itself something you might CRUD)
- some (eg TBL) say URIs should be opaque. . . why?

4.6 Representations

- may be human-readable, but should be computer-oriented
- eg collection of values, not picture of graph (unless graphing is the service!)
- need not be XML: plain text, key–value pairs, JSON, . . .
- outgoing representations should be acceptable as incoming too: client may GET, update, PUT
- use return codes, not outgoing representations, for error conditions

4.7 Conditional GET

- save bandwidth by not resending unchanged representations
- requires collaboration between client and server
- server sends `Last-Modified` header with representation
- client requests `If-Modified-Since`
- server may return code 304 ‘Not Modified’, and no representation
- similarly, `ETag` (hashcode) and `If-None-Match`
- if client requests both `If-Modified-Since` and `If-None-Match`, server should send representation only if representation is modified and has different entity tag

4.8 The trouble with cookies

- opaque data sent by server to client
- often a key into server-side table of session state
- violates principles of statelessness (ie, a hack)
  - client cannot control their own session state (breaks the ‘back’ button on the browser)
  - server must store this state (indefinitely)
  - much easier for client and server to get out of sync

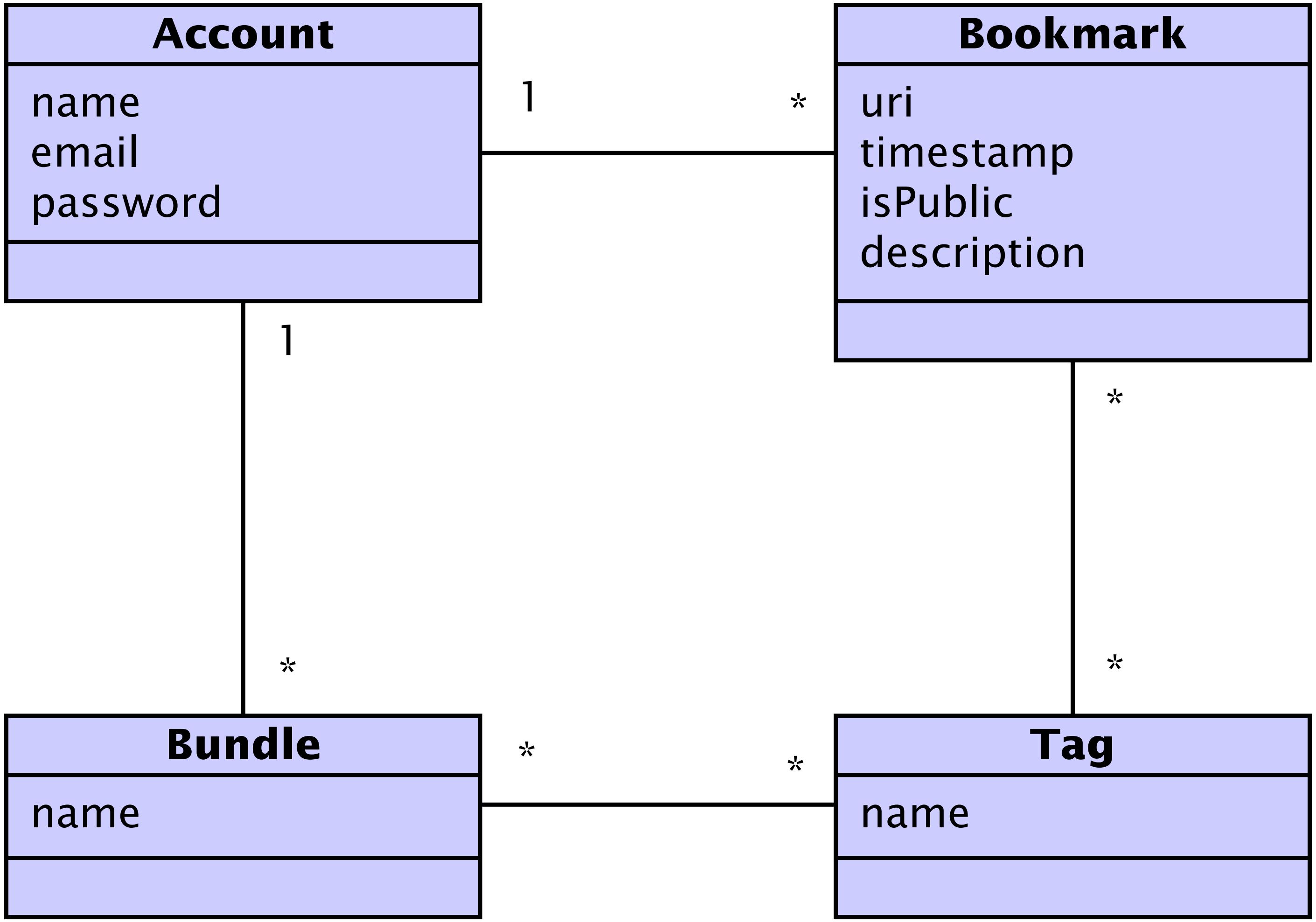
4.9 Interfaces for RESTful services

- how to describe a RESTful web service?
- what is to REST as WSDL is to SOAP?
- *Web Application Description Language* (WADL):
  - site map of resources
  - referential and causal links between resources
  - methods applicable to each resource
  - resource representation formats
  - all machine-processable
- without something like WADL, is REST service-oriented?

5 Example: social bookmarking

- user *accounts*
- *bookmarks*
- *tags* for bookmarks
- *bundles* of tags for a user
- each bookmark accumulates *currency*, *popularity*, *tag cloud* (but we won't model those)

5.1 Structure



5.2 Accounts

intention	*operation*
create account	`POST /users`
view account	`GET /users/<username>`
