

XJCO3011: Web Services and Web Data



Instructor:

Dr. Guilin Zhao

Computer Science, SWJTU-LEEDS JOINT SCHOOL, Spring 2023

Welcome to XJCO3011!

- Today's lecture
 - Course Administration
 - Instructor and Teaching Assistant
 - Assessment
 - Course Plan
 - Module Overview-Introduction
 - RESTful Web Services
 - Search Engines
 - Linked Data and the Semantic Web
 - HTTP – The Workhorse of the Web

Teaching Staff

- Instructor: Dr. Guilin Zhao, X31415, Phone: 18981960065,
Email: guilinzhaow@swjtu.edu.cn or sjugz@leeds.ac.uk
- Time & Mode: Lecture: Monday, 9:50am – 11:25am, 10 weeks;
Lab: Tuesday, 9:50am – 12:15pm, 7 weeks;
- Office Hours: Wednesday/Thursday, 9:30 am – 10:45 am
or other time by appointment via email;
- TAs: Lilan Peng (PhD student),
Phone: 15102816154, Email: lfpeng@my.swjtu.edu.cn
Lin Chen (Master student),
Phone: 13219045985, Email: 1123364568@qq.com

Grading Policy

- Coursework: 60%
 - Coursework 1: 30%, release/due date: To be announced
 - Coursework 2: 30%, release/due date: To be announced
- Examination: 40%
 - Type: Online Timed Limited Assessment (OTLA)
 - Date: To be announced
 - Time: To be announced

Course Plan – Lecture

Week	Date	Session Number	Topic	Coursework Milestones
1	20 Feb	1&2	Introduction & Principles of HTTP	
2	27 Feb	3&4	Principles of RESTful APIs & The Semantic Gap	
3	6 Mar	5&6	Hypermedia & Django	
4	13 Mar	7&8	More Django & RESTful APIs Design	
5	20 Mar	9&10	Python anywhere & Web Crawling	
6	27 Mar	11&12	Parsing and Tokenization & Link Analysis	
7	3 April	13&14	Indexing & Query Processing	
8	10 April	15&16	Introduction to Linked Data & RDF	
9	17 April	17&18	More RDF & Querying Linked Data with SPARQL	
10	24 April	19&20	SPARQL Examples & Closure	
11	28 April			

Course Plan – Lab

Week	Date	Topic
2	28 Feb	HelloWorld
3	7 Mar	Model Implementation
4	14 Mar	Templates View URL Testing
5	21 Mar	Database
6	28 Mar	BlogAPP
7	4 April	Forms
8	11 April	Custom Model

Outline

- ✓ Course Administration
 - Instructor and Teaching Assistant
 - Assessment
 - Course Plan
- Module Overview-Introduction
 - RESTful Web Services
 - Search Engines
 - Linked Data and the Semantic Web
- HTTP – The Workhorse of the Web

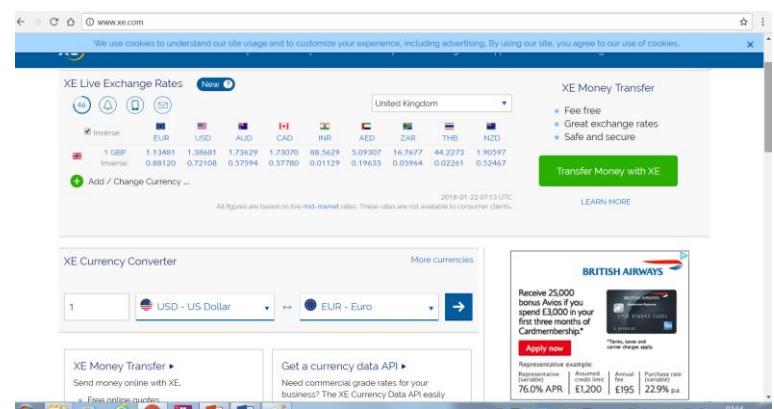
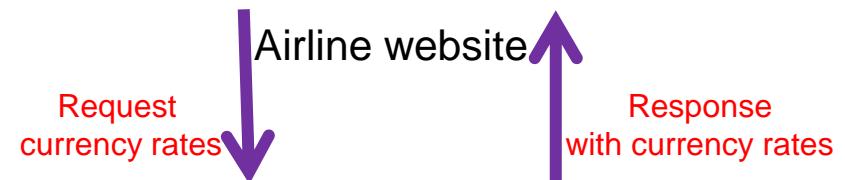
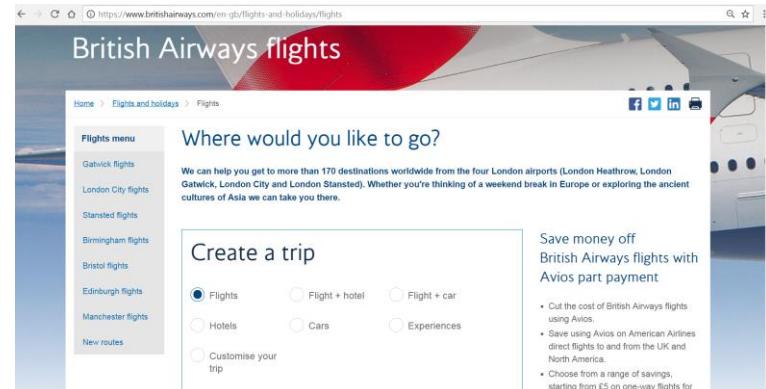
Overview

In this module, we will study the following important and highly related topics:

- RESTful Web Services (8 Units)
- Search Engines (5 Units)
- Linked Data and the Semantic Web (5 Units)

Web Services

- Modern software systems often need to exchange data with each other over the internet. For example, an airline website may require **live currency exchange rates** to compute prices in any desired currency. This data can best be provided by a financial services website.
- A web service is a method of communication that allows two software systems to exchange data over the internet.
- The software system that requests data is called a service **requester**, whereas the software system that would process the request and provides the data is called a service **provider**.



Financial services website

Mashups

- Web services allows us to create mashups.
- A mashup is a software (e.g. web application) that uses content from **more than one source** to create a single new service displayed in a single graphical interface.
- For example, a user could combine the addresses and photographs of library branches with a Google map to create a map mashup.



Web Services Paradigms

Today, we can distinguish two main paradigms for web services:

- SOAP-based services.
- RESTful APIs.

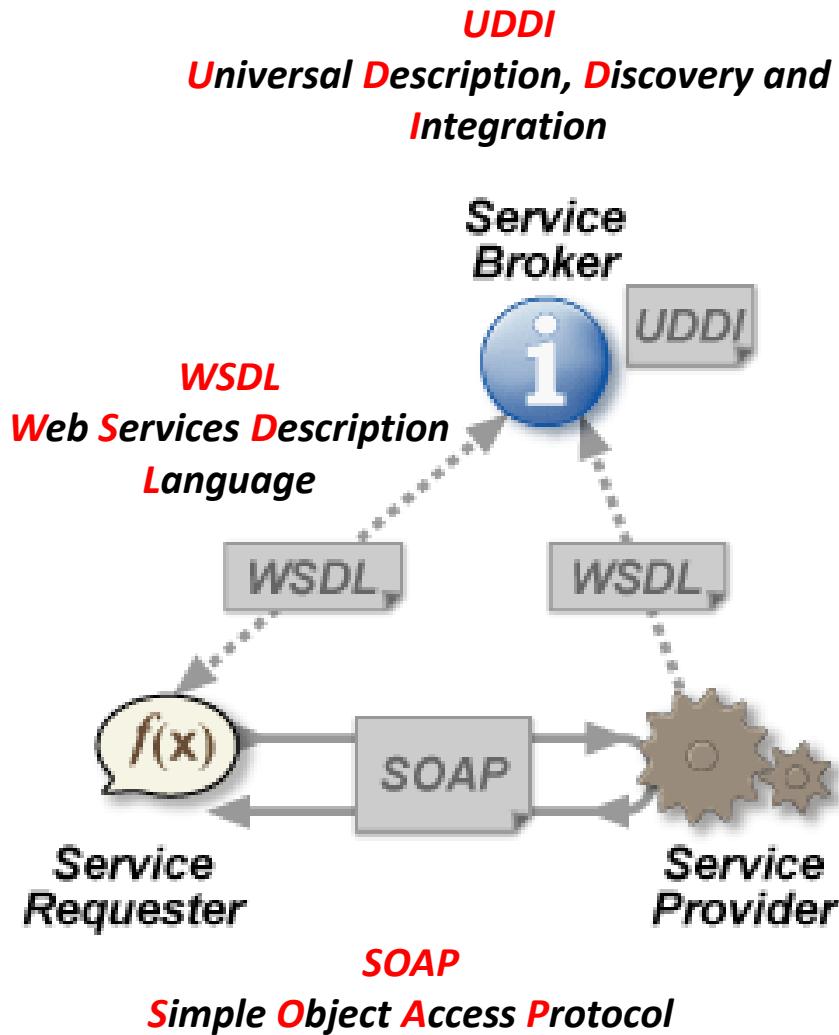
Web Services: The Challenge

- Different software may use **different programming languages, platforms, and operating systems**, hence there is a need for a method of data exchange that doesn't depend upon a particular platform.
- Fortunately, many common data formats, such as **XML** and **JSON**, are recognised by the vast majority of software and hence can be used for data exchange in web services.
- In addition, existing web technology such as **HTTP**, originally designed for human-to-machine communication, can be utilized for machine-to-machine communication, i.e. for transferring machine-readable file formats such as XML and JSON.

SOAP

The term web service originally described methods for integrating web-based applications using a number of open standards, namely: XML, SOAP, WSDL and UDDI, over an Internet Protocol backbone:

- XML is the data format used to contain the data and provide metadata around it.
- SOAP is used to transfer the data.
- WSDL is used for describing the services available.
- and UDDI lists what services are available.
- Don't worry if the above jargon sounds too complex because it is. Fortunately, we are NOT going to use SOAP in this module (you will learn why in a few slides).



SOAP is Complex

In SOAP based services , rules for communication between different systems need to be defined, such as:

- How one system can request data from another system.
- Which specific parameters are needed in the data request.
- What would be the structure of the data produced.
- What error messages to display when a certain rule of communication is not observed.
- All of these rules of communication are defined in a file called **WSDL** (Web Services Description Language)
- A directory called **UDDI** (Universal Description, Discovery and Integration) defines which software system should be contacted for which type of data.
- Once the client finds out which other system it should contact, it would then contact that system using a special protocol called **SOAP** (Simple Object Access Protocol).
- The service provider would first validate the data request by referring to the WSDL file, and then process the request and send the data under the SOAP protocol.

RESTful Web Services (APIs)

- A RESTful Web API is a development in web services where emphasis has been moving to simpler **representational state transfer** (REST) based communications.
- Unlike SOAP-based Web services, restful APIs do **not require** the XML-based web service protocols (SOAP and WSDL) to support their interfaces.
- There is no official standard for RESTful Web APIs. This is because REST is **an architectural style**, while SOAP is a protocol.
- Although REST is not a standard in itself, RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML.
- Unfortunately, many developers also describe their APIs as being RESTful, even though these APIs actually don't fulfil all of the architectural constraints described later.

REST

- Representational state transfer (REST) is a way of providing interoperability between computer systems.
- In a RESTful system, the server does **not maintain state information** about the client. Each request is served on its own merits regardless of any previous requests.
- The client is responsible for maintaining its own state transitions based on the **representations sent by the server** (hence the name of this method).
- REST-compliant services allow requesting systems to access and manipulate **resources** using a uniform and **predefined set of stateless operations**.
- In RESTful APIs, web resources (first defined for the World Wide Web as documents or files identified by URLs) have a more generic definition encompassing **every entity** that can be identified, named, or addressed (such as an object in a data base).
- In a RESTful service, requests made to a resource's URI will trigger a response that may be in XML, HTML, JSON or any other **multimedia** format. The response may **confirm that some alteration has been made to the stored resource**, and it **may provide hypertext links to other related resources** or collections of resources.
- Web APIs **use HTTP as the underlying protocol** for operations including those predefined by the CRUD (Create, Read, Update, and Delete) HTTP methods: POST, GET, PUT, and DELETE, as we will learn in Unit 2.

RESTful Architectural Constraints

There are a number of guiding **constraints** that define a RESTful system. By operating within these constraints, the service gains desirable **non-functional** properties, such as **performance, scalability, simplicity, modifiability, visibility, portability, and reliability**. If a service violates any of the required constraints, it cannot be considered RESTful. This is mainly achieved through 3 main principles:

1. **Client-server architecture** (leading to the separation of concerns). Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms
2. **Statelessness**. The client–server communication is constrained by **no client context being stored on the server** between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client.
3. **Uniform interface**. It simplifies and decouples the architecture, which enables each part to evolve independently.

SOAP vs REST

- SOAP is a protocol.
- SOAP defines standards to be strictly followed.
- SOAP requires more bandwidth and resources than REST.
- SOAP defines its own security.
- SOAP permits the XML data format only.
- SOAP can support stateful implementations.
- SOAP based reads cannot be cached.
- **SOAP is COMPLEX.**

- REST is an architectural style
- REST does not define strict standards like SOAP.
- REST requires less bandwidth and resources than SOAP.
- RESTful web services inherits security measures from the underlying transport layer.
- REST permits different data formats such as plain text, HTML, XML, JSON etc.
- REST follows the stateless model
- REST has better performance and scalability.
- REST reads can be cached.
- JSON usually parses much faster than XML
- **REST is SIMPLE.**

Battle of the Services SOAP against REST

- For some time during the last decade, there were heated **arguments between proponents of SOAP and REST**.
- Today, the battle is almost over, and **REST has won**.
- It is estimated that more than **70% of services available today use the REST architecture**.
- Major providers of cloud services, such as **Amazon** and **Google**, have now adopted the RESTful model, and they are phasing out their support for SOAP-based interfaces.
- SOAP is still preferred when higher levels of security are required.



SOAP



RESTful

Overview

In this module, we will study the following important and highly related topics:

- ✓ RESTful Web Services (8 Units)
- Search Engines (5 Units)
- Linked Data and the Semantic Web (5 Units)

Search Engines

- A web search engine is a software system designed to **search for information on the World Wide Web**.
- Search engines extract information from a **wide variety of data sources** such as HTML web pages, pdf files, MS Word docs, images, videos, and many other types of files.
- Some search engines can also **mine data** from **databases**
- Unlike **web directories**, which are **maintained by human editors**, search engines obtain real-time data by continuously running a **web crawler**.
- However, search engines cannot discover everything on the web. Web content that cannot be discovered by search engines is called the **deep web**.



The Deep Web

- The **deep web** (also called the invisible or hidden web) is that part of the World Wide Web whose contents cannot be indexed by web search engines.
- The content of the deep web **is hidden** behind **HTML forms**, web mail, online banking services, **paid services**, or services protected by a paywall, such as video on demand, some online magazines and newspapers, and many more.
- Content of the deep web can only be located and accessed by a **direct URL or IP address**, and may **require password** or other security protection beyond the public website page.



Size of the Deep Web

- It is **difficult to estimate** the size of the deep web.
- However, some estimates suggest that it is **several orders of magnitude** larger than the surface web.
- An **iceberg** analogy depicts the proportions of the surface web to the deep web.



How Search Engines Work

A search engine maintains the following processes in near real time:

- Web crawling
- Indexing
- Searching

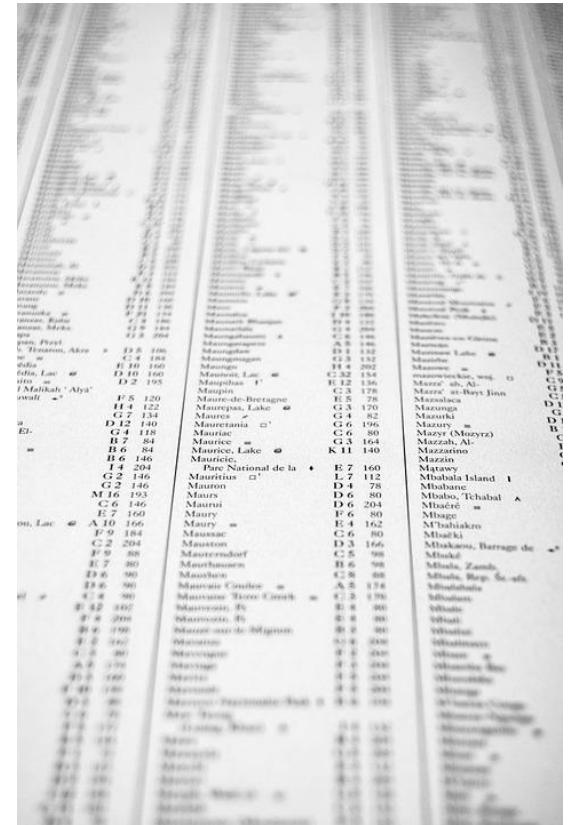
Web Crawling

- Search engines get their information by **crawling** from one web page to another (like a spider).
- The spider follows the **hyperlinks** in each page to discover other web pages.
- Yet, a web spider cannot actually crawl the entire reachable web, due to **infinite websites, spider traps, link rot**, and other factors,
- Crawlers instead apply a **crawl policy** to determine when the crawling of a site should be deemed sufficient.
- Some sites are crawled **exhaustively**, while others are crawled only **partially**.



Indexing

- Indexing means **associating words** found on web pages to their URLs and HTML-based fields.
- The information to be indexed depends on many factors, such as the titles, page content, JavaScript, Cascading Style Sheets (CSS), headings, or **metadata** in HTML meta tags.
- The associations are made in a **public database**, made available for web search queries.
- The index helps find information relating to the query as quickly as possible.



Searching

- When a user enters a query into a search engine, the index already has the URLs containing the keywords, and these are instantly obtained from the index.
- The real processing load is in **generating the list of search results of web pages**, i.e. every page in the list must be weighted according to information in the indices.
- Then search results requires the lookup, reconstruction, and markup of the **snippets** showing the context of the keywords matched.



Overview

In this module, we will study the following important and highly related topics:

- ✓ RESTful Web Services (8 Units)
- ✓ Search Engines (5 Units)
- Linked Data and the Semantic Web (5 Units)

Structure of the Traditional World Wide Web

- The WWW is a collection of linked documents that are full of data.
- Many of these documents have **little, if any, structure** imposed on the data (mostly images and free text).
- The data is available in **so many formats** such as HTML, XML, PDF, TIFF, CSV, Excel spreadsheets, embedded tables in Word documents, and many forms of plain text.
- This kind of data has a limitation: it's formatted for **human consumption**.
- It often requires a specialized utility to read it.
- It's **not easy** for **automated processes** to access, search, or reuse this data.
- Further **processing by people** is generally required for this data to be incorporated into new projects or allow someone to base decisions on it.
- With the exception of some very simple cases, only humans can analyse the semantic relationships between data in various web pages.

The Linked Data Web

- Linked Data refers to a set of techniques for publishing and connecting **structured data** on the Web
- It adheres to standards set by the World Wide Web Consortium (W3C).
- The two images below show how the same information about Star War's character *Darth Vader* can be represented in a human readable form (i.e. natural language) or a machine readable format (called RDF Turtle).

One of the persons is “Anakin,” known as “Darth Vader” and “Anakin Skywalker.” He has a wife named Padme Amidala.

Unstructured Data. Good for Humans

```
@base <http://rosemary.umw.edu/~marsha/starwars/foaf.ttl#> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
@prefix rel: <http://purl.org/vocab/relationship>.  
@prefix stars: <http://www.starwars.com/explore/encyclopedia/characters/> .  
<me> a foaf:Person;  
    foaf:family_name "Skywalker";  
    foaf:givenname "Anakin";  
    foaf:nick "Darth Vader";  
    rel:Spouse_Of <stars:padmeamidala/> .
```

Structured data. Good for Automated Agents

Django

The Practical Sessions

- In the first coursework, we will be using the **Django** framework.
- Django is a free open-source web framework, written in **Python**, which follows the model-view-template (**MVT**) architectural pattern.
- Created by Adrian Holovaty and Simon Willison in 2003, and named after the **French jazz guitarist Django Reinhardt**.
- Django can be used on its own to make a RESTful API, however, the **Django REST Framework** is an extension to the Django framework that has plenty of support for RESTful APIs.



Click to listen to music by Django Reinhardt

Reading List

- Croft W B, Metzler D, and Strohman T (2015), **Search Engines - Information Retrieval in Practice**, Previously Published by Pearson, now available as a free e-book from
<http://ciir.cs.umass.edu/downloads/SEIRiP.pdf>
- George N, **Mastering Django: Core** (The Django Book) (2016), available as a free online book here <https://djangobook.com/the-django-book/>
- Hillar G C, **Building RESTful Python Web Services** (2016), Packt Publishing, Chapters 1-4
- Richardson L and Amundsen M (2013), **RESTful Web APIs**, O'Reilly Media (available online through Safari Books Online)
- Totty B et al, **HTTP The Definitive Guide** (2009), O'Reilly Media (available online through Safari Books Online)
- Wood D et al (2013), **Linked Data - Structured data on the Web**, Manning Publications, Chapters 1-2

XJCO3011: Web Services and Web Data



Session #2 – HTTP – The Workhorse of the Web

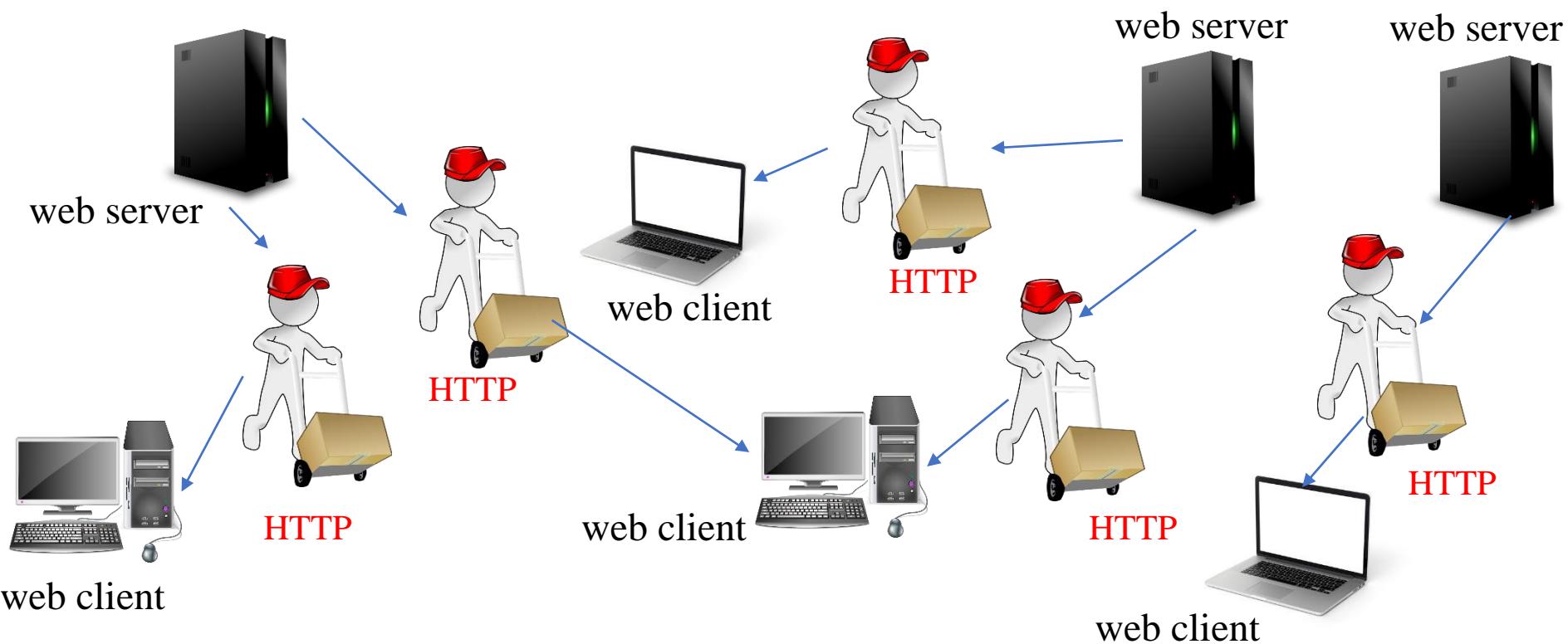
Instructor: Guilin Zhao
Spring 2023

The Hypertext Transfer Protocol (HTTP)

The Multimedia Courier of the Web



- Every day, **billions of multimedia content** (e.g. JPEG images, HTML pages, text files, MPEG movies, WAV audio files, Java applets) **cruise through the Internet** using the HTTP protocol.
- HTTP **moves information quickly** and reliably from **web servers** all around the world to **web browsers** on people's desktops.
- HTTP is an **application layer protocol** that dates back to 1991, but is still the workhorse of the world wide web.
- A **good understanding** of this protocol is **essential** for building web applications, search engines, and RESTful APIs.





A Quick Introduction to the HTTP Protocol

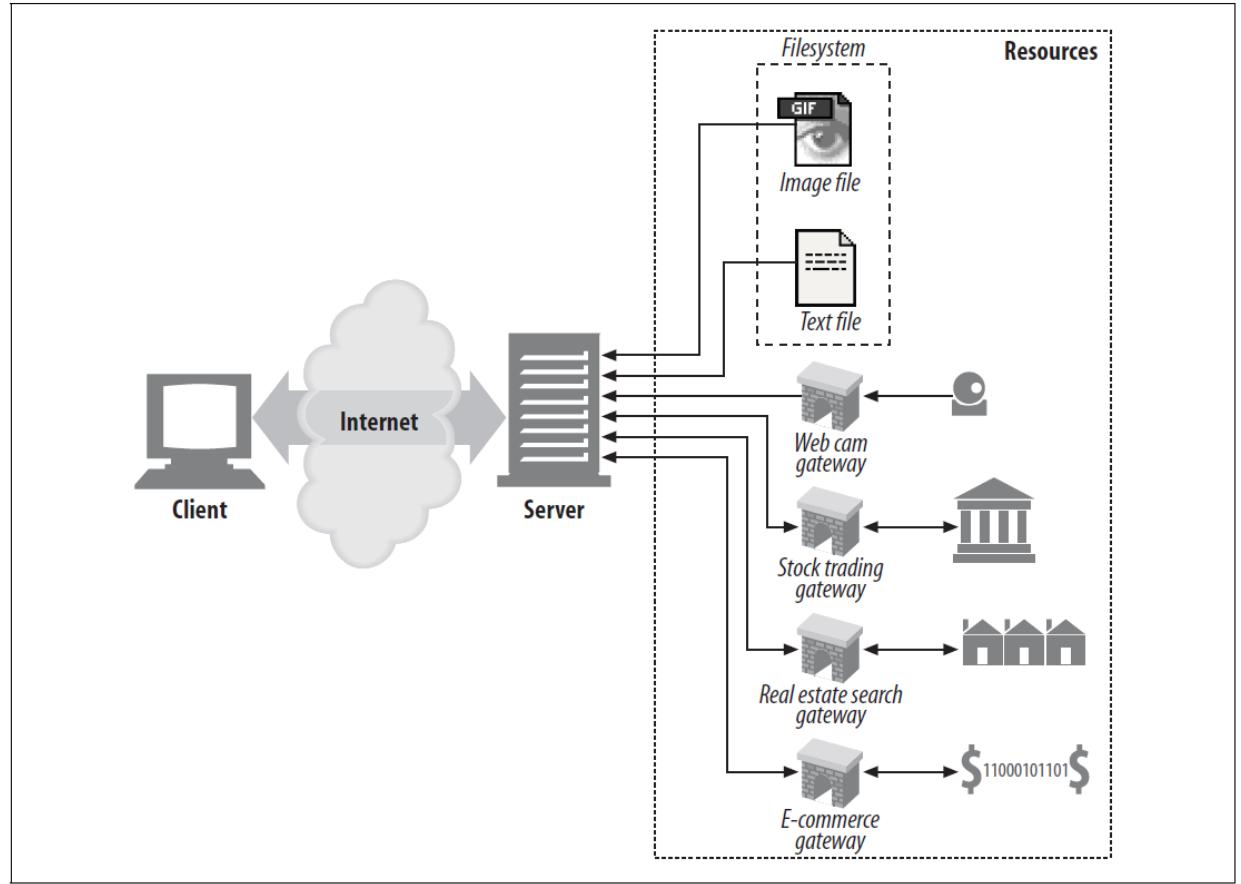
- A protocol is a set of rules and standards that govern the communication between two or more devices or systems, typically in the context of computer networking.
- HTTP is a protocol used to transfer resources over the web.

- HTTP is a protocol used to transfer resources over the web, and it forms the basis of **web resources**.
- Web resources are various types of files, documents, images, videos, audios, and other files that are transmitted from a web server to a client browser using the HTTP protocol.
- Therefore, HTTP and web resources are closely related, with **HTTP providing a standard way to transfer web resources** so that they can be accessed, shared, and utilized on the web.



Web Resources

- Web servers host *web resources*.
- A web resource is the source of web content.
- The simplest kind of web resource is a **static file** on the web server's filesystem (text, HTML, Microsoft Word, Adobe Acrobat, JPEG image, AVI movie, .. etc.).
- However, resources do **NOT** have to be static files. Resources can also be software programs that generate content on demand.

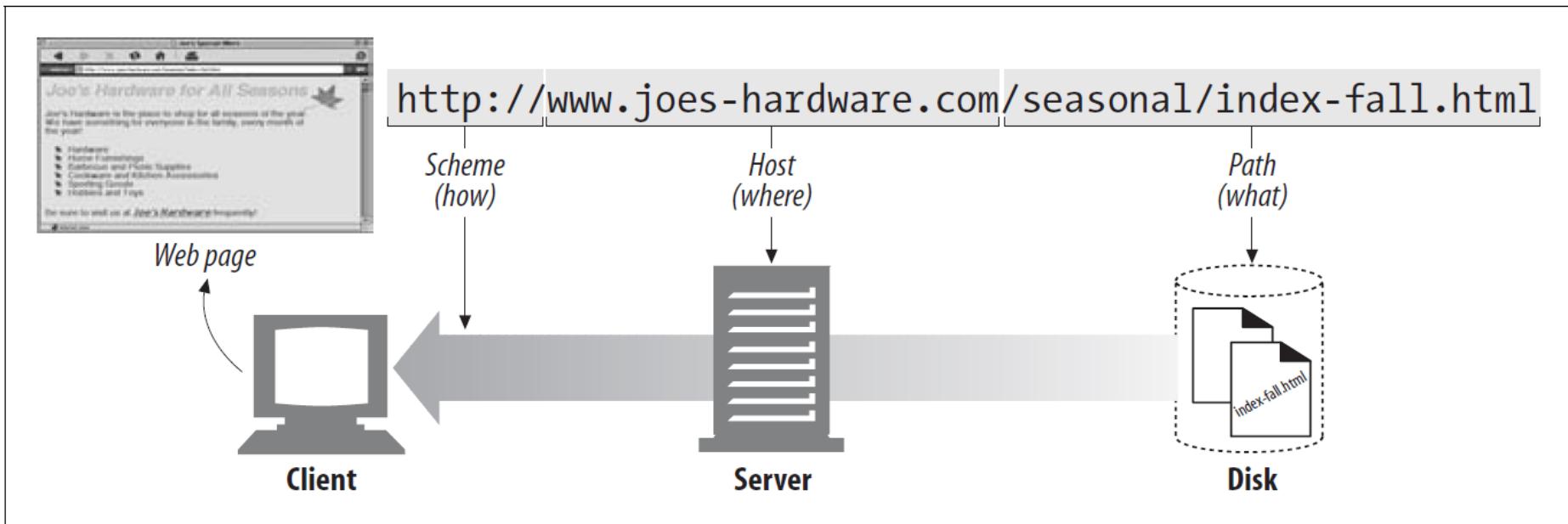


URIs

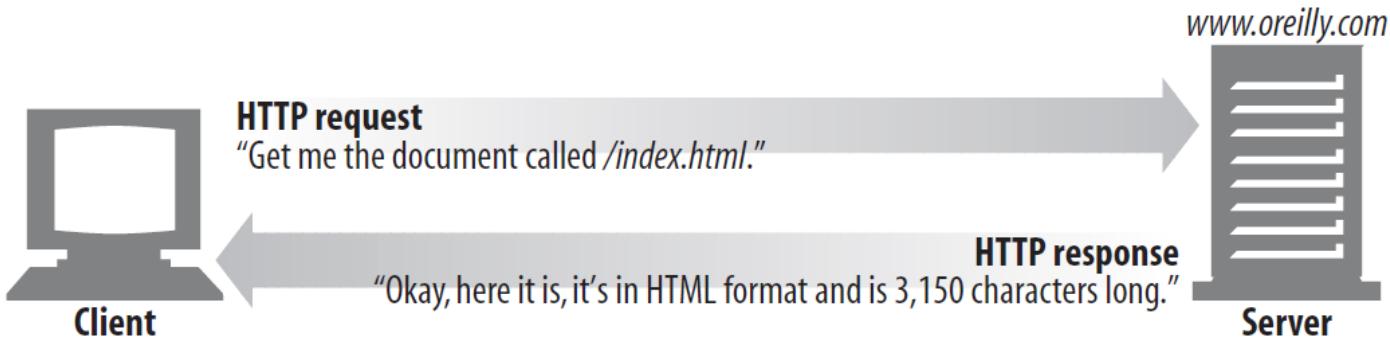
- The **HTTP protocol** defines the rules and format of communication between a client browser and a server, allowing the client browser to request **web resources** and receive them.
- Web resources are typically identified by a Uniform Resource Identifier (URI), and **the client browser uses the URI to send an HTTP request to the server**, which responds by returning the appropriate web resource.

URIs

- Each web resource has a name, so clients can point out what resources they are interested in.
- The resource name is called a *uniform resource identifier*, or **URI**.
- URIs have the general form: “scheme://server location/path”



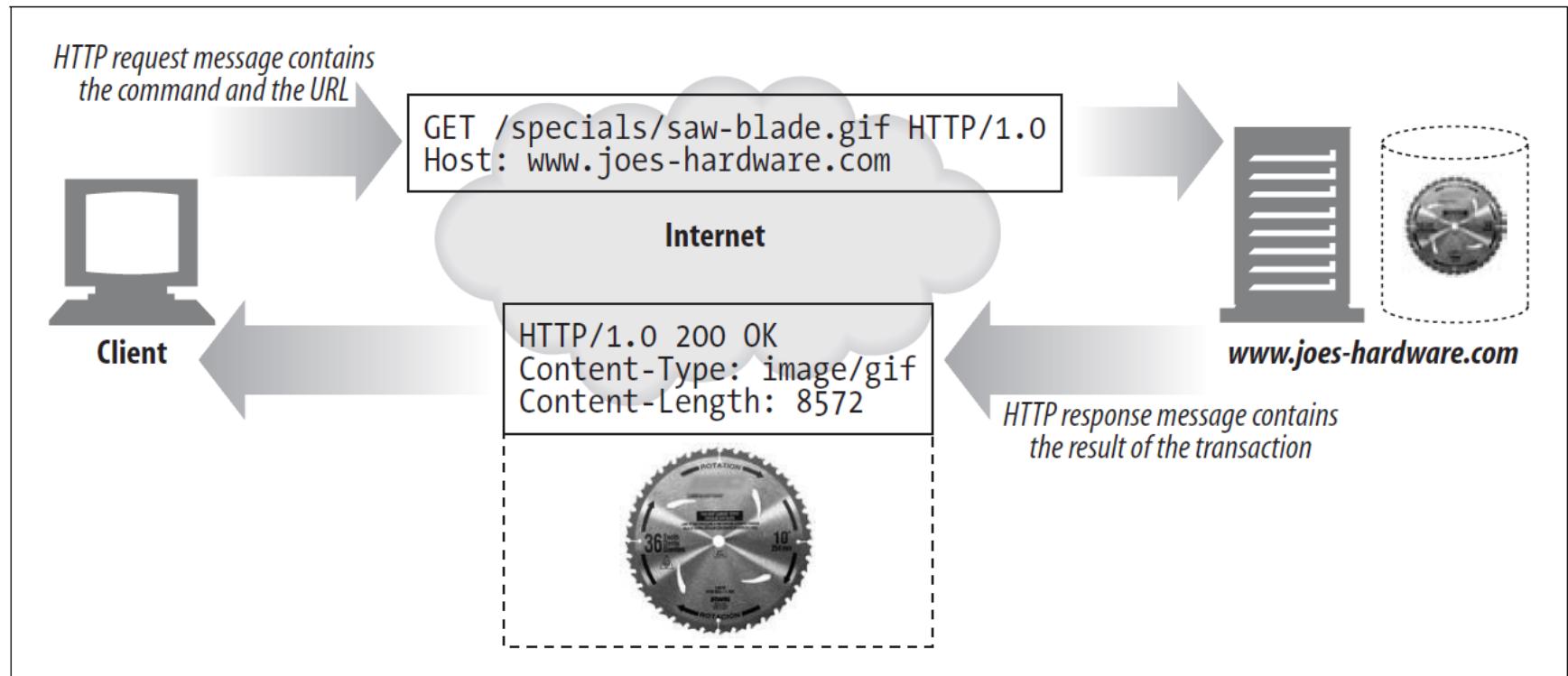
HTTP uses a Client Server Model



HTTP Message Types

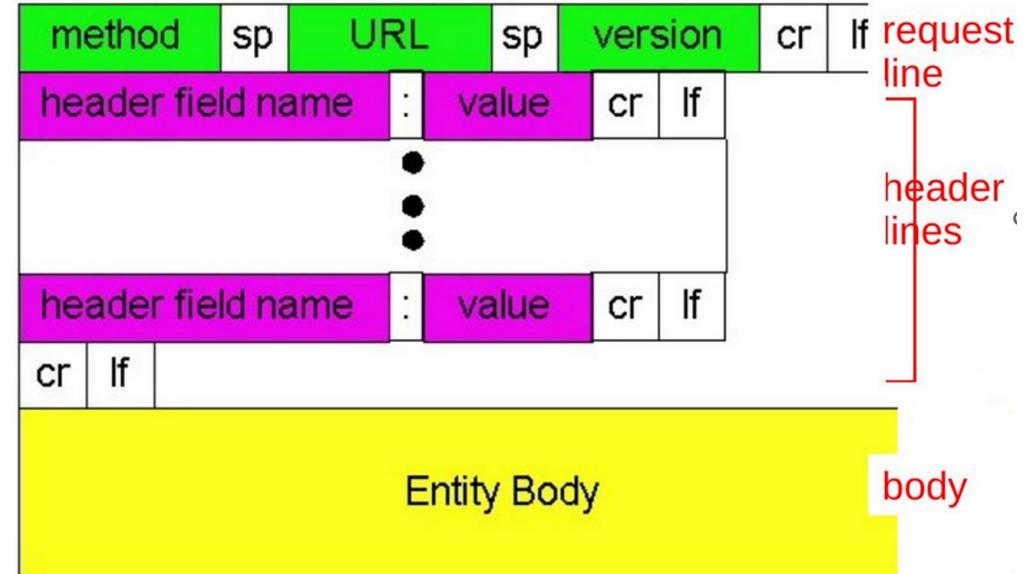
All HTTP messages fall into two types: **request messages** and **response messages**.

- Request messages **request an action from a web server**.
- Response messages **carry results of a request back to a client**.
- Both request and response messages have the same basic message structure

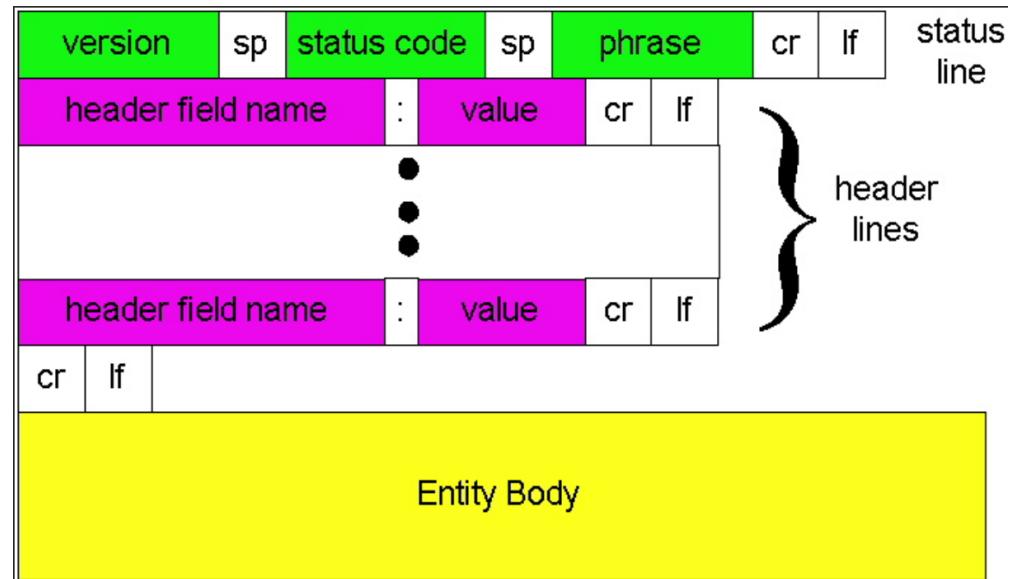


HTTP Message: General Format

- Request messages



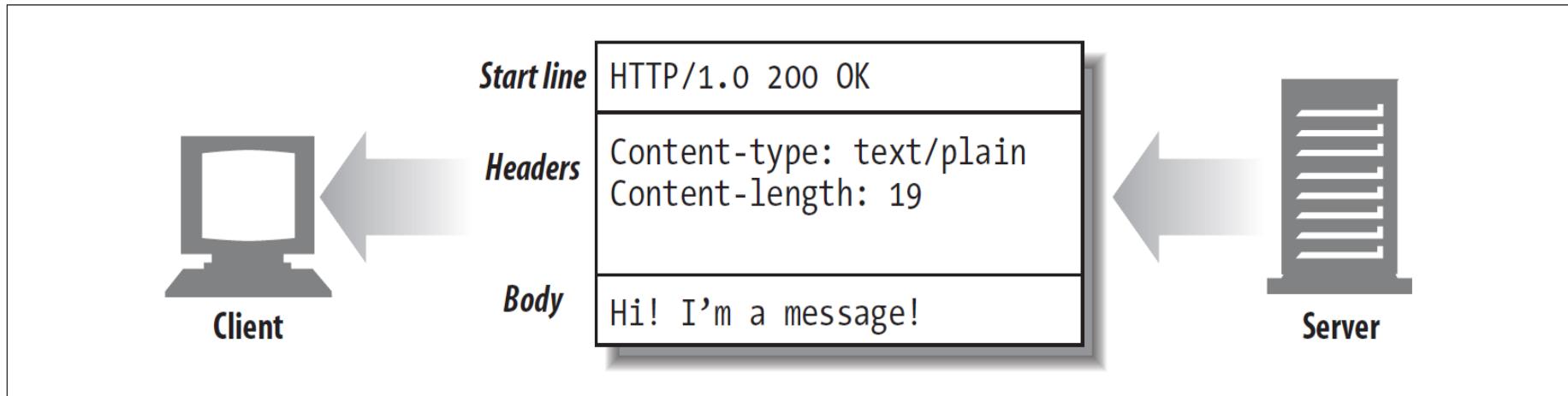
- Response messages



The Parts of an HTTP Message

- A **start line** describing the message.
- A **block of headers** containing some attributes.
- An **optional body** containing data.

