university of groningen

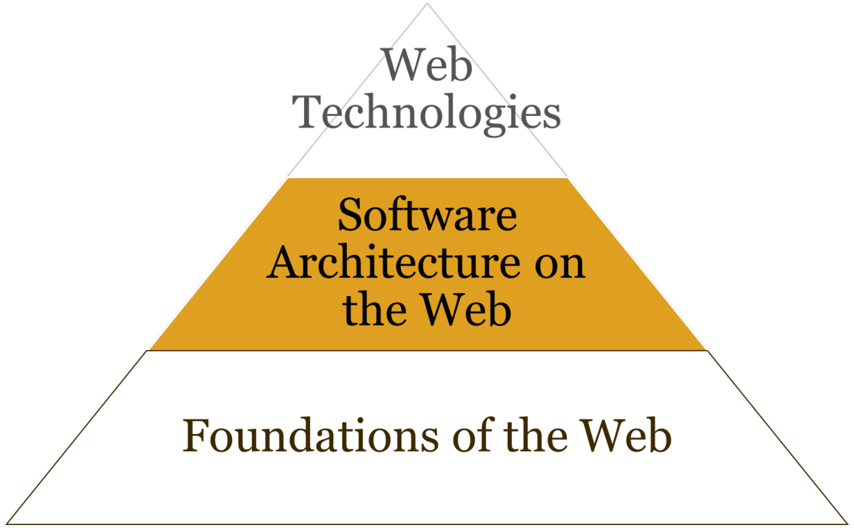
Web Engineering (WBCS008-05)

Set 3: REST

Vasilios Andrikopoulos
v.andrikopoulos@rug.nl

Outline

- REST
  - Definition
  - Principles
- Maturity levels
- API design
- API specification



Web Technologies

Software Architecture on the Web

Foundations of the Web

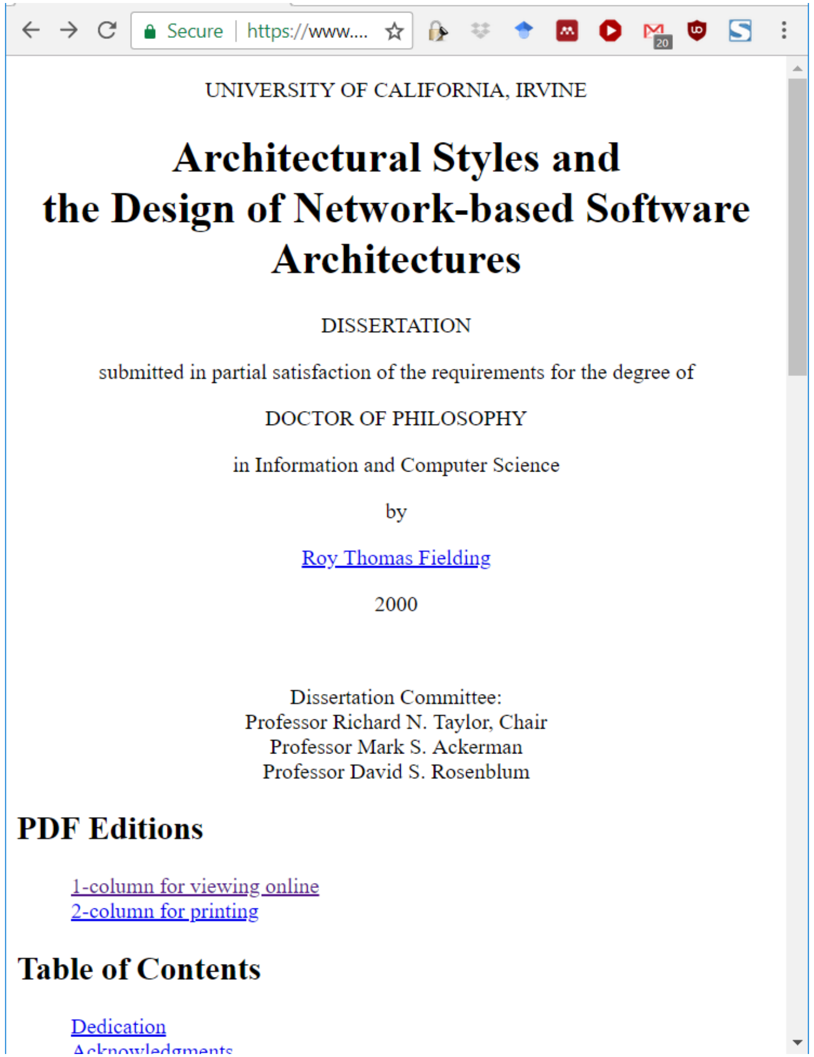
Definition

REpresentational State Transfer (REST)

› REST (as a term) introduced in Roy Fielding's PhD thesis (2000)

› An *architectural style* for building large-scale distributed hypermedia systems

- Defines set of constraints
- Web as an instance of this style



UNIVERSITY OF CALIFORNIA, IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

2000

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

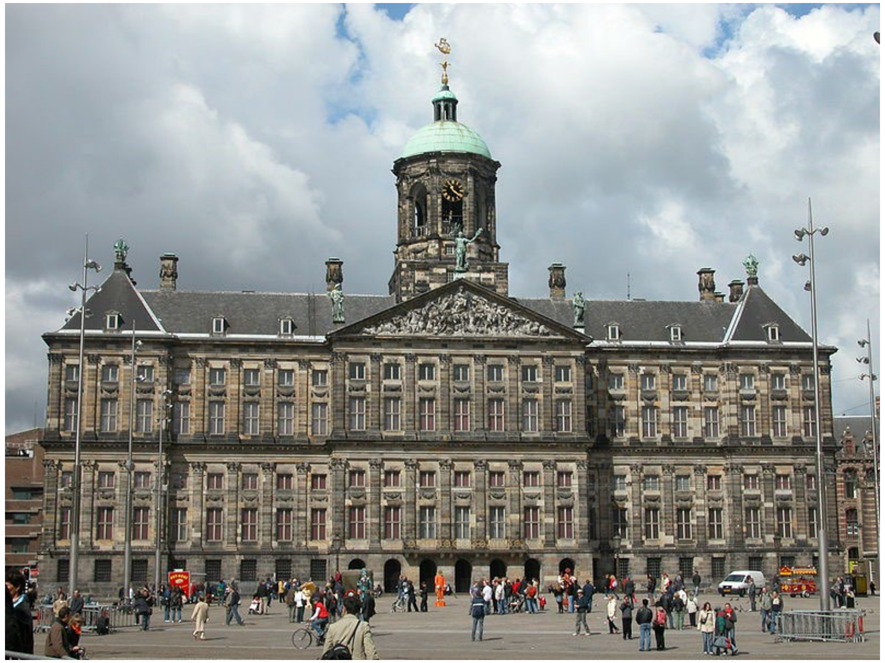
PDF Editions

1-column for viewing online
2-column for printing

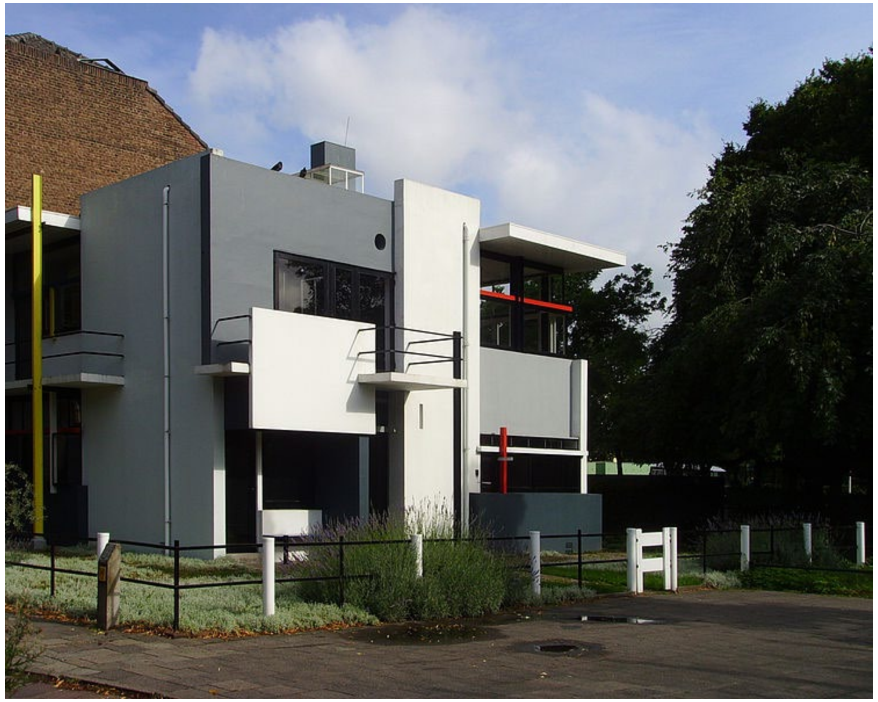
Table of Contents

Dedication
Acknowledgments

Architectural Style vs Architecture



Amsterdam Town Hall in the Dutch Baroque Style



The Rietveld Schröder House in the Modernist Style (as influenced by the De Stijl movement)

Points to keep in mind in the following

1. REST (as a style) cannot be implemented, only adhered to

2. It was defined/distilled from the Web *a posteriori*
   1. Aggregates standards and best practices
   2. The Web follows REST principles

3. It is possible to design other RESTful systems that are not the Web, i.e. they do not use URIs and HTTP

Goals of the REST style

- **Scalability**
  - System can cope with increasing amount of users
- **Simplicity**
  - Easy to use for any kind of users
- **Data independence**
  - Data of a single resource may be in different formats
  - Users may choose the desired representation
- **Performance**
  - Speed

REST in a nutshell

- A set of constraints on system architecture:
  1. Resource Identification e.g. through URIs
  2. Uniform Interface e.g. through HTTP verbs
  3. Self-describing Messages: decouple resource from representations
  4. Hypermedia as the Engine of Application State (HATEOAS): links between resources
  5. Stateless Interactions: separation of resource state from client state