modify account	`PUT /users/<username>`
delete account	`DELETE /users/<username>`

5.3 Bookmarks

intention	*operation*
post bookmark	`POST /users/<username>/bookmarks`
fetch bookmark	`GET /users/<username>/bookmarks/<uri>`
modify bookmark	`PUT /users/<username>/bookmarks/<uri>`
delete bookmark	`DELETE /users/<username>/bookmarks/<uri>`
user's last post?	use conditional `GET`
user's posting history	`GET /users/<username>/calendar`
fetch filtered history	`GET /users/<username>/calendar/<tag>`

5.4 Searching

intention	*operation*
all user's bookmarks	`GET /users/<username>/bookmarks`
…by tag	`GET /users/<username>/bookmarks/<tag>`
search	`GET /users/<username>/bookmarks?<query>`

5.5 Socialising

intention	*operation*
recent bookmarks	`GET /recent`
. . . by tag	`GET /recent/<tag>`
who bookmarked uri?	`GET /uris/<uri>`

5.6 Tags and bundles

intention	*operation*
user’s vocabulary	`GET /users/<username>/tags`
rename tag	`PUT /users/<username>/tags/<tag>`
user’s bundles	`GET /users/<username>/bundles`
bundle tags	`POST /users/<username>/bundles`
fetch bundle	`GET /users/<username>/bundles/<bundle>`
modify bundle	`PUT /users/<username>/bundles/<bundle>`
delete bundle	`DELETE /users/<username>/bundles/<bundle>`

Index

3.2 Representations

3.3 Interaction

3.4 Connectors

3.5 Components

3.6 State transitions

3.7 REST-based architecture

4 Resource-oriented architecture

4.1 RESTful operations

4.2 PUT vs POST

4.3 Designing a ROA (Richardson & Ruby)

4.4 Asynchronous operations

4.5 URI design

4.6 Representations

4.7 Conditional GET

4.8 The trouble with cookies

4.9 Interfaces for RESTful services

5 Example: social bookmarking

5.1 Structure

5.2 Accounts

5.3 Bookmarks

5.4 Searching

5.5 Socialising

5.6 Tags and bundles

Service-Oriented Architecture

Monday	Tuesday	Wednesday	Thursday	Friday
Introduction Components	REST	Composition	Architecture	Engineering
coffee	coffee	coffee	coffee	coffee
Components	REST	Composition	Architecture	Conclusion
lunch	lunch	lunch	lunch	lunch
Web Services	Qualities	Objects	Semantic Web	
tea	tea	tea	tea	
Web Services	Qualities	Objects	Semantic Web	