Each line ends with a two-character end-of-line sequence, consisting of a carriage return (ASCII 13) and a line-feed character (ASCII 10), called **CRLF** (or `\r\n` in C like languages).



Format of a Request Message

<method> <request-URL> <version>
<headers>
<entity-body>

```
GET /test/hi-there.txt HTTP/1.1
Accept: text/*
Host: www.joes-hardware.com
```

Example

- <method> The action that the client wants the server to perform on the resource. It is a single verb such as “GET,” “HEAD,” or “POST”
- <request-URL> A complete URL naming the requested resource, or the path component of the URL.
- <version> The version of HTTP that the message is using. Its format looks like: HTTP/<major>.<minor>
- <headers> Zero or more headers, each of which is a name, then a colon (:), then a value.
- Headers are terminated by a blank line (CRLF), marking the end of headers and the beginning of the entity body.
- <entity-body> The entity body contains a block of arbitrary data. Not all messages contain entity bodies, so sometimes a message terminates with a bare CRLF.

Format of a Response Message

```
<version> <status> <reason-phrase>
<headers>
<entity-body>
```

HTTP/1.0 200 OK

Content-type: text/plain
Content-length: 19

Hi! I'm a message!

Example

- **<status>** A **three-digit number** describing what happened during the request. Remember, for example, the **notorious 404** (Not Found) status code.
- The first digit describes the general class of status (“success,” “error,” etc.).
- **<reason-phrase>** A **human-readable version** of the numeric status code, consisting of all the text until the end-of-line sequence.



Status Code Classes

Overall range	Defined range	Category
100-199	100-101	Informational
200-299	200-206	Successful
300-399	300-305	Redirection
400-499	400-415	Client error
500-599	500-505	Server error

Examples

Status code	Reason phrase	Meaning
200	OK	Success! Any requested data is in the response body.
401	Unauthorized	You need to enter a username and password.
404	Not Found	The server cannot find a resource for the requested URL.

The Header Lines

Each HTTP header has a simple syntax: a name, followed by a colon (:), followed by the field value, followed by a CRLF

Examples

Header example	Description
Date: Tue, 3 Oct 1997 02:16:03 GMT	The date the server generated the response
Content-length: 15040	The entity body contains 15,040 bytes of data
Content-type: image/gif	The entity body is a GIF image
Accept: image/gif, image/jpeg, text/html	The client accepts GIF and JPEG images and HTML

Header Types

- **General headers** used by both clients and servers. For example, the Date header:

Date: Tue, 3 Oct 1974 02:16:00 GMT

- **Request headers** specific to request messages. They provide extra information to servers, such as **what type of data** the client is willing to receive. For example, to accept any media type we use:

Accept: */*

- **Response headers** provide information to the client. For example, to tell the client that it is talking to a Version 1.0 Tiki-Hut server:

Server: Tiki-Hut/1.0

- **Entity headers** refer to headers that deal with the entity body. For example, the following Content-Type header lets the application know that the data is an HTML document in the iso-latin-1 character set:

Content-Type: text/html; charset=iso-latin-1



Media Types

- In HTTP, Media types are used to indicate the type of content being transferred in the body of an HTTP message. In the HTTP protocol, when a client makes a request to a server, the server responds with an HTTP message that contains a response body, which can be any type of data, such as HTML documents, images, audio, video, and so on. **Media types are used to identify these different types of data.**
- The use of media types in HTTP provides a standardized way to represent data types, which allows different clients and servers to interact with each other. This standardization ensures that **data is correctly parsed and processed, ensuring the reliability and interoperability of data transfer.**
- Additionally, media types can **guide web browsers in displaying different content types**, such as rendering HTML files as web pages and PDF files as documents.

- In HTTP, each response message includes a "**Content-Type**" header field, which specifies the media type of the response message body.
- The Content-Type field value is made up of a **MIME (Multipurpose Internet Mail Extensions)** type and an optional MIME subtype, such as "text/html" or "image/jpeg". The client uses the Content-Type field value to determine how to handle the data in the response message body.
- MIME was **originally designed** for moving messages between different **electronic mail** systems. MIME worked very well, so HTTP adopted it to describe and label its own multimedia content.





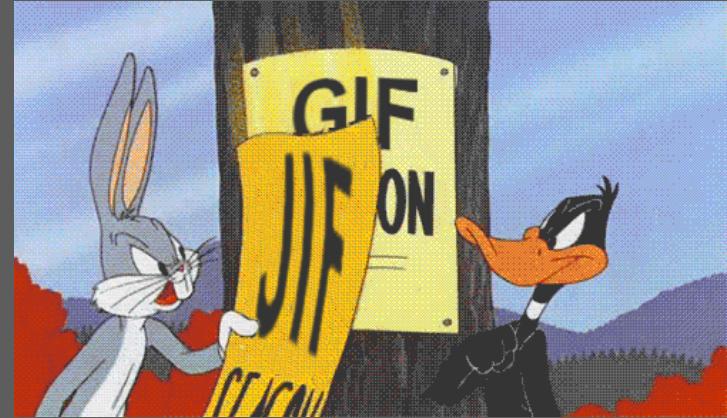
MIME types structure

- Each MIME media type consists of a **primary type**, a **subtype**, and a list of optional parameters.
- The type and subtype are separated by a **slash**, and the optional parameters begin with a **semicolon**.
- The primary type can be a predefined type, an IETF-defined (Internet Engineering Task Force) extension token, or an experimental token (beginning with “x-”). Here are a few examples:

Examples	Type	Description
	application	Application-specific content format (discrete type)
	audio	Audio format (discrete type)
	chemical	Chemical data set (discrete IETF extension type)
	image	Image format (discrete type)
	message	Message format (composite type)
	model	3-D model format (discrete IETF extension type)
	multipart	Collection of multiple objects (composite type)
	text	Text format (discrete type)
	video	Video movie format (discrete type)

MIME type examples:

- An HTML-formatted text document would be text/html.
- A plain ASCII text document would be text/plain.
- A JPEG version of an image would be image/jpeg.
- An Apple QuickTime movie would be video/quicktime.
- A Microsoft PowerPoint presentation would be application/vnd.ms-powerpoint.
- A GIF-format image would be image/gif.



MIME Type IANA Registration

- MIME types should be registered with **IANA** (Internet Assigned Numbers Authority)
- Registration is simple and open to all.
- MIME type tokens are split into **four classes**, called “**registration trees**,” each with its own registration rules.

Registration tree	Example	Description
IETF	text/html (HTML text)	The IETF tree is intended for types that are of general significance to the Internet community. New IETF tree media types require approval by the Internet Engineering Steering Group (IESG) and an accompanying standards-track RFC. IETF tree types have no periods (.) in tokens.
Vendor (vnd.)	image/vnd.fpx (Kodak FlashPix image)	The vendor tree is intended for media types used by commercially available products. Public review of new vendor types is encouraged but not required. Vendor tree types begin with “vnd.”
Personal/Vanity (prs.)	image/prs.btif (internal check-management format used by Nations Bank)	Private, personal, or vanity media types can be registered in the personal tree. These media types will not be distributed commercially. Personal tree types begin with “prs.”
Experimental (x- or x.)	application/x-tar (Unix tar archive)	The experimental tree is for unregistered or experimental media types. Because it's relatively simple to register a new vendor or personal media type, software should not be distributed widely using x- types. Experimental tree types begin with “x.” or “x-”.

Common HTTP Methods (verbs)

Method	Description	Message body?
GET	Get a document from the server.	No
HEAD	Get just the headers for a document from the server.	No
POST	Send data to the server for processing.	Yes
PUT	Store the body of the request on the server.	Yes
TRACE	Trace the message through proxy servers to the server.	No
OPTIONS	Determine what methods can operate on a server.	No
DELETE	Remove a document from the server.	No

Note that not all methods are implemented by every server. To be compliant with HTTP Version 1.1, a **server need implement only the GET and HEAD methods for its resources.**

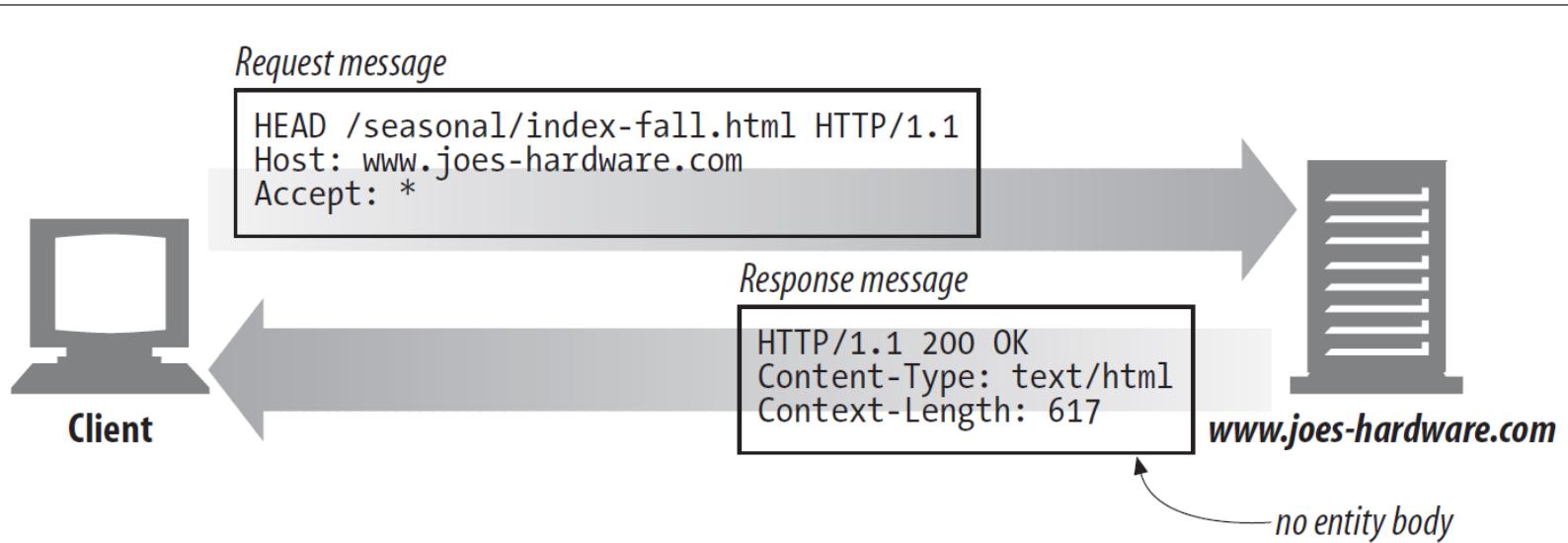
GET

GET is the most common method. It usually is used to ask a server to send a resource.



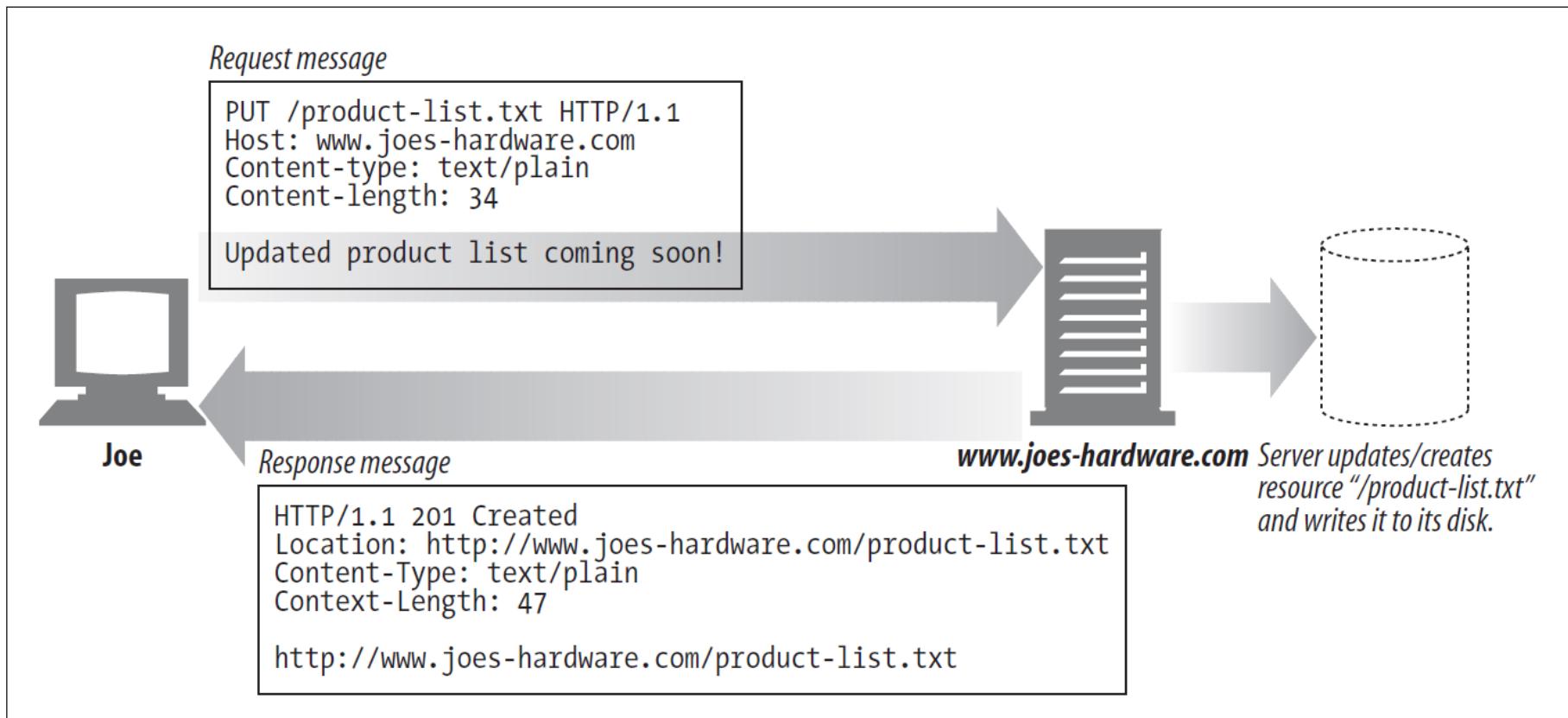
HEAD

- The HEAD method behaves exactly like the GET method, but the server returns only the headers in the response. **No entity body is ever returned.**
- This allows a client to inspect the headers for a resource without having to actually get the resource.
- Using HEAD, you can:
 - Find out about a resource (e.g., determine its type) without getting it.
 - See if an object exists, by looking at the status code of the response.
 - Test if the resource has been modified, by looking at the headers (Last-Modified).



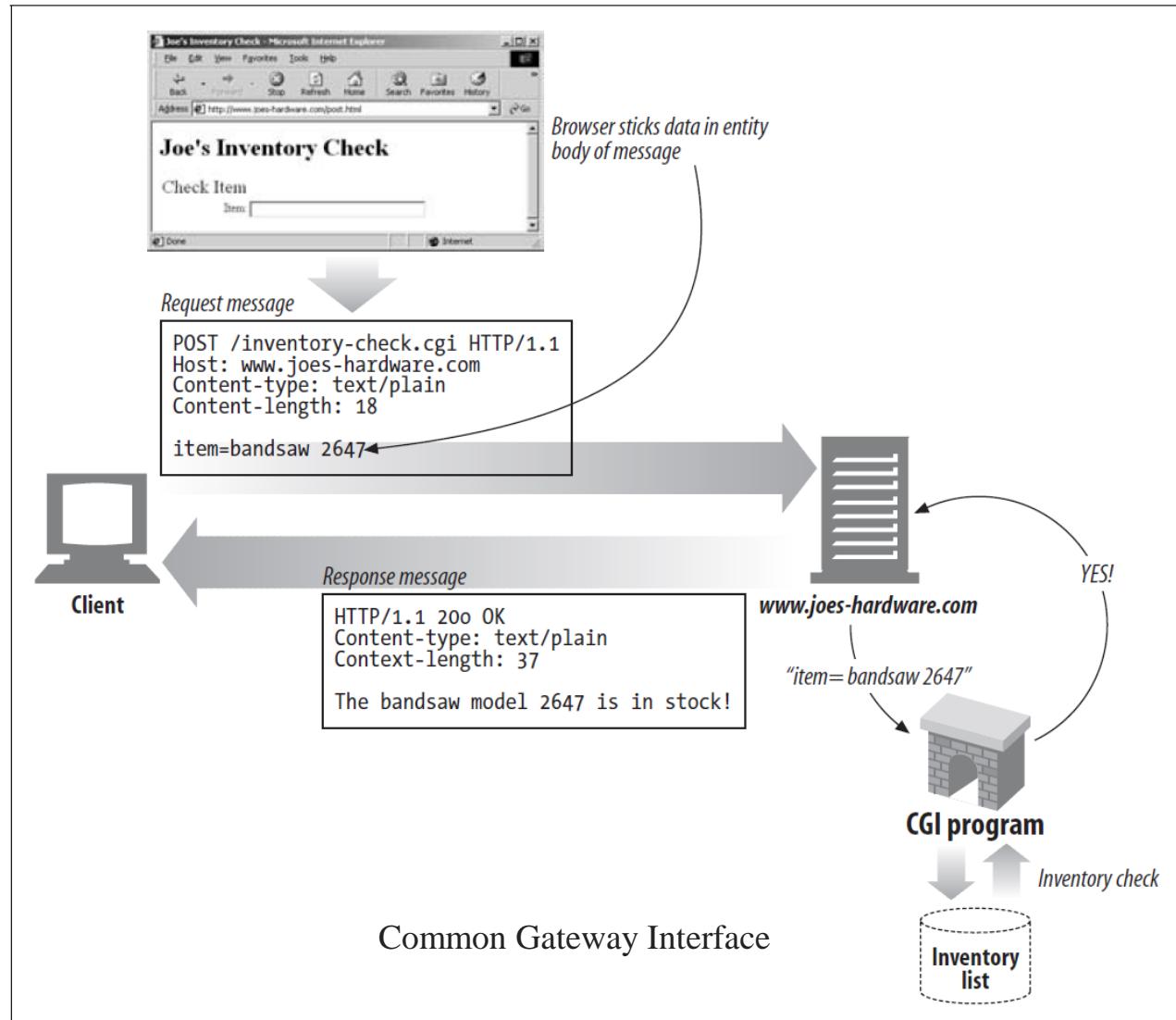
PUT

- The PUT method writes documents to a server, in the inverse of the way that GET reads documents from a server.
- Some publishing systems let you create web pages and install them directly on a web server using PUT



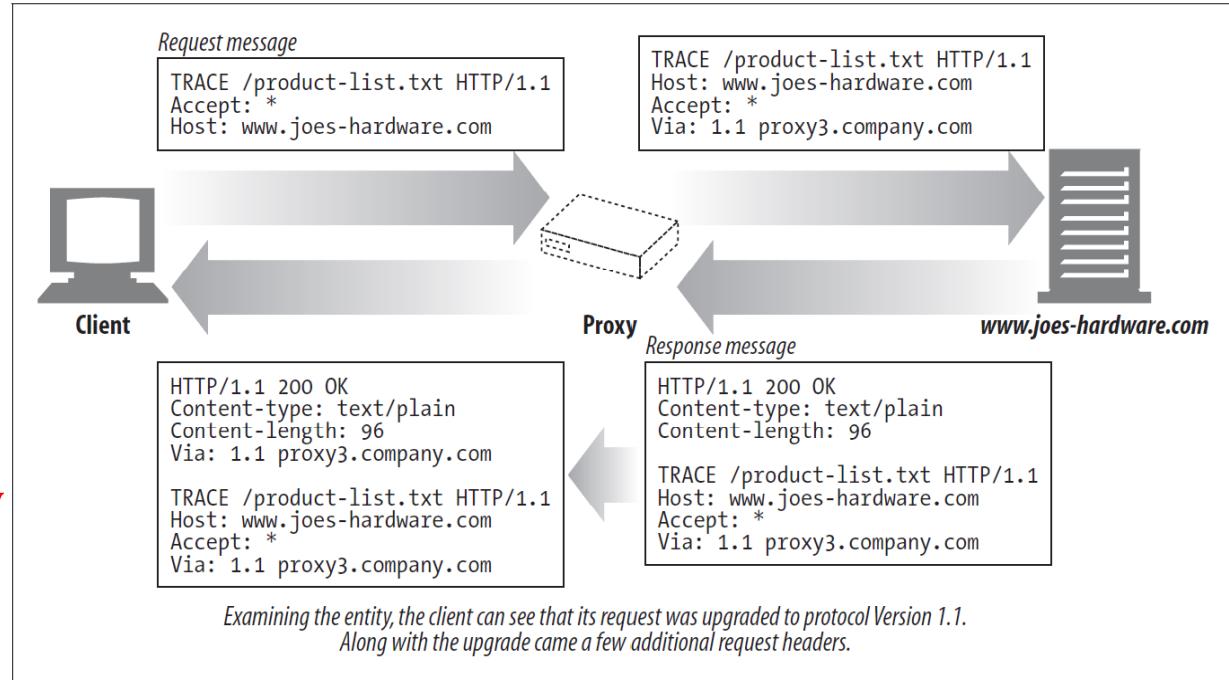
POST

- The POST method sends input data to the server. For example, it is often used to support HTML forms. The data from a filled-in form is sent to the server, which then processes it.



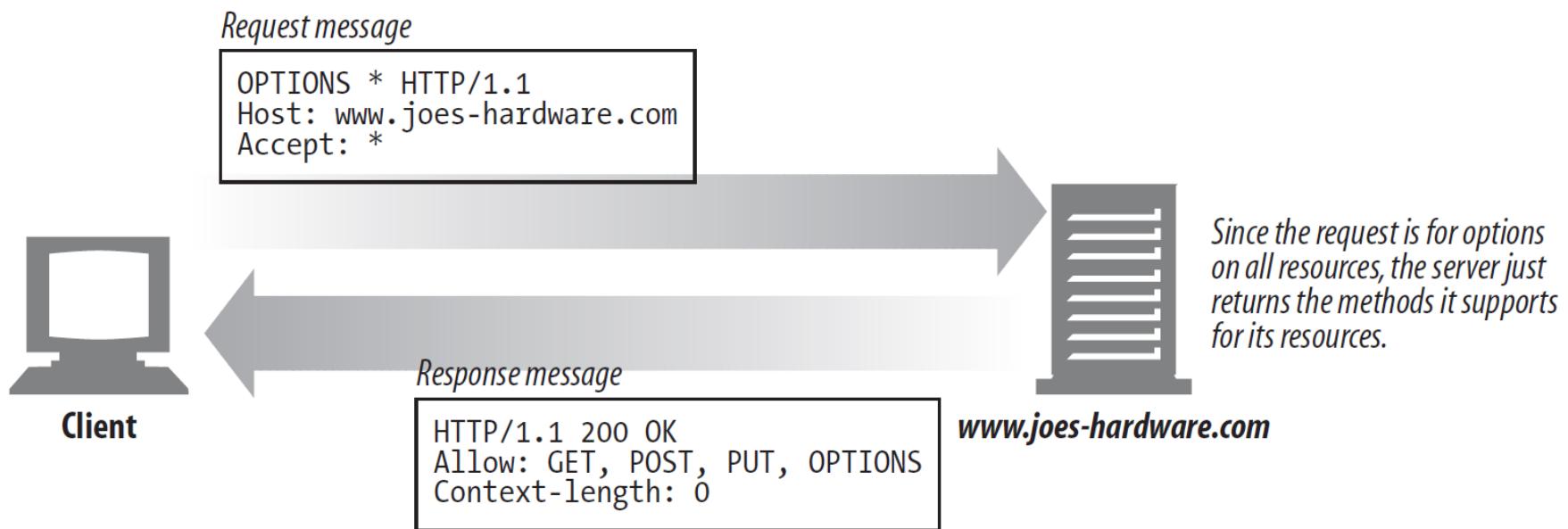
TRACE

- When a client makes a request, that **request may have to travel through firewalls, proxies, gateways, or other applications.**
- Each of these has the opportunity to modify the original HTTP request.
- The TRACE method **allows clients to see how its request looks when it finally makes it to the server.**
- A TRACE request initiates a “**loopback**” diagnostic at the destination server.
- The server at the final leg of the trip bounces back a TRACE response, with the **virgin request message** it received in the body of its response.



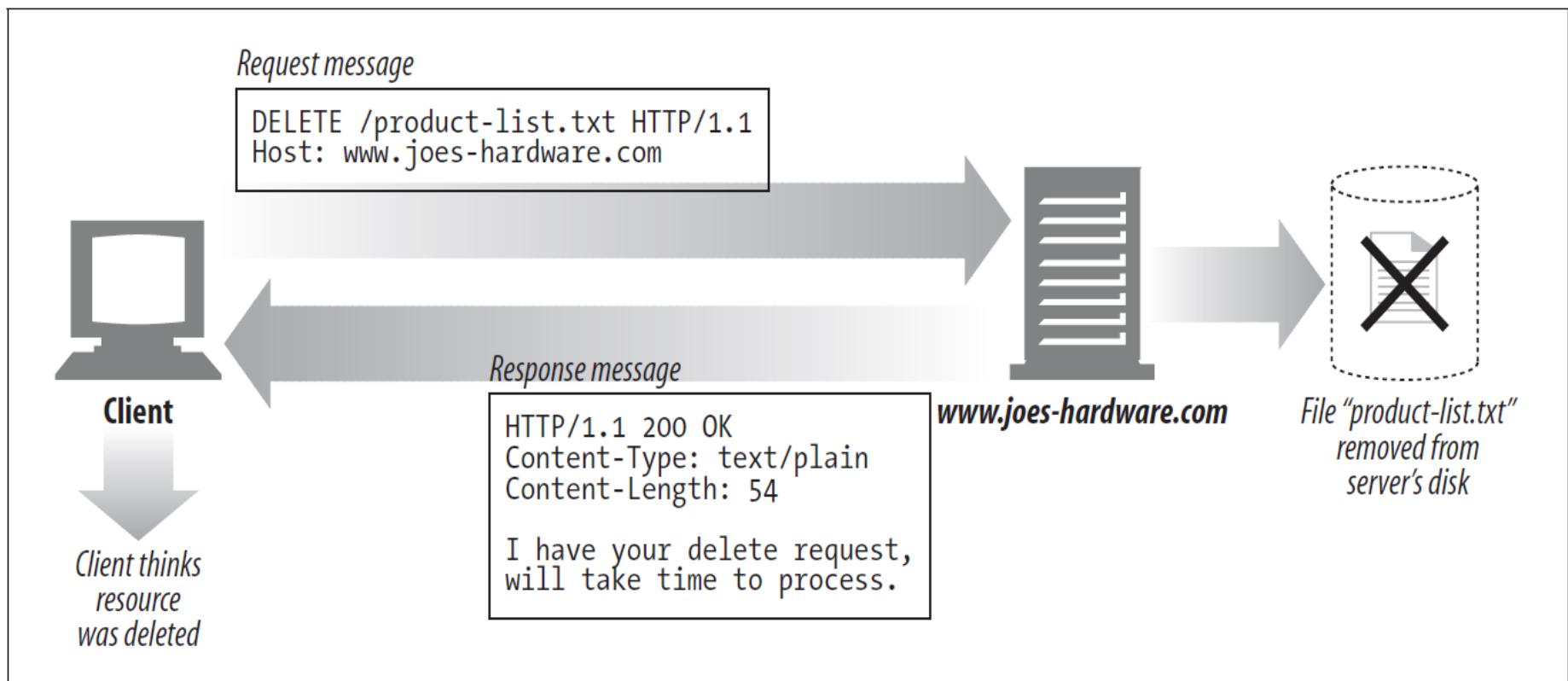
OPTIONS

- The OPTIONS method asks the server to tell us about the **various supported capabilities of the web server**.
- You can ask a server about **what methods it supports in general or for particular resources** (Some servers may support particular operations only on particular kinds of objects).
- This provides a means for client applications to **determine how best to access various resources** without actually having to access them.



DELETE

- The DELETE method asks the server to **delete the resources** specified by the request URL.
- However, the client application is **not guaranteed that the delete is carried out**.
- The HTTP specification allows the server to override the request **without telling the client**.



Extension Methods

- HTTP was designed to be **field-extensible**, so new features wouldn't cause older software to fail.
- Extension methods are **methods that are not defined in the HTTP/1.1 specification**.
- They provide developers with a means of **extending the capabilities of the HTTP** services their servers implement on the resources that the servers manage.

Example: the WebDAV HTTP extension that support publishing of web content to web servers over HTTP.

Method	Description
LOCK	Allows a user to “lock” a resource—for example, you could lock a resource while you are editing it to prevent others from editing it at the same time
MKCOL	Allows a user to create a resource
COPY	Facilitates copying resources on a server
MOVE	Moves a resource on a server

That's it Folks



Further Reading

HTTP: The Definitive Guide by Brian Totty and David Gourley

XJCO3011: Web Services and Web Data



Session #3 – Alice in Webland

Instructor: Dr. Guilin Zhao
Spring 2023

Administrative Issues

- Lab section: 1st Lab: 28 Feb. @ X7304
- TA office hour schedule:
 - ✓ Lilan Peng: Friday: 10:00am – 11:30am @X31404
 - ✓ Lin Chen: Thursday: 10:00am – 11:30am @X9432

Review of Sessions #1 and #2

True or False:

1. SOAP is a protocol, but REST is an architectural style. ()
2. SOAP is simple, REST is complex. ()
3. There are two types of HTTP messages: the request message and the response message. ()

Topic: Surfing the Web



Topic: Surfing the Web

- The World Wide Web became popular because ordinary people can use it to do really useful things with minimal training. But behind the scenes, the Web is also a powerful platform for distributed computing.
- The principles that make the Web usable by ordinary people also work when the “user” is an automated software agent. A piece of software designed to transfer money between bank accounts (or carry out any other real-world task) can accomplish the task using the same basic technologies a human being would use.
- Start off by telling a simple story about the World Wide Web, as a way of explaining the principles behind its design and the reasons for its success
- The story needs to be simple because although you’re certainly familiar with the Web, you might not have heard of the concepts that make it work. I want you to have a simple, concrete example to fall back on if you ever get confused about terminology like “hypermedia as the engine of application state.”



Alice's Story

I am sure you have heard this story
(Alice's Adventures in Wonderland) many
times before, but there are always **new**
lessons to learn from a good story.

- Episode 1: The Billboard
- Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 1: The Billboard

This is the story of Alice, a little girl who was one day walking around town when she saw this billboard



Alice was intrigued by this web address, so she pulled out her mobile phone and put <http://www.youtypeitwepostit.com/> in her browser's address bar.

But, what's at the other end of that URL?



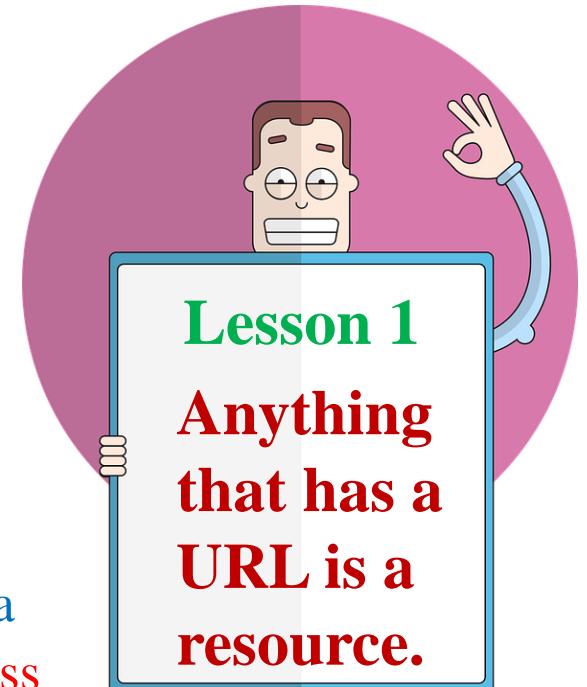
Note: Try the above link, it should work for you as it did work for Alice.

Alice's web browser sent an HTTP request to a web server—specifically, to the URL <http://www.youtypeitwepostit.com/>.

Well, you probably know that the technical term for the thing named by a URL is a **resource**.

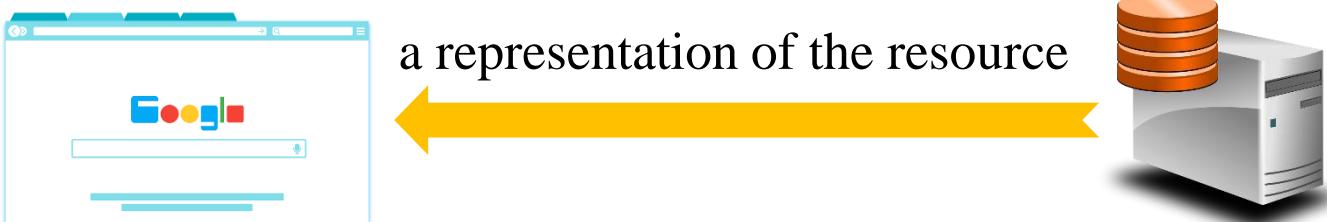


- In the context of HTTP, the term "host" typically refers to the name or IP address of the server that is hosting a particular website or web application. When a client makes a request to a server using HTTP, it typically includes a "Host" header in the request. This header specifies the hostname or IP address of the server that the client is trying to connect to.



the server sent a document in response
(maybe an HTML document, maybe a
binary image or maybe something else).

Whatever document the server sent is
called a **representation** of
the resource.



Lesson 2

When a client makes an HTTP request to a URL, it gets a representation of the underlying resource. The client never sees a resource directly.

GOT IT?

The illustration features a cartoon character with brown hair tied back, wearing a red long-sleeved shirt and blue pants, standing next to a whiteboard. The whiteboard has a blue border and contains the text from the slide. The background behind the character is a large teal circle.

Lesson 3

- A URL identifies **one and only** one resource.
- If an API provides two different functionality, the API should treat them as two resources with different URLs.
- The principle of addressability says **that every** resource should have its own URL.
- If something is important to your application, it should have a unique name, a URL, so that you and your users can refer to it unambiguously.

Got it?

Alice's Story

- ✓ Episode 1: The Billboard
- Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 2: The Home Page

So, when the web server handled this request, it sent this response:

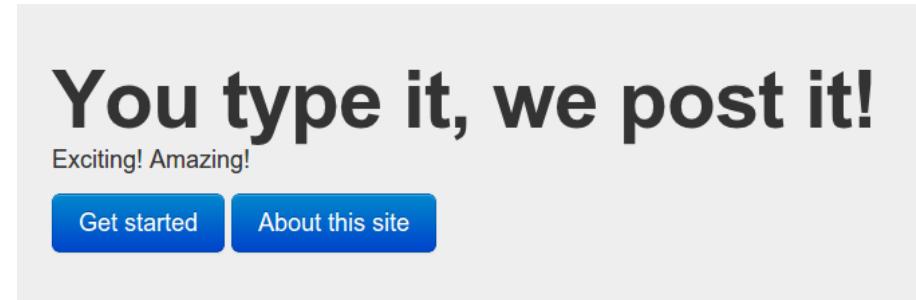
```
HTTP/1.1 200 OK  
Content-type: text/html
```

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Home</title>  
  </head>  
  <body>  
    <div>  
      <h1>You type it, we post it!</h1>  
      <p>Exciting! Amazing!</p>  
      <p class="links">  
        <a href="/messages">Get started</a>  
        <a href="/about">About this site</a>  
      </p>  
    </div>  
  </body>  
</html>
```

- ❖ In HTTP, the "[" tag \(also known as the **hyperlink tag**\) is one of the HTML tags used to create hyperlinks. The "a" tag is used to link a text or an image from one webpage to another webpage or to another part of the same webpage. The "a" tag typically contains an "\[" attribute, which specifies the URL or file path of the target of the link.\]\(#\)](#)
- ❖ For example, the following is an example of using the "a" tag to create a hyperlink:

`< a href=" https://www.example.com ">Click here to visit Example website< /a >`
- ❖ This tag will display a link text "Click here to visit Example website" on the webpage. When the user clicks the link, it will take the user to a website named "https://www.example.com".

Alice's web browser decoded the response as an HTML document and displayed it graphically



Now Alice could read the web page and understand what the billboard was talking about. It was advertising a microblogging site.

She was feeling so sleepy, so she put her phone down and fell asleep. The next morning she woke up and clicked her phone to resume what she was doing the night before.

Lesson 4

Short Sessions

- HTTP sessions last for one request. The client sends a request, and the server responds.
- This means Alice could turn her phone off overnight, and the web server will not sit up all night worrying about Alice. When she's not making an HTTP request, the server doesn't know Alice exists.
- This principle is called **statelessness**. The server doesn't care what state the client is in.

Got it?

The markup for the home page contains two links: one to the relative URL /about and one to /messages.

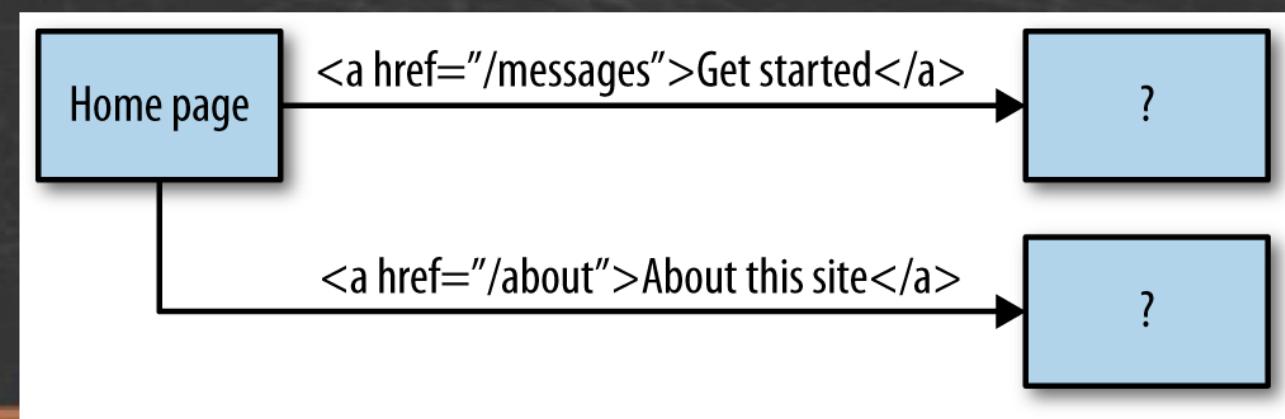
```
HTTP/1.1 200 OK
Content-type: text/html
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <div>
      <h1>You type it, we post it!</h1>
      <p>Exciting! Amazing!</p>
      <p class="links">
        <a href="/messages">Get started</a>
        <a href="/about">About this site</a>
      </p>
    </div>
  </body>
</html>
```

At first Alice only knew one URL—the URL to the home page—but now she knows three. The server is slowly revealing its structure to her.

Lesson 5

Self-Descriptive Messages

When a client requests a resource, the representation (e.g. HTML document) it receives doesn't just give immediate information about this resource. The representation can also help the client decide the next step to make (what state it must go to next).



Alice's Story

- ✓ Episode 1: The Billboard
- ✓ Episode 2: The Home Page
- Episode 3: The Link
- Episode 4: The Form and the Redirect



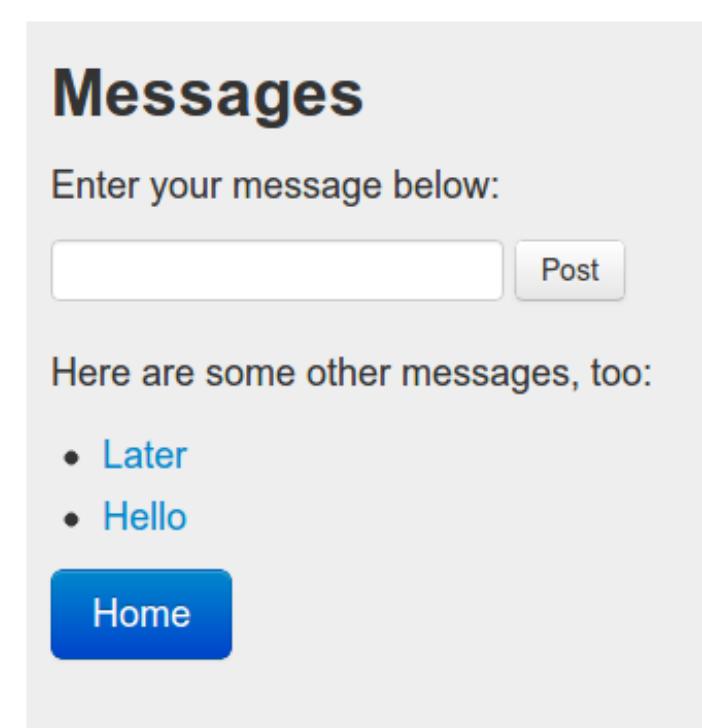
Episode 3: The Link

After reading the home page, Alice decided to give this site a try, so she clicked the link that said “Get started.” and her browser made an HTTP request:

```
GET /messages HTTP/1.1  
Host: www.youtypeitwepostit.com
```

The server handled this particular GET request and sent a representation of /messages,

and Alice’s browser rendered the HTML graphically



Alice's Story

- ✓ Episode 1: The Billboard
- ✓ Episode 2: The Home Page
- ✓ Episode 3: The Link
- Episode 4: The Form and the Redirect



Episode 4: The Form and the Redirect

Alice was tempted by the form, so she typed in “Test” and clicked the Post button.: Again, Alice’s browser made an HTTP request:

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-type: application/x-www-form-urlencoded

message=Test&submit=Post
```



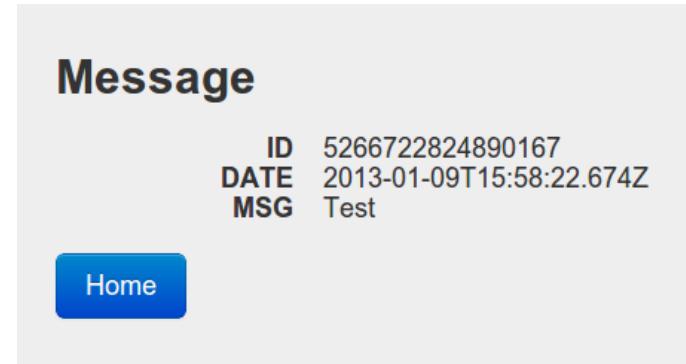
And the server responded with the following:

```
HTTP/1.1 303 See Other
Content-type: text/html
Location:
http://www.youtypeitwepostit.com/messages/5266722824890167
```

Status code 303 told Alice’s browser to automatically make a fourth HTTP request, to the URL given in the Location header. Without asking Alice’s permission, her browser did just that:

```
GET /messages/5266722824890167 HTTP/1.1
```

This time, the server responded with 200 (“OK”) and an HTML document. Alice’s browser displayed this document graphically



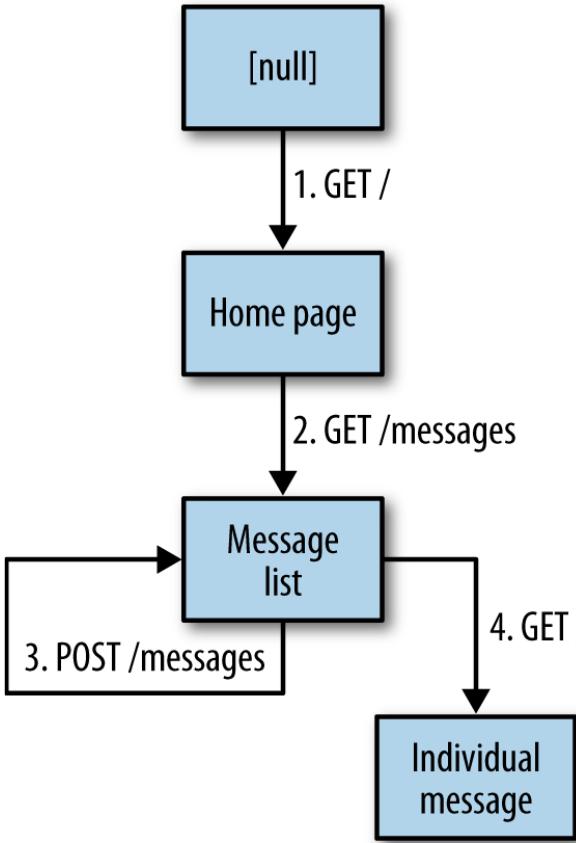
By looking at the graphical rendering, Alice saw that her message (“Test”) is now a fully fledged post on YouTypeItWePostIt.com. Our story ends here—Alice has accomplished her goal of trying out the microblogging site.