- Claimed benefits: scalability, remixability, usability, accessibility, …

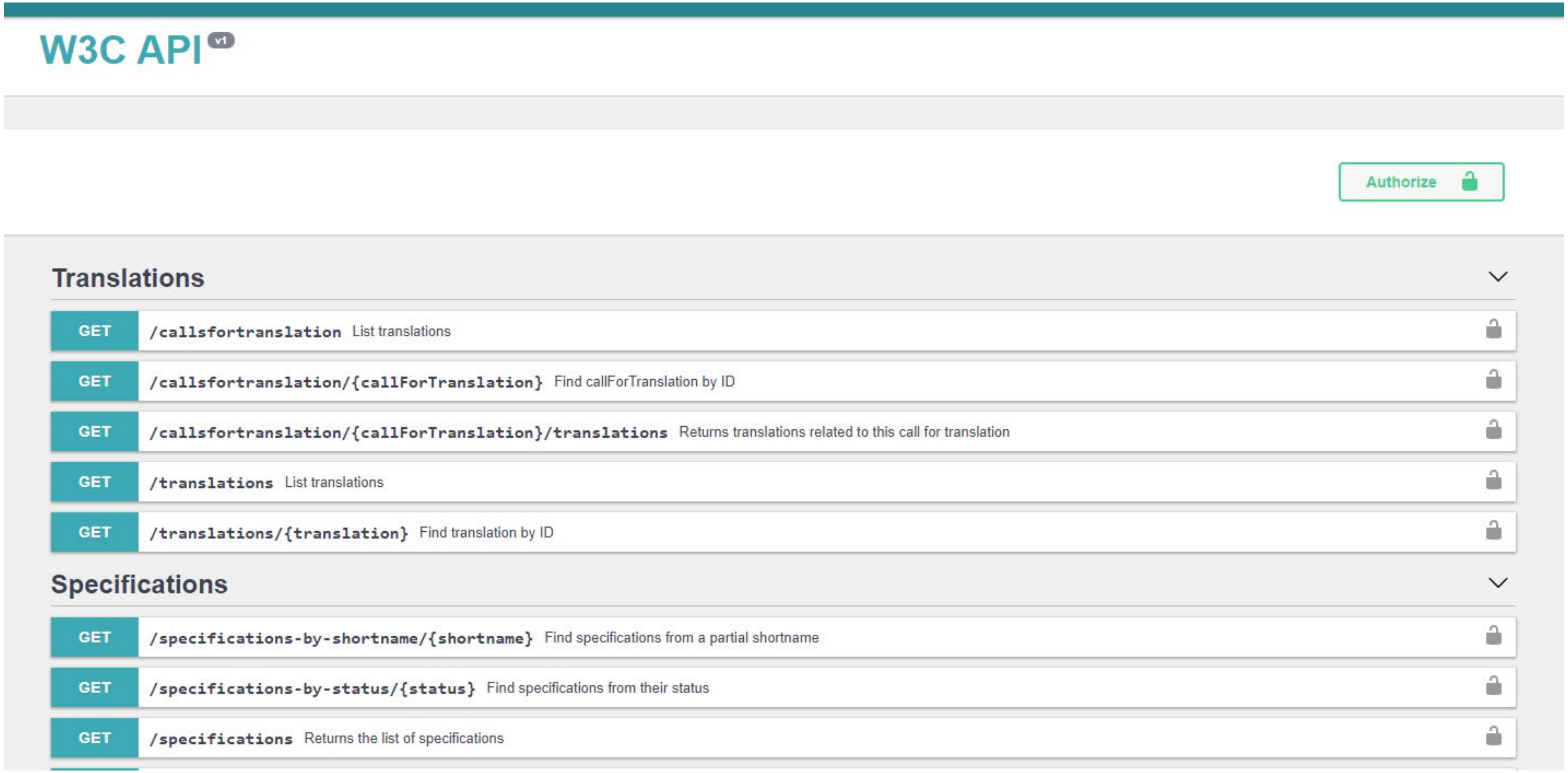
Resource Identification (RI)

- RI: Name everything (resource) that you want to talk about
  - Conversely: anything that can be named can be a resource

- For example, in the Web the use of URIs provides a global addressing space for resource and service discovery

- Application state also represented as a resource [SI]
  - Links to next pages for multi-page process
  - Paged results identifying follow-up pages

RI in practice



W3C API v1

Translations

GET `/callsfortranslation` List translations

GET `/callsfortranslation/{callForTranslation}` Find callForTranslation by ID

GET `/callsfortranslation/{callForTranslation}/translations` Returns translations related to this call for translation

GET `/translations` List translations

GET `/translations/{translation}` Find translation by ID

Specifications

GET `/specifications-by-shortname/{shortname}` Find specifications from a partial shortname

GET `/specifications-by-status/{status}` Find specifications from their status

GET `/specifications` Returns the list of specifications

Uniform Interface (UI)

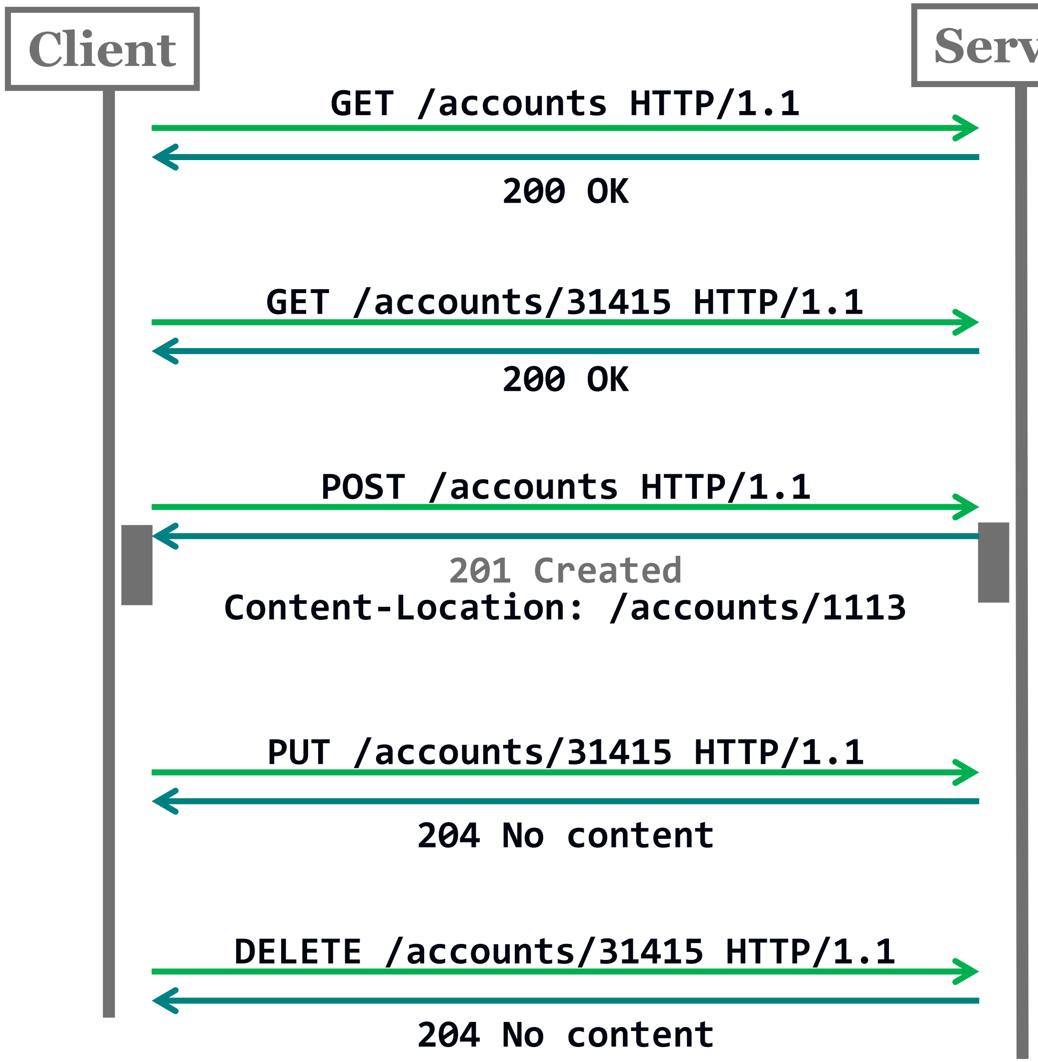
- UI: The same (small) set of operations applies to all resources [RI]

- Small set of verbs applied to large set of nouns
  - Verbs are universal and not application-specific
  - Set of verbs can be extended if there is sufficient demand

- Identify operations that can be optimized
  - Safe vs idempotent operations

- Build functionality based on useful properties of these operations

Uniform Interface example



Client

Server

GET /accounts HTTP/1.1

200 OK

GET /accounts/31415 HTTP/1.1

200 OK

POST /accounts HTTP/1.1

201 Created
Content-Location: /accounts/1113

PUT /accounts/31415 HTTP/1.1

204 No content

DELETE /accounts/31415 HTTP/1.1

204 No content

Retrieve a list of all current accounts

Retrieve a particular account

Create a new account.
Retrieve new account URI from HTTP Headers

Update an account

Delete an account

CREATE
RETRIEVE
UPDATE
DELETE

Self-Describing Messages (SDM)

- Resources are abstract entities → they cannot be used as they are, only identified [RI] and accessed [UI]

- SDM: Resources accessed using only resource representations
  - Sufficient to represent a resource
  - Made clear which representation is used for communication
  - Representation format is negotiable

- Resource representation can be based on different constraints
  - Same model, different format for different users
  - Representation must support links [HATEOAS]

XML Representations

- Extensible Markup Language (XML)
  - Markup ≡ Annotation
  - Meta-language
  - Only syntax, very little semantics
  - Simplification of SGML
- Points of distinction:
  - Document type
  - Portability
- Tree-model based

```
<sentence>This is XML</sentence>

<article>
   <title>What is XML?</title>
   <text>XML stands for
   eXtensible Markup
   Language</text>
</article>

<?xml version=“1.0”?>
<note date=“22/11/2020”>
   <from name=“me”/>
   <to name=“me”/>
   <description>Don’t forget the
   presentation </description>
</note>
```

JSON representations

- JavaScript deals with XML by parsing it first into a (Document Object Model) DOM tree
  - Inconvenient and ineffective
- JavaScript Object Notation (JSON) encodes data as JS objects
  - More efficient if client is in JS

```
{
  “sentence” : “This is JSON”
}

{
  “article” : {
    “title” : “What is JSON?”,
    “text” : “JSON stands for JavaScript
    Object Notation”
  }
}

{
  “note” : {
    “date” : “22/11/2020”,
    “from” : {
           “name” : “me”},
    “to” : {
           “name” : “me”},
    “description” : “Don’t forget the
    presentation”
  }
}
```

Other representation formats examples

- XHTML (HTML that can be parsed using an XML parser) `Media-type: application/xhtml+xml`
- Atom (XML vocabulary for describing time-stamped entries – used extensively in news feeds) `Media-type: application/atom+xml`
- SVG (Scalable Vector Graphics: XML vocabulary for storing and manipulating graphics) `Media-type: image/svg+xml`
- RDF (Resource Description Framework: a URI-based knowledge representation framework) through an XHTML Microformat like RDFa
- Specialized XML vocabularies like MathML or OpenDocument `Media-type: application/xml`

Hypermedia as the Engine of Application State (HATEOAS)

- Resource representations [SDM] contain links to identifiable resources [RI]
- HATEOAS: Resources and their state accessed through link navigation
- RESTful applications do not call but navigate
  - Traversal paths contained as links in the resources representations [SDM]
  - Navigation to next resource depends on link semantics

Stateless Interactions (SI)

- SI: State is moved to clients or resources themselves
  - Avoids state on server-side applications
- Resource state managed by the server: the same for all clients, can be changed by a client
- Client state managed by the client: maintained by each client separately, affects access to server resources but not the resources themselves
- Stateless interaction ≠ stateless application
- Security as a major concern due to lack of trust of client state

State Management on the Web

- Essential for supporting [HATEOAS] and [SI]
- State embedded in every resource representation/URI
- Cookies as a very popular alternative
  - Session state (e.g. logged in/out) as a set of variables maintained on client and server side
  - More convenient than state embedding
  - Web frameworks can handle either transparently through URI rewriting