Lesson 6

Standardized Methods

- The HTTP standard (RFC 2616) defines eight methods a client can apply to a resource.
- The most frequently used are **GET, HEAD, POST, PUT, DELETE, and OPTION**.
- We distinguish between resources not by defining new methods, but by assigning different URLs to these resources.
- A single URL can exhibit different behaviour depending on the method sent to it.

To Sum up, here is the state diagram that shows Alice's entire adventure from **the perspective of her web browser**.



- ❖ When Alice started up the browser on her phone, it didn't have any particular page loaded. It was an empty slate.
 - Then Alice typed in a URL and a GET request took the browser to the site's home page.
 - Alice clicked a link, and a second GET request took the browser to the list of messages.
 - She submitted a form, which caused a third request (a POST request).
 - The response to that was an HTTP redirect, which Alice's browser made automatically. Alice's browser ended up at a web page describing the message Alice had just created.
- ❖ Every state in this diagram corresponds to a particular page (or to no page at all) being open in Alice's browser window.
 - ❖ In REST terms, we call this bit of information—which page are you on?—**the application state**.
 - ❖ When you surf the Web, every transition from one application state to another corresponds to a link you decided to follow or a form you decided to fill out.
 - ❖ Not all transitions are available from all states. Alice can't make her POST request directly from the home page, because the home page doesn't

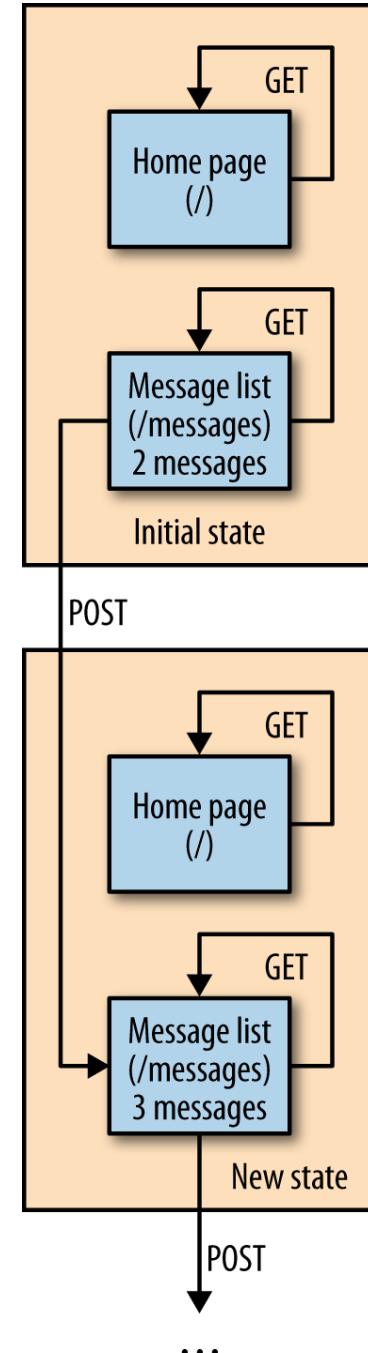
Lesson 7

Application State

- Clients can change state when they receive a new representation of a resource.
- Because HTTP sessions are so short, the server doesn't know anything about a client's application state.
- Application state is kept on the client, but the server can manipulate it by sending representations that describe the possible state transitions.

And, here is a state diagram showing Alice's adventure from **the perspective of the web server**.

- ❖ The server manages two resources: the home page (served from `/`) and the message list (served from `/messages`). The state of these resources is called *resource state*.
- ❖ When the story begins, there are two messages in the message list: “Hello” and “Later.” Sending a GET to the home page doesn’t change resource state, since the home page is a static document that never changes. Sending a GET to the message list won’t change the state either.
- ❖ But when Alice sends a POST to the message list, it puts the server in a new state. Now the message list contains three messages: “Hello,” “Later,” and “Test.” There’s no way back to the old state, but this new state is very similar. As before, sending a GET to the home page or message list won’t change anything. But sending another POST to the message list will add a fourth message to the list.



Lesson 8

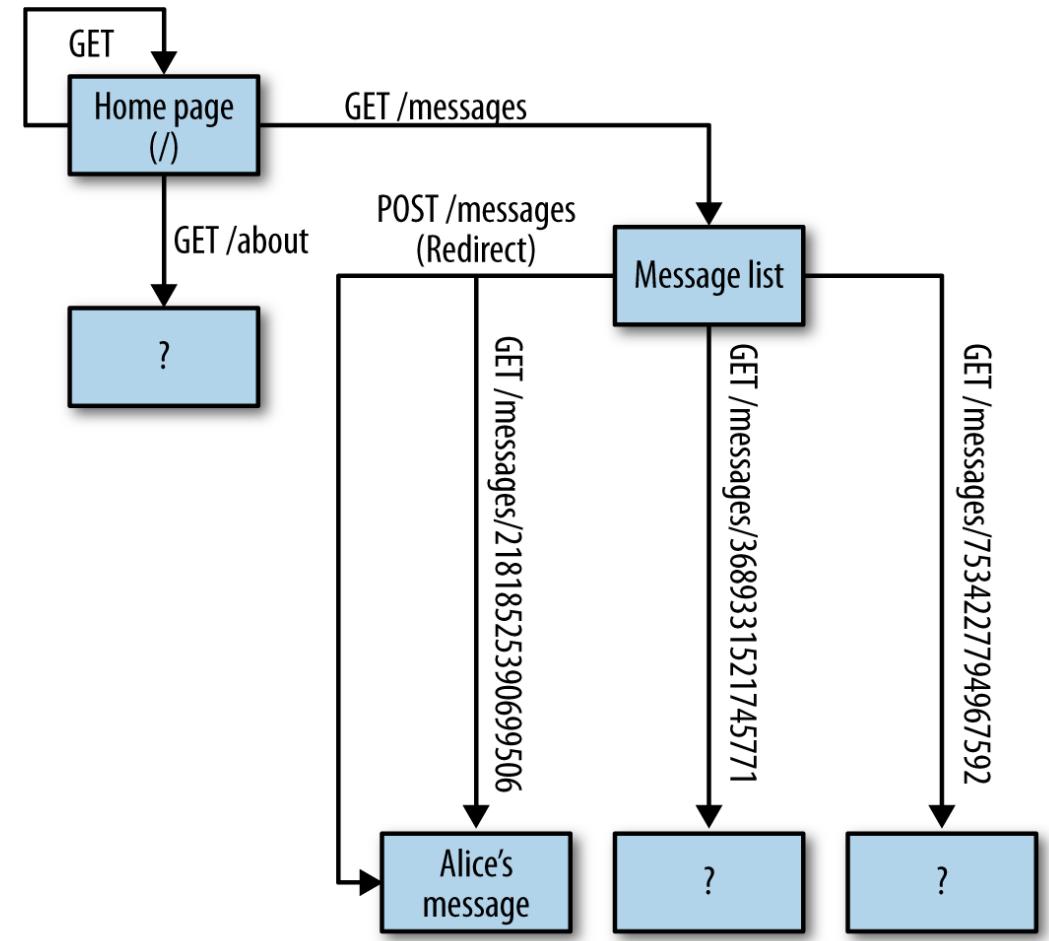
Resource State

- The client has no direct control over resource state—all that stuff is kept on the server.
- Resource state is kept on the server, but the client can manipulate it by sending the server a representation describing the desired new state.
- REST web APIs work through **representational state transfer**.

Finally, here is a partial map of the website from the client's perspective.

This is a web of HTML pages.

The **strands** of the web are the HTML `<a>` tags and `<form>` tags, each describing a GET or POST HTTP request.



Lesson 9

The principle of Connectedness

Each web page tells you how to get to the adjoining pages.

The Web as a whole works on the principle of connectedness, which is better known as "hypermedia as the engine of application state," sometimes abbreviated as **HATEOAS**

The Semantic Challenge

- The story of Alice’s trip through a website went as smoothly as it did thanks to a very slow and expensive piece of hardware: Alice herself.
- Every time her browser rendered a web page, Alice, a human being, had to look at the rendered page and decide what to do next.
- The Web works because human beings make all the decisions about which links to click and which forms to fill out.
- The whole point of web APIs is to get things done without making a human sit in front of a web browser all day.
- How can we program a computer to make the decisions about which links to click?
- A computer can parse the HTML markup `Get started`, but it can’t understand the phrase “Get started.”
- Why bother to design APIs that serve self-descriptive messages if those messages won’t be understood by their software consumers?
- This is the biggest challenge in web API design: **bridging the semantic gap between understanding a document’s structure and understanding what it means**

That's it Folks



Further Reading

Chapter 1 from RESTful Web APIs by Leonard Richardson and Mike Amundsen

XJCO3011: Web Services and Web Data

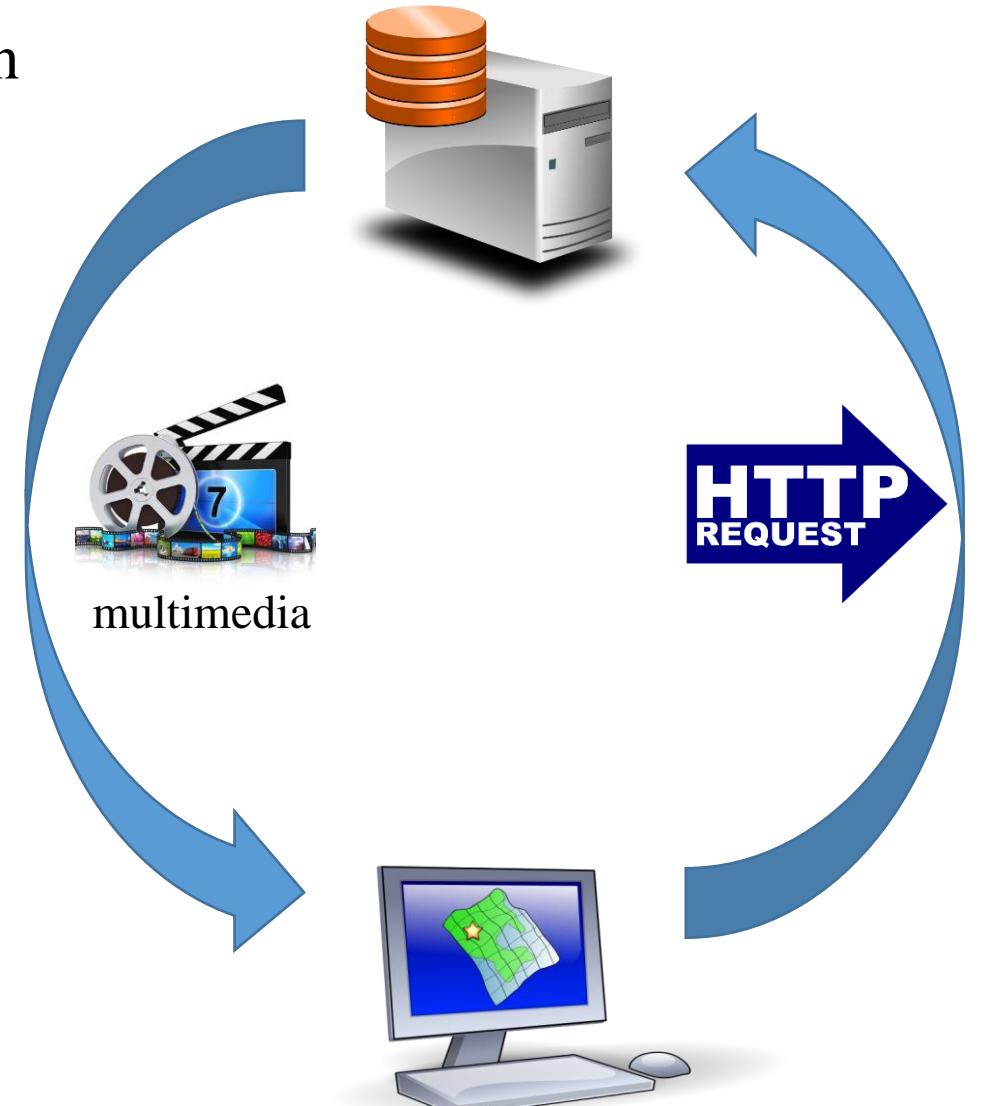


Session #4 – Trying to Bridge the Sematic Gap

Instructor: Dr. Guilin Zhao
Spring 2023

The Problem

- Representational State Transfer (REST) works when clients:
 1. request representations of resources,
 2. parse the representation (the multimedia),
 3. recognise **control structurers** that leads to further resources, e.g. HTML <a> tag,
 4. decide on a new resource to be requested, and
 5. change their current state when the new resource is received.
- Steps 1, 2, 3, and 5 are straightforward and can be easily implemented in fully autonomous code.
- Step 4 however is difficult and requires the client to have knowledge of the application domain. This is what we called the **Semantic Gap**.



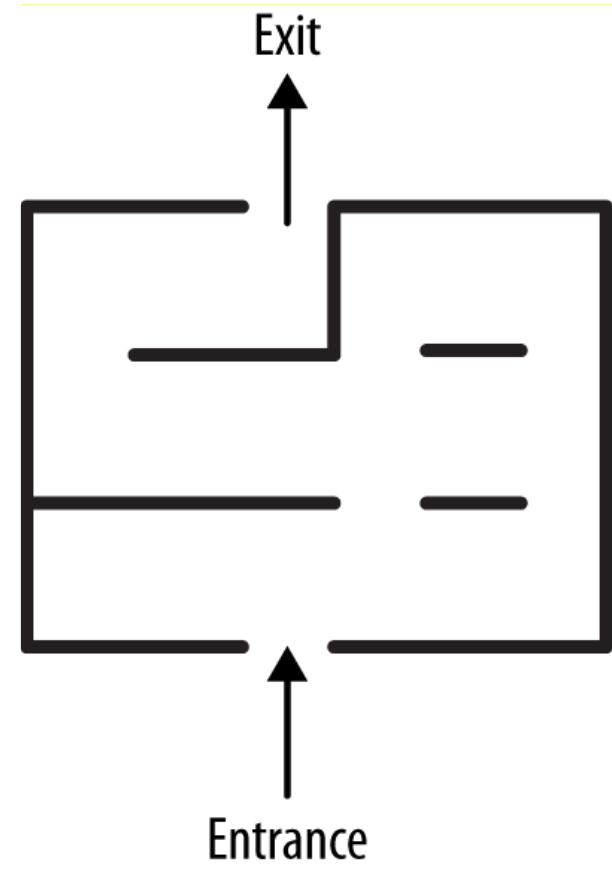
Request, Parse, Recognize, Decide

Semantic Gap

- It refers to the difference or mismatch between the way computers process and understand digital data and the way humans perceive and interpret that same data.
- Humans are able to understand and interpret information based on their **senses**, such as sight, hearing, touch, taste, and smell.
- In contrast, computers rely on **algorithms and mathematical calculations** to process and interpret digital data.
- As a result, there is often a gap between the way humans perceive and interpret digital data, and the way that computers do.
- This gap can **cause problems** in areas such as image recognition, natural language processing, and audio signal processing, where computers may struggle to accurately interpret data in the same way that humans do.

Bridging the Semantic Gap with a Domain-Specific Design

- We will design a web API for representing mazes
- The server's job will be to invent mazes like this one and present them to clients. But why this example?
- Because any complex **problem** (e.g., modifying complex insurance policies; selecting products from a catalog and paying for them) can be represented as a hypermedia maze that the client must navigate.
- These problems have commonalities:
 - ✓ The problem is too complex to be understood all at once, so it's split up into steps.
 - ✓ Every client begins the process at the same first step.
 - ✓ At each step in the process, the server presents the client with a number of possible next steps.
 - ✓ At each step, the client decides what next step to take.
 - ✓ The client knows what counts as success and when to stop.



The Maze+XML Media Type

- How can we represent the shape of a maze in a format that's easy for a computer to understand?
- There's a **personal standard** called Maze+XML, for representing mazes in a machine-readable format.
 - ✓ Maze+XML : the technology for migrating maze games to the web. It uses XML (Extensible Markup Language) to describe the maze so that web browsers can read, display, and manipulate it. With Maze+XML, developers can create complex maze games and easily modify and adjust game rules and elements (i.e., providing developers with a flexible and scalable way to create and deploy maze games).
- The media type of a Maze+XML document is application/vnd.amundsen.maze+xml.
 - ✓ application/vnd.amundsen.maze+xml is a MIME type for a Maze+XML file. MIME types are standards used to indicate the format and type of files transmitted over the internet. In this case, application/vnd.amundsen.maze+xml indicates that the file is a Maze+XML file, defined and supported by the Amundsen company. If a program supports this MIME type, it can read and process the Maze+XML file and generate and manipulate maze games based on the information described in the file.
 - ✓ By defining specific MIME types, developers can make their applications more accurate in identifying and processing different types of files. Therefore, application/vnd.amundsen.maze+xml is an identifier with a specific meaning and purpose, used to represent the type and format of a Maze+XML file.)

The Maze+XML Media Type (Cont'd)

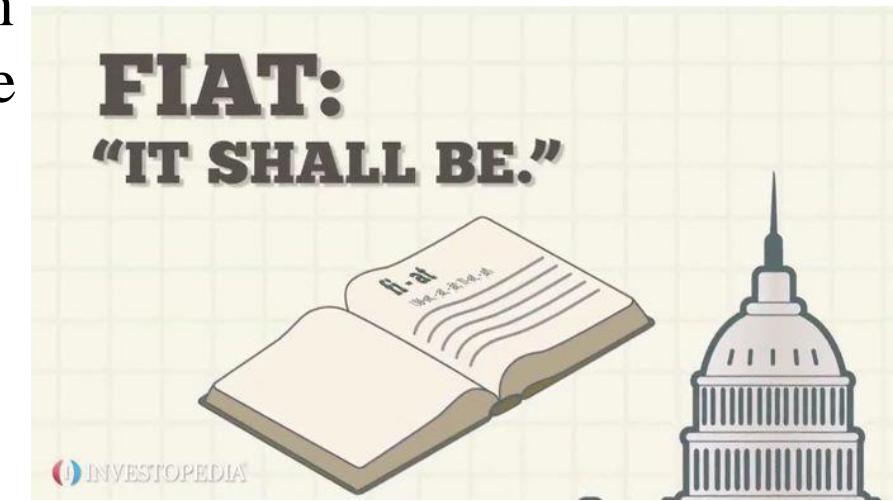
- This is one way a domain-specific design meets the semantic challenge: **by defining a document format that represents the problem** (such as the layout of a maze), **and by registering a media type for that format**, so that a client knows right away when it has encountered an instance of the problem.
- In general, it is not recommended to create new domain-specific media types. It is usually less work to add application semantics to a generic hypermedia format.
- If you set out to do a domain-specific design, you will probably end up with a **fait standard** that doesn't take advantage of the work done by your predecessors.
- But a domain-specific design is the average developer's first instinct when designing an API.

The Structured Syntax Suffix

- A suffix is an augmentation to a media type to specify the underlying structure (e.g. syntax) of that media type.
- For example, MAZE+XML means that this media type uses XML syntax but it has its own defined semantics.
- Structured syntax suffix types are registered and maintained by **IANA** (Internet Assigned Numbers Authority)
- The currently registered suffixes are: **+xml**, **+json**, **+ber**, **+der**, **+fastinfoset**, **+wbxml**, **+zip**, and **+cbor**

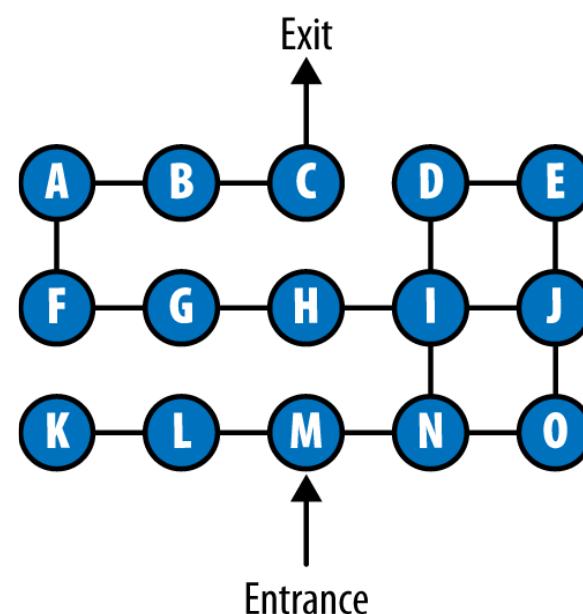
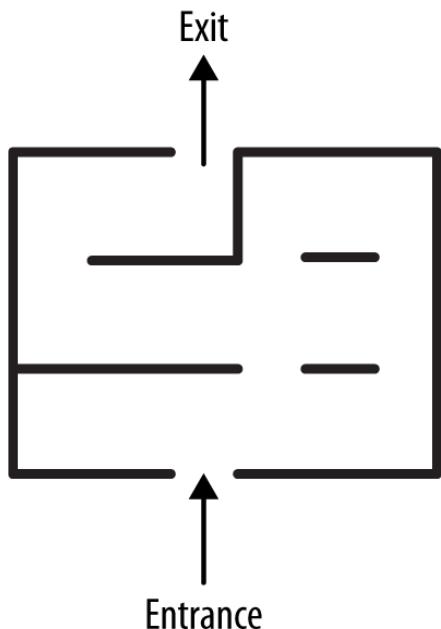
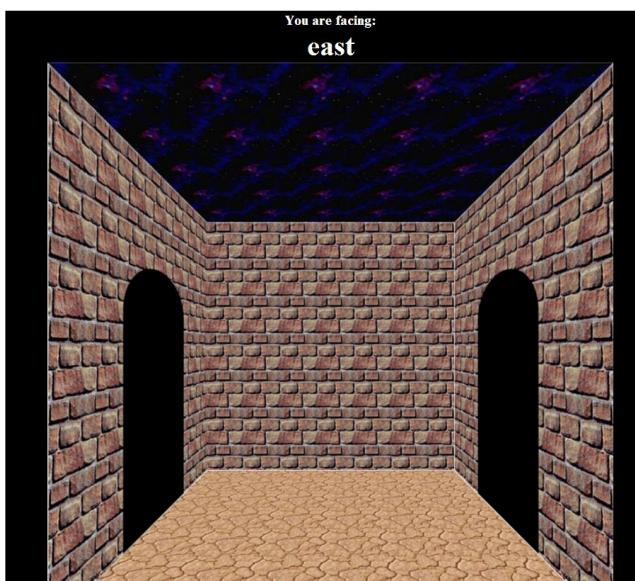
Fiat Standards

- Fiat standards are not really standards; they are behaviours.
No one agreed to them. They're just a description of the way somebody does things.
- The behaviour may be documented, but the core assumption of a standard—that other people ought to do things the same way—is missing.
- Pretty much every API today is a fiat standard, a one-off design associated with a specific company. That's why we talk about the “Twitter API,” the “Facebook API,” and the “Google+ API.”
- Even under ideal circumstances, your API will be a fiat standard, since your business requirements will be slightly different from everyone else's. But ideally a fiat standard would be just a light gloss over a number of other standards.
- Read more: [page xxii – xxiii in RESTful Web APIs by Leonard Richardson and Mike Amundsen](#)



How Maze+XML Works

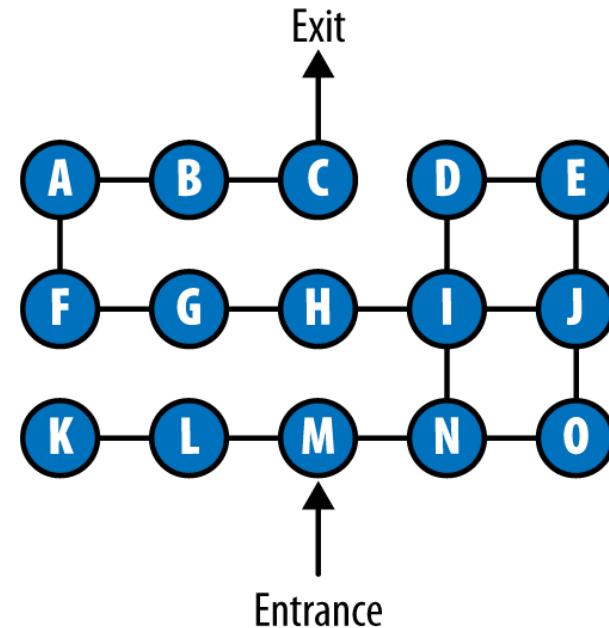
- Upon entering the maze, you'd see something like the figure, a wall in front of you and the entrance behind you. But you'd have two choices: go left or go right. You'd have no way of knowing which direction would take you to the exit.
 - The Maze+XML format simulates this rat's-eye-view by representing **a maze** as **a network of "cells"** that connect to each other. I've chopped the maze into **a grid** and created **a cell** for each grid square.
 - Maze+XML cells connect to each other in the cardinal directions: north, south, east, and west.



The Representation

- Each cell in a Maze+XML maze is an HTTP resource with its own URL.
- If you send a GET request to the first cell in this maze, you'll get a representation that looks like the one below.
- This representation includes a human-readable name of the cell, “The Entrance Hallway,” like you'd see in an old, text-based adventure game.
- The representation also includes <link> tags that connect this cell to its neighbours. From cell M, you have a choice of going west to cell L or east to cell N.

```
<maze version="1.0">
  <cell href="/cells/M" rel="current">
    <title>The Entrance Hallway</title>
    <link rel="east" href="/cells/N"/>
    <link rel="west" href="/cells/L"/>
  </cell>
</maze>
```



Link Relations

- This representation shows off a powerful hypermedia tool called *the link relation*
- By themselves, rel="east" and rel="west" don't mean anything. A computer doesn't know the words "east" and "west."
- But the Maze+XML standard defines meanings for "east" and "west" and developers can program those definitions into their clients
 - ✓ east: refers to a resource to the east of the current resource. When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the east in the active maze.
 - ✓ west: refers to a resource to the west of the current resource. When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the west in the active maze.
- In Maze+XML, following a link marked with the link relation **east** will move your client **east** through some abstract geographical space.
- This is how Maze+XML meets the semantic challenge: by defining link relations that convey its application semantics.

Link Relations (Continued)

- A link relation is a **string** associated with a **hypermedia control** (e.g., `<link>` tags).
- It explains the change in application state (for safe requests) or resource state (for unsafe requests) that will happen if the client triggers the control.
- Link relations are **managed by the Internet Assigned Numbers Authority (IANA)**.
- Currently, it contains about **60 link relations** that have been deemed to be generally useful and not tied to a particular data format
- The simplest examples are the **next** and **previous relations**, for navigating a list.
- RFC 5988 (which is a document that defines the Web Linking protocol for the World Wide Web) defines two kinds of link relations: **registered relation types** and **extension relation types**.
 - ✓ Registered link relations look like the ones you see in the IANA registry: short strings like next and previous.
 - ✓ Extension relations look like URLs. If you own mydoma.in, you can name a link relation `http://mydoma.in/whatever` and define it to mean anything you want.

Example: HTML Link Relations

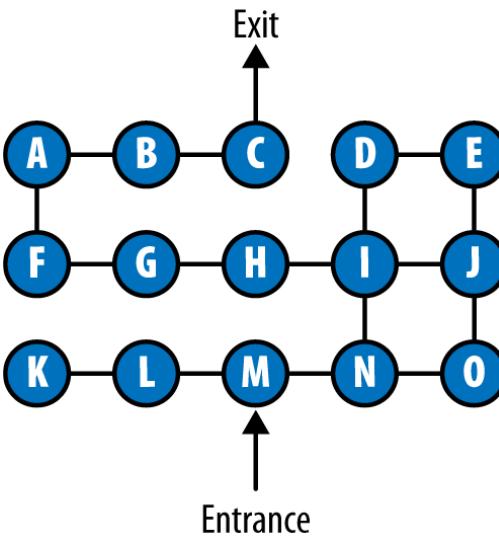
< a rel="nofollow" href="http://www.functravel.com/"> Cheap Flights

Value	Description
alternate	Provides a link to an alternate representation of the document (i.e. print page, translated or mirror)
author	Provides a link to the author of the document
bookmark	Permanent URL used for bookmarking
external	Indicates that the referenced document is not part of the same site as the current document
help	Provides a link to a help document
license	Provides a link to copyright information for the document
next	Provides a link to the next document in the series
nofollow	Links to an unendorsed document, like a paid link. ("nofollow" is used by Google, to specify that the Google search spider should not follow that link)
noreferrer	Requires that the browser should not send an HTTP referer header if the user follows the hyperlink
noopener	Requires that any browsing context created by following the hyperlink must not have an opener browsing context
prev	The previous document in a selection
search	Links to a search tool for the document
tag	A tag (keyword) for the current document

Follow a Link to Change Application State

- A client can “go east” from cell M by following the appropriate link (i.e. by sending a GET request to the URL labelled with rel="east").
- When the client does this, it will get a second Maze+XML representation, looking something like this:

```
<maze version="1.0">
  <cell href="/cells/N">
    <title>Foyer of Horrors</title>
    <link rel="north" href="/cells/I"/>
    <link rel="west" href="/cells/M"/>
    <link rel="east" href="/cells/O"/>
  </cell>
</maze>
```



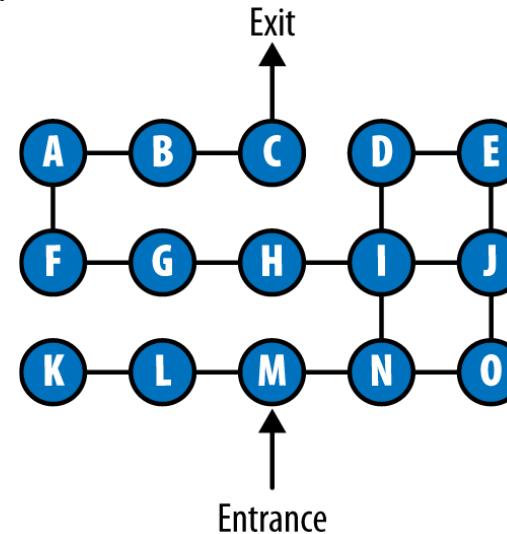
```
<maze version="1.0">
  <cell href="/cells/M" rel="current">
    <title>The Entrance Hallway</title>
    <link rel="east" href="/cells/N"/>
    <link rel="west" href="/cells/L"/>
  </cell>
</maze>
```

- This is the Maze+XML representation of cell N on the map. It links back to cell M (using the link relation west), as well as to cells I (north) and O (east).
- **The client’s application state has changed.** To borrow a term from the HTML standard, *the client was “visiting” cell M, and now it’s “visiting” cell N.*
- The client now has three new options, represented by the links in the representation of cell N.

Finally

- By following the right links (north, west, west, west, north, east, east, and finally east), a client can make its way from cell N to cell C. That cell includes the exit to the maze, indicated here by a <link> tag with the link relation exit:

```
<maze version="1.0">
  <cell href="/cells/C">
    <title>The End of the Tunnel</title>
    <link rel="west" href="/cells/B"/>
    <link rel="exit" href="/success.txt"/>
  </cell>
</maze>
```



- Here's what the Maze+XML standard says about exit:
 - ✓ exit: refers to a resource that represents the exit or end of the current client activity or process. When used in the Maze+XML media type, the associated URI points to the final exit resource of the active maze.
- Maze+XML provides no guidance as to what should appear at the other end of an exit link. It's a “resource,” which means it can be anything at all. In this implementation, we've chosen to link to a textual congratulatory message (success.txt).

The Collection of Mazes

- Cell C leads out of the maze, because its representation includes **a special link with rel="exit"**. But cell M, the entrance to the maze, doesn't include anything to distinguish it from the other fourteen cells.
- There's **no rel="entrance"**. Cell M's title is "The Entrance Hallway," but **that phrase doesn't mean anything to a computer**. How do we bridge the semantic gap? How is the client supposed to know where to start the maze?
- The Maze+XML standard solves this problem with **a collection**: a list of mazes. If you send **a GET request to the root URL of the maze API**, you might get a Maze+XML representation that looks like this.
- A collection in Maze+XML is **a <collection> tag that includes some <link> tags with the link relation maze**.

```
<maze version="1.0">
  <collection>
    <link rel="maze" title="A Beginner's Maze" href="/beginner">
    <link rel="maze" title="For Experts Only" href="/expert-maze/start">
  </collection>
</maze>
```

To Get Started

Send a GET request to a URL labelled with the relation maze (e.g. /beginner), and you'll get a representation that looks like this:

```
<maze version="1.0">
  <item>
    <title>A Beginner's Maze</title>
    <link rel="start" href="/cells/C"/>
  </item>
</maze>
```

This is a high-level representation of the maze as seen from the outside. It's got a link with the relation start which points to cell C.

Navigating

Starting with no information but this URL, you can do everything it's possible to do with a Maze+XML API:

- Start off by GETting a representation of the collection of mazes. You know how to parse the representation, because **you read the Maze+XML specification** and **programmed this knowledge into your client**.
- Your client also knows that the link relation `maze` indicates an individual maze. This gives it a URL it can use in a second GET request. Sending that GET request gives you the representation of an individual maze.
- Your client knows how to parse the representation of an individual maze (because you programmed that knowledge into it), **and it knows that the link relation 'start' indicates an entrance into the maze**. You can make a **third GET request to enter the maze**.
- Your client knows how to parse the representation of a cell. It knows what east, west, north, and south mean, so it can translate movement through an abstract maze into a series of HTTP GET requests.
- Your client knows **what exit means**, so it knows when it's completed a maze.

There's more to the Maze+XML standard, but you've now seen the basics. A collection links to a maze, which links to a cell. From one cell you can follow links to other cells. Eventually you'll find a cell with an exit link leading out of the maze.

A Maze+XML Server

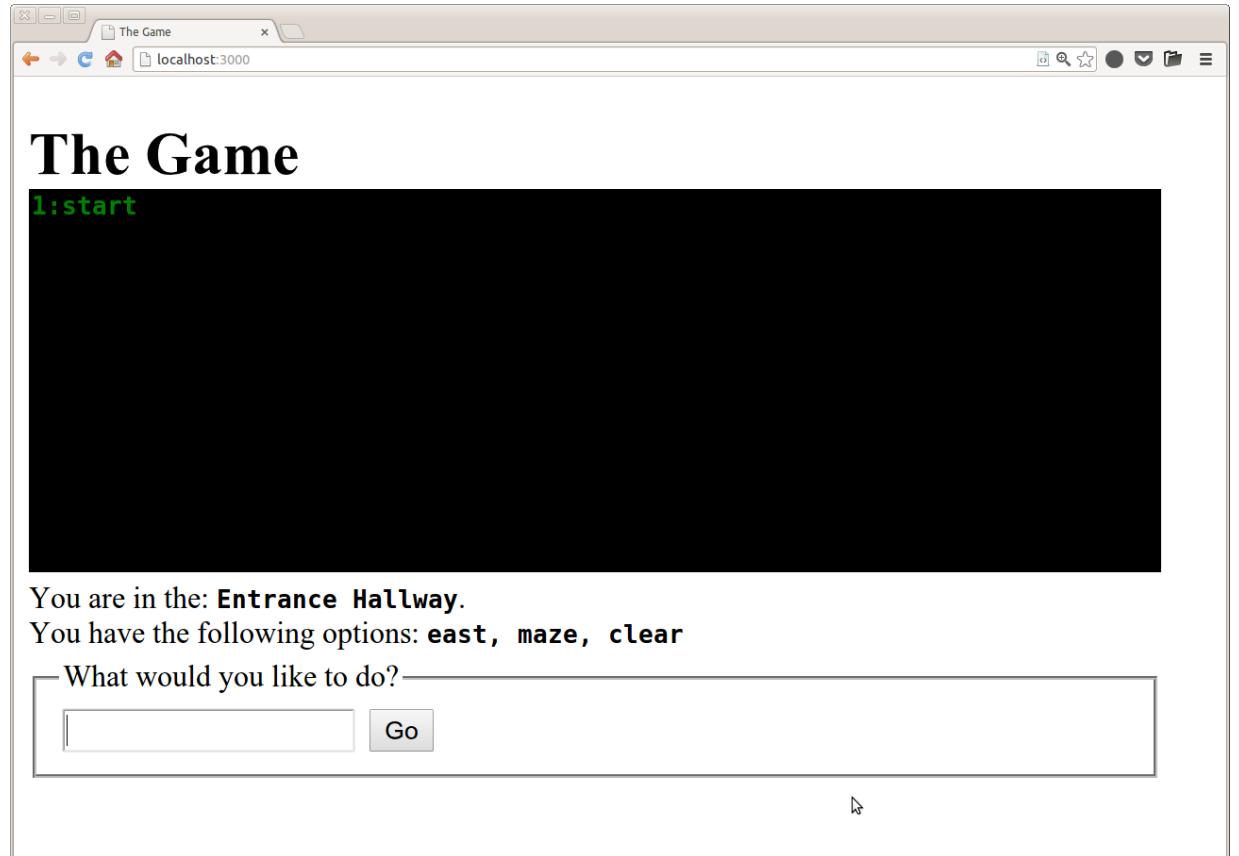
- A maze+xml server can store maze data in any format, including simple JSON documents.
- Here's the JSON document representing the “beginner” maze
- This is not a representation of the maze in the REST sense, because it will never be sent over HTTP.
- It's the raw data used to generate the Maze+XML document that *is* sent over HTTP.

```
{  
  "_id" : "five-by-five",  
  "title" : "A Beginner's Maze",  
  "cells" : {  
    "cell0": {"title": "Entrance Hallway", "doors": [1,1,1,0]},  
    "cell1": {"title": "Hall of Knives", "doors": [1,1,1,0]},  
    "cell2": {"title": "Library", "doors": [1,1,0,0]},  
    "cell3": {"title": "Trophy Room", "doors": [0,1,0,1]},  
    "cell4": {"title": "Pantry", "doors": [0,1,1,0]},  
    "cell5": {"title": "Kitchen", "doors": [1,0,1,0]},  
    "cell6": {"title": "Cloak Room", "doors": [1,0,0,1]},  
    "cell7": {"title": "Master Bedroom", "doors": [0,0,1,0]},  
    "cell8": {"title": "Fruit Closet", "doors": [1,1,0,0]},  
    "cell9": {"title": "Den of Forks", "doors": [0,0,1,1]},  
    "cell10": {"title": "Nursery", "doors": [1,0,0,1]},  
    "cell11": {"title": "Laundry Room", "doors": [0,1,1,0]},  
    "cell12": {"title": "Smoking Room", "doors": [1,0,1,1]},  
    "cell13": {"title": "Dining Room", "doors": [1,0,0,1]},  
    "cell14": {"title": "Sitting Room", "doors": [0,1,1,0]},  
    "cell15": {"title": "Standing Room", "doors": [1,1,1,0]},  
    "cell16": {"title": "Hobby Room", "doors": [1,0,1,0]},  
    "cell17": {"title": "Observatory", "doors": [1,1,0,0]},  
    "cell18": {"title": "Hot House", "doors": [0,1,0,1]},  
    "cell19": {"title": "Guest Room", "doors": [0,0,1,0]},  
    "cell20": {"title": "Servant's Quarters", "doors": [1,0,0,1]},  
    "cell21": {"title": "Garage", "doors": [0,0,0,1]},  
    "cell22": {"title": "Tool Room", "doors": [0,0,1,1]},  
    "cell23": {"title": "Banquet Hall", "doors": [1,1,0,1]},  
    "cell24": {"title": "Spoon Storage", "doors": [0,0,1,1]}  
  }  
}
```

The Clients

Client 1: A Human Directed Maze Game

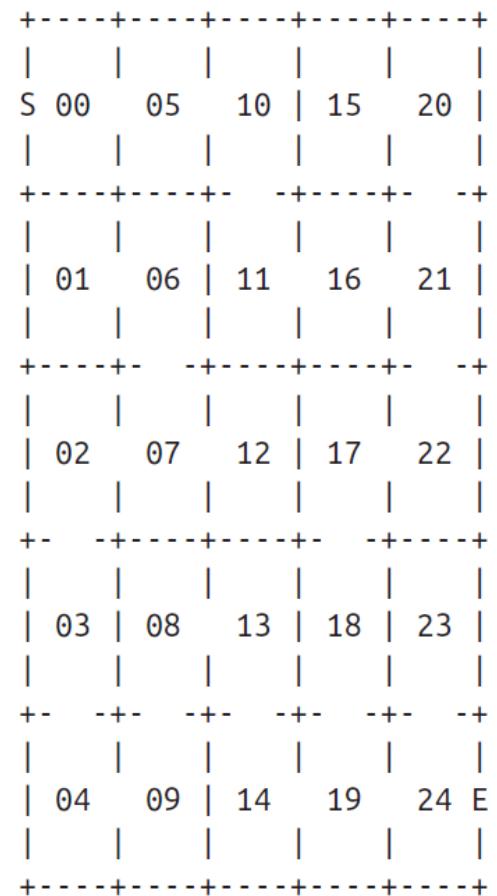
- The obvious use for the Maze+XML API is a **game to be played by a human being**
- We tend to think of an “API client” as an automated client.
- But **human-driven clients** like this have a **big part to play in the modern API ecosystem**.
- It’s very common for a mobile application, driven by a human, to communicate with a server through a web API. Best of all, **with a human in the loop, the semantic gap is no problem**.



Client 2: The Mapmaker

- The first client relied on a human being making the decisions about where to go.
- But there are algorithms for automatically solving mazes, and there's no reason we can't write an automated client to go along with the manually operated one.
- This client does something a little different. It is called the Mapmaker, and it's a client for *mapping* a maze

The server doesn't define mazes in this graphical format; they're stored as JSON documents and served as XML documents. The Mapmaker builds up this graphical view of the maze by automatic exploration.



Meeting the Semantic Challenge

- For the designer of a domain-specific API, **bridging the semantic gap** is a two-step process:
 1. Write down your application semantics in a human-readable specification (like the Maze+XML standard).
 2. Register one or more IANA media types for your design, (like *application/vnd.amundsen.maze+xml*. In the registration, associate the media types with the human-readable document you wrote.
- Your **client developers** can reverse the process to bridge the semantic gap in the other direction:
 1. Look up an unknown media type in the IANA registry.
 2. Read the human-readable specification to learn how to deal with documents of the unknown media type.

There's no magic shortcut. To get working client code, your users will have to read your human-readable document and do some work. We can't get rid of the semantic gap completely, because computers aren't as smart as humans.

That's it Folks



Further Reading

Chapter 5: RESTful Web APIs by Leonard Richardson and Mike Amundsen

Web Services and Web Data

XJCO3011



Session 5. HyperMedia for RESTful API's

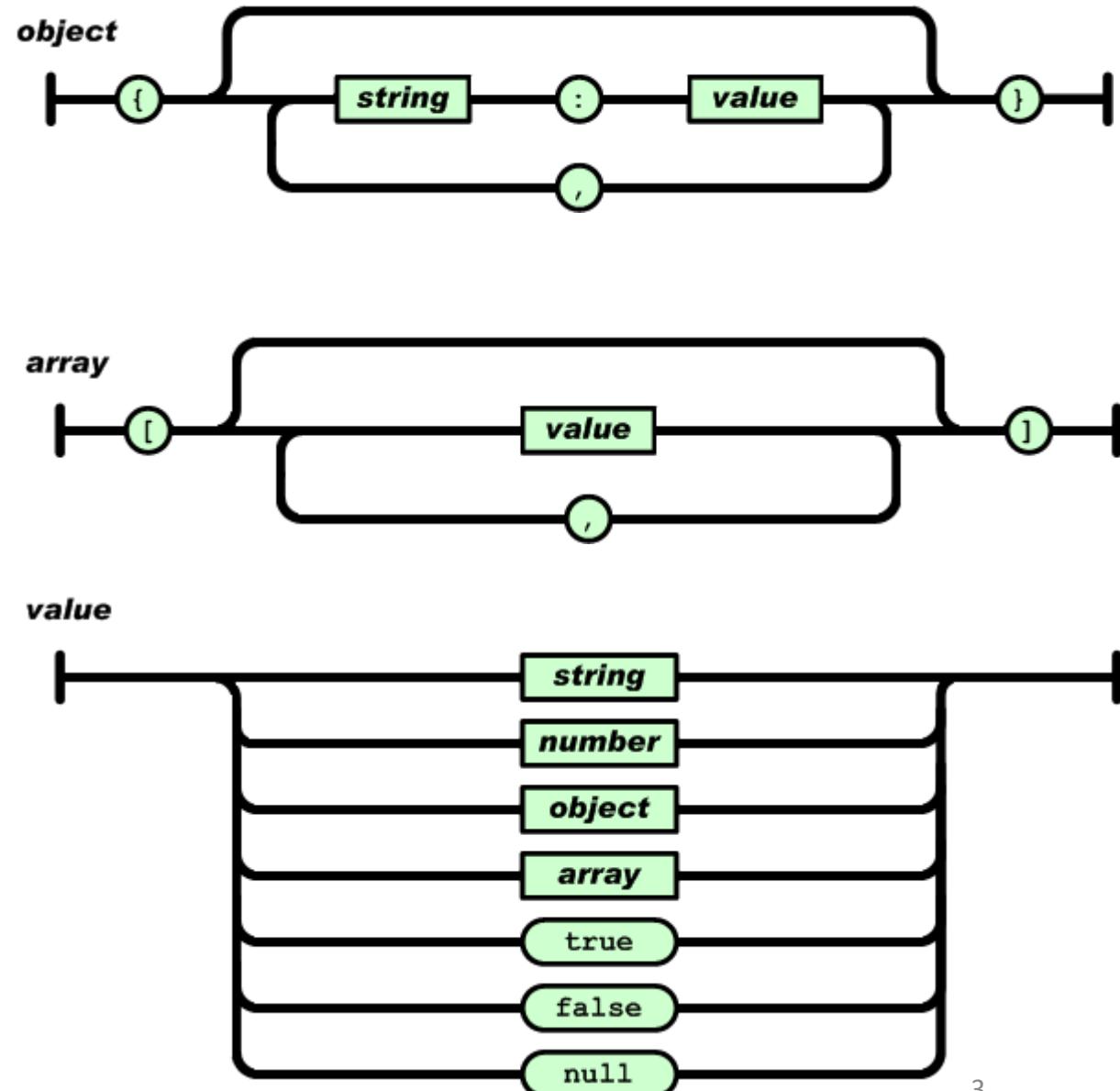
Why HyperMedia

- Hypermedia is a method of creating and representing resources and their relationships on the web. It not only focuses on how to obtain resources, but also emphasizes how to navigate and interact with resources. Through hypermedia, clients can automatically explore and interact with resources by following links and relationships between them.
- RESTful is an architectural style for designing web services that emphasizes using the semantics of the HTTP protocol to describe resources and their relationships. RESTful services typically use HTTP verbs (such as GET, POST, PUT, DELETE, etc.) to manipulate resources and use HTTP status codes to represent the results of operations. In RESTful, hypermedia is an important design principle used to represent relationships between resources and to help clients discover and interact with the service.
- hypermedia can be seen as an extension of RESTful that emphasizes relationships and navigation between resources, making web services more flexible and discoverable.
- In practice, hypermedia can be represented using different formats, such as HTML, **JSON**, XML, etc.)

JSON

JSON is built on two structures:

- A **collection of name-value pairs**. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An **ordered list of values**. In most languages, this is realized as an array, vector, list, or sequence.
- These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.



A JSON Example

A person record

Object structures and array structures can be nested inside each other, allowing for more complex data structures

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "isAlive": true,  
  "age": 27,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021-3100"  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "office",  
      "number": "646 555-4567"  
    },  
    {  
      "type": "mobile",  
      "number": "123 456-7890"  
    }  
],  
  "children": [],  
  "spouse": null  
}
```

Collection+JSON

- Collection+JSON is one of several standards designed **not** to represent one specific problem domain (the Maze+XML does) but to fit a pattern—the collection—that shows up over and over again, in all sorts of domains.
- A collection such as microblog posts, goods in a shopping cart or a collection of readings from a weather sensor would look pretty much the same and have the same **protocol semantics**.
- A collection is **a special kind of resource**. Recall that a resource is anything important enough to have been given its own URL. Recall from Chapter 3 that a resource is anything important enough to have been given its own URL. A resource can be a piece of data, a physical object, or an abstract concept—anything at all. All that matters is that it has a URL and the representation—the document the client receives when it sends a GET request to the URL.
- A collection resource is a little more specific than that. It exists **mainly to group other resources together**. Its representation focuses on links to other resources, though it may also include snippets from the representations of those other resources. (Or even the full representations!)

Collection+JSON Standard

- The Collection+JSON standard defines a representation format based on JSON.
- It also defines the **protocol semantics for the HTTP resources that serve that format** in response to GET requests.
- It's basically **an object with five special properties**, predefined slots for application specific data:
 1. **href**: A permanent link to the collection itself.
 2. **items**: Links to the members of the collection, and partial representations of them.
 3. **links**: Links to other resources related to the collection.
 4. **queries**: Hypermedia controls for searching the collection.
 5. **template**: A hypermedia control for adding a new item to the collection.

Representing the Items

Let's zoom in on items, the most important field in a Collection+JSON representation:

```
"items" : [  
  { "href" : "/api/airlines/ba",  
    "data" : [  
      { "name" : "company_name", "value" : "British Airways" },  
      { "name" : "date_started", "value" : "1974-03-31T05:33:58.930Z" }  
    ],  
    "links" : []  
  },  
  { "href" : "/api/airlines/af",  
    "data" : [  
      { "name" : "company_name", "value" : "Air France" },  
      { "name" : "date_started", "value" : "1933-10-07T12:55:59.685Z" }  
    ],  
    "links" : []  
  }  
]
```

The href attribute :A permanent link to the item as a standalone resource.

data: Any other information that's an important part of the item's representation.

links

Hypermedia links to other resources related to the item.

An item's permanent link

A member's href attribute is a link to the resource outside the context of its collection. If you GET the URL mentioned in the href attribute, the server will send you a Collection+JSON representation of a single item.

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://airlines.pythonanywhere.com/api/",  
    "items" : [  
      { "href" : "/api/airlines/ba",  
        "data": [  
          { "name" : "company_name", "value" : "British Airways" },  
          { "name" : "date_started", "value" : "1974-03-31T05:33:58.930Z" }  
        ],  
        "links" : []  
      }  
    ]  
  }  
}
```

An item's links

An item's representation may also contain a list called `links`. This contains any number of other hypermedia links to related resources. Here's a link you might see in the representation of a "book" resource:

```
"links" :  
[  
  {  
    "name" : "owner",  
    "rel" : "owner",  
    "prompt" : "Owner of the company",  
    "href" : "/owners/441",  
    "render" : "link"  
  }  
]
```

The `rel` attribute is a slot for a link relation, just like the `rel` attribute in Maze+XML

The `prompt` attribute is a place to put a human-readable description

Setting `render` to "link" tells a Collection+JSON client to present the link as an outbound link like an HTML `<a>` tag. Setting `render` to "image" tells the client to present the link as an embedded image, like HTML's `` tag

The Write Template

Suppose you want to add a new item to a collection. What HTTP request should you make? To answer this question, you need to look at the collection's write template. Here's the write template for our airline directory API:

```
"template": {  
    "data": [  
        {"prompt": "Name of company", "name": "company_name", "value": ""}  
    ]  
}
```

Interpreting this template according to the Collection+JSON standard tells you it's OK to fill in the blanks and submit a document that looks like this:

```
{ "template":  
  {  
    "data": [  
      {"prompt": "Name of company", "name": "company_name", "value": "Air France"}  
    ]  
  }  
}
```

Where does that request go? The Collection+JSON standard says you add an item to a collection by sending a POST request to the collection (i.e., to its href attribute):

"href": "http://airlines.pythonanywhere.com/api/"

How a (Generic) Collection Works

- There's not much more to Collection+JSON than what we have just shown. It was **designed without any real application semantics**, so that it can be used in many different applications.
- Because it's so general, it does a good job illustrating the common features of the collection pattern.
- Before moving on to other collection types, let us go up a level and lay out the pattern itself by describing the behaviour of a generic “collection” resource under HTTP.
- Different collection types - such as Collection+JSON , AtomPub and Odata - take different approaches to collections, but they all have more or less the same protocol semantics.

GET

- Like most resources, a collection **responds to GET by serving a representation**. Although collection standards (Collection+Json, AtomPub, and OData) don't say much about what an item should look like within a collection, they go into great detail about what a collection's representation should look like.
- The **media type of the representation tells you what you can do with the resource**. If you get an application/vnd.collection+json representation, you know that the rules of the Collection+JSON standard apply. If the representation is application/atom+xml, you know that AtomPub rules apply.
- If the representation is **application/json**, you're **out of luck**, because the JSON standard doesn't say anything about collection resources. You're using an API that went off on its own and **defined a fiat standard**. You'll need to look up the details for the specific API you're using.

POST-to-Append

- The defining characteristic of a collection is its behaviour under HTTP POST.
- Unless a collection is **read-only** (like a collection of search results), a client can **create a new item** inside it by sending it a POST request.
- When you **POST a representation** to a collection, the server creates a new resource based on your representation.
- That resource **becomes the latest member** of the collection.

PUT and PATCH

- None of the main collection standards define a collection's response to PUT or PATCH.
- Some applications implement these methods as a way of modifying several elements at once, or of removing individual elements from a collection.
- Collection+JSON, AtomPub, and OData all define an *item's response to PUT*: they say that PUT is how clients should change the state of an item. But these standards are just repeating what the HTTP standard says. They're not putting new restrictions on item resources. PUT is how clients change the state of *any* HTTP resource.

DELETE

- None of the three big standards define how a collection should respond to DELETE.
- Some applications implement DELETE by deleting the collection; others delete the collection and every resource listed as an item in the collection.
- The main collection standards all define an item's response to DELETE, but again, they're just restating what the HTTP standard says. The DELETE method is for deleting things.

Pagination

- A collection **may contain millions of items**, but the server is under no obligation to serve millions of links in a single document. The most common alternative is **pagination**.
- A server can choose to serve the first 10 items in the collection, and give the client a link to the rest:

```
<link rel="next" href="/collection/4iz6"/>
```

- The "next" link relation is registered with the IANA to mean “the next in the series.” and by following that link we get the second page of the collection.
- We can keep following rel="next" links until we reach the end of the collection.
- There are a number of generic link relations for navigating paginated lists. These include "next", "previous", "first", and "last".
- These link relations were originally defined for HTML, but now they're registered with the IANA, so you can use them with any media type.

The Atom Publishing Protocol (AtomPub)

- The Atom file format was developed for **syndicating news articles** and blog posts. It's defined in RFC 4287, which was finalized in 2005.
- The Atom Publishing Protocol is a standardized workflow for editing and publishing news articles, using the Atom file format as the representation format. It's defined in RFC 5023, which was finalized in 2007.
- Those are pretty early dates in the world of REST APIs. In fact, AtomPub was the first standard to describe the collection pattern.
- AtomPub has the same concepts as Collection+JSON, but uses different terminology. Instead of a “collection” that contains “items,” this is a “feed” that contains “entries.”

An Atom representation of a microblog

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>You Type It, We Post It</title>
  <link href="http://www.youtypeitwepostit.com/api" rel="self" />
  <id>http://www.youtypeitwepostit.com/api</id>
  <updated>2013-04-22T05:33:58.930Z</updated>
  <entry>
    <title>Test.</title>
    <link href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <link rel="edit" href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <id>http://www.youtypeitwepostit.com/api/messages/21818525390699506</id>
    <updated>2013-04-22T05:33:58.930Z</updated>
    <author><name/></author>
  </entry>
  ....
```

Some Atom Features

- The previous document is served with the media type **application/atom+xml**
- We can **POST a new Atom entry to the href of the collection**. An entry's **rel="edit"** **link** is the URL you **send a PUT** to if you want to edit the entry, or **send a DELETE** to if you want to delete the entry.
- There's one big conceptual difference between Collection+JSON and AtomPub. **Collection+ JSON defines no particular application semantics** for "item." An "item" can look like anything. But since **Atom was designed to syndicate news articles**, every AtomPub entry looks a bit like a news article.
- Every entry in an AtomPub feed must have a unique ID (the URL of the post in the previous example), a title (the text of the post in the previous example), and the date and time it was published or last updated.
- The Atom file format defines **little bits of application semantics** for news stories: fields like "subtitle" and "author"

Atom Applications

- Despite this focus on news and blog posts, AtomPub is a fully **general implementation of the collection pattern**.
- Google, the biggest corporate adopter of AtomPub, **uses Atom documents to represent videos, calendar events, cells in a spreadsheet, places on a map, and more.**
- The secret is extensibility. You're allowed to extend Atom's vocabulary with whatever application semantics you care to define.
- Google defined a common Atom extension called **GData** for all of its Atom-based APIs, then defined additional extensions for videos, calendars, spreadsheets, and so on.

A few interesting facts about AtomPub

- Since news articles are often classified under one or more categories, the Atom file format defines a **simple category system**, and AtomPub defines a separate media type for a **list of categories** (`application/atomcat+xml`).
- AtomPub also defines a media type for a **Service Document**—effectively a collection of collections.
- Atom is strictly an **XML-based file format**. AtomPub installations **do not serve JSON representations**. This makes it difficult to consume an AtomPub API from an Ajax (Asynchronous JavaScript And XML) client.
- Although Atom is an XML file format, clients may POST binary files to an AtomPub API. An uploaded file is represented on the server as two distinct resources: a **Media Resource** whose representation is the binary data, and an **Entry Resource** whose representation is metadata in Atom format.
- This feature lets you use AtomPub to store a collection of photos or audio files, along with Atom documents containing descriptions and related links.

Why Doesn't Everyone Use AtomPub?

- Many years after finalizing the AtomPub standard, it **never got much traction** outside of Google, and even Google seems to be phasing it out.
- The problem stems from the **decision to use XML documents** for AtomPub representations
- It was obviously a correct decision in 2003, but over the next 10 years, as in-browser API clients became more and more popular, **JSON gained an overwhelming popularity** as a representation format.
- It's a **lot easier to process JSON** from in-browser JavaScript code than it is to process XML.
- Today, the vast majority of APIs either serve JSON representations exclusively, or offer a choice between XML and JSON representations.
- The AtomPub story **shows that “nothing wrong with the standard” isn’t good enough.**
- People won't go through the trouble of learning a standard unless it's directly relevant to their needs.
- It's **easier to reinvent the “collection” pattern using a flat standard based on JSON**, so that's what thousands of developers did—and continue to do.
- The question is whether we'll collectively allow ourselves to reinvent the same basic ideas over and over again.

That's it Folks



Further Reading

Chapter 6: RESTful Web APIs by Leonard Richardson and Mike Amundsen

XJCO3011: Web Services and Web Data



Session #8– Let's Django

Instructor: Dr. Guilin Zhao
Spring 2023

- In the first coursework, we will be using the **Django** framework.
- Django is a free open-source web framework, written in **Python**.
- Created by Adrian Holovaty and Simon Willison in 2003, and named after the **French jazz guitarist Django Reinhardt**.



Adrian Holovaty



Django Reinhardt

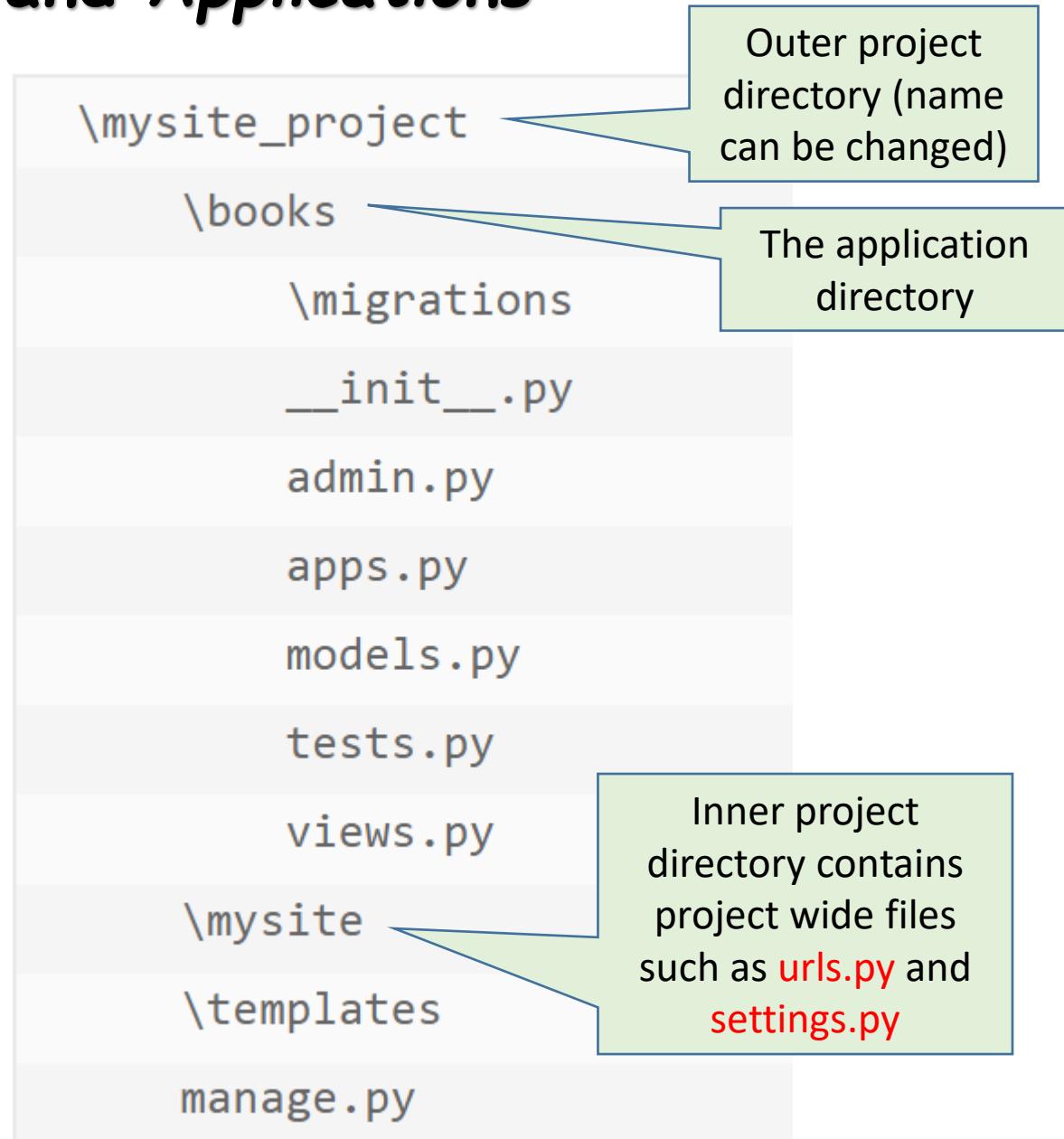


Installing Django

```
$ python -m venv vooo // to create a virtual environment
$ cd vooo
$ source bin/activate           // to activate the virtual env.
$ pip install django           // to install the latest version of the
Django framework
// to check that django has been successfully installed:
$ python                      // start the python interpreter
>>> import django
>>> django.VERSION
(2, 1 , 5, 'final' , 0)        // the version number of the software or library as 2.1.5,
                               // with a version status of 'final' (i.e., stable release), and a
                               // numerical representation of the version status as 0.
>>> quit ()
```

Django Projects and Applications

- A project is created with the command:
`django-admin startproject <project name>`
- One Django project can have a number of Django applications. An application can be created with:
`python manage.py startapp <application name>`
- Database models can only live within an application.
- For simple websites, one app is sufficient for all required functionality.



Registering the Application

The application name

- Even though our new app exists within the Django project, Django doesn't "know" about it until we **explicitly add it**.
- Your new application will not work until you register it in the project's **settings.py** file.
- Add your application configuration class to the list of **INSTALLED_APPS**

```
INSTALLED_APPS = [  
    'books.apps.BooksConfig',  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

The name of a class that is automatically created in the **apps.py** file.
The name of this class is derived from the name of the application with first letter capitalized (all other letters converted to small letters) and the word Config appended

In the **apps.py** file (in the application directory)

```
class BooksConfig(AppConfig):  
    name = 'books'
```

Django Models

- Provide an **ORM (object relational mapping) interface** to the underlying relational DBMS (**database management system**).
- Created within the **models.py** file.
- Each database table is represented as one class **derived from the Model class** in the **Django.db.models module**.
- Each column (field) is defined as an attribute using one of the predefined model Fields.
- No need to define the primary key. It is automatically created. It is called **id** in the tables, and **pk** in the class model
- A complete list of all available fields and their attributes can be found in **Appendix A of the Django Book**.

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

- ❖ **CharField** is a data type in the Django framework that represents character fields.
- ❖ The **max_length** parameter specifies the maximum number of characters allowed for this field.

[from Chapter 4 of the Django book](#)

Django Models

A basic book/author/publisher data layout

- The conceptual relationships between books, authors, and publishers are well known; a book that was written by authors and produced by a publisher!
- Some assumptions:
 - An author has a first name, a last name, and an email address.
 - A publisher has a name, a street address, a city, a state/province, a country, and a website.
 - A book has a title and a publication date. It also has one or more authors (a many-to- many relationship with authors) and a single publisher (a one-to-many relationship—aka foreign key—to publishers).
- The first step in using this database layout with Django is to express it as Python code. In the **models.py file** that was created by the startapp command, enter the codes as shown in the picture.
- Each model is represented by a Python class that is a subclass of **django.db.models.Model**. The parent class, Model, contains all the machinery necessary to make these objects capable of interacting with a database – and that leaves our models responsible solely for defining their fields, in a nice and compact syntax.

Field Creation Arguments

- **default**: assigns a default value to the field when it is created, e.g.

```
BusinessName = models.CharField(max_length = 64 , default = 'Not Assigned')
```

- **choices**: allows the value of a field to be chosen from a list, e.g:

```
BusinessTypes = [ ('airline', 'Airline Company'), ('payment', 'Payment Service Provider') ]
```

```
BusinessType = models.CharField(max_length = 32 , choices = BusinessTypes , default = 'unknown').
```

The list should contain tuples of items, the first one is the field value and the second one is a human readable form

- **unique**: enforces the rule that the field value should be unique, e.g.

```
BusinessCode = models.CharField(max_length = 8 , unique = True)
```

- **null**: allows the field to store null values

- **on_delete**: used with `models.ForeignKey` fields to define what happens to the record when the record in the primary table (the one containing the primary key) is deleted, e.g.:

```
destination_airport = models.ForeignKey(Airport, on_delete=models.CASCADE)
```

The following options can be used as values for the `on_delete` argument:

- `models.CASCADE`: when the referenced object is deleted, also delete the objects that have references to it.
- `models.PROTECT`: forbid the deletion of the `referenced` object.
- `models.SET_NULL`: set the reference to NULL (requires the field to be nullable).

Creating the underlying relational data base check, makemigrations and migrate!

- The process of transforming model classes into actual database tables.
- In Django, each model class corresponds to a database table, where the class attributes are mapped to table columns and class instances are mapped to table rows. Once the model classes are defined, a Django database migration command needs to be run to generate the corresponding database tables based on the model classes.
- To create the underlying relational database for a Django model, we need to do the following steps:

1. Check the validity of the model using the check command:

```
python manage.py check
```

2. If the model is correct, you can proceed to creating the migration files (files from which the database will be created and synchronized), using the makemigrations command:

```
python manage.py makemigrations <application name>
```

3. Finally, create the database using the migrate command:

```
python manage.py migrate
```

The admin Site

- Django can automatically creates an admin site for you.
- This site provides an easy way for site administration, and can be used to populate your data base with actual data.
- To create the admin site, follow the following steps:
 1. Create a super user using the `createsuperuser` command. You will be prompted for the user name, email, and password.

python manage.py createsuperuser

2. Edit the `admin.py` file, to import and register your models (tables), for example:

In the `admin.py` file (in the application directory)

```
...  
from .models import Publisher, Author, Book  
  
admin.site.register(Publisher)  
admin.site.register(Author)  
admin.site.register(Book)
```

Testing your data base

To test your database using the admin site, follow the following steps:

1- run the development server using the runserver command:

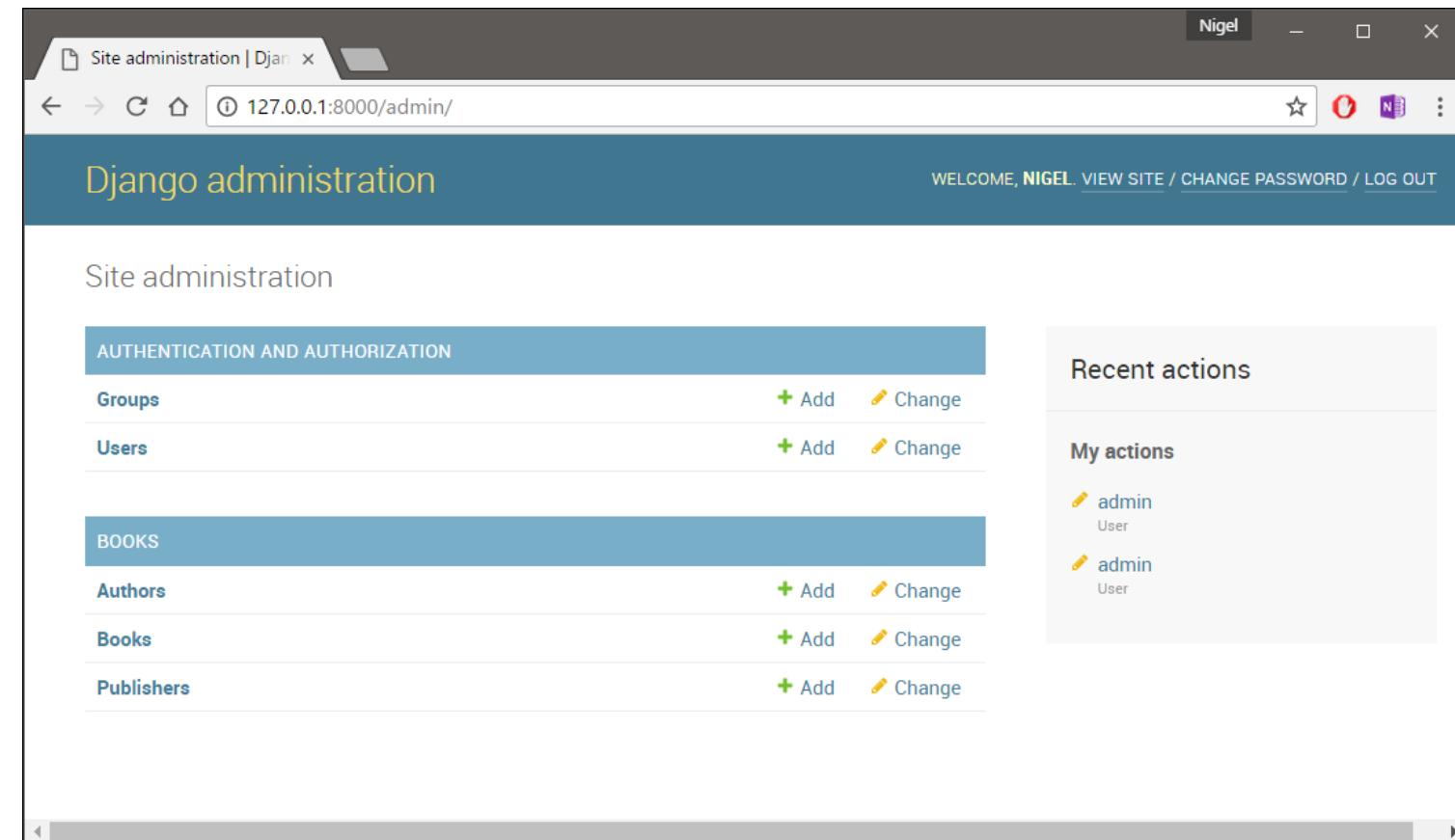
python manage.py runserver

2. Open a web browser and type the url of the admin site, in this case it will be

<http://127.0.0.1:8000/admin/>

3. You will be asked for the admin user and password.

4. You can then see an interface for the database tables where you can populate and edit the database content.



Creating the API view functions

- Now that your database model is ready, you can create an API to interact with the database.
- The first step is to create one function to handle each request. These functions are called **view functions** in Django.
- The functions should be defined in the views.py file (in the application directory).
- The first positional argument that will always be passed to this function is an object of type HttpRequest that encapsulates all the elements of the incoming HTTP request.
- Initially you can define a skeleton function for each of the requests, for example:

```
def HandleRegisterRequest (request):  
    return HttpResponse ('not yet implemented')
```

In the views.py file (in the application directory)

- ❖ HandleRegisterRequest is a Python function that takes a request object as a parameter and returns an HttpResponse object.

Linking view functions to resource URIs

- Each view function should handle a request sent to one of the resources of the API
- Django can automatically call the appropriate view function when a request is received if a resource URI is linked to the functions in the urls.py file (in the project directory)
- To do this, edit the url.py file, and:
 1. Import the view functions from your views.py file
 2. For each function (resource) add a call to path (for fixed URIs), or re_path (for variable URI defined with a regular expressions) to the **urlpatterns** list, for example

The diagram illustrates the structure of a Django URL configuration file (urls.py). It shows a yellow rectangular area containing Python code, with three callout boxes pointing to specific parts of the code:

- The name of your application**: Points to the line `from MyApp.views import Register, List`.
- The URI of the resource**: Points to the line `path('admin/', admin.site.urls),`.
- The view function that will be called when a request is sent to the URI**: Points to the line `path('api/register/', HandleRegisterRequest),`.

```
from MyApp.views import Register, List

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/register/', HandleRegisterRequest),
    path('api/list/', HandleListRequest),
]
```

Testing the view functions

- Before proceeding to detailed implementation of each view function, you may like to test that you can fire the appropriate function by sending a request to the resource associated with that function.
- To do this compose a request (using a command line tool such as curl or HTTPie), or better still by using the Requests python library in a client application.

A python program to send a
GET request to a resource

```
import requests  
r = requests.get('https://api.github.com/events')
```

HttpRequest Objects

- The request argument that is passed to a view function is an instance of an HttpRequest class. This object has a number of useful attributes that can be used to probe the different parts of a request:
- `HttpRequest.body`: The raw HTTP request body as a byte string.
- `HttpRequest.method`: A string representing the HTTP method used in the request. This is guaranteed to be uppercase. e.g.

```
if request.method == 'GET':  
    do_something()  
elif request.method == 'POST':  
    do_something_else()
```

- `HttpRequest.META`: A standard Python dictionary containing all available HTTP headers, e.g:

```
headers = request.META  
content_type = headers ['CONTENT_TYPE']
```
- `HttpRequest.COOKIES` : A standard Python dictionary containing all cookies in the. Keys and values are strings.
- Likewise the `HttpResponse` object has a number of attributes that can be set to return different response types.
- Full description of the `HttpRequest` and `HttpResponse` classes are available in the Django book.

Overriding the default CSRF protection in Django

- Sending a POST request to a view function will create an exception.
- This is because by default Django does not allow POST requests to be sent to view functions without a CSRF (cross site request forgery) token being attached to the request.
- CSRF is a malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts.
- For the time being, we will override this feature by attaching the following python decorator before each view function that will receive a POST request:

```
@csrf_exempt  
def HandleRegisterRequest (request):  
    return HttpResponse ('not yet implemented')
```

That's it Folks



Further Reading

The Django book

Web Services and Web Data

XJCO3011



Session 7. More Django: Accessing and Filtering Data

In this lecture

- All we need now is to implement the business logic for each of the view functions.
 - As we explained, a view is responsible for doing some arbitrary logic, and then returning a response.
 - In modern web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a database-driven website connects to a database server, retrieves some data out of it, and displays that data on a web page. The site might also provide ways for site visitors to populate the database on their own.
 - Many complex websites provide some combination of the two. www.amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially a query into Amazon's product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.
 - Django is well suited for making database-driven websites because it comes with easy yet powerful tools for performing database queries using Python.
- For this, we need to know how to retrieve and manipulate data from that database within the Django models, which we will discuss in this lecture.

Django Girls Tutorial

<https://tutorial.djangogirls.org/en/>

It works for boys as well!

MVC

- Best web frameworks are built around the MVC concept. The MVC design pattern is really simple to understand:
- The **Model(M)** is a model or representation of your data. It's not the actual data, but an interface to the data. The model allows you to pull data from your database without knowing the intricacies of the underlying database. The model usually also provides an *abstraction* layer with your database, so that you can use the same model with multiple databases.
- The **View(V)** is what you see. It's the presentation layer for your model. On your computer, the view is what you see in the browser for a web app, or the UI for a desktop app. The view also provides an interface to collect user input.
- The **Controller(C)** controls the flow of information between the model and the view. It uses programmed logic to decide what information is pulled from the database via the model and what information is passed to the view. It also gets information from the user via the view and implements business logic: either by changing the view, or modifying data through the model, or both.

MTV

- Django follows the MVC pattern closely, however, it does use its own logic in the implementation. Because the C is handled by the framework itself and most of the work in Django happens in models, templates and views, Django is often referred to as an *MTV framework*. In the MTV development pattern:
- **M stands for “Model,”** the data access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data.
- **T stands for “Template,”** the presentation layer. This layer contains presentation related decisions: how something should be displayed on web page or other type of document.
- **V stands for “View,”** the business logic layer. This layer contains the logic that accesses the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates.

Difference between MVC and MTV

Django's **View** is more like the **Controller** in MVC,
and
MVC's **View** is actually a **Template** in Django.

Django Models

A basic book/author/publisher data layout

- The conceptual relationships between books, authors, and publishers are well known; a book that was written by authors and produced by a publisher!
- Some assumptions:
 - An author has a first name, a last name, and an email address.
 - A publisher has a name, a street address, a city, a state/province, a country, and a website.
 - A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship—aka foreign key—to publishers).

Example

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

Step 6 of Lab 2 in models.py file

Creating and retrieving records

Once you've created a model, Django automatically provides a high-level Python API for working with those models.

Creates a new Publisher object

```
>>> from books.models import Publisher  
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',  
...     city='Berkeley', state_province='CA', country='U.S.A.',  
...     website='http://www.apress.com/')
```

Saves the object to the underlying database table database

```
>>> p1.save()  
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',  
...     city='Cambridge', state_province='MA', country='U.S.A.',  
...     website='http://www.oreilly.com/')
```

The QuerySet is a list-like object that contains model objects

```
>>> p2.save()  
>>> publisher_list = Publisher.objects.all()  
>>> publisher_list  
<QuerySet [ <Publisher: Publisher object>, <Publisher: Publisher  
object> ]>
```

import our Publisher model class

Fetch a list of all Publisher objects in the database with the statement Publisher.objects.all()

The objects.create() method

If you want to create an object and save it to the database in a single step, use the objects.create() method

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')

>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')

>>> publisher_list = Publisher.objects.all()

>>> publisher_list
<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher
object>]>
```

Adding Model String Representations

- A `__str__()` method tells Python how to display a human-readable representation of an object.
- Always define this method for each of your models

```
class Author(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=40)  
    email = models.EmailField()  
  
    def __str__(self):  
        return u'%s %s' % (self.first_name, self.last_name)
```

So now, we can get a more meaningful QuerySet (or a desired representation of an object)

```
>>> from books.models import Publisher  
>>> publisher_list = Publisher.objects.all()  
>>> publisher_list  
<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>]>
```

Filtering Data

Naturally, it's rare to want to select everything from a database at once; in most cases, you'll want to deal with a subset of your data. You can filter your data using the `filter()` method:

```
>>> Publisher.objects.filter(name='Apress')  
<QuerySet [<Publisher: Apress>]>
```

You can pass **multiple arguments** into `filter()` to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.",  
state_province="CA")  
<QuerySet [<Publisher: Apress>]>
```

- Retrieve all the Publisher objects from the database where the name attribute is equal to 'Apress'
- The `filter()` method has successfully retrieved a subset of Publisher objects from the database, and returned them as a list-like object containing the matching Publisher objects

Approximate matching of field content can be done by appending **`_contains`** to the field's name, i.e. a double underscore then the word contains.

```
>>> Publisher.objects.filter(name__contains="press")  
<QuerySet [<Publisher: Apress>]>
```

Retrieving Single Objects

- The filter() examples above all returned a **QuerySet**, which you can treat like a list.
- To fetch only a single object, instead of a QuerySet. Use the get() method :

```
>>> Publisher.objects.get(name="Apress")  
<Publisher: Apress>
```

- only a single object is returned

- Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")  
Traceback (most recent call last):  
...  
books.models.MultipleObjectsReturned: get() returned more than one  
Publisher - it returned 2!
```

- A query that returns no objects also causes a `models.DoesNotExist` exception, try to catch it:

```
>>> Publisher.objects.get(name="Penguin")  
Traceback (most recent call last):  
...  
books.models.DoesNotExist: Publisher matching query does not exist.
```

Deleting Objects

- To delete an object from your database, simply call the object's `delete()` method:

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all()
<QuerySet [<Publisher: Apress>, <Publisher: GNW Independent
Publishing>]>
```

- You can also delete objects in bulk by calling `delete()` on the result of any `QuerySet`.

```
>>> Publisher.objects.filter(country='USA').delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all().delete()
(1, {'books.Publisher': 1})
>>> Publisher.objects.all()
<QuerySet []>
```

- deletes all Publisher objects where the country is 'USA'
- deletes all Publisher objects
- queries all Publisher objects

Accessing Foreign Key Values (1)

- When you access a field that's a ForeignKey, you'll get the related model object.

```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
'http://www.apress.com/'
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

- queries the Book object with an id of 50 from the database and assigns it to the variable b.
- accesses the Publisher object associated with the variable b. This is achieved through the foreign key field in the Book model, which establishes a one-to-many relationship between the Book and Publisher models, , where each Book instance is associated with one Publisher instance.)
- accesses the website attribute of the Publisher object associated with the variable b. This is a string representing the website address of the publishing house.

Accessing Foreign Key Values (2)

- With ForeignKey fields, it works the other way, too, but it's slightly different due to the non-symmetrical nature of the relationship. To get a list of books for a given publisher, use `publisher.book_set.all()`, like this:

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.all()
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()
```

- queries the Publisher object with a name of "Apress Publishing" from the database and assigns it to the variable `p`.
- accesses the set of Book objects associated with the variable `p`. This is achieved through the foreign key relationship defined in the Book model, where each Book instance is associated with one Publisher instance. Django automatically adds a property named `book_set` to each Publisher
- The attribute name **book_set is generated by appending the lower case model name to _set**.
- `book_set` is just a QuerySet, and it can be filtered and sliced like any other QuerySet. For example:

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.filter(title__icontains='django')
[<Book: The Django Book>, <Book: Pro Django>]
```

[Chapter 9 from Django Book](#)

- query the Publisher object with a name of "Apress Publishing" from the database, and retrieve a filtered list of associated Book objects based on the substring "django" appearing in their title attribute in a case-insensitive manner

Accessing Many-to-Many Values

Many-to-many values work like foreign-key values, except we deal with QuerySet values instead of model instances. For example, here's how to view the authors for a book:

```
>>> b = Book.objects.get(id=50)
>>> b.authors.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.authors.filter(first_name='Adrian')
[<Author: Adrian Holovaty>]
>>> b.authors.filter(first_name='Adam')
[]
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

It works in reverse, too. To view all of the books for an author, use `author.book_set`, like this:

```
>>> a = Author.objects.get(first_name='Adrian',
last_name='Holovaty')
>>> a.book_set.all()
[<Book: The Django Book>, <Book: Adrian's Other Book>]
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
```

Model methods

- Model methods define custom methods on a model to add custom **row-level functionality** to your objects.
- Model methods act on a particular model instance.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        # Returns the person's baby-boomer status.
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        # Returns the person's full name.
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

- a class called "Person" that inherits from Django's "models.Model" class.
- three attributes: "first_name", "last_name", and "birth_date", which are defined as character fields and a date field

A baby boomer is a person born in the years following the Second World War, when there was a temporary marked increase in the birth rate.