Cookies side-effects

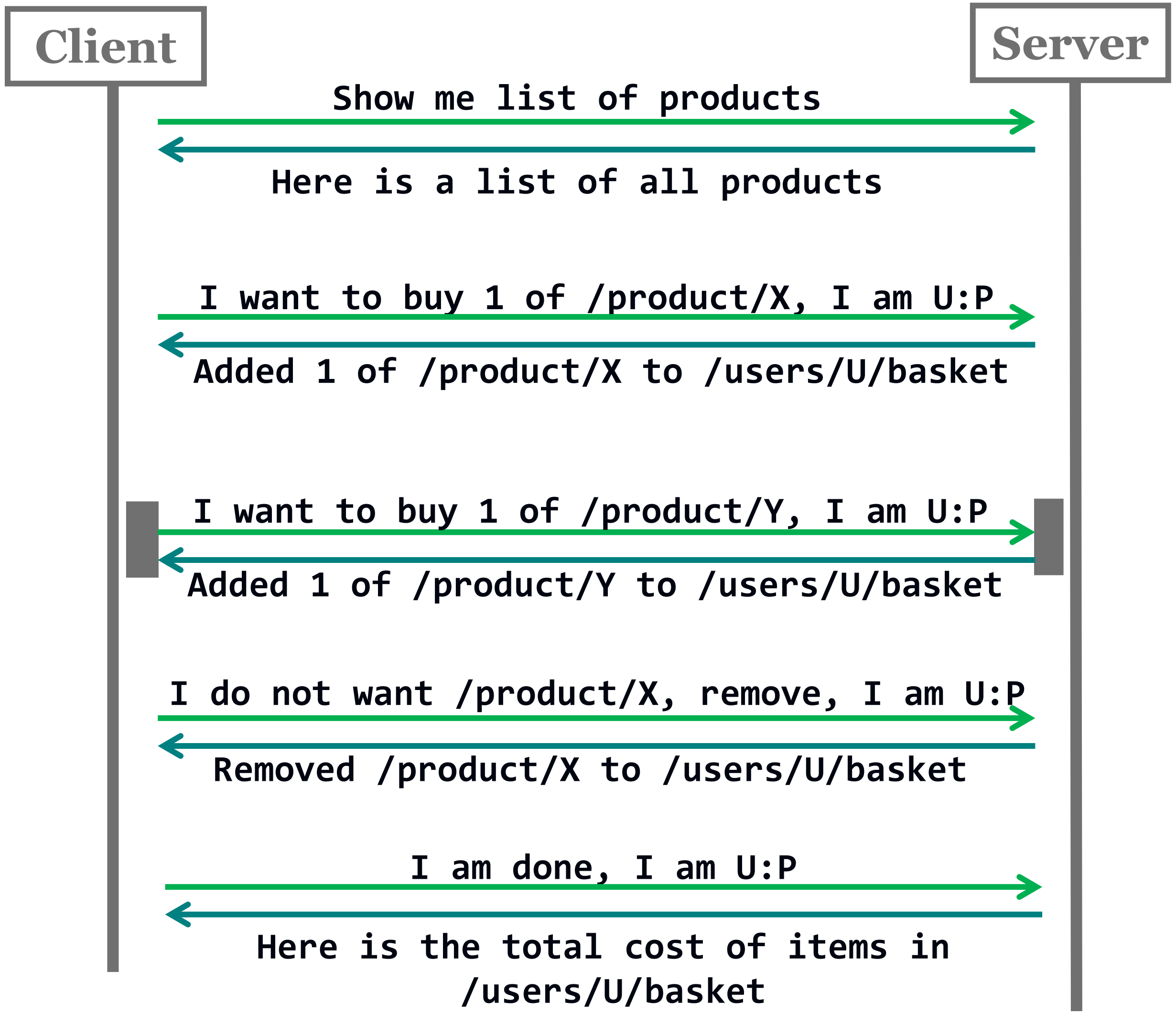
› Client-side:

- Stored persistently independent of resource representation (!)
- Effectively "shared state" within the context of the same client application, e.g. browser

› Server-side (through Session IDs)

- Require expensive tracking
- Potentially global/cross-resource
- Load balancing to be cookie-aware
- Resource-based state as a useful alternative solution

Resource-based state by example



Show me list of products

Here is a list of all products

I want to buy 1 of /product/X, I am U:P

Added 1 of /product/X to /users/U/basket

I want to buy 1 of /product/Y, I am U:P

Added 1 of /product/Y to /users/U/basket

I do not want /product/X, remove, I am U:P

Removed /product/X to /users/U/basket

I am done, I am U:P

Here is the total cost of items in /users/U/basket

Achieving the REST Goals

- Performance
  - [RI+UI] Caching is enabled → “the closer the cache, the faster the response”

- Scalability
  - Caching allows serving requests without accessing origin server
  - [HATEOAS+SI] All required state is contained in request
    - No server affinity to requests
    - Requests can be sprayed across clustered servers (load balancing)

Achieving the REST Goals (cont.)

› Simplicity

- [UI] Standardized interface to all resources allows of simpler interaction design

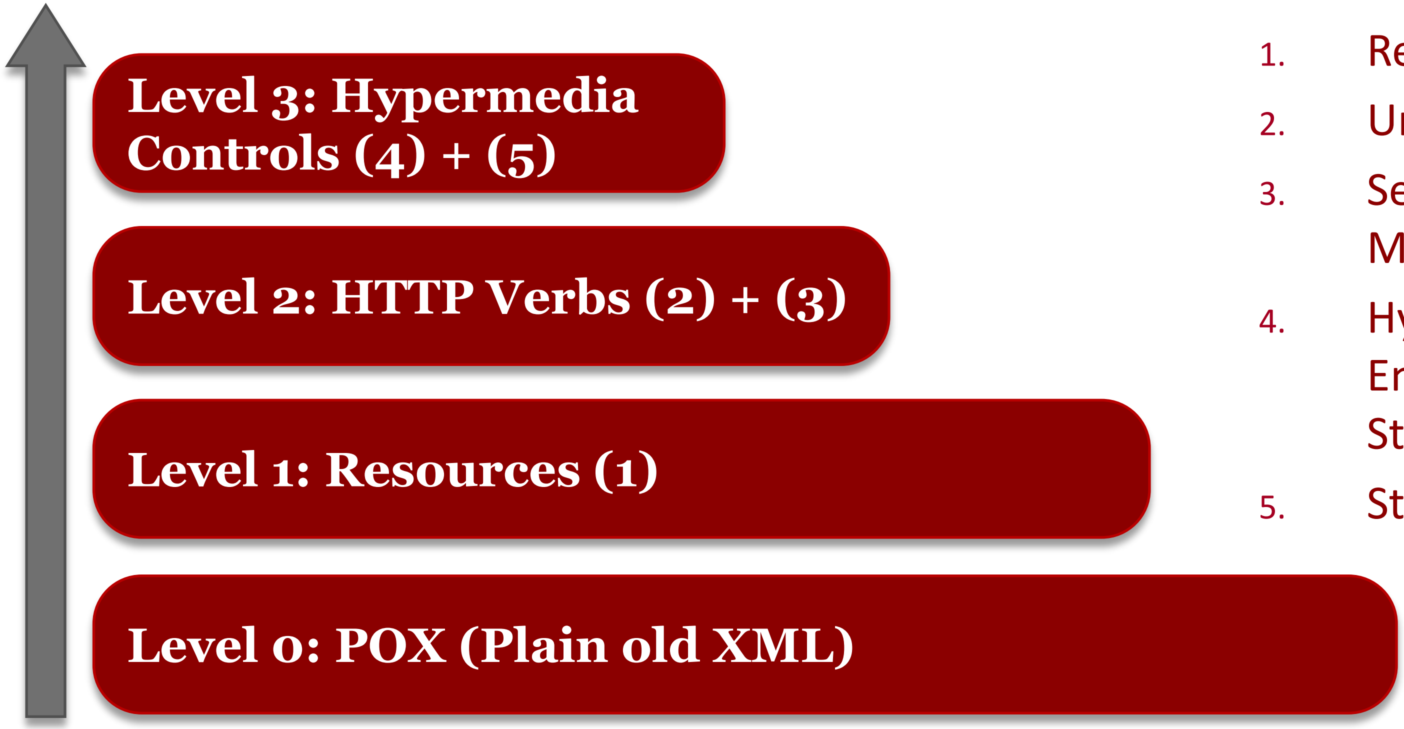
› Data independence

- [SDM] Different formats are available for each resource
- Content can be tailored to client’s capabilities

Maturity Model

Maturity Levels of REST Applications

- Introduced by Richardson in 2008
- Helps explain properties of applications
- Used as a metric for compliance to REST principles



Level 3: Hypermedia Controls (4) + (5)

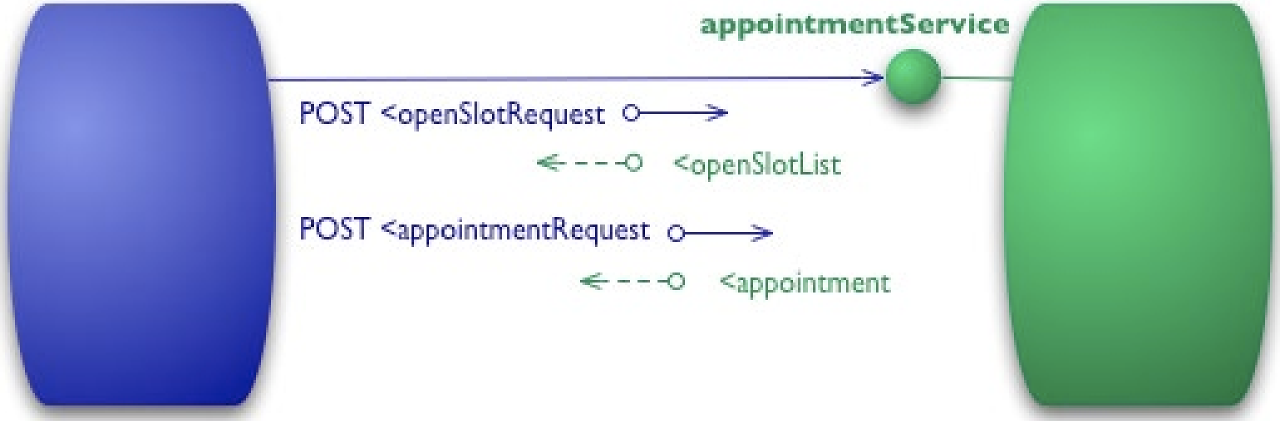
Level 2: HTTP Verbs (2) + (3)

Level 1: Resources (1)

Level 0: POX (Plain old XML)