- an attribute named `full_name`, which uses the Python built-in `property` function to convert the `_get_full_name` method into a property.
- This allows accessing the person's full name by accessing the `full_name` attribute of a `Person` object.

Overriding Predefined Model Methods

- There's another set of model methods that encapsulate a bunch of database behaviour that you'll want to customize. In particular, you'll often want to change the way `save()` and `delete()` work.
- A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example, to update **a calculated field** (such as the arrival date for a flight)

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real"
        save() method.
        do_something_else()
```

- It calls a custom `"do_something()"` function before saving the instance.
- It calls the `"super()"` method to invoke the original `"save"` method from the parent class, passing any arguments and keyword arguments received by this method.
- It calls another custom `"do_something_else()"` function after saving the instance.

The QuerySet

- QuerySets are list-like objects
- A QuerySet is iterable, e.g.:

```
for e in Entry.objects.all(): # retrieve all objects of the Entry model from the database, and  
then iterates over each object and prints its headline attribute.  
    print(e.headline)
```

- It can be sliced like any Python list.
- It has the **values()** method that returns dictionaries, rather than model instances, e.g.:

```
# This list contains a Blog object.  
>>> Blog.objects.filter(name__startswith='Beatles')  
<QuerySet [<Blog: Beatles Blog>]>  
  
# This list contains a dictionary.  
>>> Blog.objects.filter(name__startswith='Beatles').values()  
<QuerySet [{"id": 1, "name": 'Beatles Blog', "tagline": 'All the latest Beatles  
news.'}]>
```

Example

Implementing the view function for the directory List service

List

Service Aim: to get a list of all agencies in the directory

Service Details:

The client sends a **GET** request to **/api/list/** with an empty payload.

If the request is processed successfully, the server responds with **200 OK** and the list of agencies ("agency_list", array) in a JSON payload. For each agency in the list, the following data must be given:

- The name of the agency ("agency_name", string)
- The URL of the agency's website ("url", string)
- The agency's unique code ("agency_code", string)

If the server is unable to process the request for any reason, the server should respond with the appropriate error code.

```
112 def List (request):
113     # prepare a bad request-object to return to the client if an error is encountered
114     http_bad_response = HttpResponseBadRequest ()
115     http_bad_response['Content-Type'] = 'text/plain'
116
117     if (request.method != 'GET'):
118         http_bad_response.content = 'Only GET requests are allowed for this resource\n'
119         return http_bad_response
120
121
122     # if reach this point then this means that it is a good request
123     # get the list of companies from the database
124     company_list = DirectoryEntry.objects.all().values('BusinessName' , 'BusinessType' , 'BusinessURL' , 'BusinessCode')
125     # collect the list items and put a new list with appropriate json names as per requirements
126     the_list = []
127     for record in company_list:
128         item = {'agency_name': record['BusinessName'] , 'url' : record['BusinessURL'] , 'agency_code': record['BusinessCode']}
129         the_list.append(item)
130     # now make the final json response payload
131     payload = {'agency_list' : the_list}
132     # create and return a normal response
133     http_response = HttpResponse (json.dumps(payload))
134     http_response['Content-Type'] = 'application/json'
135     http_response.status_code = 200
136     http_response.reason_phrase = 'OK'
137     return http_response
```

- **views.py** is where we handle the request/response logic for our web app

That's it Folks



Further Reading

The Django Book

XJCO3011: Web Services and Web Data



Session #8– Designing a RESTful System

Instructor: Dr. Guilin Zhao
Spring 2023

Design Process

1. Determine overall system architecture.
2. Clearly describe processes and procedures.
3. Design the database models.
4. Design the Web API.
 - Determine required services.
 - Assign URLs.
 - Choose suitable media types.
 - Media types are used in HTTP to indicate the type of content being transferred in the body of an HTTP message. Media types are used to identify these different types of data. such as HTML documents, images, audio, video, and so on.
 - Clearly define request and response messages.

Example *Flight Aggregation Systems*

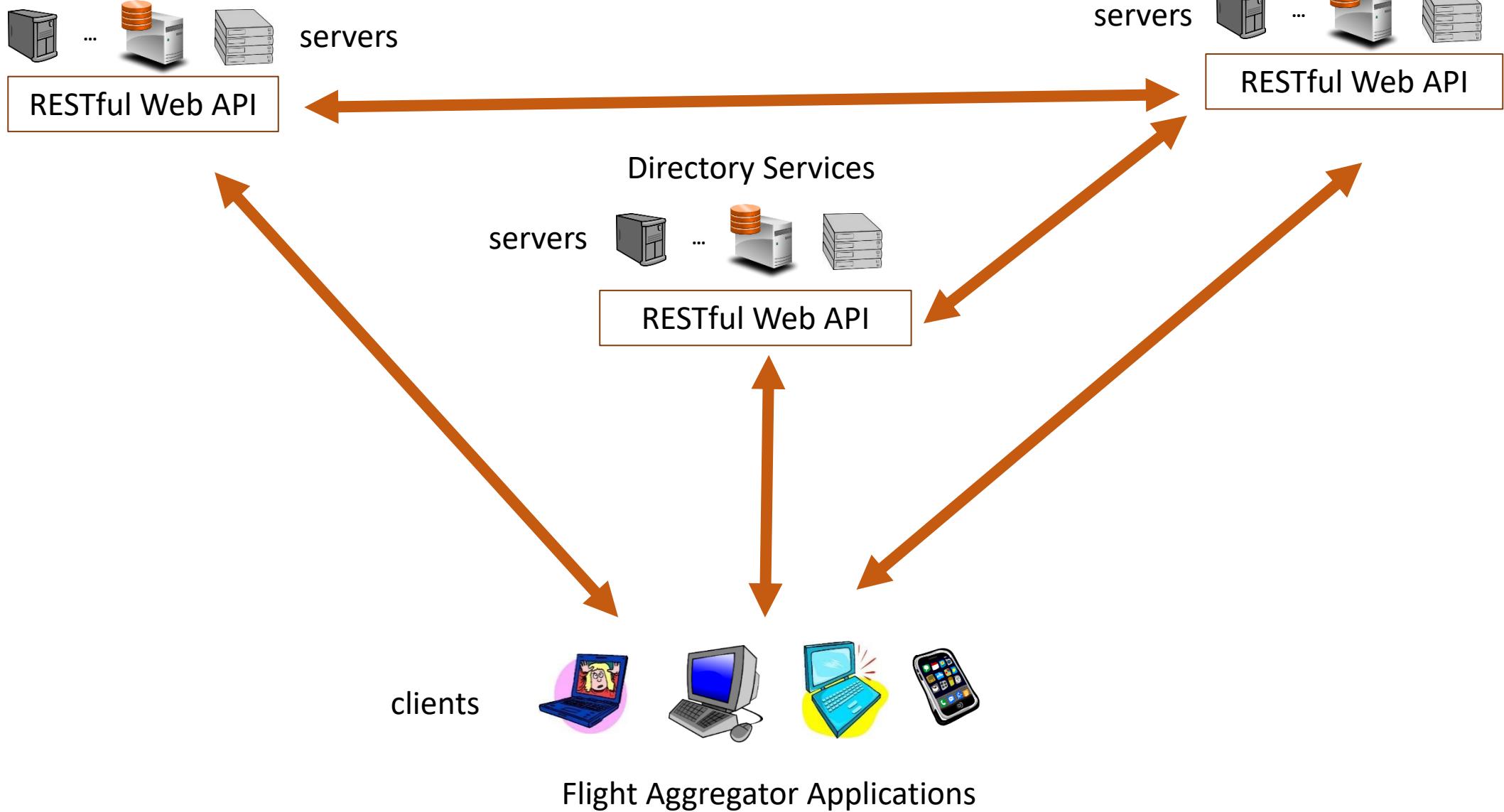
- People trying to book a flight do not search the website of one single airline
- Most people **shop** around in **many airlines**, then choose the flight that is most suitable for them in terms of **price** and **total journey time** (and some other factors).
- **Flight search aggregator** applications have emerged to assist customers to **find the best flight** for their needs, and then to **book and pay for it online**.

Payment Service Providers



The System Architecture

Airlines



The Booking Process

1. Client sends a request to find a flight (request)
2. Airline server responds with a list of candidate flights (response)
3. User selects a flight to book
4. Client sends a request to book the flight (request)
5. Airline server responds with booking number (response)
6. Client sends a request for available payment methods (request)
7. Airline responds with list of payment providers (response)
8. User chooses a payment provider
9. Client sends a request to pay for the booking with this provider (request)
10. Airline server contact payment server to create an electronic bill
11. Payment server responds to airline server with electronic bill
12. Airline responds to client with electronic invoice (response)
13. Client sends a request to payment provider to pay the electronic bill
 - A. If the user (client) is not logged in to the payment server the server responds with a prompt to log in
 - B. The user logs in (if necessary)
14. Payment server makes sure that *the user does have an account and enough balance*
15. Payment server responds with electronic receipt (confirmation of payment)
16. Client sends receipt to airline server (request)
17. Airline finalizes booking and responds with the confirmed booking details (response)

User



Client Software



1. Find me a flight

2. find a flight

3. list of flights

6. book a flight

7. booking number

8. request payment
methods

9. list of payment
providers

12. pay for booking

15. electronic invoice

16. pay invoice

10. select a payment
provider

11. provider selected

log in
user name & pass.

Airline Sever



The Booking Process

Payment Service
Server



13. request
electronic invoice

14. electronic invoice

17. Not logged in

log in success

18. electronic stamp

19. electronic stamp

The Airline Database Model

- A **database model** is a blueprint or a set of rules that defines the structure, relationships, and constraints of data stored in a database.
- It's essential to have a database model to **store and manage data efficiently** (in data organization, data consistency, scalability, security, maintenance).
- It provides a framework for organizing, securing, scaling, and maintaining data. It ensures that the system can handle large amounts of data, enforce data consistency, and protect data from unauthorized access.

Aircraft

- An airline business needs a number of *aircraft*
- For each aircraft the airline must maintain the following information:
 - The aircraft **type** (e.g. Airbus A320)
 - A unique tail or registration **number**
 - ❖ An aircraft tail number or registration number.
 - ❖ Each aircraft has a unique identifier to ensure accurate identification and tracking within the aviation industry.
 - The seating **capacity** (e.g. 150)
 - Other technical information (number of engines ...)
 - This data can be maintained in a relational data base table such as this:



Primary Key	Tail Number	Type	Seats	Other
	G-STBA	Airbus A320	150	

Airports

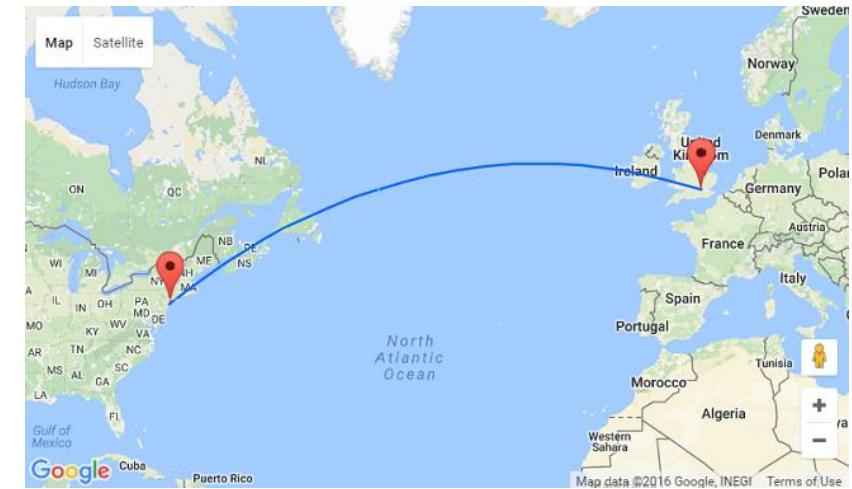
- An airline must maintain a list of airports travelled to by the company
- For each airport the airline must maintain the following information:
 1. The airport unique **name** (e.g. New York JFK, Chengdu CTU)
 2. **Country** (e.g. USA)
 3. The time zone of the airport (e.g. USA EASTERN EST (**Eastern Standard Time**)), why ?
 4. Other information
- This data can be maintained in a relational data base table such as this:



Primary Key	Name	Country	Time Zone	Other
	New York JFK	USA	USA EST	

Flights

- An airline must provide **flights**
- For each flight the following information must be maintained:
 1. A unique **flight number** to distinguish it from other flights (e.g. BA1349)
 2. Departure airport (e.g. London Heathrow LHR)
 3. Destination airport (e.g. New York JFK)
 4. Departure date-time (e.g. 2018.04.01, 14:45:00)
 5. Arrival date-time
 6. Flight duration (e.g. 5 hours)
 7. The aircraft used for this flight
 8. Seat cost
 9. Seat price
 10. Other



Primary Key	Flight Num.	Departure Airport (FK)	Destin. Airport (FK)	Departure DateTime	Arrival DateTime	Duration	Aircraft (Foreign Key)	Seat Price	Seat Cost
	BA1349	→	→	2018.04.01	-	5:00	→	£500	£450
	BA1349	→	→	2018.04.02	-	5:00	→	£500	£450

FK

- "FK" typically refers to a **foreign key**.
- A foreign key is a column or set of columns in a database table that refers to the primary key of another table. It is used to establish a link or relationship between two tables in a database. FK can be used to link related resources and ensure consistency between them.
 - E.g., consider a database with two tables: "Orders" and "Customers". The "Orders" table has a foreign key column "customer_id" that refers to the "id" column in the "Customers" table. This establishes a one-to-many relationship between the two tables, where each order is associated with a single customer, and each customer can have many orders.

Primary Key	Flight Num.	Departure Airport (FK)	Destin. Airport (FK)	Departure DateTime	Arrival DateTime	Duration	Aircraft (Foreign Key)	Seat Price	Seat Cost
	BA1349	→	→	2018.04.01	-	5:00	→	£500	£450
	BA1349	→	→	2018.04.02	-	5:00	→	£500	£450

Bookings

- To allow passengers to book seats on a flight, an airline provides flight bookings.
- For each flight booking the following information must be maintained:
 - A unique **booking number** to distinguish it from other bookings (e.g. WXY12Z)
 - The **flight** of this booking
 - Number of seats booked
 - Details of each passenger (full name, nationality, passport number)
 - Booking status (e.g. ON_HOLD, CONFIRMED, CANCELLED, TRAVELED ...)
 - Other information

This is NOT one of the fields in the relational database table. I have just added it to indicate that there exists a many-to-many relation with the passengers table through the join table

PK	Booking Number	Flight (FK)	No Seats	Passengers	Status	Other
8	WXY12Z	→	2	ManyToMany	ON_HOLD	

passenger = models.ManyToManyField(Passenger)

PK	Booking FK	Passenger FK
50	8	12
51	8	13

Join Table (passenger on each booking)

In the Django model this relation is created like this

PK	Name	Nationality	Passport Number
12	John West	UK	123456789
13	Ben East	UK	987654321

Many-to-Many

- Flight bookings and passengers have a many-to-many relationship, which means that a single flight booking can have multiple passengers, and a single passenger can have multiple flight bookings.
- In the context of a flight booking system, this relationship can be represented by a junction table that links the flight booking table and the passenger table. This junction table typically includes foreign keys to the flight booking and passenger tables, creating a many-to-many relationship between the two.
- This type of relationship is common in many industries, and it's often represented in database systems using a junction or bridging table to establish the association between the two entities.
- It allows for greater flexibility and scalability when managing large amounts of data related to multiple entities, and it also makes it possible to retrieve and query data more efficiently.

Other Tables

Payment providers

To allow an airline to accept payments from customers, an airline must have accounts with a number of payment service providers. For each account the airline must maintain the following data:

- The name of the payment service provider (e.g. SalPay)
- The address of the website of the payment service provider (e.g. ‘www. salpay.co.uk’)
- The account number.
- The login name and password (this is needed when the airline server must access their account to create an electronic invoice for a customer).

Invoices

To be able to charge customers and accept payment, the airline must ask a payment service provider to create an invoice (see below). A copy of the invoice is kept at the airline database to match against customer payments. For each invoice the following information is maintained in the database:

- A unique reference number for this invoice at the airline’s database.
- The unique reference number for this invoice within the database of the payment service provider.
- The booking number for which the invoice was issued.
- The amount of the invoice.
- A Boolean value indicating whether this invoice was paid or not.

A unique 10 digit alphanumeric code (electronic stamp). This key is generated by the payment service provider when the invoice is created. The airline should remove this code from the invoice before forwarding it to the client. When payment is made for this invoice through the payment provider’s website, the payment provider releases this code to the client (payer). The client will then send this electronic stamp code to the airline as a confirmation of payment. The airline verifies this code against that stored in its copy of the invoice and confirms payment.

The Airline Web API

- What **web services** (web APIs) **the airline must provide** to client applications to facilitate finding and booking a flight, and what data must be exchanged between clients and servers for each service (request)?
- We can answer this question by **examining the steps of the booking process** and identify those related to the **client's interaction with the airline web API**, then:
 1. Decide if this should be a **GET** or **POST** request.
 2. Assign a **resource URI** for this service.
 3. Identify the data the client must provide in order for the server to be able to serve the request
 4. Identify the appropriate response and the data the server must send back to the client to fulfil the request
 5. Decide on a media type for the data in the response
- Remember that a **GET** request should be used **when we do NOT intend to change the state** of the server (read only), while a **POST** request are used **when we need to change the server's state** (write or modify data).

The API

Payment Service Providers



RESTful Web API
/signup
/signin
/deposit
/pay
/transfer
/balance
/statement

Directory Services



RESTful Web API

Airlines



RESTful Web API
/findflight
/bookflight
/paymentmethods
/payforbooking
/finalizebooking
/bookingstatus
/cancelbooking

Hypermedia

Hypermedia

Hypermedia



Flight Aggregator Applications

Client sends a request to find a flight and server responds with a list of candidate flights

Service Aim: *Find a flight from a departure airport to a destination airport at some departure date.*

Service Details:

The client sends a **GET** request to **/api/findflight** with the following data:

1. Departure airport ("dep_airport", string)
2. Destination airport ("dest_airport", string)
3. Departure date ("dep_date", string)
4. Number of passengers ("num_passengers", number)
5. A Boolean value indicating whether the departure date is exact or flexible ("is_flex", true or false)

If the request is processed successfully, the server responds with **200 OK** and a **list of flights** in a JSON payload. For each flight in the list, the following data must be provided:

1. Flight identifier that uniquely identifies a flight on a certain date ("flight_id", string)
2. Flight number ("flight_num", string)
3. Departure airport ("dep_airport", string)
4. Destination airport ("dest_airport", string)
5. Departure date and time ("dep_datetime", string)
6. Arrival date and time ("arr_datetime", string)
7. Flight duration ("duration", string)
8. Seat price for one passenger ("price", number)

If no flights are found the server should respond with **503 Service Unavailable** with **text/plain** payload giving reason.

Client sends a request to book a flight and server responds with booking number

Service Aim: Book a flight given by its unique identifier.

Service Details:

The client sends a **POST** request to **/api/bookflight** with the following data in a JSON payload:

1. Flight unique identifier ("flight_id", string)
2. A list of passengers ("passengers", array) with the following details about each passenger:
 - First name ("first_name", string)
 - Surname ("surname", string)
 - email ("email", string)
 - phone number ("phone", string)

If the request is processed successfully, the server responds with **201 CREATED** and a JSON payload containing the following data:

1. Booking number ("booking_num", string)
2. Booking status ("booking_status", string) = "ON_HOLD"
3. Total price for this booking ("tot_price", number)

If a booking cannot be made for any reason (e.g. no seats are available), the server should respond with **503 Service Unavailable** with **text/plain** payload giving reason

Client sends a request for available payment methods and server responds with a list of payment providers

Service Aim: Request Payment Methods

Service Details:

The client sends a **GET** request to **/api/paymentmethods** :

If the request is processed successfully, the server responds with **200 OK** and a **list of payment service providers** ("pay_providers", array) in a JSON payload. For each provider in the list, the following data must be given:

1. The provider unique identifier in the airline's database ("pay_provider_id", string)
2. Provider name ("pay_provider_name", string)

If no providers are available, the server should respond with **503 Service Unavailable** with **text/plain** payload giving reason.

Client sends a request to pay for a booking with a certain provider and server responds with electronic invoice

Service Aim: Pay for a Booking

Service Details:

The client sends a **POST** request to **/api/payforbooking** with the following data in a JSON payload:

1. Booking number ("booking_num", string)
2. Payment provider identifier ("pay_provider_id", string)

If the request is processed successfully, the server responds with **201 CREATED** and a JSON payload with the following data:

1. Payment provider identifier ("pay_provider_id", string)
2. Payment provider invoice unique id ("invoice_id", string)
3. Booking number ("booking_num", string)

If the server is unable to process the request for any reason (e.g. could not establish connection to the server of the payment service provider), the server should respond with a **503 Service Unavailable** with **text/plain** payload giving reason.

Client sends a request to finalize (confirm) a booking and server responds with the confirmed booking details

Service Aim: Finalize a Booking

Service Details:

The client sends a **POST** request to **/api/finalizebooking** with the following data in a JSON payload:

1. Booking number ("booking_num", string)
2. Payment provider identifier ("pay_provider_id", string)
3. Payment provider receipt id ("receipt_id", string)

If the request is processed successfully, the server responds with **201 CREATED** and a JSON payload with the following data:

1. Booking number ("booking_num", string)
2. Booking status ("booking_status", string) = "CONFIRMED"

If the server is unable to process the request for any reason, the server should respond with a **503 Service Unavailable** with **text/plain** payload giving reason.

Client sends a request to know the status of a booking and server responds with the booking details and status

Service Aim: Provide Booking Status

Service Details:

The client sends a **GET** request to /api/bookingstatus with the following data:

1. Booking number ("booking_num", string)

If the request is processed successfully, the server responds with **200 OK** and a JSON payload with the following data:

1. Booking number ("booking_num", string)
2. Booking status ("booking_status", string)
3. Flight number ("flight_num", string)
4. Departure airport ("dep_airport", string)
5. Destination airport ("dest_airport", string)
6. Departure date and time ("dep_datetime", string)
7. Arrival date and time ("arr_datetime", string)
8. Flight duration ("duration", string)

If the server is unable to process the request for any reason, the server should respond with a **503 Service Unavailable** with **text/plain** payload giving reason.

Client sends a request to cancel a booking and server responds with the booking status

Service Aim: Cancel a Booking

Service Details:

The client sends a **POST** request to /api/cancelbooking with the following data in a JSON payload:

1. Booking number ("booking_num", string)

If the request is processed successfully, the server responds with **200 OK** and a JSON payload with the following data:

1. Booking number ("booking_num", string)
2. Booking status ("booking_status", string) "CANCELLED"

If the server is unable to process the request for any reason, the server should respond with a **503 Service Unavailable** with **text/plain** payload giving reason.

Web Services and Web Data

XJCO 3011



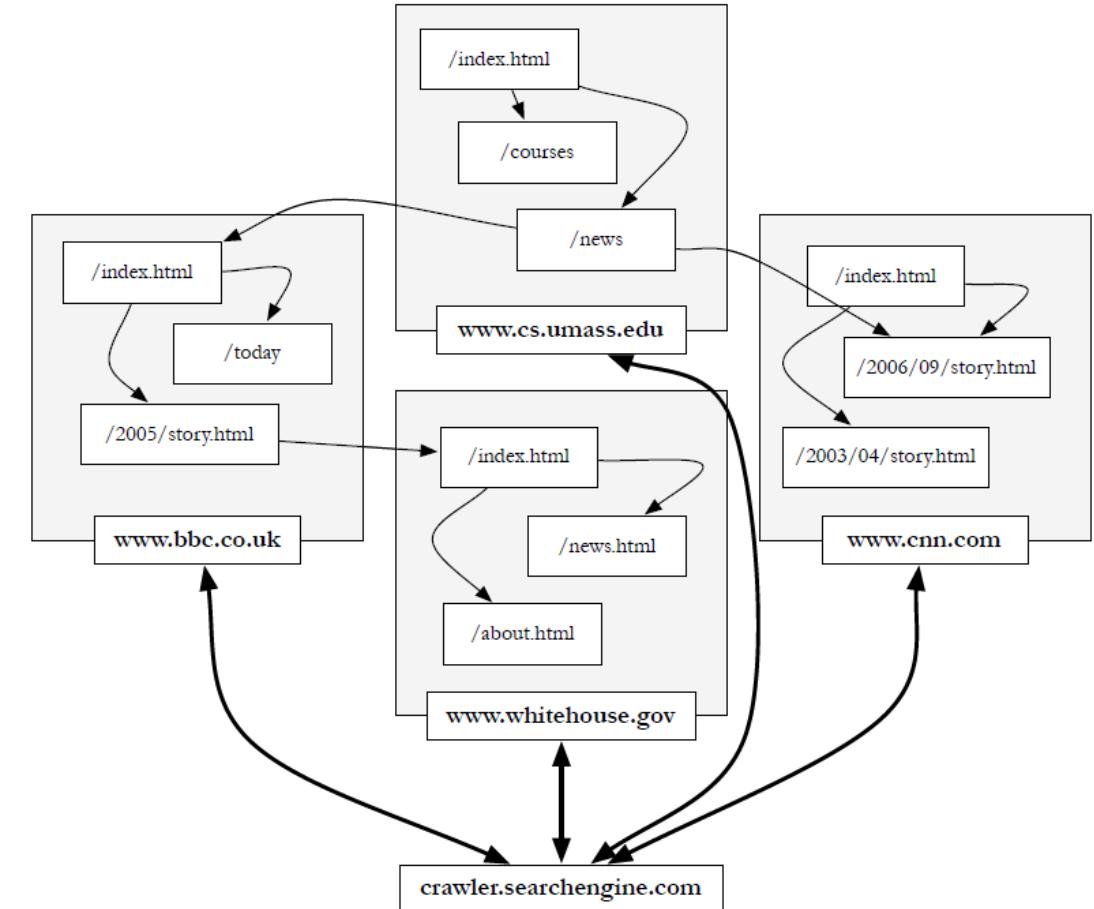
Session 9. Web Crawling

Crawlers for Search Engines

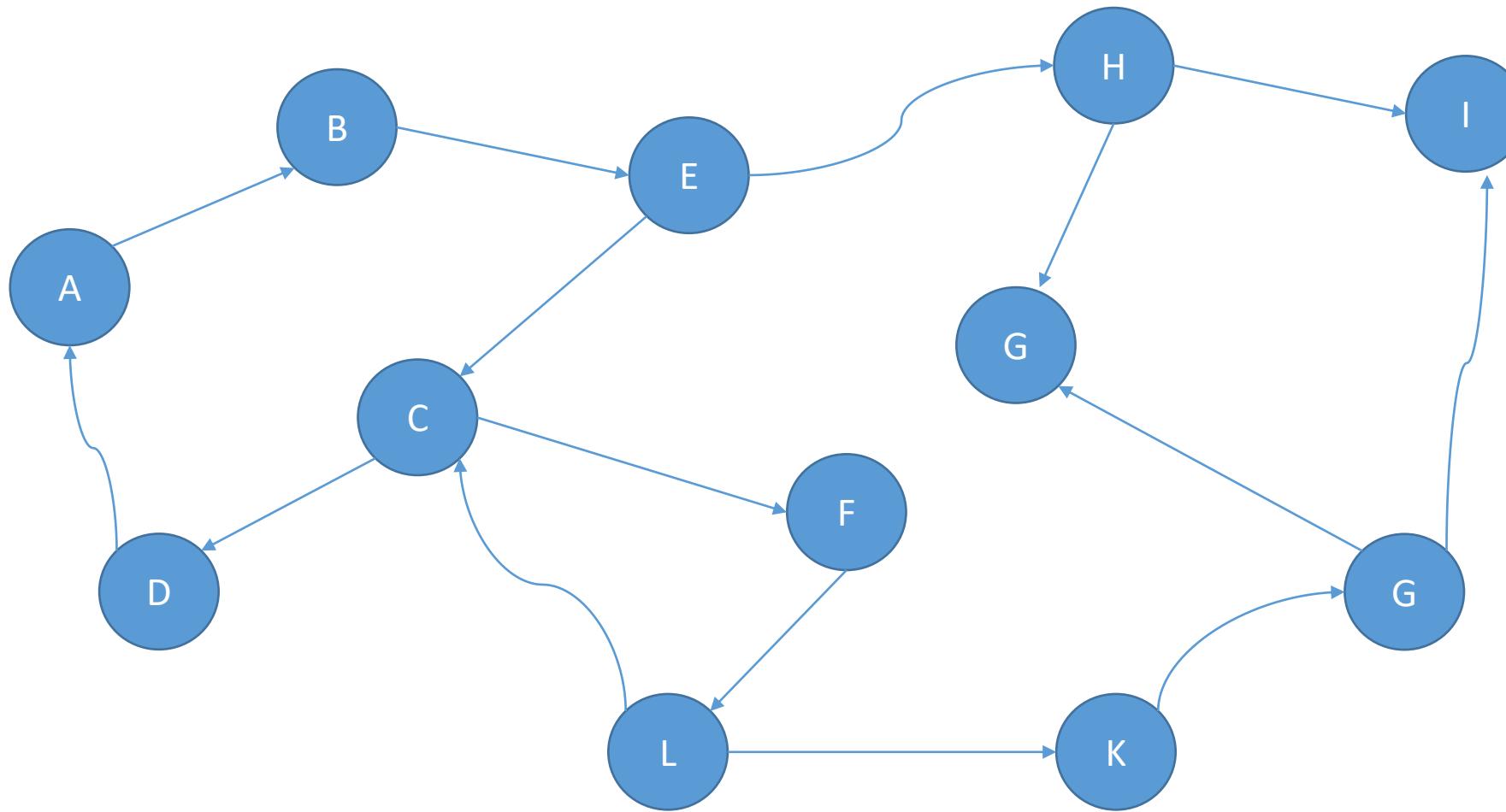
- To build a search engine, you first need a copy of the pages that you want to search. Unlike some of the other sources of text we will consider later, web pages are particularly easy to copy, since they are meant to be retrieved over the Internet by browsers. This instantly solves one of the major problems of getting information to search, which is how to get the data from the place it is stored to the search engine.
- Finding and downloading web pages automatically is called **crawling or spidering**, and a program that downloads pages is called a **web crawler**.
- There are some unique challenges to crawling web pages:
 1. The sheer scale of the Web (billions of pages on the Internet)
 2. Pages are constantly being created and removed.
 3. Web pages are usually not under the control of the people building the search engine.
 4. Website owners do not like crawlers.
 5. Some of the data you want to copy may be available only by typing a request into a form, which is a difficult process to automate.

How Web Crawlers Work

- The web crawler has two jobs: **downloading** pages and **finding URLs**.
- The crawler starts with a set of **seeds**, which are a set of URLs given to it (the crawler) as parameters.
- These seeds are added to a URL request **queue**.
- The crawler starts **fetching** pages from the request queue. Once a page is **downloaded**, it is parsed to **find link tags** that might contain other useful URLs to fetch.
- If the crawler finds a new URL that it has not seen before, it is added to **the crawler's request queue**, or **frontier**.
- The frontier may be a standard queue, or it may be ordered so that **important pages move to the front of the list**.
- This process continues until the crawler either runs out of disk space to store pages or runs out of useful links to add to the request queue.



Web Crawling is a Kind of Graph Traversal



- A web crawler starts by visiting a specific website or URL. The crawler then follows the links on that page to other pages, and continues to follow links until it has visited all the pages it needs to.

Web Crawlers Are Not Always Welcome

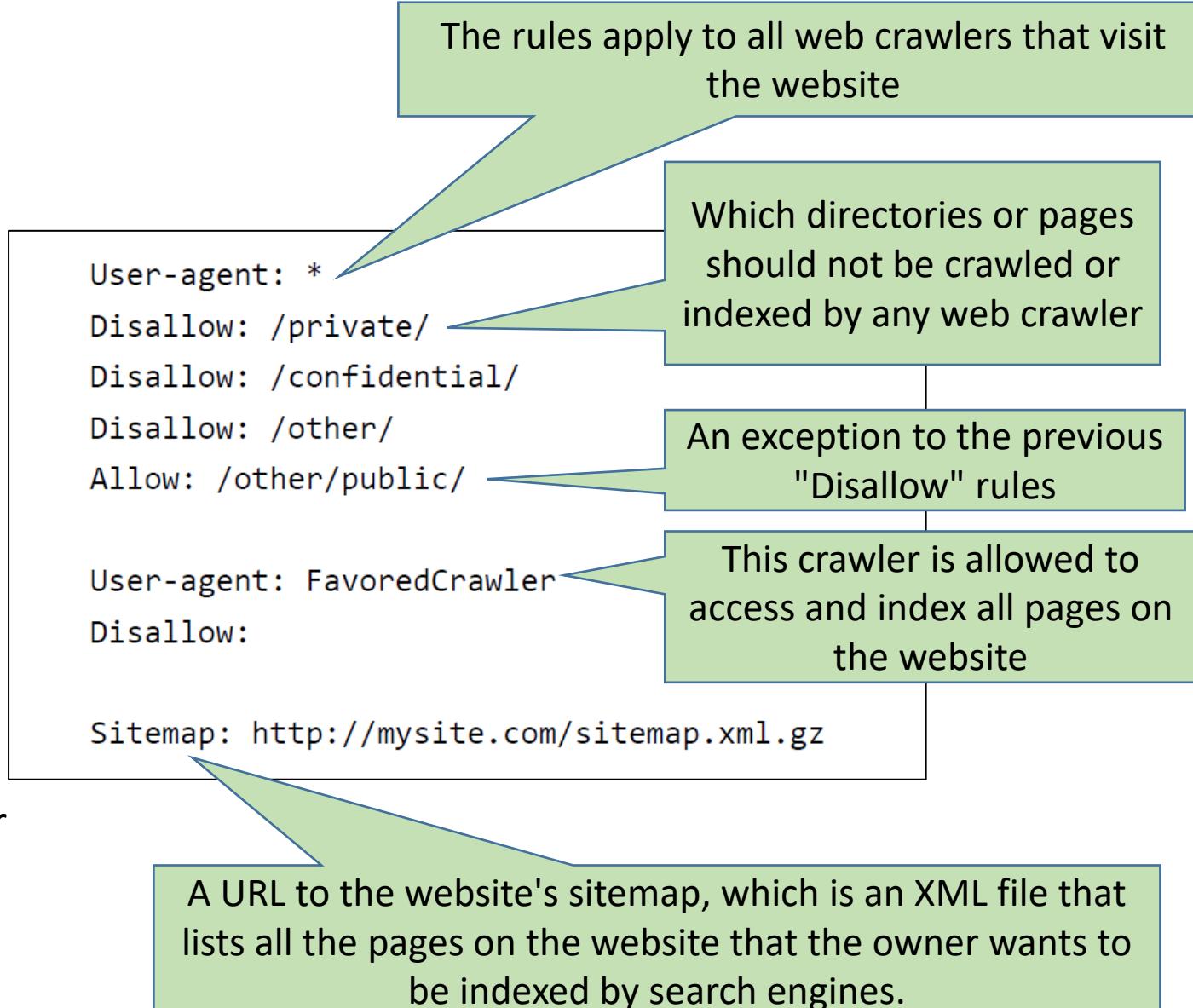
- Fetching hundreds of pages at once is good for the web crawler, but not good for the web server on the other end.
- If the web server is not very powerful, it might spend all of its time handling requests from crawlers instead of handling requests from 'real' users
- This kind of behaviour from web crawlers can make web server administrators very angry
- They can consequently block the IP address from which the crawler is working.
- To avoid this problem, web crawlers use **politeness policies**.
 1. Reasonable web crawlers do **not** fetch **more than one page at a time** from a **particular web server**.
 2. In addition, web crawlers **wait at least a few seconds**, and sometimes minutes, between requests to the same web server.
 3. To support this, the **request queue is logically split into a single queue per web server**.
 4. The crawler is free to read page requests only from queues that haven't been accessed within the specified politeness window.

The request queue can be very large

- Suppose a web crawler can fetch 100 pages each second, and that its politeness policy dictates that it cannot fetch more than **one page each 30 seconds from a particular web server**.
- The web crawler needs to have URLs from at least 3,000 **different web servers** in its request queue in order to achieve the required 100 pages/second throughput.
- Since many URLs will come from the same servers, the request queue needs to have **tens of thousands of URLs** in it before a crawler can reach its peak throughput.

The robots.txt file

- Even crawling a site slowly will anger some web server administrators who object to any copying of their data. Web server administrators who feel this way can store a file called **/robots.txt** on their web servers.
- The file is split into blocks of commands that start with a **User-agent: specification**.
- The User-agent: line identifies a crawler, or group of crawlers, affected by the following rules.
- Following this line are **Allow** and **Disallow** rules that dictate which resources the crawler is allowed to access.
- An optional Crawl-delay (non standard extension) can be included.



Detecting Robots

Websites can 'catch' unwanted crawlers in many ways:

- Very **quick page requests** to the same website is the first and most obvious feature of a badly designed robot. Hence, a robot should always respect the fair use policy of websites and servers by allowing sufficient and reasonable time between requests.
- A hidden **trap link** in some pages (for example /trap_unwanted_robots) that human users cannot see (and hence will not click on). However, When a naive robot 'steps' into this 'booby trap', it is detected and further action can be taken against the robot. Robots can avoid this kind of trap by **reading the robots.txt file**, which usually points to these robot trap links.
- A dynamically created link to a sequence **fictitious resources** that never terminates, for example /followme/1/1, /followme/1/2, /followme/1/3, This will cause naive robots to go into infinite recursion, and destabilizes the robot. Crawlers, can avoid this by having a limit on the **maximum number of levels it can go deep down the same page hierarchy**.
- The User-agent HTTP header field in a request usually contains information about the user agent (web browser). Unwanted bots which identify themselves through this field can be black listed. Crawlers can avoid this by changing their User-agent name (**spoofing**).
- Crawlers that make requests to the same website at a regular and monotonous rate can also be easily detected, hence sophisticated bots should randomize the timing of page requests.

```
# section 1
User-agent: BadCrawler
Disallow: /

# section 2
User-agent: *
Crawl-delay: 5
Disallow: /trap

# section 3
Sitemap: http://example.webscraping.com/sitemap.xml
```

How do you know if your robot is being blocked

- CAPTCHA pages.
- Unusual content delivery delay.
- Frequent appearance of these status codes is also indication of blocking:
 - 301 Moved Temporarily
 - 401 Unauthorized
 - 403 Forbidden
 - 404 Not Found
 - 408 Request Timeout
 - 429 Too Many Requests
 - 503 Service Unavailable

A Crawler Thread

```
procedure CRAWLERTHREAD(frontier)
    while not frontier.done() do
        website ← frontier.nextSite()
        url ← website.nextURL()
        if website.permitsCrawl(url) then
            text ← retrieveURL(url)
            storeDocument(url, text)
            for each url in parse(text) do
                frontier.addURL(url)
            end for
        end if
        frontier.releaseSite(website)
    end while
end procedure
```

- The crawler first retrieves a website from the frontier.
- The crawler then identifies the next URL in the website's queue.
- The crawler checks to see if the URL is okay to crawl according to the website's robots.txt file.
- If it can be crawled, the crawler fetches the document contents. This is the most expensive part of the loop, and the crawler thread may block here for many seconds.
- Once the text has been retrieved the document is stored in the database, the document text is then parsed so that other URLs can be found.
- These URLs are added to the frontier, which adds them to the appropriate website queues.
- When all this is finished, the website object is returned to the frontier, which takes care to enforce its politeness policy by not giving the website to another crawler thread until an appropriate amount of time has passed.

Freshness

- Web pages are constantly being added, deleted, and modified.
- To keep an accurate view of the Web, a web crawler must continually revisit pages it has already crawled to see if they have changed in order to maintain the freshness of the document collection.
- The opposite of a fresh copy is a **stale** copy
- The HTTP HEAD request, which only returns header information, is useful for checking for page changes.
- The Last-Modified value indicates the last time the page content was changed. This compared with the date it received from a previous GET request for the same page.
- A HEAD request reduces the cost of checking on a page, but does not eliminate it. It simply is not possible to check every page every minute.
- It does little good to continuously check sites that are rarely updated.
- Hence, a crawler can be made to learn about the frequency of change of a web page, and guess when it is time to check the freshness of a page.

Client request:	HEAD /csinfo/people.html HTTP/1.1 Host: www.cs.umass.edu
Server response:	HTTP/1.1 200 OK Date: Thu, 03 Apr 2008 05:17:54 GMT Server: Apache/2.0.52 (CentOS) Last-Modified: Fri, 04 Jan 2008 15:28:39 GMT ETag: "239c33-2576-2a2837c0" Accept-Ranges: bytes Content-Length: 9590 Connection: close Content-Type: text/html; charset=ISO-8859-1

Sitemaps

- A robots.txt file can contain a reference to a sitemap. A sitemap contains a list of URLs and data about those URLs, such as modification time and modification frequency.
- There are three URL entries shown in the example sitemap.
- Each one contains a URL in a **loc tag**.
- The **changefreq tag** indicates how often this resource is likely to change.
- The **lastmod tag** indicates the last time it was changed.
- The first entry also includes a **priority tag** with a value of 0.7, which is higher than the default of 0.5. This tells crawlers that this page is more important than other pages on this site.
- A sitemap tells search engines about pages it might not otherwise find (e.g. because they require form entry).

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.company.com/</loc>
    <lastmod>2008-01-15</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.7</priority>
  </url>
  <url>
    <loc>http://www.company.com/items?item=truck</loc>
    <changefreq>weekly</changefreq>
  </url>
  <url>
    <loc>http://www.company.com/items?item=bicycle</loc>
    <changefreq>daily</changefreq>
  </url>
</urlset>
```

Sitemap XML tags

Attribute	Description
<urlset>	required Encapsulates the file and references the current protocol standard.
<url>	required Parent tag for each URL entry. The remaining tags are children of this tag.
<loc>	required URL of the page. This URL must begin with the protocol (such as http) and end with a trailing slash, if your web server requires it. This value must be less than 2,048 characters.
<lastmod>	optional The date of last modification of the file. This date should be in W3C Datetime format. This format allows you to omit the time portion, if desired, and use YYYY-MM-DD. Note that this tag is separate from the If-Modified-Since (304) header the server can return, and search engines may use the information from both sources differently.
<changefreq>	optional How frequently the page is likely to change. This value provides general information to search engines and may not correlate exactly to how often they crawl the page. Valid values are: <ul style="list-style-type: none"> • always • hourly • daily • weekly • monthly • yearly • never The value "always" should be used to describe documents that change each time they are accessed. The value "never" should be used to describe archived URLs. Please note that the value of this tag is considered a <i>hint</i> and not a command. Even though search engine crawlers may consider this information when making decisions, they may crawl pages marked "hourly" less frequently than that, and they may crawl pages marked "yearly" more frequently than that. Crawlers may periodically crawl pages marked "never" so that they can handle unexpected changes to those pages.
<priority>	optional The priority of this URL relative to other URLs on your site. Valid values range from 0.0 to 1.0. This value does not affect how your pages are compared to pages on other sites—it only lets the search engines know which pages you deem most important for the crawlers. The default priority of a page is 0.5. Please note that the priority you assign to a page is not likely to influence the position of your URLs in a search engine's result pages. Search engines may use this information when selecting between URLs on the same site, so you can use this tag to increase the likelihood that your most important pages are present in a search index. Also, please note that assigning a high priority to all of the URLs on your site is not likely to help you. Since the priority is relative, it is only used to select between URLs on your site.

Distributed Crawling

- For crawling individual websites, a single computer is sufficient.
- However, crawling the entire Web requires many computers devoted to crawling to:
 1. put the crawler closer to the sites it crawls,
 2. reduce the number of sites the crawler has to remember (the queue and the visited list), and
 3. reduce individual computing resources, including CPU resources for parsing and network bandwidth for crawling pages. Crawling a large portion of the Web is too much work for a single computer to handle.
- In a distributed crawler there are many queues. The distributed crawler uses a **hash function**, on the URL's host part, to assign URLs to crawling computers.
- When a crawler sees a new URL, it computes a hash function on that URL to decide which crawling computer is responsible for it.
- These URLs are gathered in batches, then sent periodically to reduce the network overhead of sending a single URL at a time.

The Conversion Problem

- Search engines are built to search through text.
- Unfortunately, text is stored on computers in hundreds of incompatible file formats, such as plain text, RTF, HTML, XML, Microsoft Word, ODF (Open Document Format) and PDF (Portable Document Format), to name just a few.
- It is not uncommon for a commercial search engine to support more than a hundred file types.
- The most common way to handle a new file format is to use a [conversion tool](#) that converts the document content into a tagged text format such as HTML or XML.
- Documents could be converted to plain however this would strip the file of information about headings and font sizes that could be useful to the indexer.
- Headings and bold text tend to contain words that describe the document content well, so these words must be given preferential treatment during scoring.
 - Scoring: used to determine the order in which search results are presented. The score is usually based on the importance of the keywords found in the document

That's it Folks



Further Reading

Chapter 3: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman

Web Services and Web Data

XJCO3011



Session 10. Parsing and Tokenization

Document Parsing

- Document parsing involves the recognition of the **content** and **structure** of text documents.
- Extracting words from the sequence of characters in a document is called **tokenizing** or **lexical analysis**
- In addition to natural language words, there can be many other types of content in a document, such as metadata, images, graphics, code, and tables. Among them, **metadata** is information about a document that is not part of the text content
- Metadata is information about a document that is not part of the text, and includes:
 1. Document **attributes** such as date, author, and most importantly, the tags.
 2. The **tags** are used by *markup languages* to identify document components.
 3. The most popular *markup languages* are **HTML** (Hypertext Markup Language) and **XML** (Extensible Markup Language).
- After tokenization, a **parser** uses the tags and other metadata recognized in the document to interpret the document's structure based on the syntax of the markup language (**syntactic analysis**) and to produce a representation of the document that **includes both the structure and content**.
- For example, an HTML parser can interpret the structure of a web page, and creates a **Document Object Model (DOM)** representation of the page that is used by a web browser.

Tokenizing

- Tokenizing is the process of **forming words from the sequence of characters** in a document.
- In many early systems, a “word” was defined as any sequence of alphanumeric characters of length 3 or more, terminated by a space or other special character. All uppercase letters were also converted to lowercase, for example, the text:

Bigcorp's 2007 bi-annual report showed profits rose 10%.

would produce the following tokens:

bigcorp 2007 annual report showed profits rose.
- However, this simple tokenizing process is not adequate for most search applications because too much information is discarded

Tokenization is more complicated than it seems

- **Small words** (one or two characters) can be important in some queries, usually in combinations with other words. For example, xp, pm, ben e king, el paso, master p, j lo, world war ii.
- Both **hyphenated** and **non-hyphenated** forms of many words are common. In some cases the hyphen is not needed. For example, e-bay, wal-mart, active-x, cd-rom, t-shirts. At other times, hyphens should be considered either as part of the word or a word separator. For example, winston-salem, mazda rx-7, e-cards, pre-diabetes, t-mobile, spanish-speaking.
- **Special characters** (e.g. ! or &) are an important part of the tags, URLs, code, and other important parts of documents that must be correctly tokenized.
- **Capitalized words can have different meaning** from lowercase words. For example, “Bush” and “Apple”.
- **Apostrophes** can be a part of a word, a part of a possessive, or just a mistake. For example, rosie o'donnell (a famous American actress, comedian, talk show host, and social activist name), can't, don't, 80's, 1890's, men's straw hats, master's degree, England's ten largest cities.
- **Numbers** can be important, including decimals. For example, Nokia 3250, top 10 courses, united 93, QuickTime 6.5 pro, 92.3 the beat, 288358 (yes, this was a real query; it's a patent number).
- **Periods** can occur in numbers, abbreviations (e.g., “I.B.M.”, “Ph.D.”), URLs, ends of sentences, and other situations.

Document Structure and Markup

- In database applications, the fields or attributes of database records are a critical part of searching. Queries are specified in terms of the required values of these fields. In the case of web search, queries usually do not refer to document structure or fields. Some parts of the structure of web pages, indicated by HTML markup, are very significant features used by the ranking algorithm. The document parser must recognize this structure and make it available for indexing.
- The **main heading** for the page, e.g. “tropical fish”, indicates that this phrase is particularly important.
- If the same phrase is also in **bold** and **italics** in the body of the text, then this is further evidence of its importance.
- Other words and phrases are used as the **anchor text** for links and are likely to be good terms to represent the content of the page.
`the somewhere page`.

Tropical fish

From Wikipedia, the free encyclopedia

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species. Fishkeepers often use the term *tropical fish* to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

Tropical fish are popular aquarium fish , due to their often bright coloration. In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Document Structure and Markup

National Flag:	
Area:	83,858 square kilometres
Population:	8,205,000
Iso:	AT
Country:	Austria
Capital:	Vienna
Continent:	EU
Tld:	.at
Currency Code:	EUR
Currency Name:	Euro
Phone:	43
Postal Code Format:	####
Postal Code Regex:	^(\\d{4})\$
Languages:	de-AT,hr,hu,sl
Neighbours:	CH DE HU SK CZ IT SI LI
Edit	

Rendered Page

```
id="places_currency_name_label">Currency Name: </label><td class="w2p_fc">Euro</td><td class="w2p_fc"></td><tr id="places_phone_row"><td class="w2p_f1"><label class="readonly" for="places_phone">Phone: </label><td class="w2p_fc">43</td><td class="w2p_fc"></td></tr><tr id="places_postal_code_format_row"><td class="w2p_f1"><label class="readonly" for="places_postal_code_format">Postal Code Format: </label><td class="w2p_fc">####</td><td class="w2p_fc"></td></tr><tr id="places_postal_code_regex_row"><td class="w2p_f1"><label class="readonly" for="places_postal_code_regex">Postal Code Regex: </label><td class="w2p_fc">^((\\d{4}))$</td><td class="w2p_fc"></td></tr><tr id="places_languages_row"><td class="w2p_f1"><label class="readonly" for="places_languages">Languages: </label><td class="w2p_fc">de-AT,hr,hu,sl</td><td class="w2p_fc"></td></tr><tr id="places_neighbours_row"><td class="w2p_f1"><label class="readonly" for="places_neighbours">Neighbours: </label><td class="w2p_fc"><div><a href="/places/default/iso/CH">CH </a><a href="/places/default/iso/DE">DE </a><a href="/places/default/iso/HU">HU </a><a href="/places/default/iso/SK">SK </a><a href="/places/default/iso/CZ">CZ </a><a href="/places/default/iso/IT">IT </a><a href="/places/default/iso/SI">SI </a><a href="/places/default/iso/LI">LI </a></div></td><td class="w2p_fc"></td></tr></table><div style="display:none;"><input name="id" type="hidden" value="1602483" /></div></form>
```

Part of the Page's HTML Source

Two-pass Tokenization

- The tokenizing process can be divided into two passes.
- In the first pass we focus entirely on identifying markup or tags in the document. This could be done using a tokenizer and parser designed for the specific markup language used (e.g., HTML). One such HTML parser in Python is called **Beautiful Soup** which is a Python library for pulling data out of HTML and XML files.
- In the second pass we focus on the appropriate parts of the document structure (e.g. headings or body text or tables.. etc). Parts that are not useful for searching, such as those containing HTML code, are ignored in this pass.

Stopwords

- Human language is filled with **function words**, i.e. words that have little meaning in isolation from other words.
 - e.g. “the,” “a,” “an,” “that,” and “those. These words are part of how we **describe nouns in text**, and express concepts like **location or quantity**.
 - Prepositions, such as “over,” “under,” “above,” and “below,” represent relative position between two nouns.
- Two properties of these function words cause us to want to treat them in a special way in text processing.
 - These function words are extremely common in English (see table); however, they rarely indicate anything about document relevance **on their own**.
 - If we are considering **individual words** in the retrieval process and not phrases, these function words will **help us very little**.
- They are also called **stopwords** because text processing usually stops when one is seen, and they are thrown out. Throwing out these words decreases index size, increases retrieval efficiency, and generally improves retrieval effectiveness. But on the other hand, removing too many of these will negatively impact retrieval effectiveness. For instance, the query “**to be or not to be**” consists entirely of words that are usually considered stopwords.

Word	Freq.	r	P _r (%)	r.P _r	Word	Freq	r	P _r (%)	r.P _r
the	2,420,778	1	6.49	0.065	has	136,007	26	0.37	0.095
of	1,045,733	2	2.80	0.056	are	130,322	27	0.35	0.094
to	968,882	3	2.60	0.078	not	127,493	28	0.34	0.096
a	892,429	4	2.39	0.096	who	116,364	29	0.31	0.090
and	865,644	5	2.32	0.120	they	111,024	30	0.30	0.089
in	847,825	6	2.27	0.140	its	111,021	31	0.30	0.092
said	504,593	7	1.35	0.095	had	103,943	32	0.28	0.089
for	363,865	8	0.98	0.078	will	102,949	33	0.28	0.091
that	347,072	9	0.93	0.084	would	99,503	34	0.27	0.091
was	293,027	10	0.79	0.079	about	92,983	35	0.25	0.087
on	291,947	11	0.78	0.086	i	92,005	36	0.25	0.089
he	250,919	12	0.67	0.081	been	88,786	37	0.24	0.088
is	245,843	13	0.65	0.086	this	87,286	38	0.23	0.089
with	223,846	14	0.60	0.084	their	84,638	39	0.23	0.089
at	210,064	15	0.56	0.085	new	83,449	40	0.22	0.090
by	209,586	16	0.56	0.090	or	81,796	41	0.22	0.090
it	195,621	17	0.52	0.089	which	80,385	42	0.22	0.091
from	189,451	18	0.51	0.091	we	80,245	43	0.22	0.093
as	181,714	19	0.49	0.093	more	76,388	44	0.21	0.090
be	157,300	20	0.42	0.084	after	75,165	45	0.20	0.091
were	153,913	21	0.41	0.087	us	72,045	46	0.19	0.089
an	152,576	22	0.41	0.090	percent	71,956	47	0.19	0.091
have	149,749	23	0.40	0.092	up	71,082	48	0.19	0.092
his	142,285	24	0.38	0.092	one	70,266	49	0.19	0.092
but	140,880	25	0.38	0.094	people	68,988	50	0.19	0.093

Stemming

- **Stemming**, also called **conflation**, is a component of text processing that captures the relationships between different **variations of a word**.
- More precisely, stemming **reduces** the different **forms** of a word that occur because of **inflection** (e.g. plurals, tenses) or **derivation** (e.g. making a verb into a noun by adding the suffix -ation) to a **common stem**.
- For example to search for Mark Spitz's Olympic swimming career, you might type "Mark Spitz swimming" into a search engine, but a relevant page might contain the word swam. It is the job of the stemmer to reduce "swimming" and "swam" to the same stem (swim) and thereby allow the search engine to determine that there is a match between these two words.
- In general, using a stemmer for search applications with English text **produces** a **small but noticeable improvement** in the quality of results. However, in **highly inflected** languages, such as Arabic or Russian, stemming is a crucial part of effective search.

Stemmer Types

There are two basic types of stemmers: **algorithmic** and **dictionary-based**

- An **algorithmic stemmer** uses a small program to decide whether two words are related, usually based on knowledge of word suffixes for a particular language
- By contrast, a dictionary-based stemmer has no logic of its own, but instead relies on pre-created dictionaries of related terms to store term relationships (e.g. swimming and swim)
- In dictionary-based stemmers, the related words do not even need to look similar; a dictionary stemmer can recognize that “is,” “be,” and “was” are all forms of the same verb.

Porter stemmer

- One of the most popular algorithmic stemmers is the Porter stemmer (dating back to the 1970s)
- The stemmer consists of a number of steps, each containing a **set of rules for removing suffixes**.
- At each step, the rule for ***the longest applicable suffix*** is executed.
- As an example, here are the part a of step 1 (of 5 steps) of the Porter stemmer

Step 1a:

- Replace *sses* by *ss* (e.g., *stresses*→*stress*).
- Delete *s* if the preceding word part contains a vowel not immediately before the *s* (e.g., *gaps*→*gap* but *gas*→*gas*).
- Replace *ied* or *ies* by *i* if preceded by more than one letter, otherwise by *ie* (e.g., *ties*→*tie*, *cries*→*cri*).
- If suffix is *us* or *ss* do nothing (e.g., *stress*→*stress*).

- The original version of the Porter stemmer made a number of errors
- A more recent form of the stemmer (called Porter2) fixes some of these problems and provides a mechanism to specify exceptions.
- The stemmer is also available for many other languages, such as Russian and Turkish ...

Highly Inflectional Languages

- Some languages are **highly inflectional** which means that the root word can have many variants, e.g. Arabic and Spanish. For example, the following table shows some Arabic words derived from the same root.
- Clearly, a stemming algorithm that reduced Arabic words to their roots would not help in search (there are less than 2,000 roots in Arabic), but a **broad range of prefixes and suffixes must be considered**.
- In a highly inflectional language, proper stemming can make a large difference to the accuracy of the ranking. An Arabic search engine with high-quality stemming can be more than **50% more effective**, on average, at finding relevant documents than a system without stemming.
- By contrast, improvements for an English search engine vary from less than 5% on average for large collections to about 10% for small, domain-specific collections.

kitab	<i>a book</i>
kitabi	<i>my book</i>
alkitab	<i>the book</i>
kitabuki	<i>your book (f)</i>
kitabuka	<i>your book (m)</i>
kitabuhu	<i>his book</i>
kataba	<i>to write</i>
maktaba	<i>library, bookstore</i>
maktab	<i>office</i>

Table 4.8. Examples of words with the Arabic root ktb

Phrases and N-grams

- Many of the two- and three-word queries submitted to search engines are **phrases**, and finding documents that contain those phrases will be part of any effective ranking algorithm.
- Phrases are more precise than single words as topic descriptions (e.g., “tropical fish” versus “fish”) and usually less ambiguous
- **The impact of phrases on retrieval can be complex**; for example given a query such as “fishing supplies”, should the retrieved documents contain exactly that phrase, or should they get credit for containing the words “fish”, “fishing”, and “supplies” in the same paragraph, or even the same document? The details of how phrases affect ranking will depend on the specific retrieval model that is incorporated into the search engine
- There are a number of possible definitions of a phrase. Since a phrase has a grammatical definition, it seems reasonable to identify phrases using the syntactic structure of sentences. The **definition of a phrase** that is used most frequently in information retrieval is that a phrase is equivalent to a **simple noun phrase**. This is often **restricted even further** to include just **sequences of nouns, or adjectives followed by nouns**.
- Phrases defined by these criteria can be identified using a **part-of-speech (POS) tagger**.

Part-of-speech (POS) taggers

- A POS tagger **marks the words** in a text with **labels** corresponding to the **part-of-speech** of the word in that context.
- Taggers are based on statistical or rule-based approaches and are trained using **large corpora** that have been manually labelled.
- Typical tags that are used to label the words include NN (singular noun), NNS (plural noun), VB (verb), VBD (verb, past tense), VBN (verb, past participle), IN (preposition, e.g. in, out), JJ (adjective), CC (conjunction, e.g., “and”, “or”), PRP (pronoun, e.g. she, it), and MD (modal auxiliary, e.g., “can”, “will”).

Original text:

Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales.

Brill tagger:

Document/NN will/MD describe/VB marketing/NN strategies/NNS carried/VBD out/IN by/IN U.S./NNP companies/NNS for/IN their/PRP agricultural/JJ chemicals/NNS /, report/NN predictions/NNS for/IN market/NN share/NN of/IN such/JJ chemicals/NNS ,/, or/CC report/NN market/NN statistics/NNS for/IN agrochemicals/NNS ,/, pesticide/NN ,/, herbicide/NN ,/, fungicide/NN ,/, insecticide/NN ,/, fertilizer/NN ,/, predicted/VBN sales/NNS ./, market/NN share/NN ,/, stimulate/VB demand/NN ,/, price/NN cut/NN ,/, volume/NN of/IN sales/NNS ./.

That's it Folks



Further Reading

Chapter 4: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman

Web Services and Web Data

XJCO3011



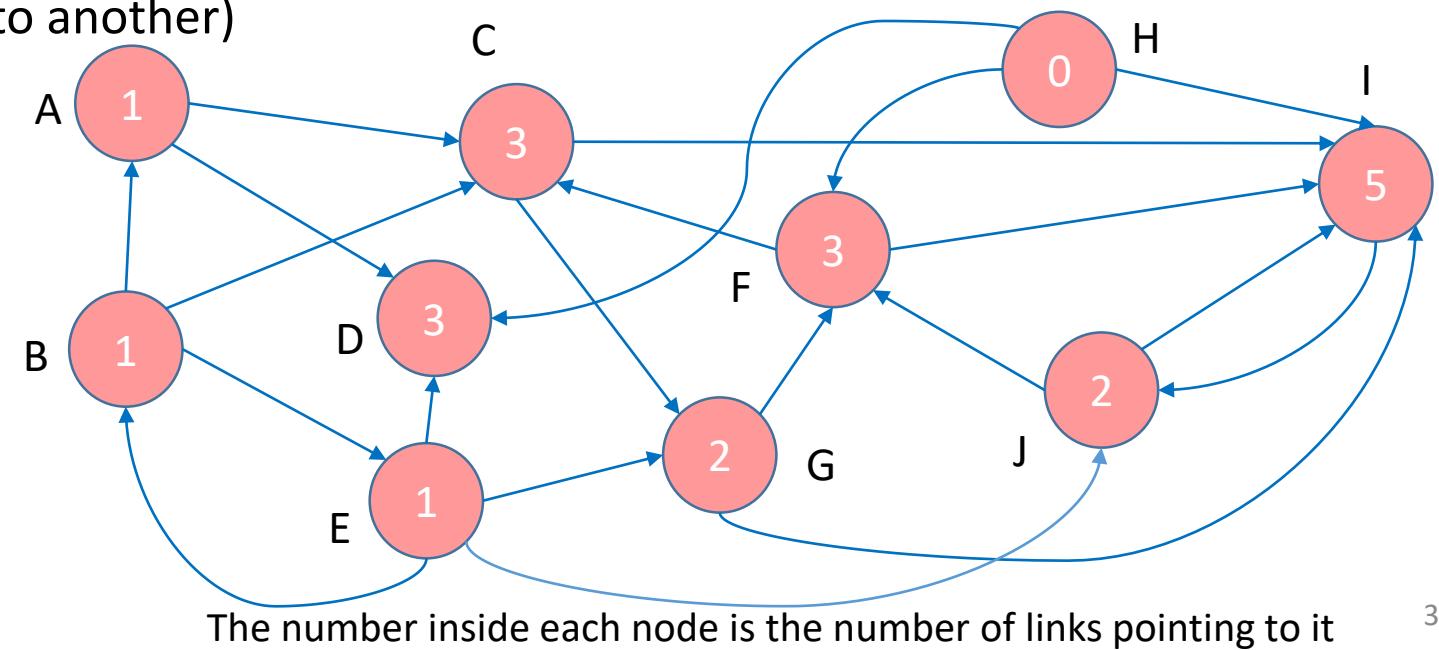
*Session 11. Link Analysis and the
PageRank Algorithm*

Ranking Web Pages

- There are **tens of billions of web pages**, but most of them are not very interesting.
- **Many** of those pages **are spam** and contain **no useful** content at all.
- Other pages are **personal blogs**, **wedding announcements**, or **family picture albums**. These pages are interesting to **a small audience**, but not broadly.
- On the other hand, there are **a few pages that are popular** and useful to many people, including news sites and the websites of popular companies.
- How can search engine choose the most popular (and probably the correct) page?

Simple Link Analysis

- Links connecting pages are a key component of the Web.
- They help search engines **rank web pages** effectively.
- The simplest form of page ranking is based on counting the **number of links pointing to a particular web page**. The fact that a link exists at all is a vote of importance for the **destination page**.
- The **higher this count**, the **higher the rank** of this page in the search results, see the web shown below.
- This is based on the assumption that **important pages are referenced (linked to) by many other pages**.
- The problem with this method is that it **does not take** into account the **importance (rank)** of the **source page** (the page that points to the other page), for example , in the web sample shown below, which page is more important D, F or C?
- A second problem is that this method does not take into account the topic of the link (a page could be important to one topic but less important to another)
 - E.g., evaluating the importance of a health website
 - Page 1: A page specifically discussing heart disease, with lots of relevant information and links.
 - Page 2: An article about fitness, which includes some links about heart disease but is not the primary focus of the website.



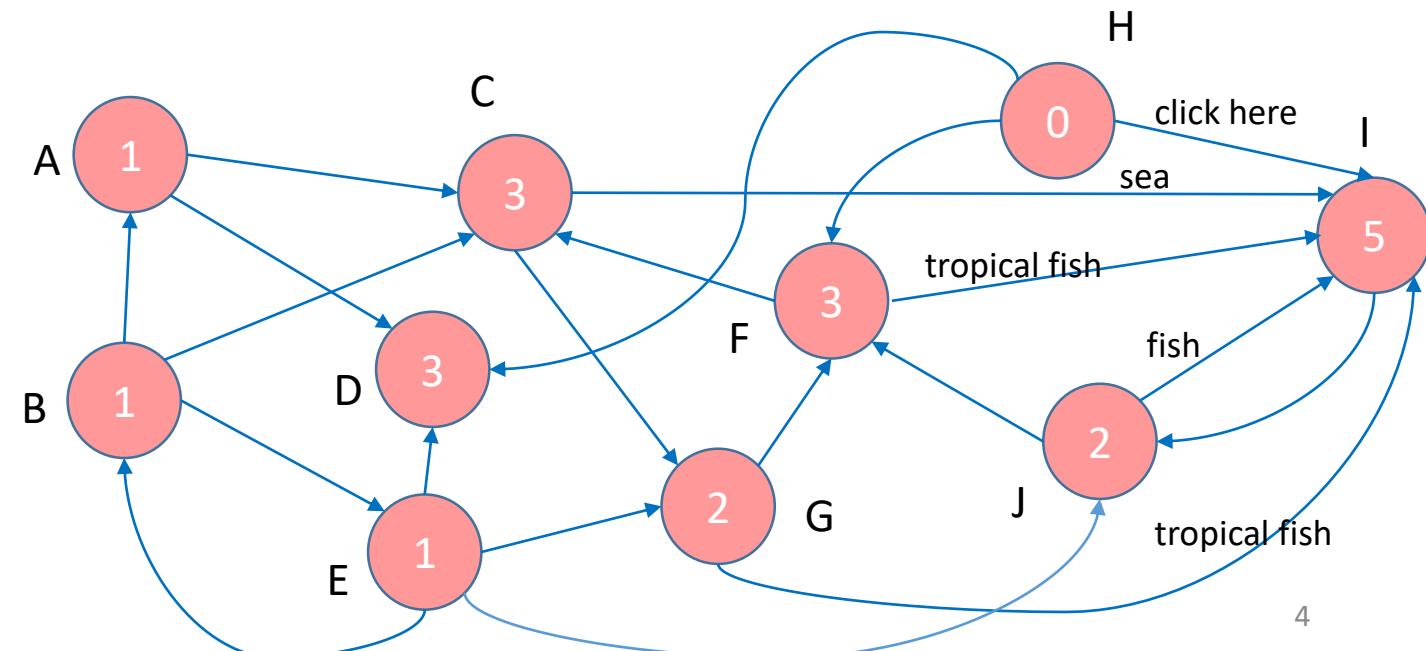
Anchor Text Analysis

- A link in a web page is encoded in HTML with a statement such as:

`tropical fish.`

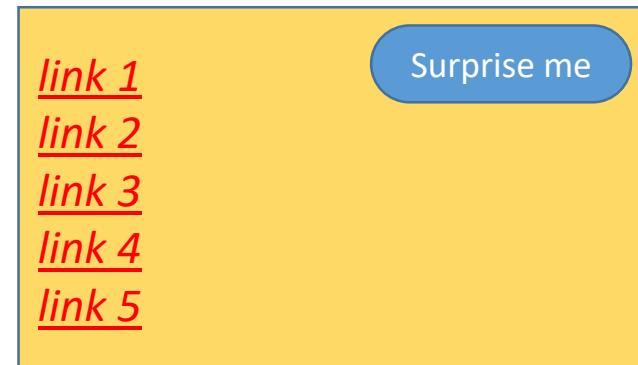
In this link, “tropical fish” is called the **anchor text**, and `http://www.somewhere.com` is the *destination*. Both components are useful in the ranking process.

- Anchor text is particularly useful for ranking web pages because it is **very short**, two or three words, and those words **succinctly describe the topic** of the linked page.
- Many **queries** are very **similar to anchor text** in that they are also short topical descriptions of web pages.
- Search through all links in the collection of pages, looking for anchor text that matches the user’s query. Each time there is a **match**, **add 1 to the score** of the destination page. Pages would then be ranked in **decreasing order** of this score.
- This algorithm has some glaring faults, not the least of which is how to handle the query “click here”, and it does **not** take into account the rank of the source page.



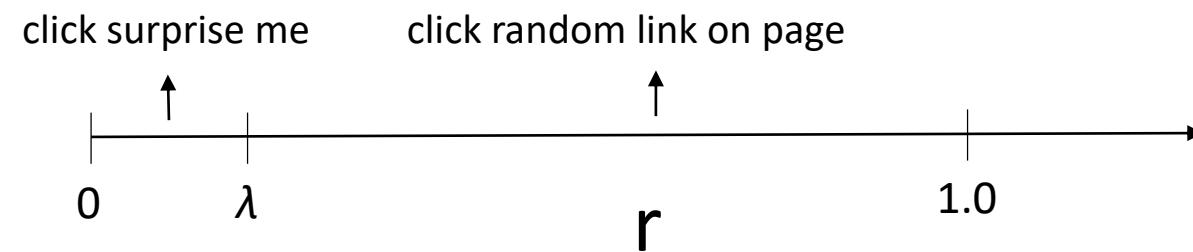
Alice Again!

- One of the most popular page ranking algorithms is called **PageRank** and has been used by Google for many years.
- PageRank is based on the idea of a **random surfer** (as in web surfer).
- Imagine that Alice is using her web browser to **wander aimlessly** between web pages.
- Her browser has a **special “surprise me” button** at the top that will jump to a **random web page** when she clicks it.
- Each time a web page loads, she chooses whether to click the “surprise me” button or whether to click one of the links on the web page.
- If she clicks a link on the page, she has **no preference for any particular link**; she just picks one randomly.
- Alice is sufficiently bored that she intends to keep **browsing** the Web like this **forever**



Lambda

- To put this in a more structured form, Alice browses the Web using this algorithm:
 1. Choose a random number r between 0 and 1.
 2. If $r < \lambda$:
 - Click the “surprise me” button.
 3. If $r \geq \lambda$:
 - Click a link at random on the current page.
 4. Start again.
- Typically we assume that λ is fairly small, so Alice is much more likely to click a link than to pick the “surprise me” button



The PageRank Web Surfing Assumptions

- Even though Alice's path through the web pages is random, Alice **will still see popular pages more often than unpopular ones** because Alice often follows links, and links tend to point to popular pages
- So, we expect that Alice will end up at a university website, for example, more often than a personal website, but less often than the CNN website.
- The “surprise me” button can guarantee that eventually she will reach every page on the Internet.
- Without the “surprise me” button, she would get stuck on pages that no longer pointed to any page, or pages that formed a loop

A Page Rank is a Probability Value

- Now suppose that while Alice is browsing, you happened to walk into her room and glance at the web page on her screen.
- What is the probability that she will be looking at the CNN web page when you walk in? **That probability is CNN's PageRank.**
- Every web page on the Internet has a PageRank, and it is uniquely determined by the link structure of web pages.
- PageRank has the ability to distinguish popular pages, **those with many incoming links, or those that have links from popular pages,** from unpopular ones.
- The PageRank value can help search engines sift through millions of pages that contain the word “eBay” to find the one that is most popular (www.ebay.com).

Computing PageRank

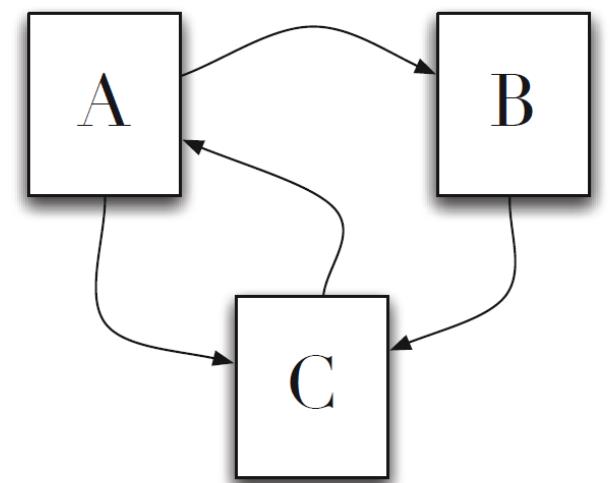
- Suppose for the moment that the Web consists of just three pages, A, B, and C, with links as shown.
- The PageRank of page C, which is the probability that Alice will be looking at this page, will depend on the PageRank of pages A and B.
- Since Alice chooses randomly between links on a given page, if she starts in page A, there is a 50% chance that she will go to page C
- Another way of saying this is that the PageRank for a page is divided evenly between all the outgoing links
- If we ignore the “surprise me” button, this means that the PageRank of page C, represented as $PR(C)$, can be calculated as

$$PR(C) = \frac{PR(A)}{2} + \frac{PR(B)}{1}$$

- More generally, we could calculate the PageRank for any page u as

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

where B_u is the set of pages that point to u ,
and L_v is the number of outgoing links from page v (not counting duplicate links).



Computing PageRank ...

- There is an obvious problem here: we don't know the PageRank values for the pages, because that is what we are trying to calculate.
- If we start by assuming that the PageRank values for all pages are the same (1/3 in this case), then it is easy to see that we could perform multiple iterations of the calculation.

- For example, in the first iteration

$$PR(C) = 0.33/2 + 0.33 = 0.5, PR(A) = 0.33, \text{ and } PR(B) = 0.17.$$

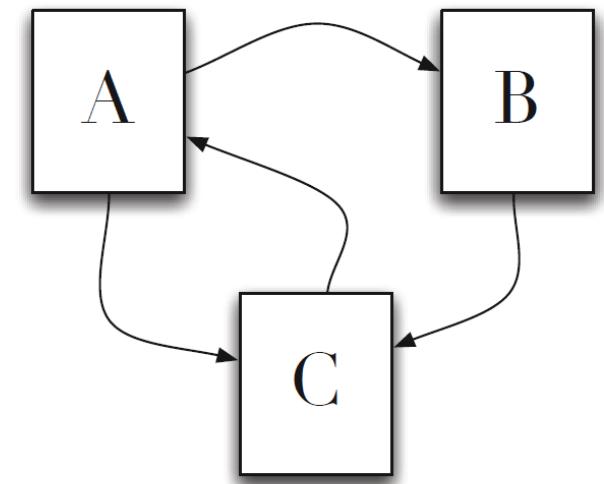
- In the next iteration:

$$PR(C) = 0.33/2 + 0.17 = 0.33, PR(A) = 0.5, \text{ and } PR(B) = 0.17.$$

- In the third iteration, $PR(C) = 0.42$, $PR(A) = 0.33$, and $PR(B) = 0.25$.

- After a few more iterations, the PageRank values converge to the final values of

$$PR(C) = 0.4, PR(A) = 0.4, \text{ and } PR(B) = 0.2.$$



Computing PageRank ...

- If we take the “surprise me” button into account, part of the PageRank for page C will be due to the probability of coming to that page by pushing the button.
- Given that there is a $1/3$ chance of going to any page when the button is pushed, and that the probability of pushing the button is λ , the contribution of the button to the PageRank of C will be $\lambda/3$. This means that the total PageRank for C is now:

$$PR(C) = \frac{\lambda}{3} + (1 - \lambda) \cdot \left(\frac{PR(A)}{2} + \frac{PR(B)}{1} \right)$$

- Similarly, the general formula for PageRank is:

$$PR(u) = \frac{\lambda}{N} + (1 - \lambda) \cdot \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

where N is the number of pages being considered. The typical value for λ is 0.15

Simplified PageRank Algorithm

```
// initialize all page rank values
for each page p in web
    page rank of p = 1/number of pages

repeat until page rank values converge
{
    for each page p in web
    {
        new page rank of p = 0 // we will accumulate a new value here
        for each page q pointing at p
            new page rank of p += old page rank of q / number of links coming out of q
    }
    // now make the old values = the new values, to prepare for the next iteration
    for each page p in web
        old value of page rank of p = new value of page rank of p
}
```

PageRank Algorithm with λ

Note: Pages with no outbound links are **rank sinks**, in that they accumulate PageRank but do not distribute it. In this algorithm, we assume that these pages link to all other pages in the collection.

```
1: procedure PAGERANK( $G$ )
2:            $\triangleright G$  is the web graph, consisting of vertices (pages) and edges (links).
3:            $(P, L) \leftarrow G$                                  $\triangleright$  Split graph into pages and links
4:            $I \leftarrow$  a vector of length  $|P|$                  $\triangleright$  The current PageRank estimate
5:            $R \leftarrow$  a vector of length  $|P|$                  $\triangleright$  The resulting better PageRank estimate
6:           for all entries  $I_i \in I$  do
7:                $I_i \leftarrow 1/|P|$                              $\triangleright$  Start with each page being equally likely
8:           end for
9:           while  $R$  has not converged do
10:              for all entries  $R_i \in R$  do
11:                   $R_i \leftarrow \lambda/|P|$      $\triangleright$  Each page has a  $\lambda/|P|$  chance of random selection
12:              end for
13:              for all pages  $p \in P$  do
14:                   $Q \leftarrow$  the set of pages such that  $(p, q) \in L$  and  $q \in P$ 
15:                  if  $|Q| > 0$  then
16:                      for all pages  $q \in Q$  do
17:                           $R_q \leftarrow R_q + (1 - \lambda)I_p/|Q|$   $\triangleright$  Probability  $I_p$  of being at page  $p$ 
18:                      end for
19:                  else
20:                      for all pages  $q \in P$  do
21:                           $R_q \leftarrow R_q + (1 - \lambda)I_p/|P|$ 
22:                      end for
23:                  end if
24:                   $I \leftarrow R$                                  $\triangleright$  Update our current PageRank estimate
25:              end for
26:          end while
27:          return  $R$ 
28: end procedure
```

That's it Folks



Further Reading

Chapter 4: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman

Web Services and Web Data

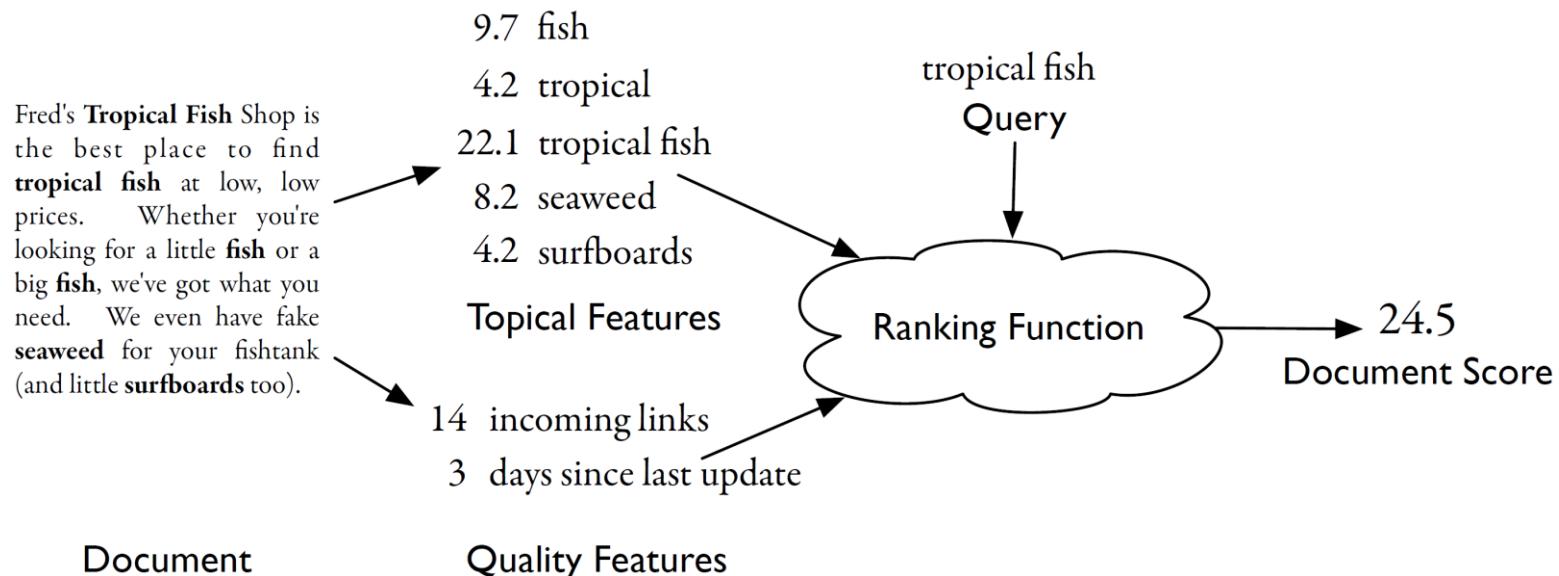
XJCO3011



Session 12 - Indexing

Abstract Model of Ranking

- Documents are written in natural human languages, which are difficult for computers to analyse directly. So, the text is transformed into *index terms* or *document features*.
- A document feature is some attribute of the document we can *express numerically*. In the figure, we show two kinds of features:
 - **Topical features** estimate the degree to which the document is about a particular subject.
 - Document **quality features**, e.g. the number of web pages that link to this document, or the number of days since this page was last updated.
- Quality features don't address whether the document is a good topical match for a query, but address its quality; a page with no incoming links that hasn't been edited in years is probably a poor match for any query

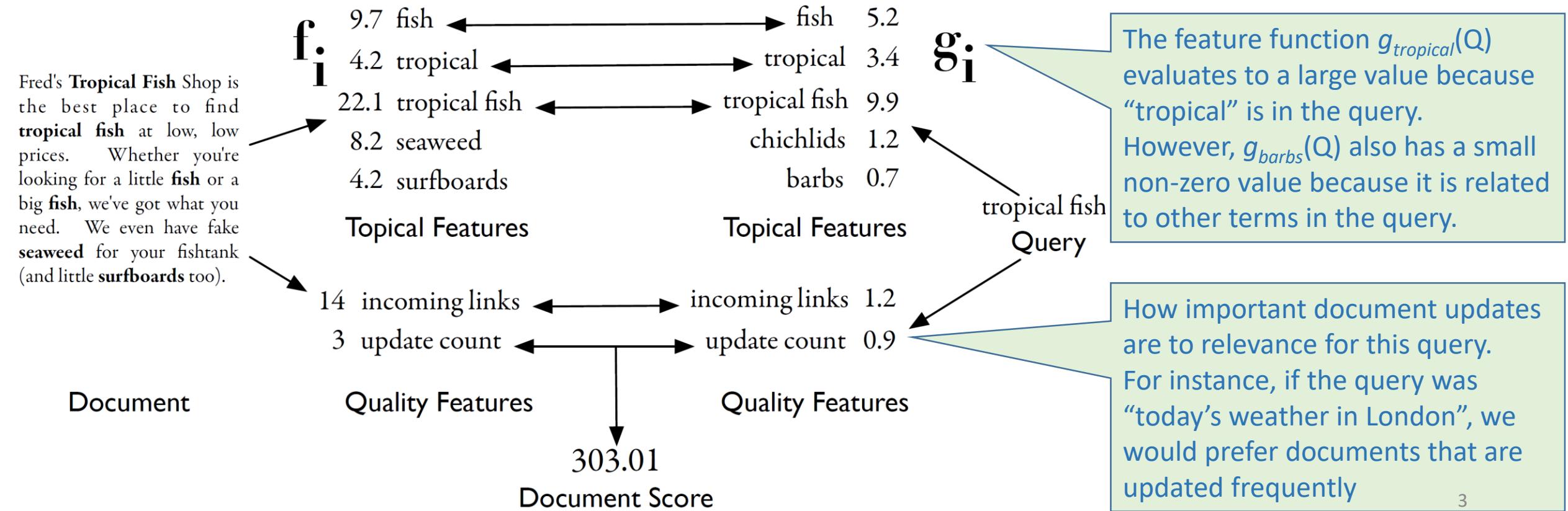


A more concrete ranking model

- We assume that the ranking function R takes the following form:

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

- Here, f_i is some feature function that extracts a number from the document text, and g_i is a similar feature function that extracts a value from the query



Inverted Indices

Index

- All modern search engine indices are based on *inverted indices*
- An inverted index is the computational equivalent of the index found in the back of all textbooks.
- The index is inverted because usually we think of words being part of documents, but if we invert this idea, documents are associated with words.

fluid
 incompressible, see incompressible fluid

Airy wave theory, 25

algorithm, 33

amortized running time, 33

amplitude, 26

angular frequency, 25

approximation, 7

Bessel function
 zeroth order of the first kind, 30

big O notation, 33

cell, 7

cell face, 7

CFMM, 25, 32

Continuous Fast Multipole Method, see CFMM

convolution, 27

convolution filter, 27

convolution kernel, 27, 28

convolution theorem, 28

core, 33

density, 25

depth
 effective, see effective depth
 water, see water depth

differentiation

filter
 convolution, see convolution filter
 low-pass, see low-pass filter

Finite Volume Method, see FVM

fluid flux, 7

flux
 fluid, see fluid flux

FMM, 32

Fourier transform
 non-uniform, see non-uniform Fourier transform

reverse, see reverse Fourier transform

free surface elevation, 25

frequency domain, 26

FVM, 7

gradient, 26

gravitational acceleration, 25

grid point
 surface, see surface grid point

Hankel transform
 zeroth order, 30

incompressible fluid, 7

instability, 7

kernel
 convolution, see convolution kernel

Index structure

- Index terms are often alphabetized like a traditional book index, but they need not be, since they are often found directly using a hash table.
- Each index term has its own inverted list that holds the relevant data for that term. For a book index, the relevant data is a list of page numbers. In a search engine, the data might be a list of documents.
- Each list entry is called a *posting*, and the part of the posting that refers to a specific document or location is often called a *pointer*.
- Each document in the collection is given a unique number to make it efficient for storing document pointers.
- Indices in books store more than just location information. For important words, often one of the page numbers is marked in boldface, indicating that this page contains a definition or extended discussion about the term.
- Inverted files can also have extended information, where postings can contain a range of information other than just locations. By storing the right information along with each posting, the feature functions can be computed efficiently.
- In the next few slides, we will look at some different kinds of inverted files.