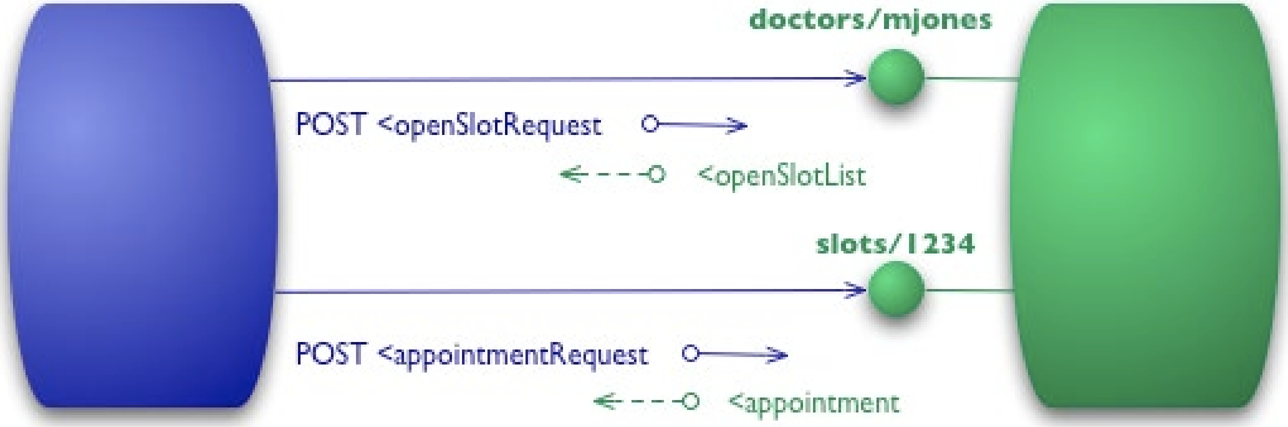
1. Resource Identification
2. Uniform Interface
3. Self-describing Messages
4. Hypermedia as the Engine of Application State
5. Stateless Interactions

Level 0



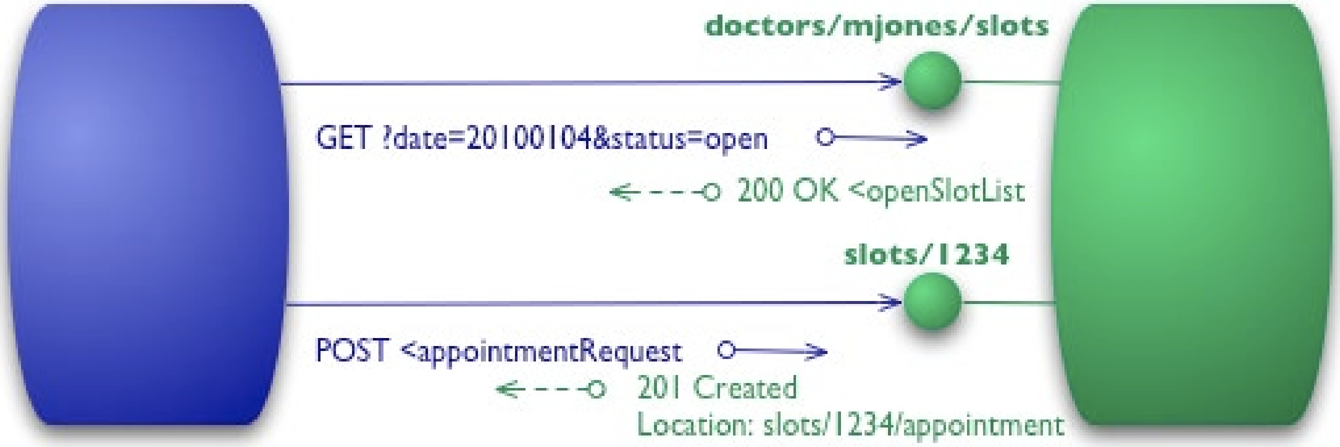
- › Simply using HTTP as a transport system for remote interactions
  - Without using any of the mechanisms of the Web
  - Essentially using HTTP as a tunneling mechanism for RPC
  - Any representation of resources possible
  - Resources are identifiable
  - Data and meta-data in the message body

Level 1



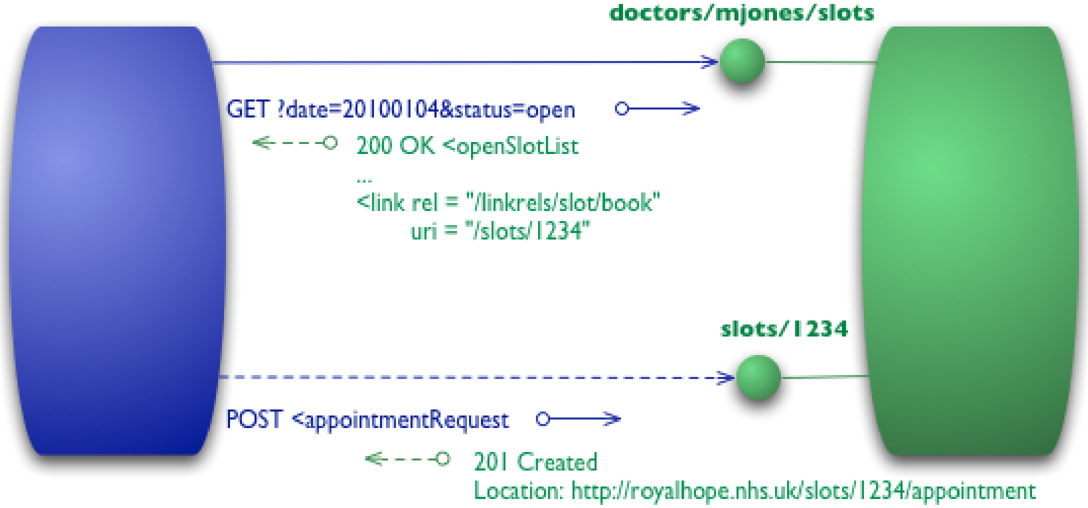
- Resources are uniquely identifiable by URIs
  - Example resources: doctor, appointment slot
  - Interactions target these resources
  - Data and meta-data in the message body

Level 2



› Uses HTTP Verbs correctly to interact with resources
  - All verbs and all standard responses (response codes – even for faults)
  - Meta-data used to identify the resource in URI

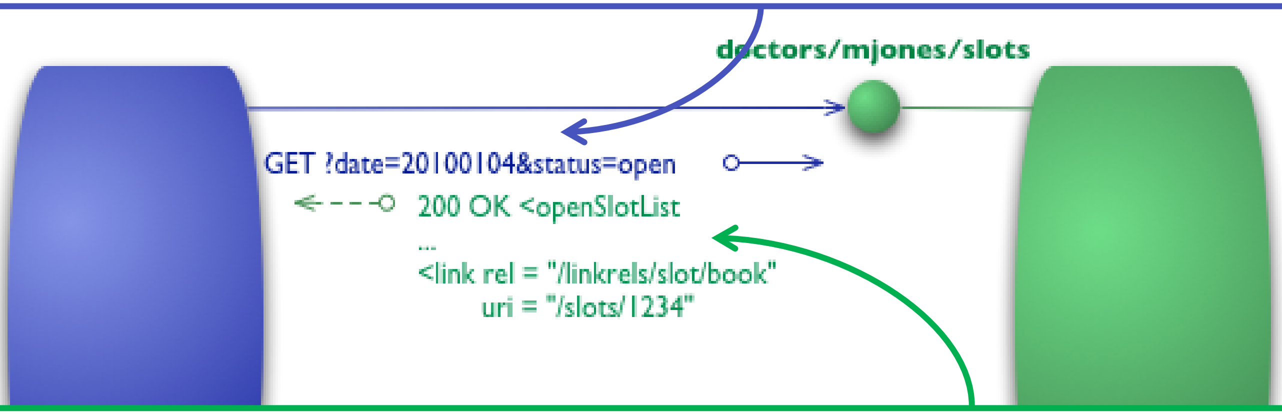
Level 3


doctors/mjones/slots
GET ?date=20100104&status=open
200 OK <openSlotList
...
<link rel = "/linkrels/slot/book"
uri = "/slots/1234"
slots/1234
POST <appointmentRequest
201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment

› Introduces HATEOAS & SI – hypermedia controls
- Provides the representation to the next valid state change operations on the resource (link)
- Provides the URI of the resource
- Allows for dynamic changes in the URIs, extend the set of links to a resource
- Warning: no universal standard for representing hypermedia controls
  - Here ATOM: e.g. `<link rel = "/linkrels/help" uri = "/help/appointment"/>`

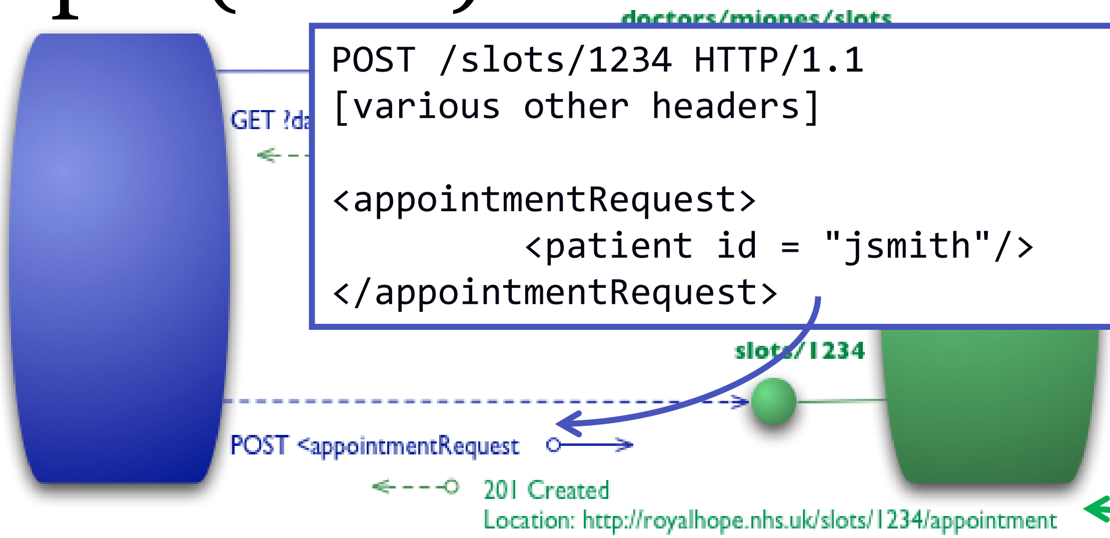
Level 3 example

```
GET /doctors/mjones/slots?date=20100104&status=open  HTTP/1.1
Host: royalhope.nhs.uk
```


doctors/mjones/slots
GET ?date=20100104&status=open
200 OK <openSlotList
...
<link rel = "/linkrels/slot/book"
uri = "/slots/1234"