Documents

- The simplest form of an inverted list **stores just the documents** that contain each word, and no additional information
- Notice that this index does not record the number of times each word appears; it only records the documents in which each word appears.
- Inverted lists become **more interesting** when we **consider their intersection**; for example find the sentence that contains the words “coloration” and “freshwater”.

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

The “documents”

and	1	only	2
aquarium	3	pigmented	4
are	3	popular	3
around	4	refer	2
as	2	referred	2
both	1	requiring	2
bright	3	salt	1
coloration	3	saltwater	4
derives	4	species	1
due	3	term	2
environments	1	the	1
fish	2	their	2
fishkeepers	3	this	3
found	1	those	4
fresh	2	to	2
freshwater	4	tropical	3
from	1	typically	1
generally	4	use	2
in	1	water	2
include	4	while	4
including	1	with	2
iridescence	2	world	1
marine	3		
often	2		

The Index

Counts

- In this method, each posting now has a second number which is **the number of times the word appears in the document**
- Example: find the most relevant document for the query “tropical fish”
- We prefer S2 over S1 and S3 for the query “tropical fish”, since S2 contains “tropical” twice and “fish” three times.
- In general, word counts can be a **powerful predictor** of document relevance. In particular, word counts can help distinguish documents that are about a particular subject from those that discuss that subject in passing

and	1:1	only	2:1
aquarium	3:1	pigmented	4:1
are	3:1	popular	3:1
around	1:1	refer	2:1
as	2:1	referred	2:1
both	1:1	requiring	2:1
bright	3:1	salt	1:1
coloration	3:1	saltwater	2:1
derives	4:1	species	1:1
due	3:1	term	2:1
environments	1:1	the	1:1
fish	1:2	their	3:1
fishkeepers	2:1	this	4:1
found	1:1	those	2:1
fresh	2:1	to	2:2
freshwater	1:1	tropical	1:2
from	4:1	typically	4:1
generally	4:1	use	2:1
in	1:1	water	1:1
include	1:1	while	4:1
including	1:1	with	2:1
iridescence	4:1	world	1:1
marine	2:1		
often	2:1		
	3:1		

An inverted index, with word counts

Positions

and	1,15		marine	2,22	
aquarium	3,5		often	2,2	3,10
are	3,3	4,14	only	2,10	
around	1,9		pigmented	4,16	
as	2,21		popular	3,4	
both	1,13		refer	2,9	
bright	3,11		referred	2,19	
coloration	3,12	4,5	requiring	2,12	
derives	4,7		salt	1,16	4,11
due	3,7		saltwater	2,16	
environments	1,8		species	1,18	
fish	1,2	1,4	term	2,5	
		2,7	the	1,10	2,4
		2,18	their	3,9	
		2,23	this	4,4	
		3,2	those	2,11	
		3,6	to	2,8	2,20 3,8
		4,3	tropical	1,1	1,7 2,6 2,17 3,1
		4,13	typically	4,6	●
fishkeepers	2,1		use	2,3	
found	1,5		water	1,17	2,14 4,12
fresh	2,13		while	4,10	
freshwater	1,14	4,2	with	2,15	
from	4,8		world	1,11	
generally	4,15				
in	1,6	4,1			
include	1,3				
including	1,12				
iridescence	4,9				

An inverted index, with word positions,

tropical	1,1	1,7	2,6	2,17	3,1		
fish	1,2	1,4	2,7	2,18	2,23	3,2	3,6 4,3 4,13

Aligning posting lists for “tropical” and “fish” to find the phrase “tropical fish”

- When looking for matches for a query like “tropical fish”, the location of the words in the document is an important predictor of relevance
 - Imagine a document about food that included a section on **tropical fruits** followed by a section on **saltwater fish**. So far, none of the indices we have considered contain enough information to tell us that this document is not relevant
 - We want to know if the document contains the exact phrase “tropical fish”.
- To determine this, we can add position information to our index
- Each posting contains two numbers: a document number first, followed by a word position
 - By aligning the posting lists, we can find the best match for a phrase. The phrase matches are easy to see in the figure; they happen at the points where the postings are lined up in columns.⁸

Fields and Extents

- Real documents are not just lists of words. They have sentences and paragraphs that separate concepts into logical units
- Some documents have **titles** and headings that provide short summaries of the rest of the content.
- This is where **extent lists** come in. An extent is a contiguous region of a document. We can represent these extents using word positions
- For example, if the title of a book started on the fifth word and ended just before the ninth word, we could encode that as (5,9).
- E.g., aligning posting lists for “fish” and title to find matches of the word “fish” in the title field of a document

fish	1,2	1,4	2,7	2,18	2,23	3,2	3,6	4,3	4,13
title	1:(1,3)		2:(1,5)						4:(9,15)

Scores

- If the inverted lists are going to be used to generate feature function values, why not just store the value of the feature function? This approach makes it possible to store feature function values that would be too **computationally intensive** to compute during the query processing phase
- We could make a list for “fish” that has postings like [(1:3.6), (3:2.2)], meaning that the total feature value for “fish” in document 1 is 3.6, and in document 3 it is 2.2
- The number 3.6 came from taking into account how many times “fish” appeared in the **title**, in the **headings**, in **large fonts**, in **bold**, and in **links to the document**. Maybe the document doesn’t contain the word “fish” at all, but instead many names of fish, such as “carp” or “trout”. The value 3.6 is then some indicator of how much this document is about fish.
- Storing scores like this **both increases and decreases the system’s flexibility**. It increases flexibility because computationally expensive scoring becomes possible, since much of the hard work of scoring documents is moved into the index.
- However, flexibility is lost, since we can no longer change the scoring mechanism once the index is built.
- More importantly, information about word proximity is gone in this model, meaning that we can’t include phrase information in scoring unless we build inverted lists for phrases, too.
- These precomputed phrase lists require considerable additional space.

Ordering

- So far, we have assumed that the postings of each inverted list would be ordered by document number.
- Although this is the most popular option, this is not the only way to order an inverted list.
- An inverted list can also be ordered by score, so that the highest-scoring documents come first.
- This makes sense only when the lists already store the score, or when only one kind of score is likely to be computed from the inverted list.
- By storing scores instead of documents, the query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded

Auxiliary Structures

- An inverted file is just a collection of inverted lists.
- To search the index, some kind of data structure is necessary to find the inverted list for a particular term.
- The simplest way to solve this problem is to store each inverted list as a separate file, where each file is named after the corresponding search term. To find the inverted list for “dog,” the system can simply open the file named dog and read the contents.
- However, document collections can have millions of unique words, and most of these words will occur only once or twice in the collection. This means that an index, if stored in files, would consist of millions of files, most of which are very small
- Unfortunately, modern file systems are not optimized for this kind of storage.
- A file system typically will reserve a few kilobytes of space for each file, even though most files will contain just a few bytes of data. The result is a huge amount of wasted space.
- Imagine for example, a collection of 70,000 words which only occur once. These inverted lists would require about 20 bytes each, for a total of about 2MB of space. However, if the file system requires 1KB for each file, the result is 70MB of space used to store 2MB of data.
- In addition, many file systems still store directory information in unsorted arrays, meaning that file lookups can be very slow for large file directories.

Inverted Files

- To fix these problems, inverted lists are usually stored together in a single file, which explains the name inverted file.
- An additional directory structure, called the vocabulary or lexicon, contains a lookup table from index terms to the byte offset of the inverted list in the inverted file.
- In many cases, this vocabulary lookup table will be small enough to fit into memory. In this case, the vocabulary data can be stored in any reasonable way on disk and loaded into a hash table at search engine startup.
- If the search engine needs to handle larger vocabularies, some kind of tree-based data structure should be used to minimize disk accesses during the search process.

That's it Folks



Further Reading

Chapter 5: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman

Web Services and Web Data

XJCO3011



Session 13 – Query Processing

Query Processing

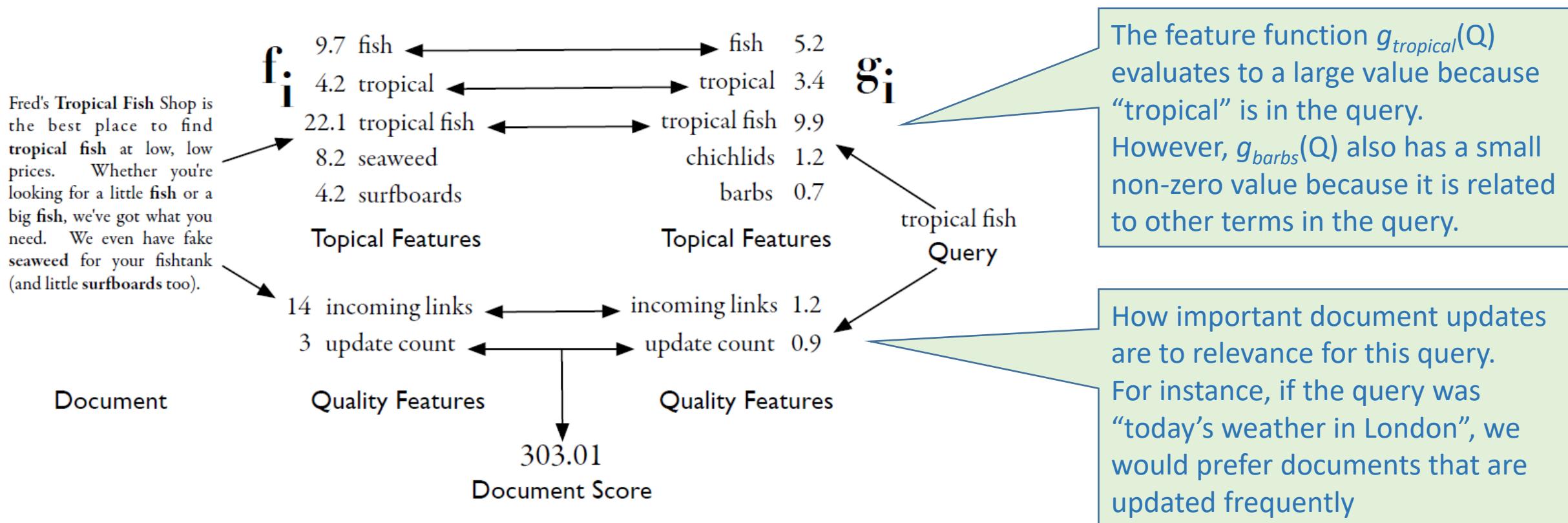
- Once an index is built, we need to process the data in it to produce query results.
- Even with simple algorithms, processing queries using an index is much faster than without one.
- However, **clever algorithms can boost query processing speed by ten to a hundred times over the simplest versions.**
- We will explore the simplest two query processing techniques first, called **document-at-a-time** and **term-at-a-time**, and then move on to faster and more flexible variants.

The General Ranking Model

- We assume that the ranking function R takes the following form:

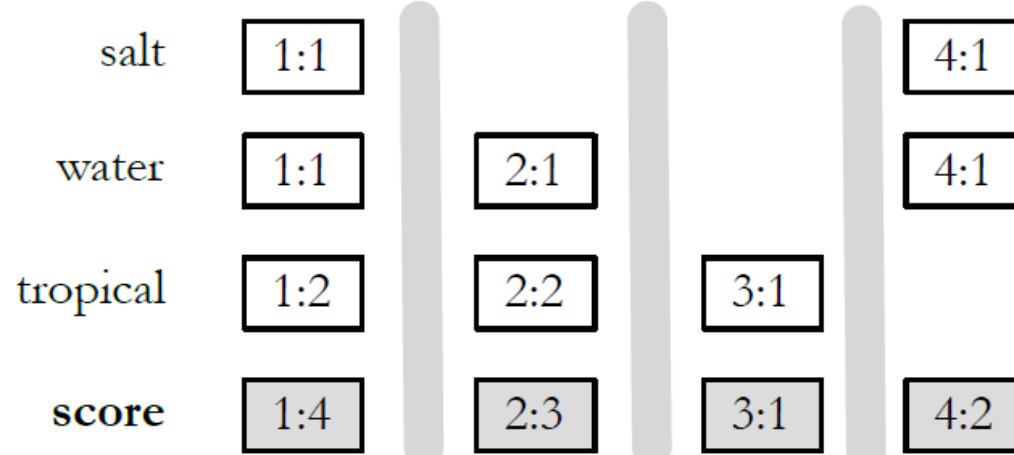
$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

- Here, f_i is some feature function that extracts a number from the document text, and g_i is a similar feature function that extracts a value from the query



Document-at-a-time Evaluation

- Document-at-a-time retrieval is the simplest way to perform retrieval with an inverted file.
- The figure below shows a document-at-a-time retrieval for the query “salt water tropical”.
- The inverted lists are shown horizontally, and the postings have been aligned so that each column represents a different document
- The inverted lists hold word counts, and **the score is just the sum of the word counts** in each document.
- The vertical grey lines indicate the different steps of retrieval
- In the first step, all the counts for the first document are added to produce the score for that document.
- Once the scoring for the first document has completed, the second document is scored, then the third, and then the fourth.



Document-at-a-time Pseudocode

```
procedure DOCUMENTATATIMERETRIEVAL( $Q, I, f, g, k$ )
```

```
     $L \leftarrow \text{Array}()$ 
```

```
     $R \leftarrow \text{PriorityQueue}(k)$ 
```

```
    for all terms  $w_i$  in  $Q$  do
```

```
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
```

```
         $L.\text{add}( l_i )$ 
```

```
    end for
```

```
    for all documents  $d \in I$  do
```

```
         $s_d \leftarrow 0$ 
```

```
        for all inverted lists  $l_i$  in  $L$  do
```

```
            if  $l_i.\text{getCurrentDocument}() = d$  then
```

```
                 $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$ 
```

```
            end if
```

```
             $l_i.\text{movePastDocument}( d )$ 
```

```
        end for
```

```
         $R.\text{add}( s_d, d )$ 
```

```
    end for
```

```
    return the top  $k$  results from  $R$ 
```

```
end procedure
```

Where Q: the query; I: the index; f and g: the feature functions; and k: the number of documents to retrieve

Iterate for each doc. in the Index

For each word w_i in the query, an **inverted list** is fetched from the index. These inverted lists are assumed to be sorted in order by document number. All of the fetched inverted lists are stored in an array, L.

For each document, all of the inverted lists are checked. If the document appears in one of the inverted lists, the feature function f_i is evaluated, and the document score s_D is computed by adding up the weighted function values

the document score is added to the priority queue R.

Document-at-a-time Algorithm Features

```
procedure DOCUMENTATATIMERETRIEVAL( $Q, I, f, g, k$ )
```

```
     $L \leftarrow \text{Array}()$ 
```

```
     $R \leftarrow \text{PriorityQueue}(k)$ 
```

```
    for all terms  $w_i$  in  $Q$  do
```

```
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
```

```
         $L.\text{add}( l_i )$ 
```

```
    end for
```

```
    for all documents  $d \in I$  do
```

```
         $s_d \leftarrow 0$ 
```

```
        for all inverted lists  $l_i$  in  $L$  do
```

```
            if  $l_i.\text{getCurrentDocument}() = d$  then
```

```
                 $s_d \leftarrow s_d + g_i(Q)f_i(l_i)$ 
```

▷ Update the document score

```
            end if
```

```
             $l_i.\text{movePastDocument}( d )$ 
```

```
        end for
```

```
         $R.\text{add}( s_d, d )$ 
```

The primary benefit of this method is its **economic use of memory**. The only major use of memory comes from the priority queue, which only needs to store k entries at a time. However, in a realistic implementation, **large portions of the inverted lists would also be buffered in memory during evaluation**.

Looping over all documents in the collection is unnecessary; we can change the algorithm to score only documents that appear in at least one of the inverted lists.

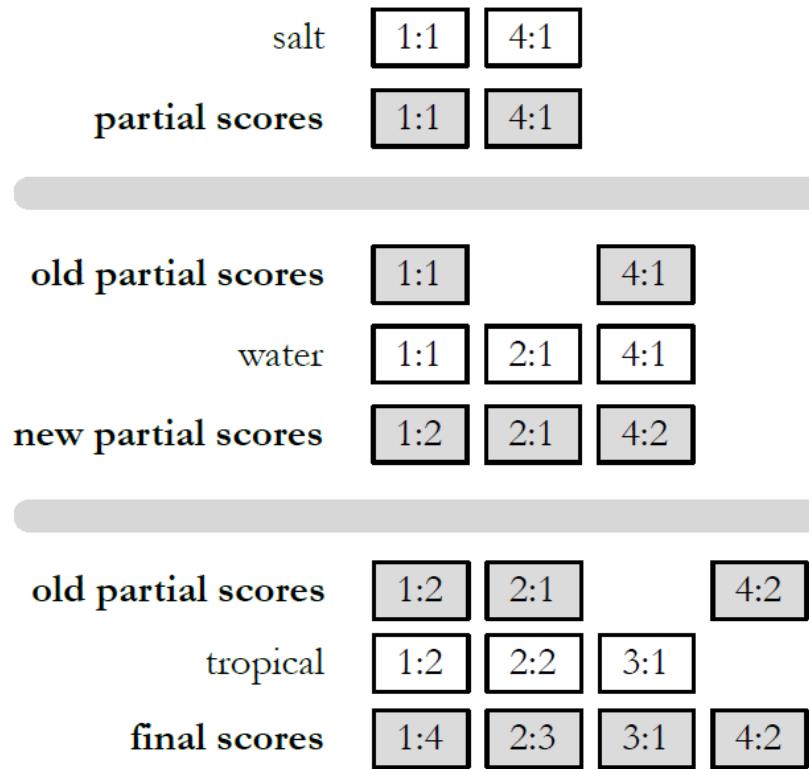
```
    end for
```

```
    return the top  $k$  results from  $R$ 
```

the priority queue R only needs to hold the top k results at any one time. If the priority queue ever contains more than k results, the lowest-scoring documents can be removed until only k remain, in order to save memory.

```
end procedure
```

Term-at-a-time Evaluation



- The figure shows term-at-a-time retrieval, using the same query, scoring function, and inverted list data as in the document-at-a-time example.
- Notice that the computed scores are exactly the same in both figures, although the structure of each figure is different.
- As before, the grey lines indicate the boundaries between each step. In the first step, the inverted list for “salt” is decoded, and **partial scores are stored in accumulators**.
- These scores are called partial scores because they are only a part of the final document score
- In the second step, partial scores from the accumulators are combined with data from the inverted list for “water” to produce a new set of partial scores.
- After the data from the list for “tropical” is added in the third step, the scoring process is complete.
- In practice **accumulators are stored in a hash table**. The information for each document is updated as the inverted list data is processed

term-at-a-time retrieval algorithm

```
procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
```

```
     $A \leftarrow \text{HashTable}()$ 
```

```
     $L \leftarrow \text{Array}()$ 
```

```
     $R \leftarrow \text{PriorityQueue}(k)$ 
```

```
    for all terms  $w_i$  in  $Q$  do
```

```
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
```

```
         $L.add(l_i)$ 
```

```
    end for
```

```
    for all lists  $l_i \in L$  do
```

```
        while  $l_i$  is not finished do
```

```
             $d \leftarrow l_i.\text{getCurrentDocument}()$ 
```

```
             $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
```

```
             $l_i.\text{moveToNextDocument}()$ 
```

```
        end while
```

```
    end for
```

```
    for all accumulators  $A_d$  in  $A$  do
```

```
         $s_d \leftarrow A_d$ 
```

```
         $R.add(s_d, d)$ 
```

```
    end for
```

```
    return the top  $k$  results from  $R$ 
```

```
end procedure
```

For each word w_i in the query, an inverted list is fetched from the index. These inverted lists are assumed to be sorted in order by document number. All of the fetched inverted lists are stored in an array, L .

the outer loop is over each list

reads each posting of the list, computing the feature functions f_i and g_i and adding its weighted contribution to the accumulator A_d .

After the main loop completes, the accumulators are scanned and added to a priority queue, which determines the top k results to be returned.

term-at-a-time retrieval algorithm features

```
procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
```

```
     $A \leftarrow \text{HashTable}()$ 
```

```
     $L \leftarrow \text{Array}()$ 
```

```
     $R \leftarrow \text{PriorityQueue}(k)$ 
```

```
    for all terms  $w_i$  in  $Q$  do
```

```
         $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
```

```
         $L.add(l_i)$ 
```

```
    end for
```

```
    for all lists  $l_i \in L$  do
```

```
        while  $l_i$  is not finished do
```

```
             $d \leftarrow l_i.\text{getCurrentDocument}()$ 
```

```
             $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
```

```
             $l_i.\text{moveToNextDocument}()$ 
```

```
        end while
```

```
    end for
```

```
    for all accumulators  $A_d$  in  $A$  do
```

```
         $s_d \leftarrow A_d$ 
```

```
         $R.add(s_d, d)$ 
```

```
    end for
```

```
    return the top  $k$  results from  $R$ 
```

```
end procedure
```

The primary disadvantage of the term-at-a-time algorithm is the **memory usage required by the accumulator table A**. Remember that the document-at-a-time strategy requires only the small priority queue R, which holds a limited number of results.

However, the term-at-a-time algorithm makes up for this because of its **more efficient disk access**. Since it reads each inverted list from start to finish, it requires **minimal disk seeking**, and it needs very little **list buffering** to achieve high speeds. In contrast, the document-at-a-time algorithm switches between lists and requires large list buffers to help reduce the cost of seeking.

In practice, neither the document-at-a-time nor term-at-a-time algorithms are used without additional optimizations. These optimizations dramatically improve the running speed of the algorithms, and can have a large effect on the memory footprint.

Optimization Techniques

- There are two main classes of optimizations for query processing.
- The first is to **read less data from the index**, and the second is to **process fewer documents**.
- The two are related, since it would be hard to score the same number of documents while reading less data.
- When using **feature functions that are particularly complex**, focusing on scoring fewer documents should be the main concern.
- For **simple feature functions**, the best speed comes from ignoring as much of the inverted list data as possible.

Conjunctive Processing

- One of the simplest kind of query optimization is **conjunctive processing**.
- By conjunctive processing, we just mean that every document returned to the user needs to contain all of the query terms.
- Conjunctive processing is the **default mode** for many web search engines, in part because of speed and in part because **users have come to expect it**.
- With short queries, conjunctive processing can actually improve efficiency.
- In contrast, search engines that use longer queries, such as entire paragraphs, will not be good candidates for conjunctive processing.
- Conjunctive processing **works best when one of the query terms is rare**, as in the query “fish locomotion”. The word **“fish” occurs about 100 times as often as the word “locomotion”**.
- Since we are only interested in documents that contain both words, the system can skip over most of the inverted list for “fish” in order to find only the postings in documents that also contain the word “locomotion”.
- Conjunctive processing can be **employed with both term-at-a-time and document-at-a-time systems**.

Processing Conjunctive Queries

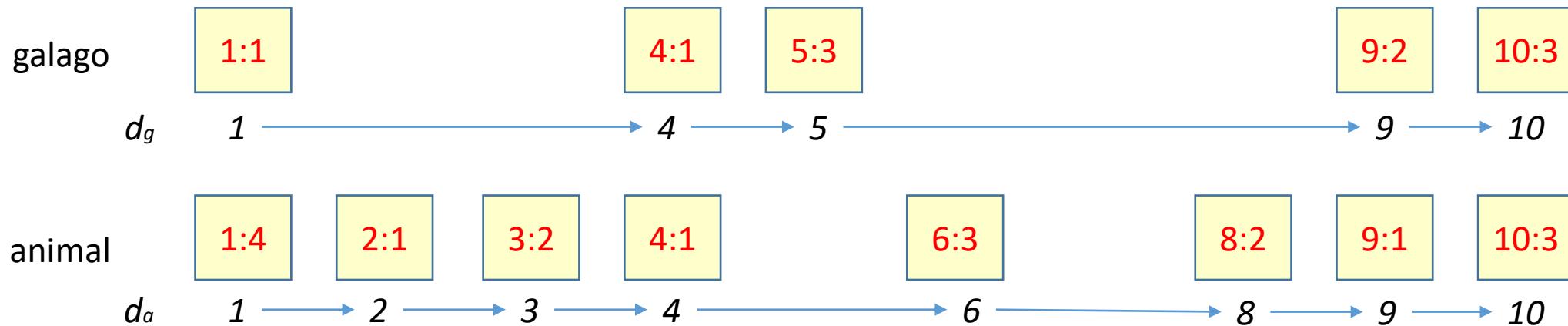
- For many queries, we don't need all of the information stored in a particular inverted list. Instead, it would be more efficient to read just the small portion of the data that is relevant to the query
- Consider the Boolean query “galago AND animal”. The word “animal” occurs in about 300 million documents on the Web versus approximately 1 million for “galago.”
- If we assume that the inverted lists for “galago” and “animal” are in document order, there is a very simple algorithm for processing this query (next slide)



Galagos

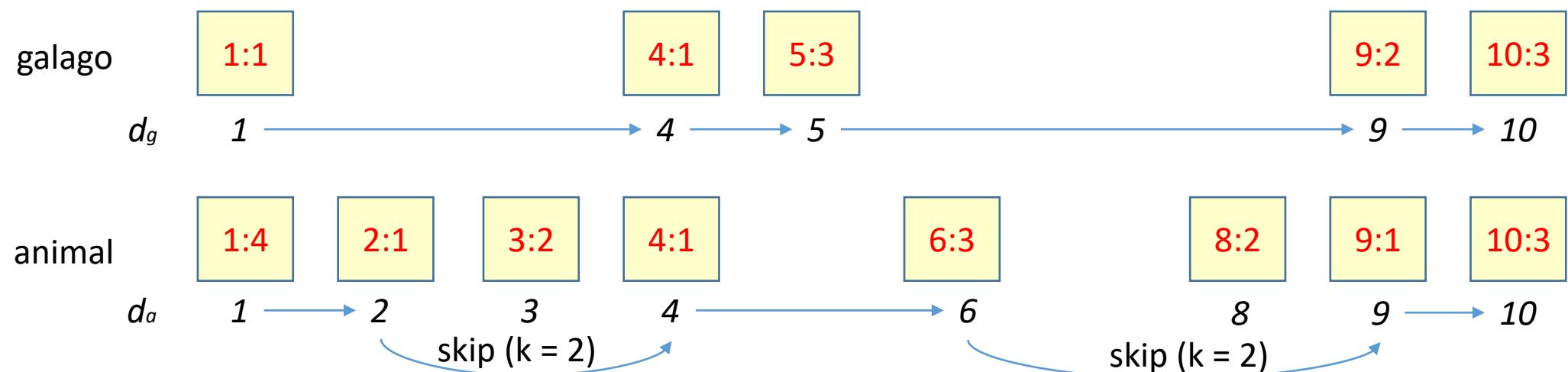
Processing Conjunctive Queries, Simple Algorithm

- Let d_g be the first document number in the inverted list for “galago.”
- Let d_a be the first document number in the inverted list for “animal.”
- While there are still documents in the lists for “galago” and “animal,” loop:
 - If $d_a = d_g$, the document da contains both “galago” and “animal”. Move both d_g and d_a to the next documents in the inverted lists for “galago” and “animal,” respectively.
 - If $d_a < d_g$, set d_a to the next document number in the “animal” list.
 - If $d_g < d_a$, set d_g to the next document number in the “galago” list.



List Skipping

- Unfortunately, the previous algorithm is very expensive. It processes almost all documents in both inverted lists, so we expect the computer to process this loop about 300 million times.
- Over 99% of the processing time will be spent processing the 299 million documents that contain “animal” but do not contain “galago.”
- We can change this algorithm slightly by skipping forward in the “animal” list.
- Every time we find that $da < dg$, we skip ahead k documents in the “animal” list to a new document, sa . If $sa < dg$, we skip ahead by another k documents. We do this until $sa \geq dg$. At this point, we have narrowed our search down to a range of k documents that might contain dg , which we can search linearly. (Chap. 5.4.7)



Skip Pointers

- List skipping can be achieved with skip pointers
- Skipping, however, does not improve the asymptotic running time of reading an inverted list.
- Suppose we have an inverted list that is n bytes long, but we add skip pointers after each c bytes, and the pointers are k bytes long.
- Reading the entire list requires reading(n) bytes, but jumping through the list using the skip pointers requires $\Theta(kn/c)$ time, which is equivalent to $\Theta(n)$.
- Even though there is no asymptotic gain in runtime, the factor of c can be huge.
- For typical values of $c = 100$ and $k = 4$, skipping through a list results in reading just 2.5% of the total data.

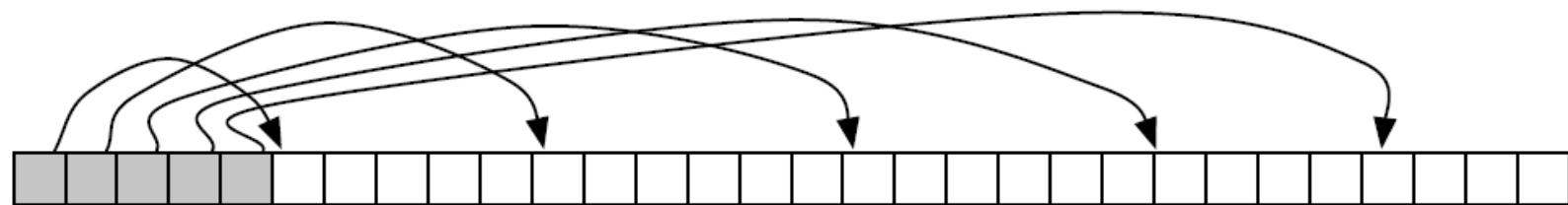


Fig. 5.19. Skip pointers in an inverted list. The gray boxes show skip pointers, which point into the white boxes, which are inverted list postings.

List Skipping Optimization

- Notice that in kn/c , as c gets bigger, the amount of data you need to read to skip through the list drops. So, why not make c as big as possible?
- The problem is that if c gets too large, the average performance drops. Let's look at this problem in more detail.
- Suppose you want to find p particular **postings** in an inverted list, and **the list is n bytes long**, with **k -byte skip pointers located at c -byte intervals**. Therefore, **there are n/c total intervals** in the list. To find those p postings, we need to **read kn/c bytes in skip pointers**, but we also need to read data in p intervals. On average, **we assume that the postings we want are about halfway between two skip pointers**, so we read an additional $pc/2$ bytes to find those postings. The total number of bytes read is then:

$$\frac{kn}{c} + \frac{pc}{2}$$

- you can see that while **a larger value of c makes the first term smaller, it also makes the second term bigger**.
- Therefore, picking the perfect value for c depends on the value of p , and we don't know what p is until a query is executed.
- However, it is possible to use previous queries to simulate skipping behaviour and to get a good estimate for c .

```

1: procedure TERMATATIMERETRIEVAL( $Q, I, f, g, k$ )
2:    $A \leftarrow \text{Map}()$ 
3:    $L \leftarrow \text{Array}()$ 
4:    $R \leftarrow \text{PriorityQueue}(k)$ 
5:   for all terms  $w_i$  in  $Q$  do
6:      $l_i \leftarrow \text{InvertedList}(w_i, I)$ 
7:      $L.\text{add}(l_i)$ 
8:   end for
9:   for all lists  $l_i \in L$  do
10:     $d_0 \leftarrow -1$ 
11:    while  $l_i$  is not finished do
12:      if  $i = 0$  then
13:         $d \leftarrow l_i.\text{getCurrentDocument}()$ 
14:         $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
15:         $l_i.\text{moveToNextDocument}()$ 
16:      else
17:         $d \leftarrow l_i.\text{getCurrentDocument}()$ 
18:         $d' \leftarrow A.\text{getNextAccumulator}(d)$ 
19:         $A.\text{removeAccumulatorsBetween}(d_0, d')$ 
20:        if  $d = d'$  then
21:           $A_d \leftarrow A_d + g_i(Q)f(l_i)$ 
22:           $l_i.\text{moveToNextDocument}()$ 
23:        else
24:           $l_i.\text{skipForwardToDocument}(d')$ 
25:        end if
26:         $d_0 \leftarrow d'$ 
27:      end if
28:    end while
29:  end for
30:  for all accumulators  $A_d$  in  $A$  do
31:     $s_d \leftarrow A_d$             $\triangleright$  Accumulator contains the document score
32:     $R.\text{add}(s_d, d)$ 
33:  end for
34:  return the top  $k$  results from  $R$ 
35: end procedure

```

term-at-a-time algorithm for conjunctive processing

When processing the first term, ($i = 0$), processing proceeds normally. However, for the remaining terms, ($i > 0$), the algorithm processes postings starting at line 17. It checks the accumulator table for the next document that contains all of the previous query terms, and instructs list l_i to skip forward to that document if there is a posting for it. If there is a posting, the accumulator is updated. If the posting does not exist, the accumulator is deleted.

That's it Folks



Further Reading

Chapter 5: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman

Web Services and Web Data

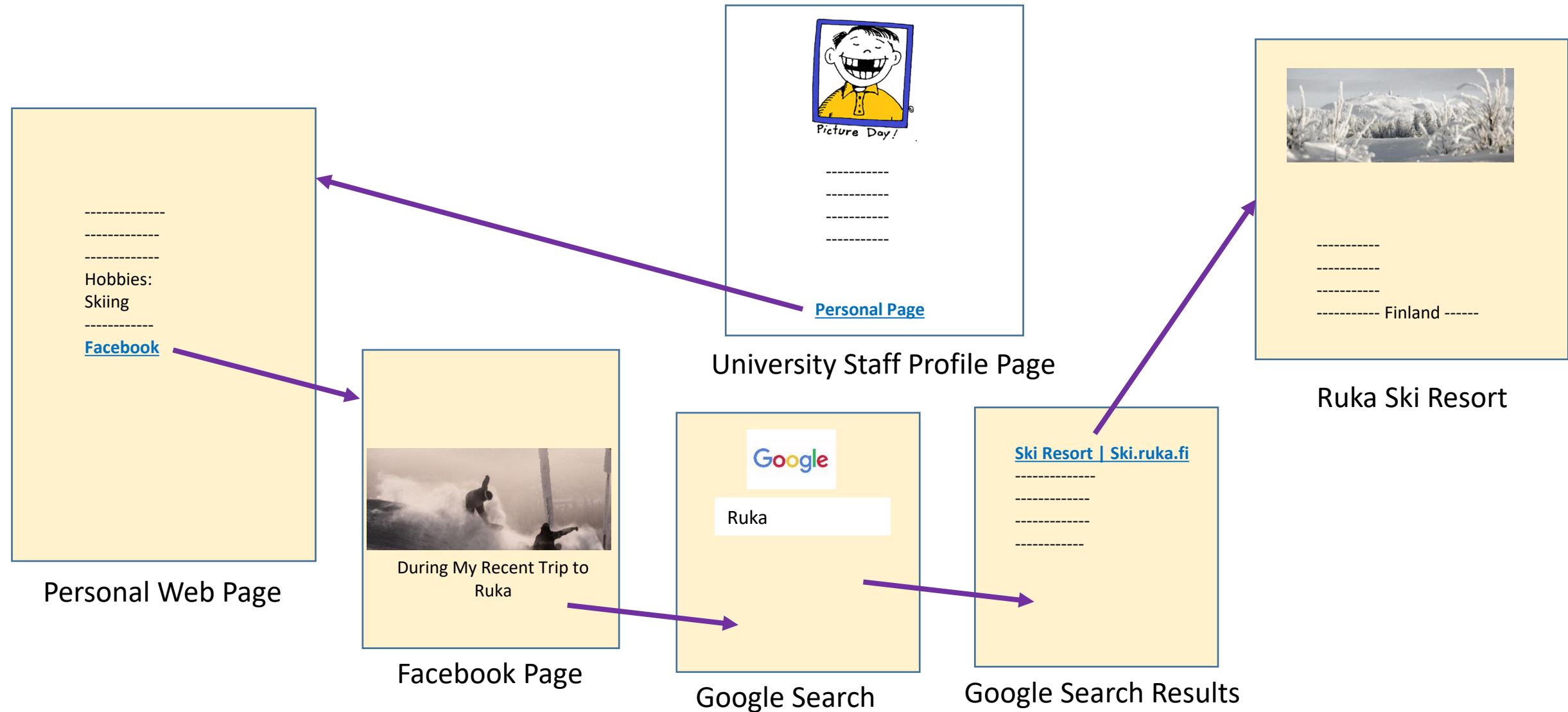
XJCO3011



Session 14 - Linked Data

A Motivating Example

How can you answer the question: Did my teacher go to Finland? by examining his profile page on the university website?



The World Wide Web

- A collection of linked documents that are full of data.
- Many of these documents have **little, if any, structure** imposed on the data (mostly images and free text).
- The data is available in **so many formats** such as HTML, XML, PDF, TIFF, CSV, Excel spreadsheets, embedded tables in Word documents, and many forms of plain text.
- This kind of data has a limitation: it's formatted for **human consumption**.
- It often requires a specialized utility to read it.
- It's **not easy** for automated processes to access, search, or reuse this data.
- Further **processing by people** is generally required for this data to be incorporated into new projects or allow someone to base decisions on it.
- With the exception of some very simple cases, only humans can analyse the semantic relationships between data in various web pages.

The Linked Data Web

- Linked Data refers to a set of techniques for publishing and connecting structured data on the Web
- It adheres to standards set by the World Wide Web Consortium (W3C).`

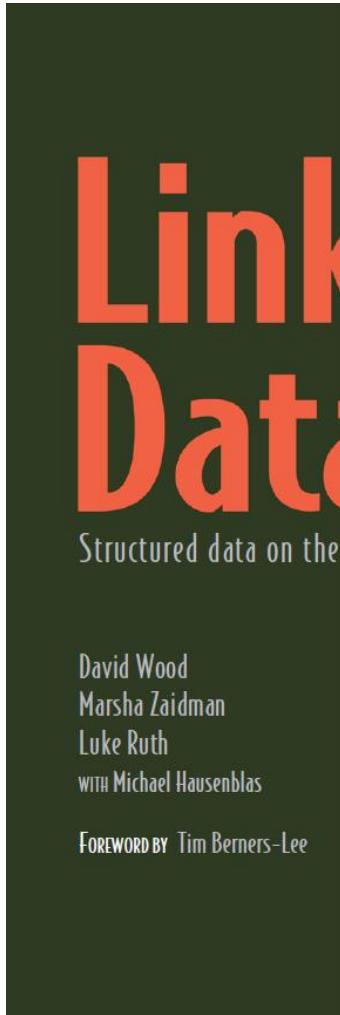
One of the persons is “Anakin,” known as “Darth Vader” and “Anakin Skywalker.” He has a wife named Padme Amidala.