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
          <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
                <link rel = "/linkrels/slot/book" uri = "/slots/1234"/>
         </slot>
         <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
                <link rel = "/linkrels/slot/book" uri = "/slots/5678"/>
         </slot>
</openSlotList>
```

Level 3 example (cont.)



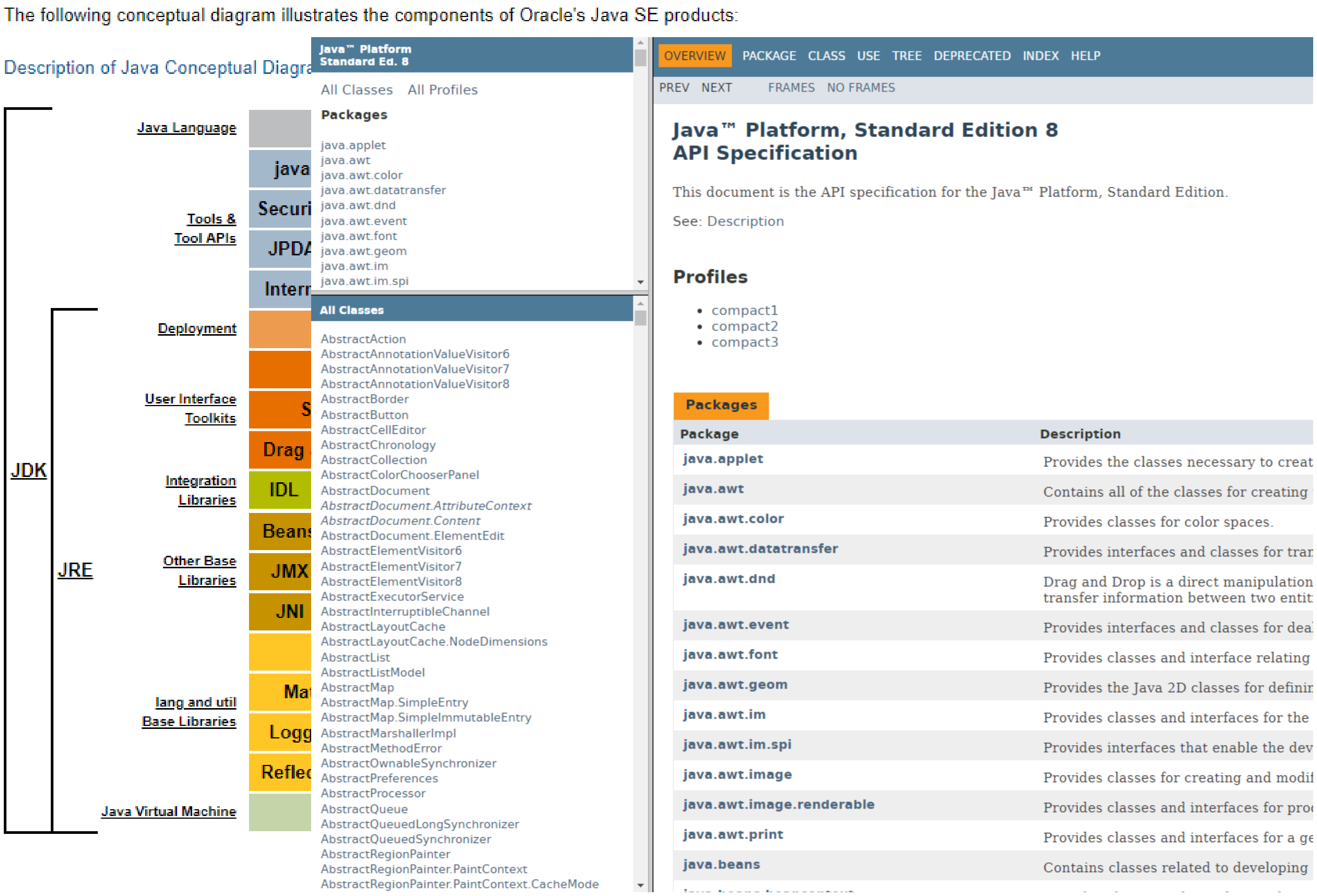
```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
        <patient id = "jsmith"/>
</appointmentRequest>
```

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]
<appointment>
        <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
        <patient id = "jsmith"/>
        <link rel = "/linkrels/appointment/cancel" uri =
"/slots/1234/appointment"/>
        <link rel = "/linkrels/appointment/addTest" uri =
"/slots/1234/appointment/tests"/>
        <link rel = "self" uri = "/slots/1234/appointment"/>
        …
</appointment>
```

API Design

SWEBOK’s definition

An *application programming interface (API)* is the set of signatures that are exported and available to the users of a library or a framework to write their applications. Besides signatures, an API should always include statements about the program’s effects and/or behaviors (i.e., its semantics).



The following conceptual diagram illustrates the components of Oracle's Java SE products:

Description of Java Conceptual Diagra

Java™ Platform Standard Ed. 8

All Classes   All Profiles

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractAnnotationValueVisitor8
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractChronology
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractElementVisitor8
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer
AbstractPreferences
AbstractProcessor
AbstractQueue
AbstractQueuedLongSynchronizer
AbstractQueuedSynchronizer
AbstractRegionPainter
AbstractRegionPainter.PaintContext
AbstractRegionPainter.PaintContext.CacheMode

OVERVIEW   PACKAGE   CLASS   USE   TREE   DEPRECATED   INDEX   HELP

PREV   NEXT   FRAMES   NO FRAMES

Java™ Platform, Standard Edition 8 API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

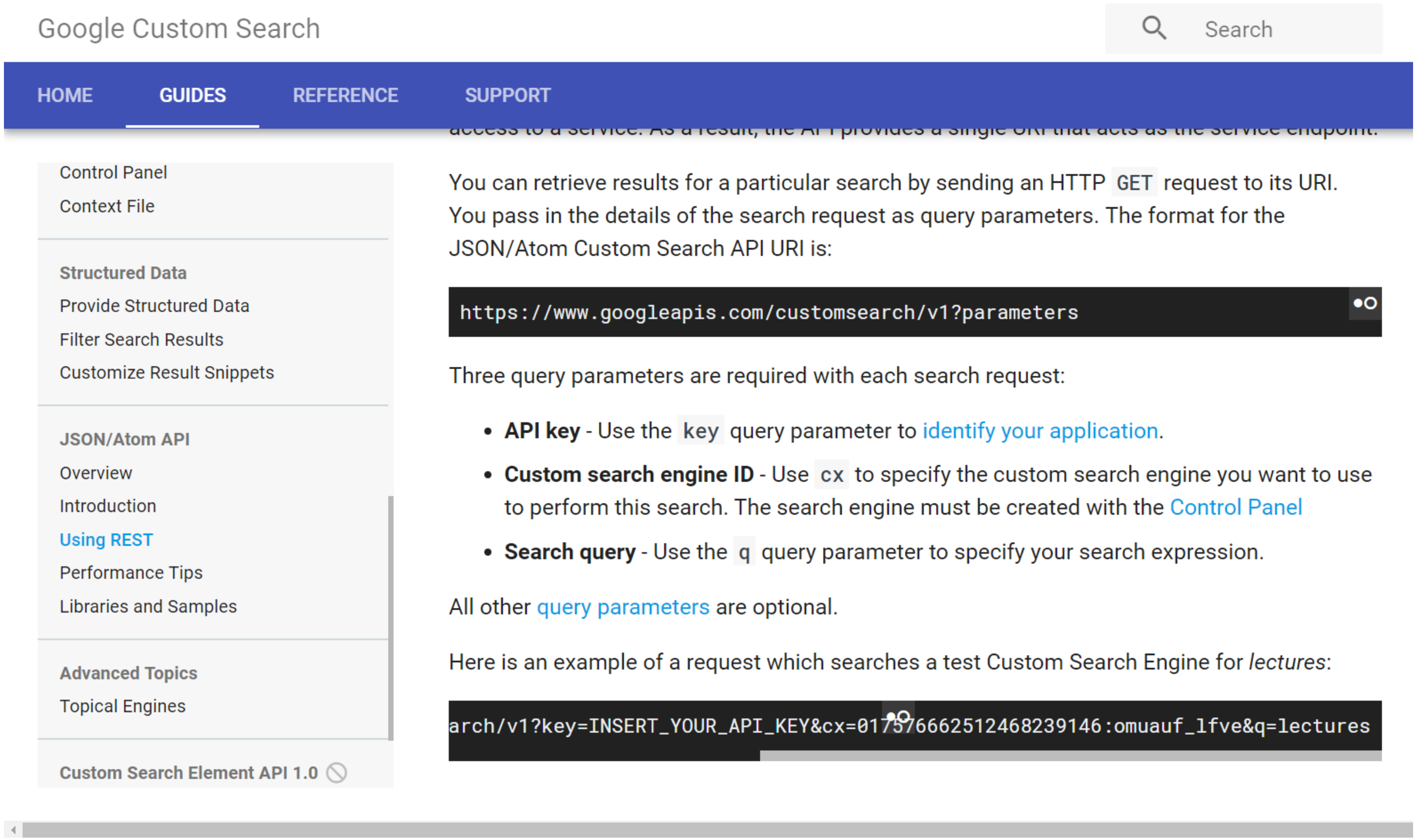
See: Description

Profiles

- compact1
- compact2
- compact3

Packages

Package	Description
java.applet	Provides the classes necessary to creat
java.awt	Contains all of the classes for creating
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for tran
java.awt.dnd	Drag and Drop is a direct manipulation transfer information between two entit
java.awt.event	Provides interfaces and classes for deal
java.awt.font	Provides classes and interface relating
java.awt.geom	Provides the Java 2D classes for definir
java.awt.im	Provides classes and interfaces for the
java.awt.im.spi	Provides interfaces that enable the dev
java.awt.image	Provides classes for creating and modif
java.awt.image.renderable	Provides classes and interfaces for proc
java.awt.print	Provides classes and interfaces for a ge
java.beans	Contains classes related to developing



access to a service. As a result, the API provides a single URI that acts as the service endpoint.

You can retrieve results for a particular search by sending an HTTP `GET` request to its URI. You pass in the details of the search request as query parameters. The format for the JSON/Atom Custom Search API URI is:

```
https://www.googleapis.com/customsearch/v1?parameters
```

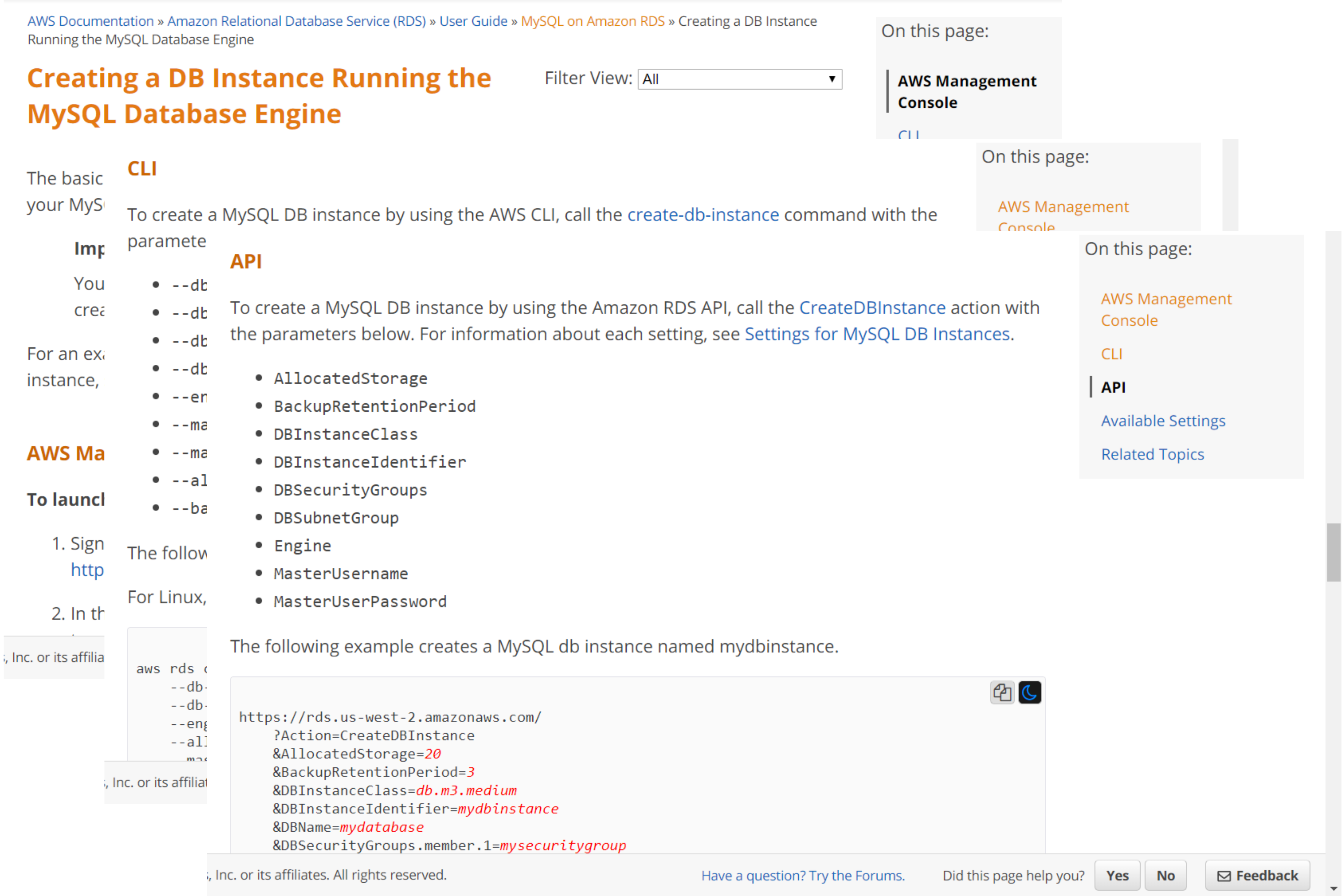
Three query parameters are required with each search request:

- **API key** - Use the `key` query parameter to identify your application.
- **Custom search engine ID** - Use `cx` to specify the custom search engine you want to use to perform this search. The search engine must be created with the Control Panel
- **Search query** - Use the `q` query parameter to specify your search expression.

All other query parameters are optional.

Here is an example of a request which searches a test Custom Search Engine for *lectures*:

```
arch/v1?key=INSERT_YOUR_API_KEY&cx=017576662512468239146:omuauf_lfve&q=lectures
```



AWS Documentation » Amazon Relational Database Service (RDS) » User Guide » MySQL on Amazon RDS » Creating a DB Instance Running the MySQL Database Engine

Creating a DB Instance Running the MySQL Database Engine

On this page:

AWS Management Console

CLI

The basic your MyS

Imp

You crea

For an ex instance,

AWS Ma

To launc

1. Sign http
2. In th

, Inc. or its affilia

CLI

To create a MySQL DB instance by using the AWS CLI, call the create-db-instance command with the paramete

- --db
- --db
- --db
- --db
- --en
- --ma
- --ma
- --al
- --ba

The follow

For Linux,

```
aws rds c
    --db-
    --db-
    --eng
    --all
```

, Inc. or its affiliat

On this page:

AWS Management Console

API

To create a MySQL DB instance by using the Amazon RDS API, call the CreateDBInstance action with the parameters below. For information about each setting, see Settings for MySQL DB Instances.

- `AllocatedStorage`
- `BackupRetentionPeriod`
- `DBInstanceClass`
- `DBInstanceIdentifier`
- `DBSecurityGroups`
- `DBSubnetGroup`
- `Engine`
- `MasterUsername`
- `MasterUserPassword`

The following example creates a MySQL db instance named mydbinstance.

```
https://rds.us-west-2.amazonaws.com/
    ?Action=CreateDBInstance
    &AllocatedStorage=20
    &BackupRetentionPeriod=3
    &DBInstanceClass=db.m3.medium
    &DBInstanceIdentifier=mydbinstance
    &DBName=mydatabase
    &DBSecurityGroups.member.1=mysecuritygroup
```

On this page:

AWS Management Console

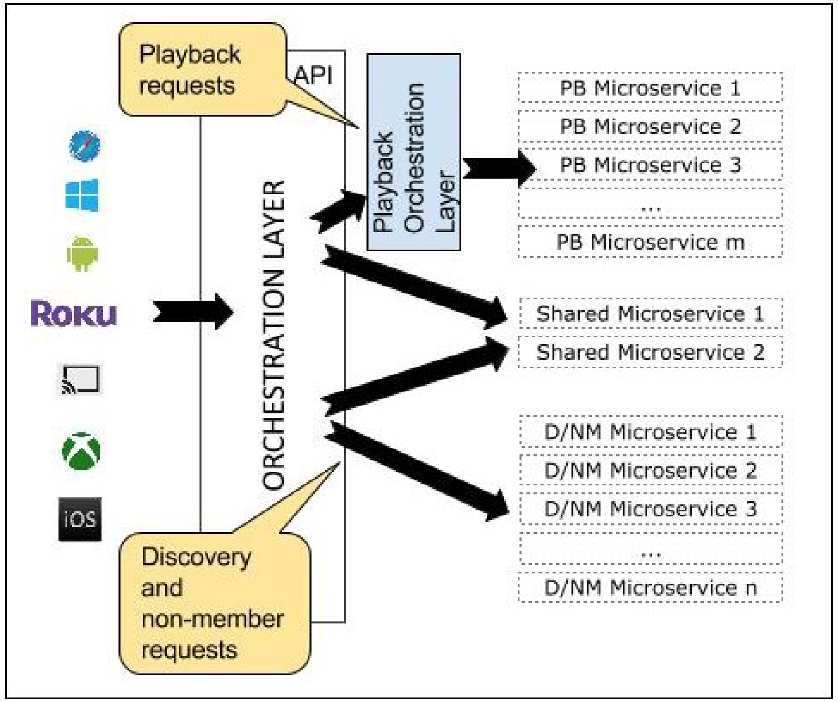
CLI

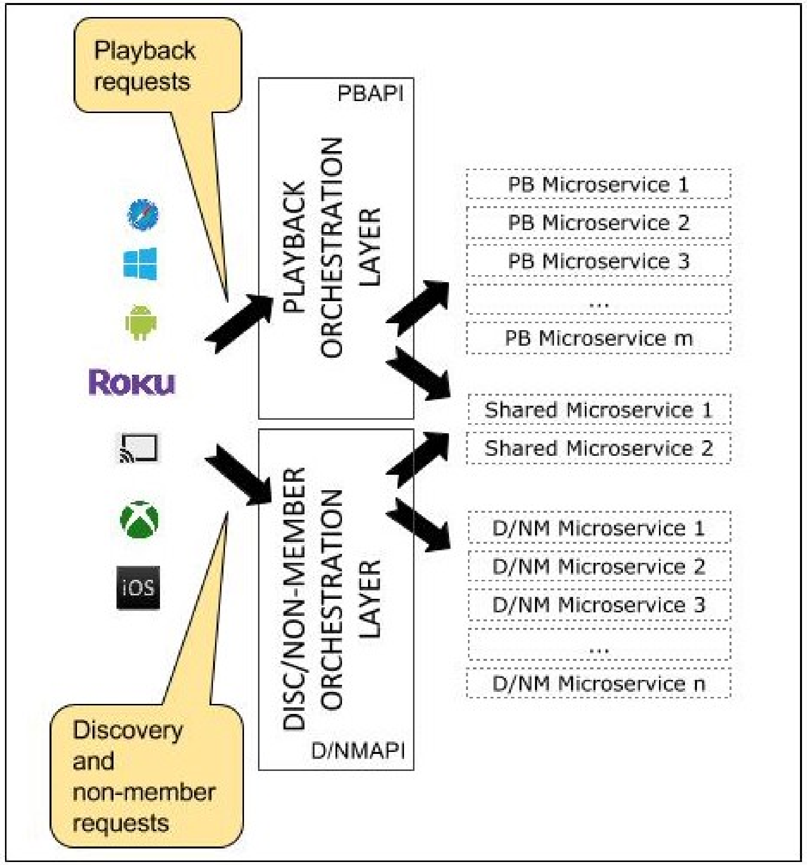
API

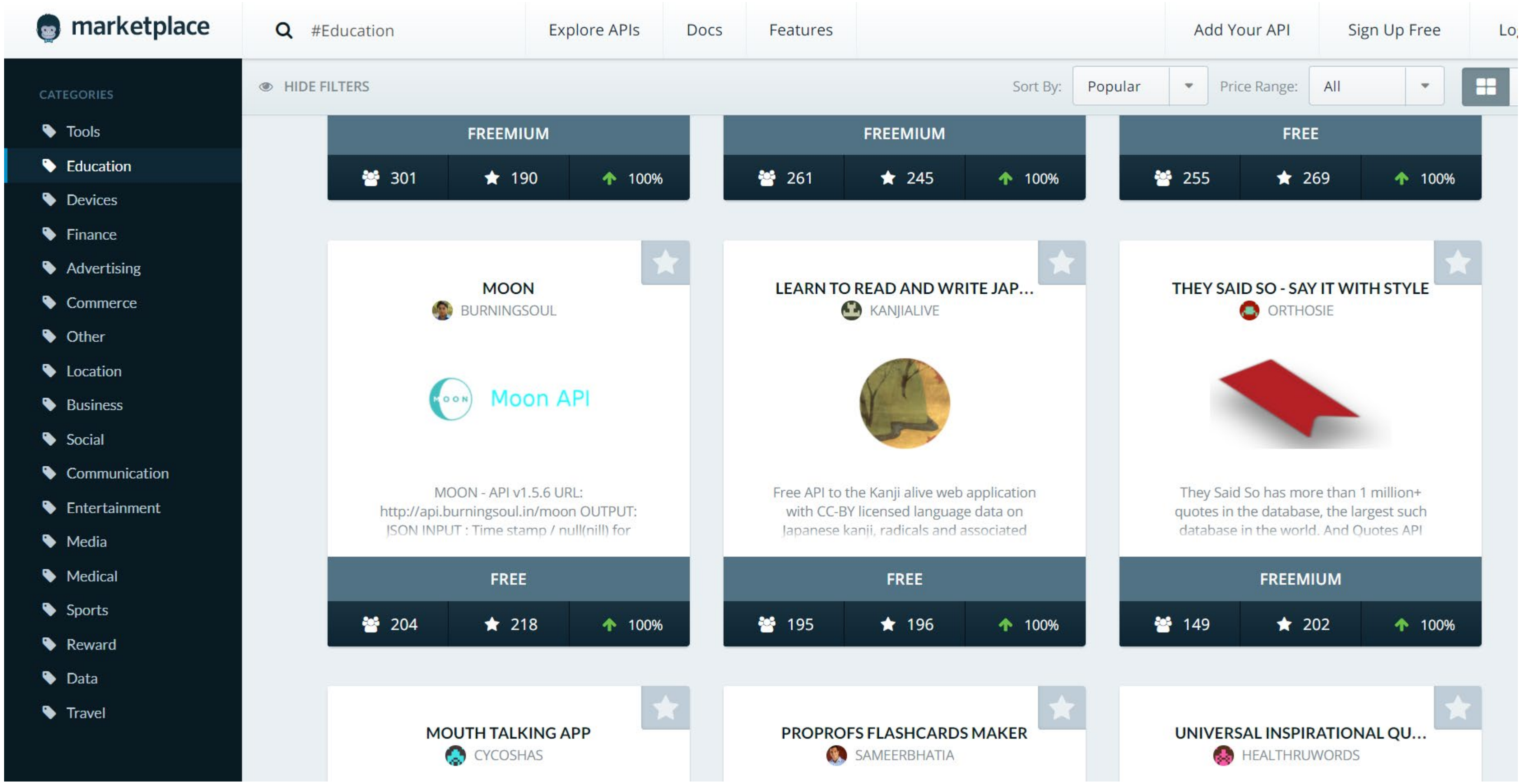
Available Settings

Related Topics

, Inc. or its affiliates. All rights reserved.


Playback requests
API
Playback Orchestration Layer
ORCHESTRATION LAYER
Roku
iOS
PB Microservice 1
PB Microservice 2
PB Microservice 3
...
PB Microservice m
Shared Microservice 1
Shared Microservice 2
D/NM Microservice 1
D/NM Microservice 2
D/NM Microservice 3
...
D/NM Microservice n
Discovery and non-member requests


Playback requests
PBAPI
PLAYBACK ORCHESTRATION LAYER
Roku
iOS
PB Microservice 1
PB Microservice 2
PB Microservice 3
...
PB Microservice m
Shared Microservice 1
Shared Microservice 2
D/NM Microservice 1
D/NM Microservice 2
D/NM Microservice 3
...
D/NM Microservice n
DISC/NON-MEMBER ORCHESTRATION LAYER
D/NMAPI
Discovery and non-member requests



marketplace

#Education

Explore APIs | Docs | Features | Add Your API | Sign Up Free | Lo

CATEGORIES

- Tools
- Education
- Devices
- Finance
- Advertising
- Commerce
- Other
- Location
- Business
- Social
- Communication
- Entertainment
- Media
- Medical
- Sports
- Reward
- Data
- Travel

HIDE FILTERS

Sort By: Popular

Price Range: All

FREEMIUM

301 ★ 190 ↑ 100%

FREEMIUM

261 ★ 245 ↑ 100%

FREE

255 ★ 269 ↑ 100%

MOON

BURNINGSOUL

Moon API

MOON - API v1.5.6 URL: http://api.burningsoul.in/moon OUTPUT: JSON INPUT : Time stamp / null(nill) for

FREE

204 ★ 218 ↑ 100%

LEARN TO READ AND WRITE JAP…

KANJIALIVE

Free API to the Kanji alive web application with CC-BY licensed language data on Japanese kanji, radicals and associated

FREE

195 ★ 196 ↑ 100%

THEY SAID SO - SAY IT WITH STYLE

ORTHOSIE

They Said So has more than 1 million+ quotes in the database, the largest such database in the world. And Quotes API

FREEMIUM

149 ★ 202 ↑ 100%

MOUTH TALKING APP

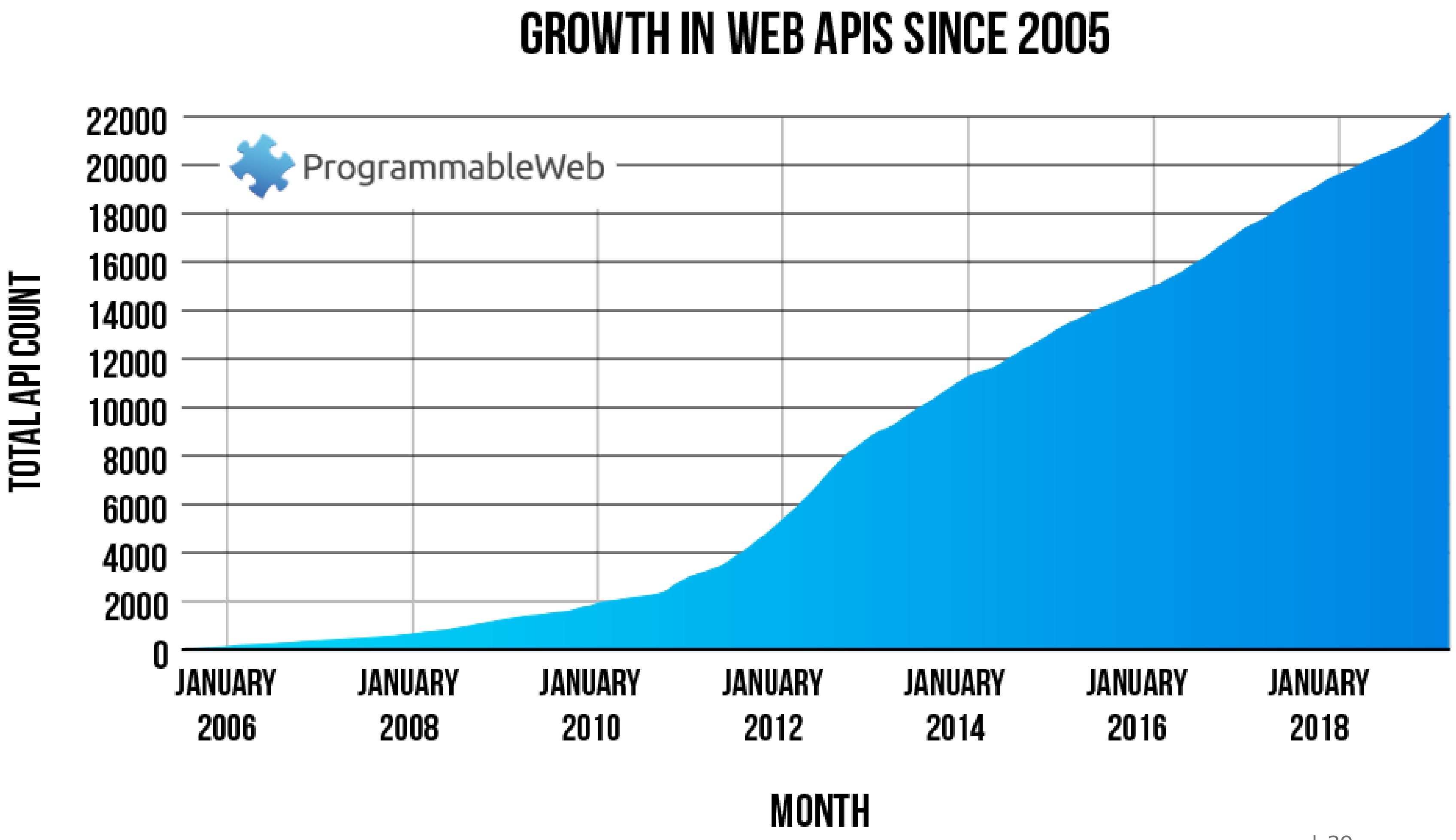
CYCOSHAS

PROPROFS FLASHCARDS MAKER

SAMEERBHATIA

UNIVERSAL INSPIRATIONAL QU…

HEALTHRUWORDS


GROWTH IN WEB APIS SINCE 2005
TOTAL API COUNT
22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
ProgrammableWeb
JANUARY 2006
JANUARY 2008
JANUARY 2010
JANUARY 2012
JANUARY 2014
JANUARY 2016
JANUARY 2018
MONTH

RESTful APIs Design Principles

Information abstraction of a key element constitutes **a resource**

Resource representation is a sequence of bytes, plus representation **metadata**; the representation is **negotiable**

All **interactions are context-free** i.e. state agnostic

Components can perform only a small set of **well-defined methods**

Idempotency of operations and representation metadata is encouraged

Presence of **intermediaries** is promoted

Note on rule formulation

- Requirement verbs in the following are compliant with RFC 2119 definitions, namely:

1. MUST/REQUIRED/SHALL: absolute requirement
2. MUST/SHALL NOT : absolute prohibition
3. SHOULD/RECOMMENDED: there may be valid reasons to ignore this item, but full implications must be taken into consideration
4. SHOULD NOT/NOT RECOMMENDED: (as above, but for accepting a particular behavior)
5. MAY/OPTIONAL: truly optional

Interface

- URI format
  - / operator must be used to indicate a hierarchical relationship (no trailing /es)
  - Design for readability (use -, avoid _, prefer lower case letters) since they may be used in browsers
  - File extensions should not be included in URIs to emphasize content negotiation

- URI Authority
  - Consistent subdomain names should be used for APIs & client developer portal (if any)

Interface: Path Design

- Singular noun should be used for document names i.e. single **resources**, e.g. `http://api.soccer.restapi.org/leagues/seattle`

- Plural nouns should be used for **collections** or *stores* (client-managed resource repositories) e.g. `http://api.music.restapi.org/artists/mikemassedotcom/playlists`

- Verbs or verb phrases for **controllers** (executable functions) e.g. `POST /alerts/245743/resend`

- CRUD function names should not be used

Interface: Query Design

- Query component may be used to filter collections or stores e.g. `GET /users?role=admin`

- Query component may be used for pagination or subsetting e.g. `GET /users?pageSize=25&pageStartIndex=50`

Interactions

- Request methods
  - GET and POST must not be used to tunnel other request methods
  - GET must be used to retrieve a representation of a resource (only)

  - PUT must be used for both insert (in a store) and update
  - PUT must be used to update (any) mutable resources

Interactions (cont.)

- Request methods (cont.)
  - POST must be used to create a new resource in a collection e.g. `POST /leagues/seattle/teams/trebuchet/players`
  - POST must be used to execute controllers e.g. `POST /alerts/245743/resend`
  - DELETE must be used to remove a resource from its parent
- Response Status Codes semantics must be **strictly** enforced

Metadata: HTTP Headers

- `Content-Type` must be used
- `Content-Length` should be used
- `Last-Modified` should be used in responses
- Stores must support conditional PUT requests
- `Location` header must be used to specify the URI of a new resource
- `Cache-Control`, `Expires`, and `Date` response headers should be used (caching)
- Custom HTTP headers must not be used to change the behavior of HTTP methods

Metadata: Media types

- Application-specific media types should be used e.g. not simply "`application/json`"
  - But it is preferable to "`text/plain`"
- Media type negotiations should be supported
- Media type selection by query parameter may be supported e.g. `GET /bookmarks/mikemassedotcom?accept=application/xml`

Representation: Message Body Format

- JSON should be supported for resource representation by default
- JSON must be well-formed (mixed lower case without special characters when possible)
- XML and other formats may optionally be used
- Additional envelopes must not be created (i.e. body contains the resource state representation only)

Representation: Hypermedia

- Hypermedia Representation
  - A consistent form should be used for link representation, link relation representation, and link advertisement
  - A self link should be included in response message body representations
  - Minimize the number of advertised entry point URLs
  - Links should be used to advertise a resource’s available actions in a state-sensitive manner

- Media Type Representation
  - A consistent form should be used to represent media type formats and schemas

Representation: Errors

- A consistent form should be used to represent errors and error responses

- Consistent error types should be used for common error conditions

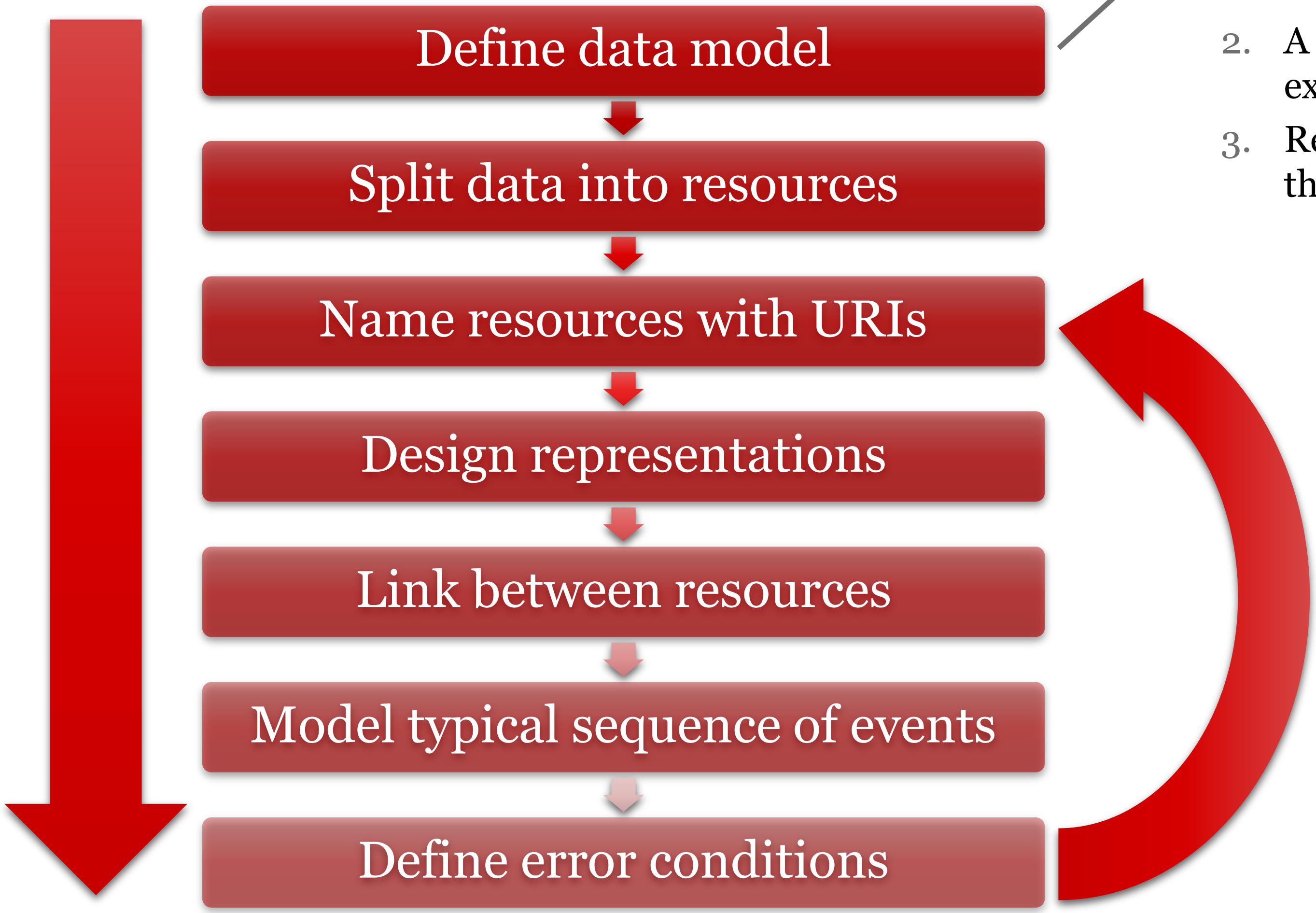
Client concerns

- Versioning
  - New URIs should be used to introduce new concepts
  - Schemas should be used to manage representational form versions

- Security
  - Oauth (Open Authorization) may be used to protect resources
  - API Management solutions may be used to protect resources

- Response Representation Composition
  - URI query component should be used to support partial responses or to embed linked resources

Read-only resource design



Define data model

Split data into resources

Name resources with URIs

Design representations

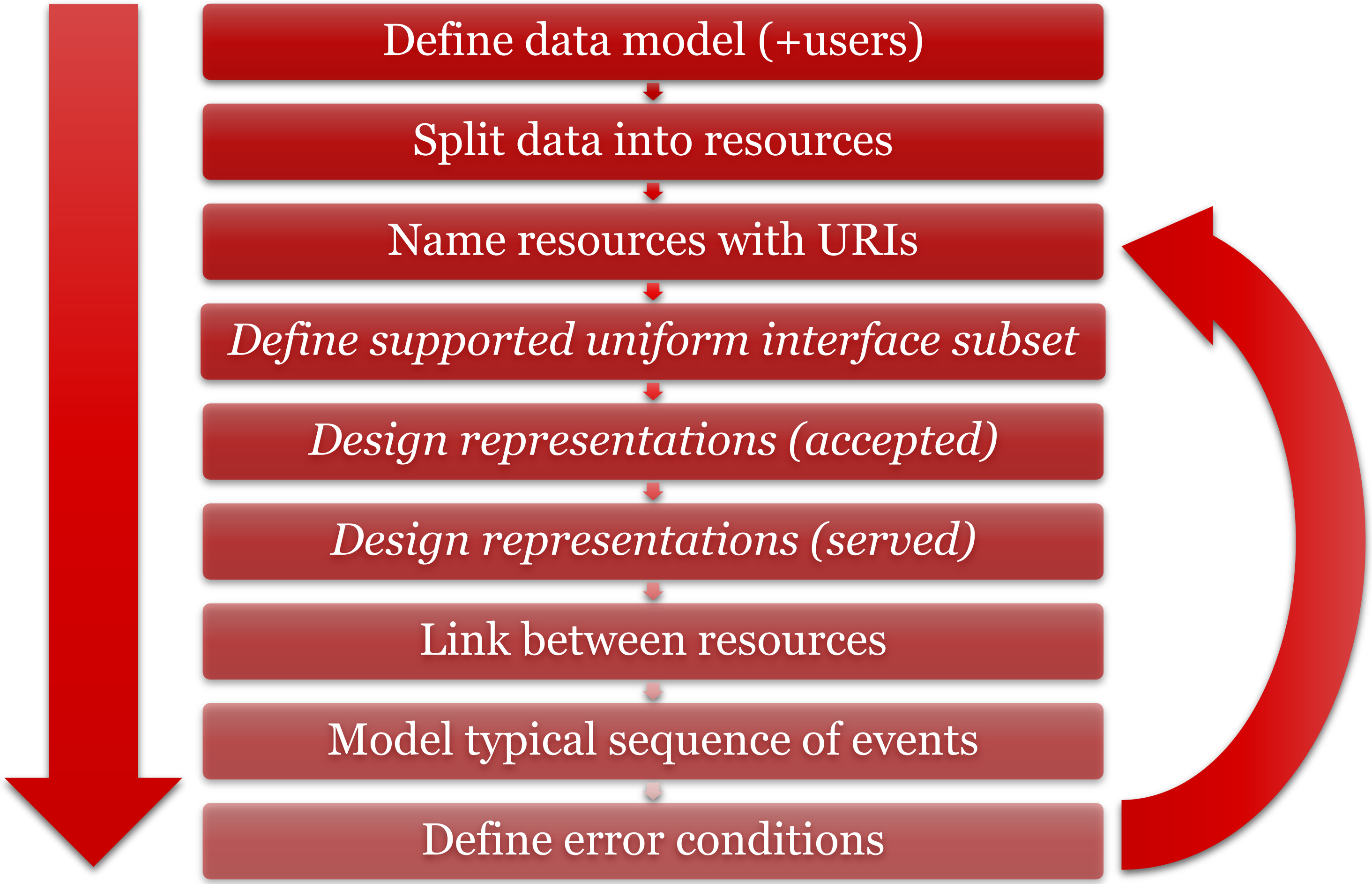
Link between resources

Model typical sequence of events

Define error conditions

1. One-off resources like top-level directories
2. A resource for each exposed object
3. Resources representing the results of algorithms

Read/Write resource design



Define data model (+users)

Split data into resources

Name resources with URIs

Define supported uniform interface subset

Design representations (accepted)

Design representations (served)

Link between resources

Model typical sequence of events

Define error conditions

Other guidelines for RESTful API design

- Guidelines available by multiple sources such as
  - Google applicable also to non-REST APIs
  - Microsoft with plenty of examples
  - Zalando (!) leaning heavily on OpenAPI (see next slides)
- Take a look at the “live” collection maintained by Erik Wilde for more
- The ProgrammableWeb offers a wide list of APIs for inspiration

API Specification

API Specification

- Defines in an ideally both machine- and human-readable format:
  - What the API offers
  - How to interact with it (in terms of payload)
- Code generation a common desirable feature
- Documentation generation the main goal
- OpenAPI as the evolution of Swagger

OpenAPI

- Programming-language agnostic interface description language for (RESTful) APIs
  - Current version 3.0.2 (2018)
  - Take a look at the OpenAPI3 map

- Specifications a OpenAPI Specification-compliant documents
  - In JSON or YAML (~~Yet Another Markup Language~~ YAML Ain't Markup Language) format
  - Fixed (name) or (regex) Patterned fields
  - Builds on JSON Schema (!) for data type definitions
  - Single document, or spread across multiple documents with JSON Schema `$ref` fields pointing to each other

Samples

```
{
      "title": "Sample Pet Store App",
      "description": "This is a sample server for a pet store.",
      "termsOfService": "http://example.com/terms/",
      "contact": {
            "name": "API Support",
            "url": "http://www.example.com/support",
            "email": "support@example.com"
      },
      "license": {
            "name": "Apache 2.0",
            "url": "https://www.apache.org/licenses/LICENSE-2.0.html"
      },
      "version": "1.0.1“
}
```

Samples

```
{
  "/pets": {
    "get": {
        "description": "Returns all pets from the system that the user has access to",
        "responses": {
            "200": {
                "description": "A list of pets.",
                "content": {
                    "application/json": {
                        "schema": {
                            "type": "array",
                            "items": {
                                "$ref": "#/components/schemas/pet"
                            }
                        }
                    }
                }
            }
        }
    }
  }
}
```

Other specification efforts

- Other standards
  - Web Services Description Language (WSDL) 2.0 (how-to)
- Non-standards
  - json:api
  - Web Resource Modeling Language (WRML)
  - RESTful API Modeling Language (RAML)
  - API Blueprint

- Note: documentation ⊂ specification
  - Almost all back-end frameworks come with documentation generation features
  - Examples of good API documentation

Self-evaluation questions

- Which constraints define the REST architectural style, and how are they related with each other?
- How are the REST style constraints related to its goals?
- What are the maturity levels in Richardson's model? Which REST principles are they related to?
- What are the principles that should govern the design of RESTful APIs?
- What steps should be followed for the design of a RESTful API according to Masse's book?

Source material

- Erik Wilde’s lecture on REST http://dret.net/lectures/web-fall10/rest
- Masse, Mark. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.", 2011.

Next lecture

Architectural concerns