Unstructured Data. Good for Humans

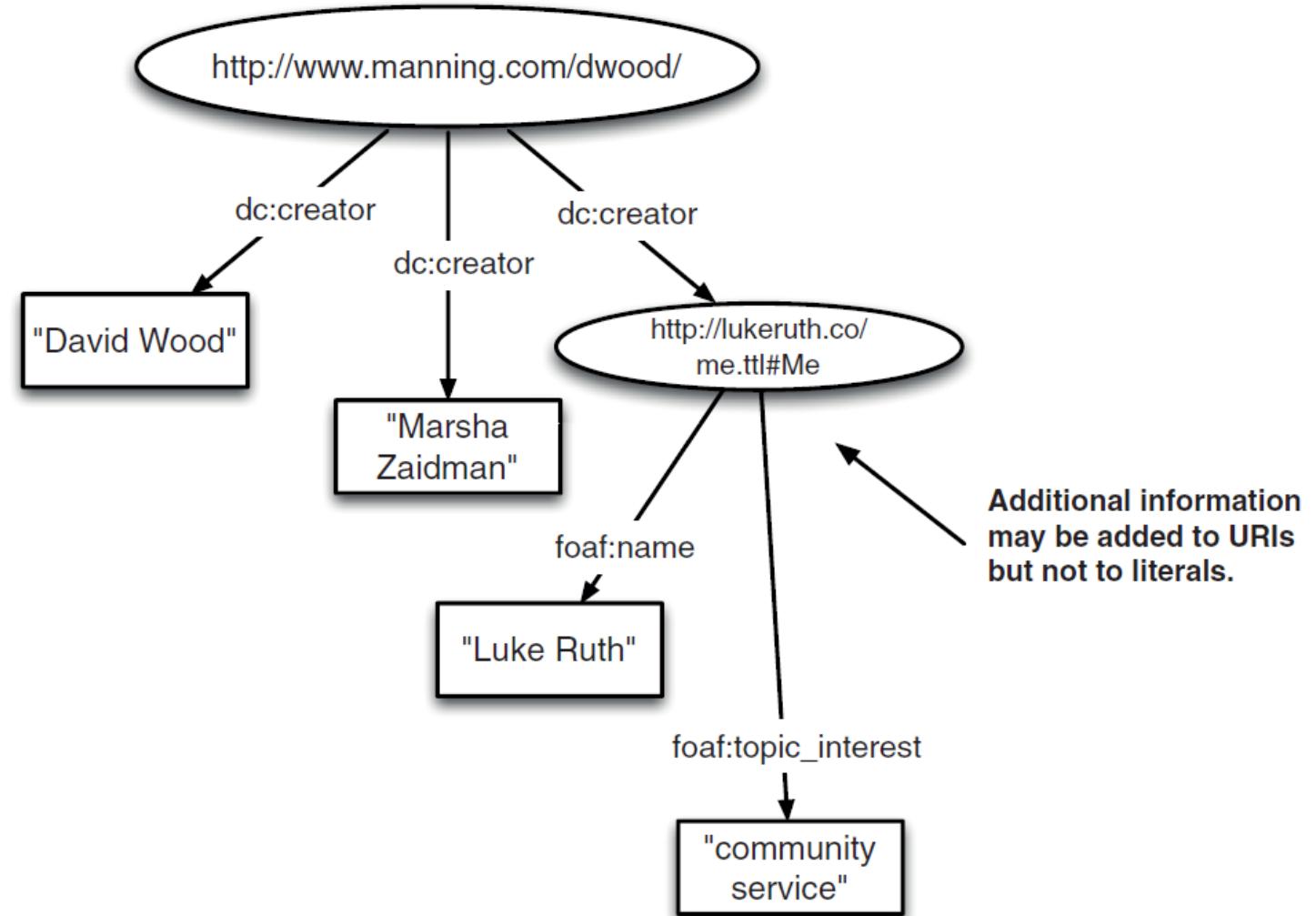
```
@base <http://rosemary.umw.edu/~marsha/starwars/foaf.ttl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rel: <http://purl.org/vocab/relationship>.
@prefix stars: <http://www.starwars.com/explore/encyclopedia/characters/> .
<me> a foaf:Person;
    foaf:family_name "Skywalker";
    foaf:givenname "Anakin";
    foaf:nick "Darth Vader";
    rel:Spouse_Of <stars:padmeamidala/> .
```

Structured data (RDF Turtle format). Good for Automated Agents

The Linked Data Web



Unstructured Data (Picture)



Linked Data (Structured)

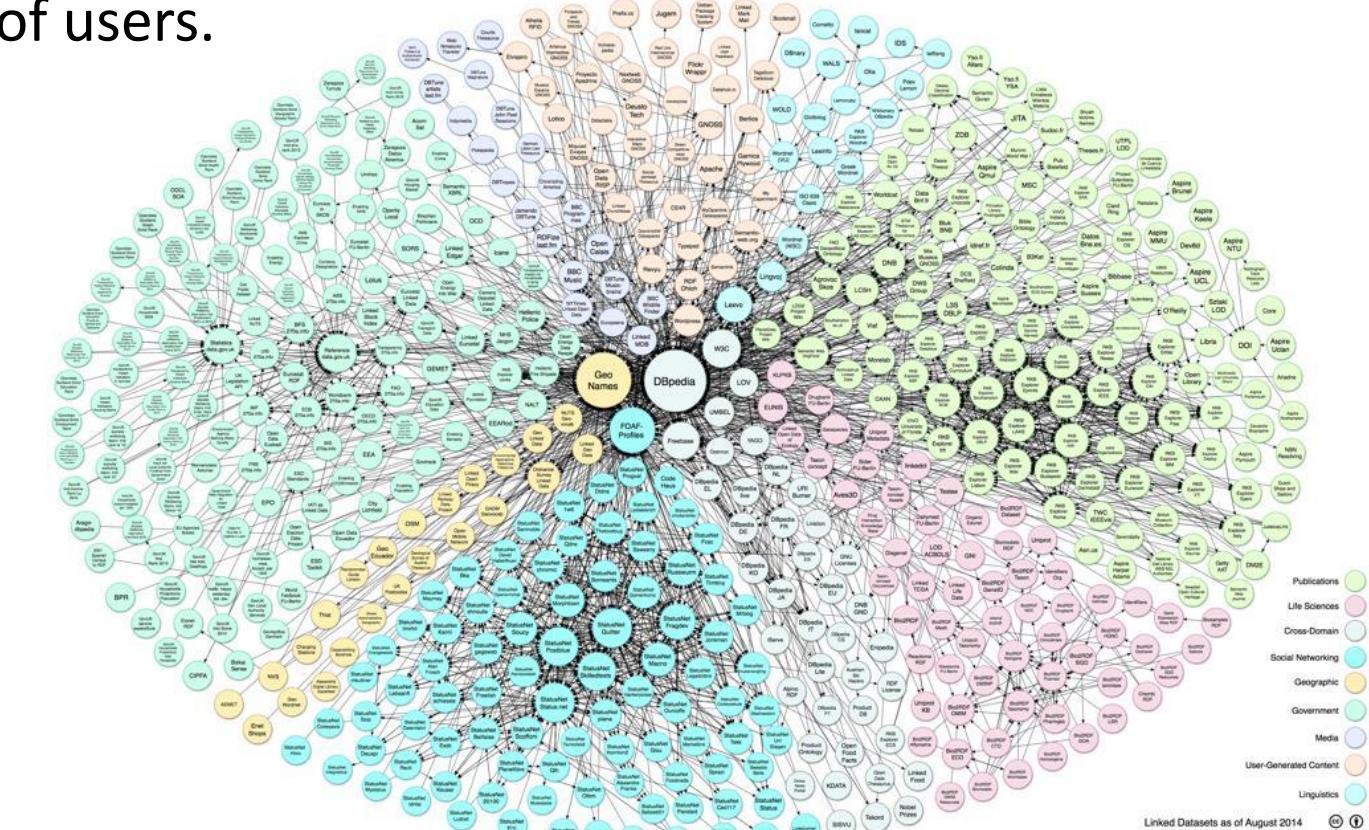
DBpedia

- The DBpedia project (<http://dbpedia.org>) extracts factual information from Wikipedia articles and publish them, on the Web, as structured data.
- Fortunately, most of the content in Wikipedia is highly structured (e.g. the "infobox" in the upper right of a Wikipedia page).
- The 2016-04 release of the DBpedia data set describes **6 million entities**, including 1.5 million persons, 800,000 places, 135,000 music albums, 100,000 films, 300,000 species and 5,000 diseases.
- This dataset is open and may be explored by anyone to extract or create new knowledge.

 DBpedia	
Developer(s)	Leipzig University University of Mannheim OpenLink Software
Initial release	10 January 2007 (11 years ago)
Stable release	DBpedia 2016-10 / July 4, 2017
Repository	https://github.com/dbpedia/ 
Written in	Scala · Java · VSP
Operating system	Virtuoso Universal Server
Type	Semantic Web · Linked Data
License	GNU General Public License
Alexa rank	▲ 81,381 (as of September 2016) ^[1]
Website	dbpedia.org 

The Semantic Web

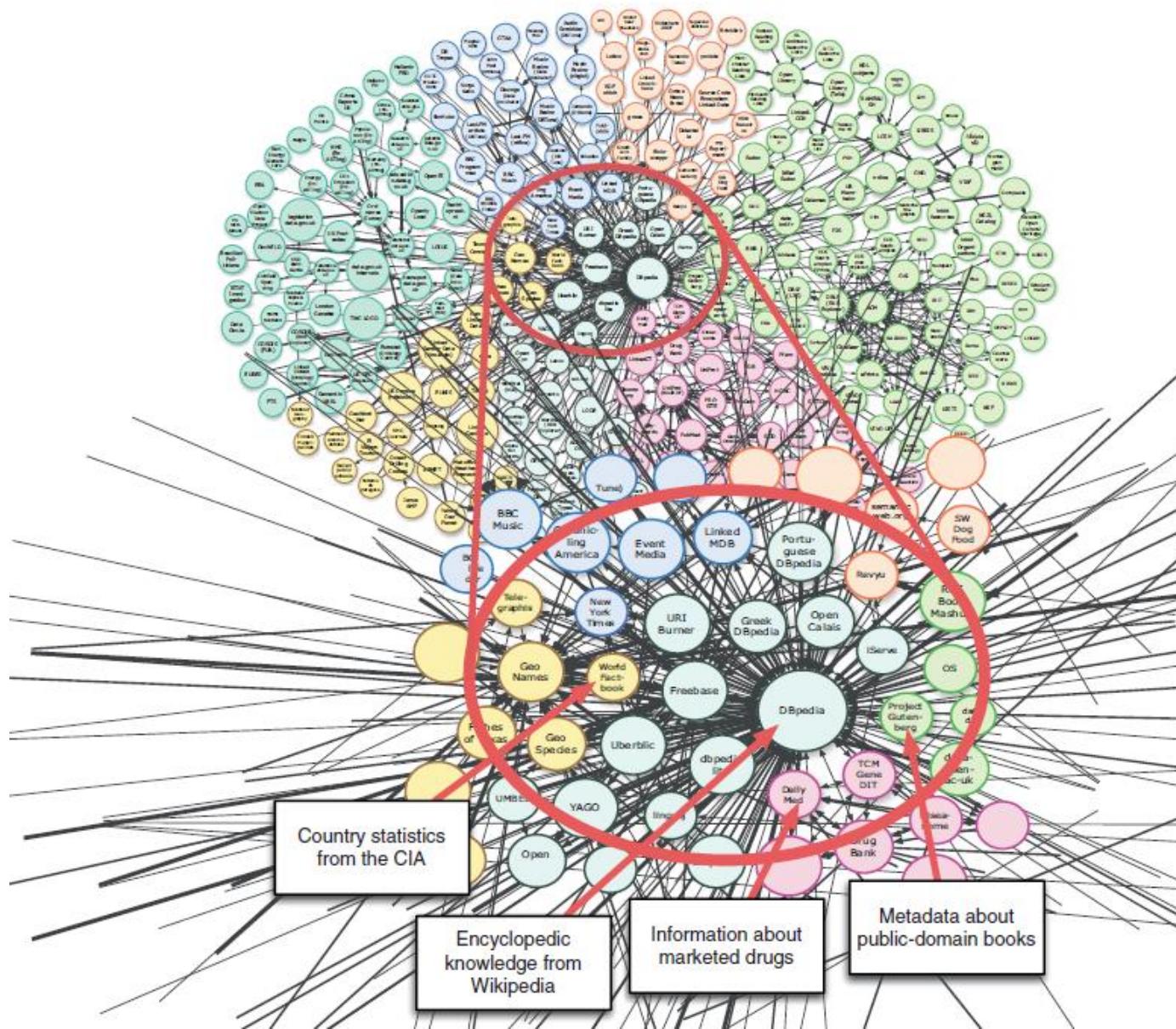
- When the elements of the web are structured data entities, connected to each other with semantic relations, the resulting graph is called the **semantic web**.
- This enables automated agents to access the Web more intelligently and perform more complicated queries on behalf of users.
- WWW: emphasizes the connection between people and information
- The semantic web: helps machines understand the relationships and meanings between pieces of information



The Linking Open Data project

- There exist a **huge body of datasets** that are open and freely available on the Web. These open-content projects are as diverse as encyclopedias, dictionaries, government statistics, chemical and biological collections, endangered species, bibliographic data, music, artists, songs, academic research papers. They are all available through the **same data format (RDF)**. This is all due to the Linking Open Data (LOD) project.
- The Linked Open Data (LOD) project is a community activity started in 2007 by the W3C's **Semantic Web Education and Outreach** (SWEO) Interest Group. The project's goal is to "make data freely available to everyone."
- This collection of Linked Data published on the Web is referred to as the **LOD cloud**. An attempt to visualizing the LOD cloud is shown in the next slide.

The Linking Open Data project



- The Linked Open Data cloud in late 2011.
- The circles represent freely available datasets and the arrows represent links between them.
- Some quick facts regarding the LOD cloud:
 - The LOD cloud has doubled in size every 10 months since 2007 and currently consists of more than 300 datasets from various domains. All of this data is available for use by developers!
 - As of late 2011, the LOD cloud contained over 295 datasets from various domains, including geography, media, government, and life sciences. In total the LOD cloud contained over 31 billion data items and some 500 million links between them.
 - etc....

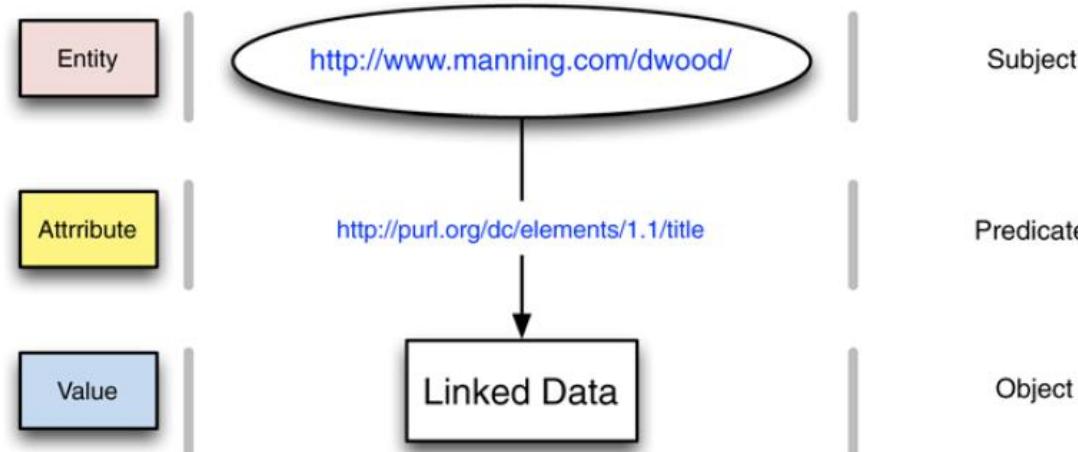
Quick Facts About the LOD Cloud.

- The LOD cloud has grown so large that no attempt was made to visualize after 2011.
- More than **40%** of the Linked Data in the LOD cloud is contributed **by governments** (mainly from the **United Kingdom** and the United States), followed by geographical data (22%) and data from the life sciences domain (almost 10%).
- Life sciences (including some large pharmaceutical companies) contribute over 50% of the links between datasets.
- Publication data (from books, journals, and the like) comes in second with 19%, and the media domain (the BBC, the New York Times, and others) provides another 12%.
- The original data owners themselves publish one-third of the data contained in the LOD cloud, whereas third parties publish 67%. For example, many universities republish data from their respective governments in Linked Data formats, often cleaning and enhancing data descriptions in the process.

The Resource Description Framework (RDF)

- Linked Data uses RDF as a data model.
- A single RDF statement describes *two things and a relationship between them*.
- This is called an Entity-Attribute-Value (EAV) data model
- Linked Data people often call the three elements in a statement the **subject**, the **predicate**, and the **object**.

	A	B
1	id	title
2	http://www.manning.com/dwood/	Linked Data
3		



The Elements of RDF Statements

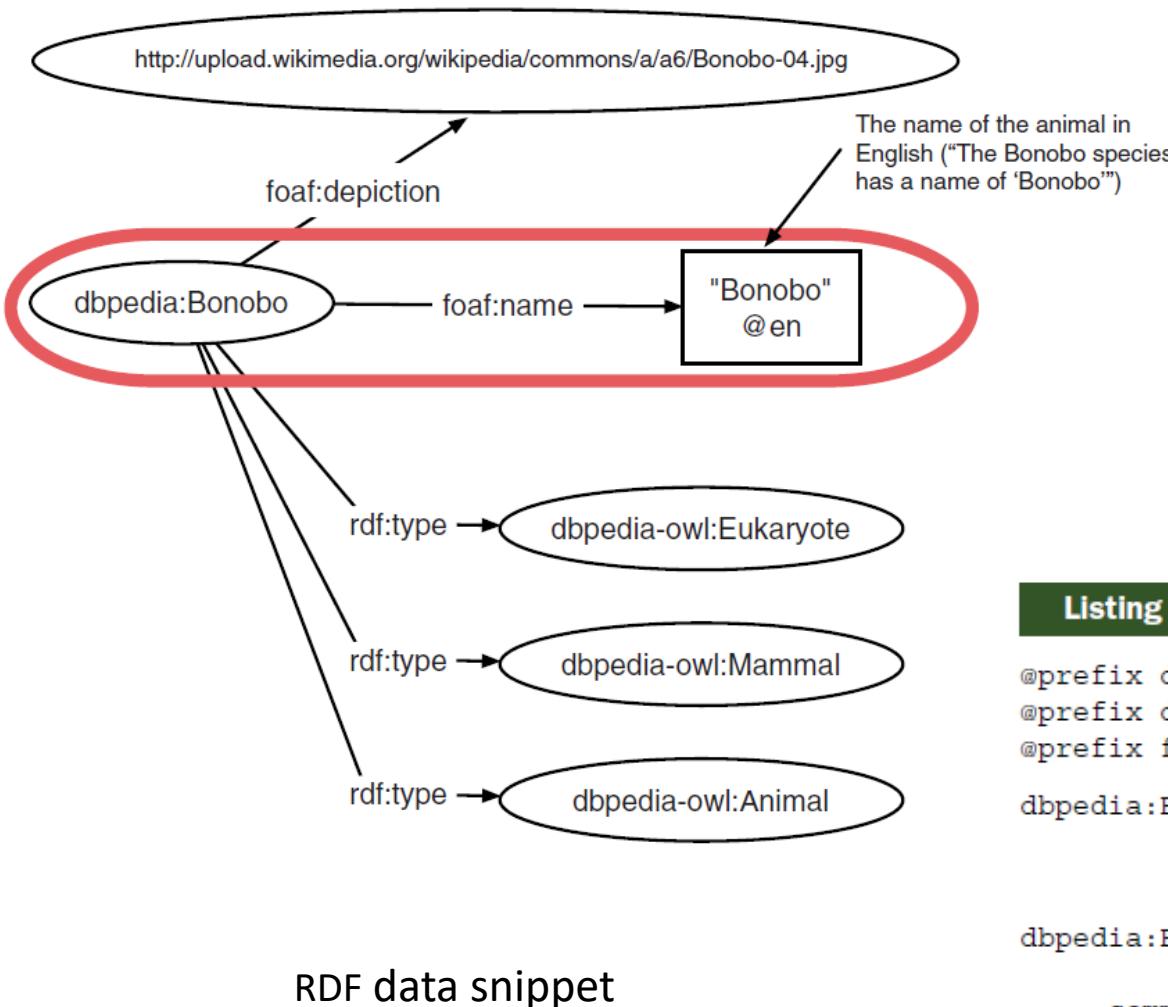
- An entity (or **subject**) is anything that we can name by a URI, such as a person, a book, a car, or a web page. In the previous case, the subject is the URI that uniquely identifies the book.
- An entity's attribute (or **predicate**) relates the subject to another entity or provides information about the entity itself (which we call a property of the subject). In the previous case, we're using a relationship for the book's title as the predicate and the book's title as the **object**.
- Through this standardized data model, an API is created that's consistent over all Linked Data sources. You only have to learn the Linked Data model once and you can use any kind of data source that complies with it.
- Anytime you **want to know what a predicate means**, you can **type its URI into a web browser** and look for information about it.

RDF Example

```
<http://example.com/my_temperature_data> rdfs:label "Temperature observations";  
rdfs:comment "Temperature observations at Galway Airport";
```

- <http://example.com/my_temperature_data> is a URI representing a sample spreadsheet of temperature data, which forms **the entity (subject)** of an RDF statement.
- The two components in rdfs:label "Temperature observations"; are the **attribute (predicate)** and the **property (object)** of the first RDF statement. In this case, we're saying that the spreadsheet may be given a human-readable label of "Temperature observations".
- rdfs:comment "Temperature observations at Galway Airport"; provides another attribute and property for the same subject, which forms another RDF statement.
- We can keep adding information about our spreadsheet that way until we're finished.
- There's no restriction in RDF about what you can link to or describe. RDF statements create graphs of metadata. We often use the term RDF graph because of this.

RDF Diagrams



A Bonobo

Listing 1.2 Excerpt of the Linked Data about bonobos in Turtle format

```
@prefix dbpedia: <http://dbpedia.org/resource/> .  
@prefix dbpedia-owl: <http://dbpedia.org/ontology/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
dbpedia:Bonobo rdf:type dbpedia-owl:Eukaryote ,  
                dbpedia-owl:Mammal ,  
                dbpedia-owl:Animal .  
  
dbpedia:Bonobo foaf:name "Bonobo"@en ;  
                 foaf:depiction <http://upload.wikimedia.org/wikipedia/  
                               commons/a/a6/Bonobo-04.jpg> ;
```

Name of the animal in English ("A Bonobo has a name of 'Bonobo'")

- A useful link defines a textual syntax for RDF called Turtle that allows an RDF graph to be completely written in a compact and natural text form : <https://www.w3.org/TR/turtle/#grammar-production-predicateObjectList>

That's it Folks



Further Reading

Chapter 1 and 2: Linked Data Structured Data on the Web, David Wood et al.

Web Services and Web Data

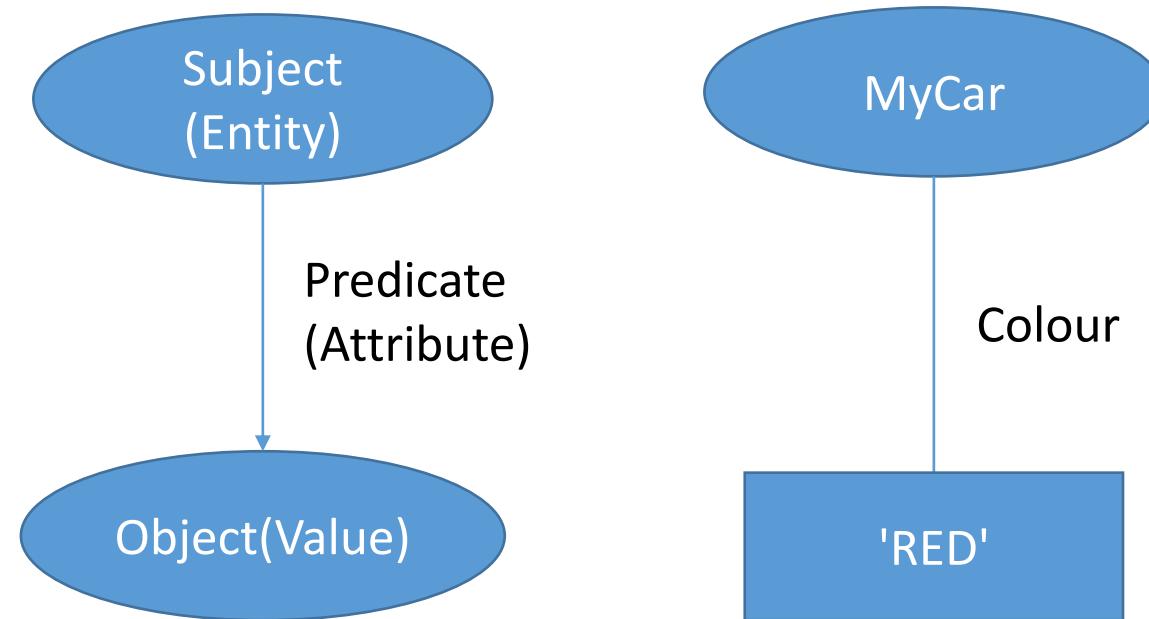
XJCO3011



Session 15 - RDF: the Resource Description Framework

RDF Statements

- RDF descriptions are comprised of a number of statements, called **triples**.
- Each statement must have three parts: a subject, a predicate, and an object (hence the name triple)
- RDF is a World Wide Web Consortium (W3C) specification (<http://www.w3.org/standards/techs/rdf>)



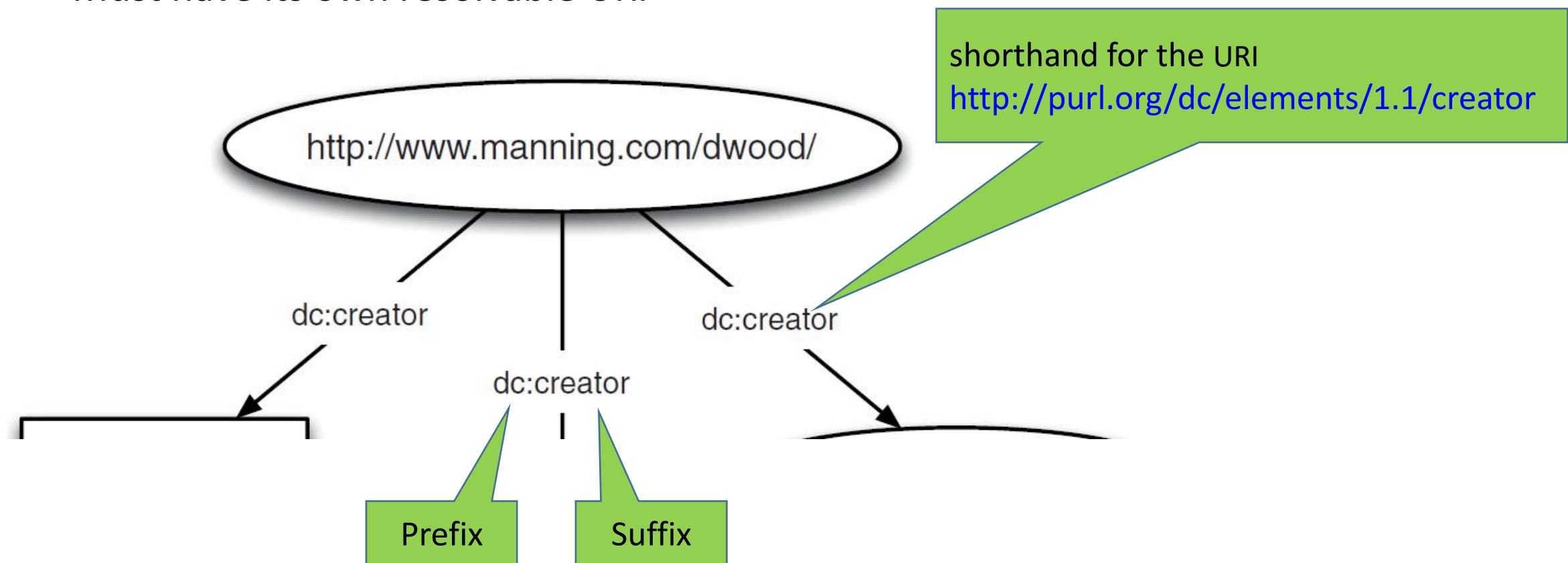
The Subject

- Represents the entity that is being described
- Must be a resolvable URI (Web URL)
- Requesting the URL should return information about the subject (entity)



The Predicate

- Determines an attribute of the subject
- Must have its own resolvable URI



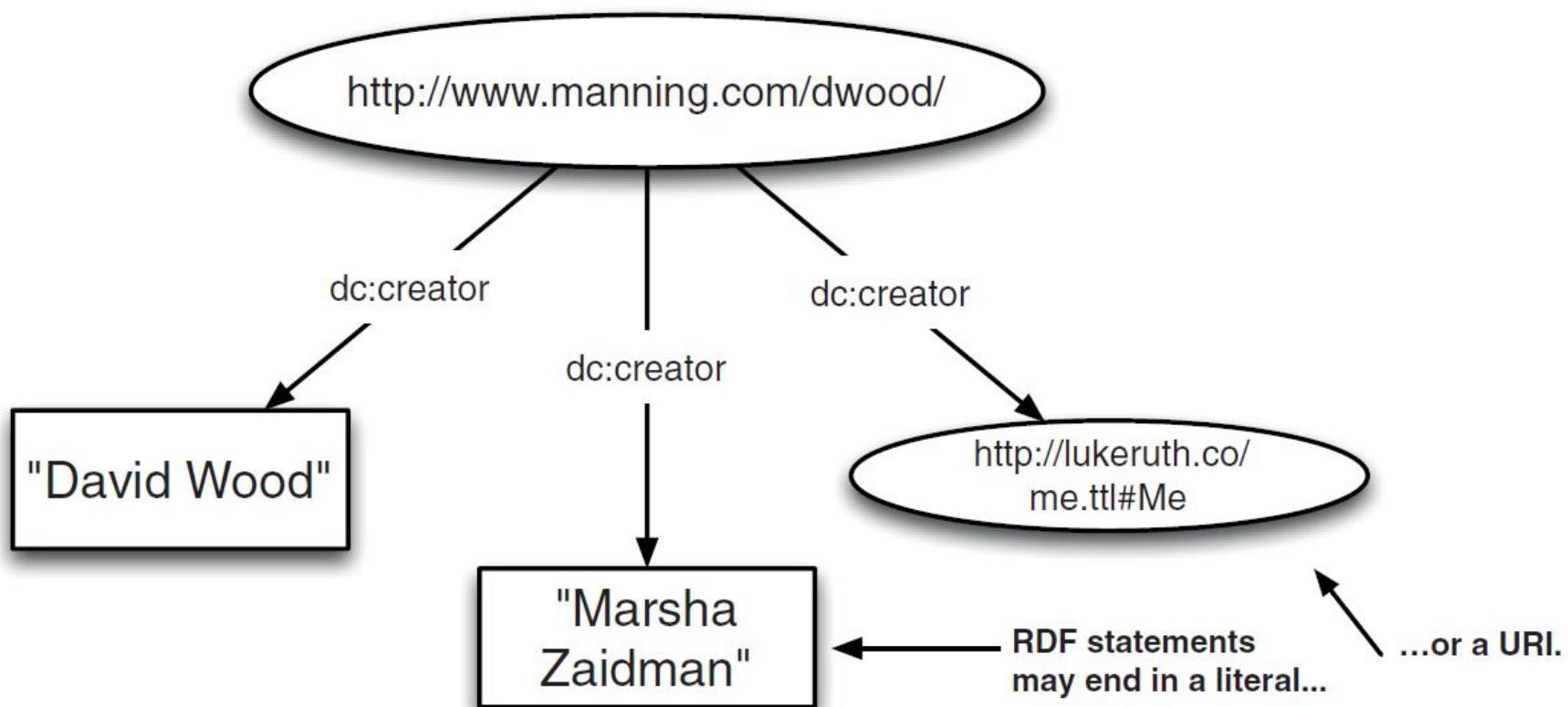
Resolving Predicates

- If you need to know what the predicate means and find other information about it, just type the full URL of the predicate into your browser.
- For example, Here is the result of resolving the URL
<http://purl.org/dc/elements/1.1/creator>.
- You can find that the predicate itself defined as an RDF statement

```
<http://purl.org/dc/elements/1.1/creator>
  dcterms:description "Examples of a Creator include a person, an organization, or a service. Typically,
  dcterms:hasVersion <http://dublincore.org/usage/terms/history/#creator-006> ;
  dcterms:issued "1999-07-02"^^<http://www.w3.org/2001/XMLSchema#date> ;
  dcterms:modified "2008-01-14"^^<http://www.w3.org/2001/XMLSchema#date> ;
  a rdf:Property ;
  rdfs:comment "An entity primarily responsible for making the resource."@en ;
  rdfs:isDefinedBy <http://purl.org/dc/elements/1.1/> ;
  rdfs:label "Creator"@en ;
  skos:note "A second property with the same name as this property has been declared in the dcterms: name
document \"DCMI Metadata Terms\" (http://dublincore.org/documents/dcmi-terms/) for an explanation."@en .
```

The Object

- Determines the value of the attribute, or the object of the subject
- Can be a resolvable URI or a literal



URI Abbreviations

- A URL can be abbreviated into a prefix:suffix, such as dc:Creator
- Here are some commonly used abbreviations in RDF files, but you can always define your own abbreviations if needed.
- Each one of these domains contains definitions for standard RDF vocabulary that are commonly used in Linked Data.

Prefix	Namespace URI
dc:	http://purl.org/dc/elements/1.1/
foaf:	http://xmlns.com/foaf/0.1/
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
vcard:	http://www.w3.org/2006/vcard/ns#

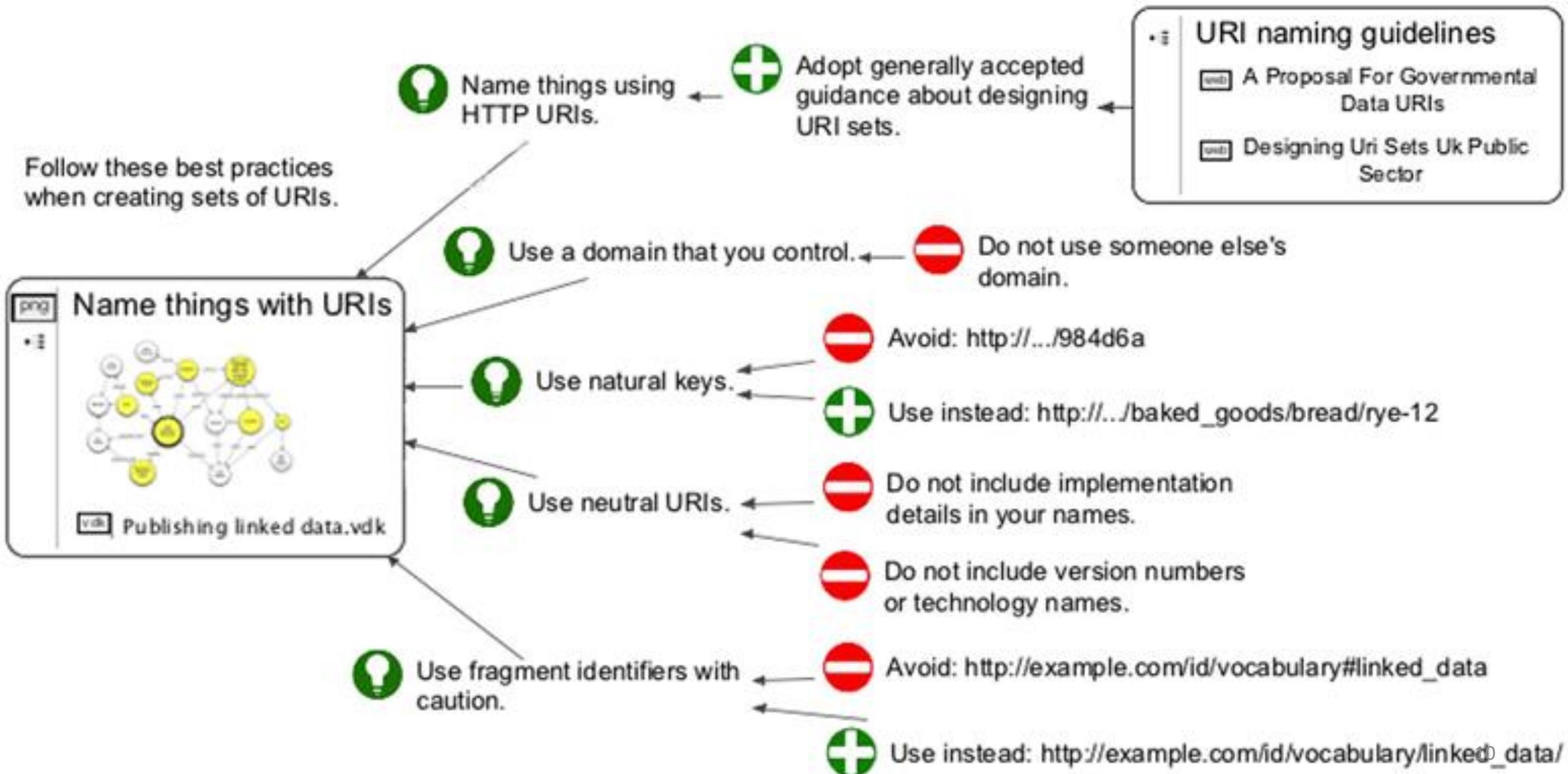
Guidelines for Minting URIs

- Name things with URIs most of the time.
- Use a DNS domain that you control.
- Use natural keys that can be easily understood by humans. For example, do not use
`(http://paulsbakery.example.com/984d6a)`
but use:
`(http://paulsbakery.example.com/baked_goods/bread/rye-12).`
- Make your URIs neutral to implementation details. For example, do not use
`http://example.com/index.aspx`

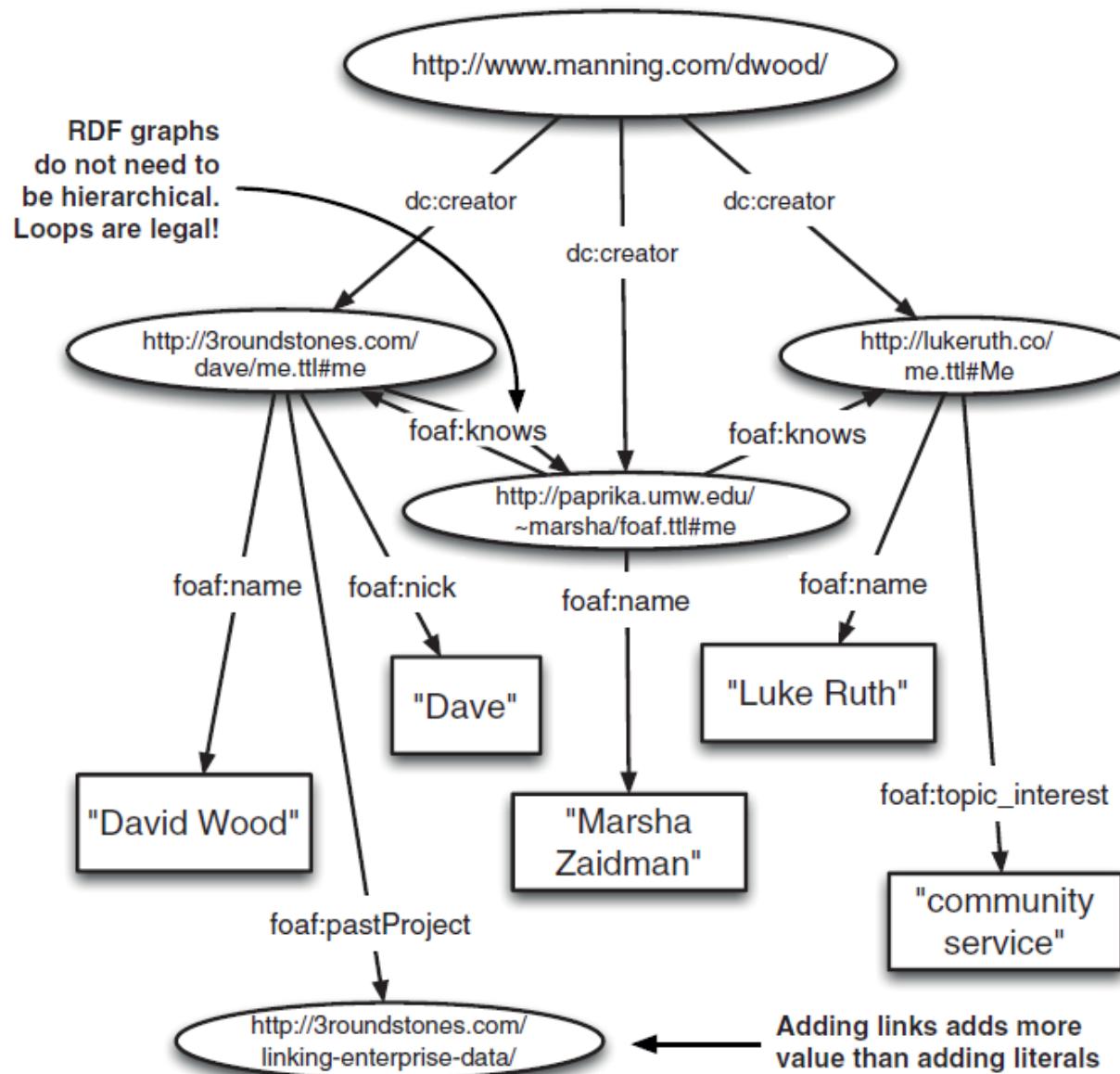
Fragment Identifiers (the #)

- Fragment identifiers are the part of the URL that follows the hash symbol (#)
- Fragment identifiers are often used in HTML pages to point to a particular section in the page
- Fragment identifiers are not passed to web servers by web clients
- This means that if you try to resolve a URI such as
http://example.com/id/vocabulary#linked_data, your browser will just send
<http://example.com/id/vocabulary> to the server.
- Fragment identifiers are used by many Linked Data vocabularies because **the vocabulary is often served as a document and the fragment is used to address a particular term within that document.**

Minting URIs Summary

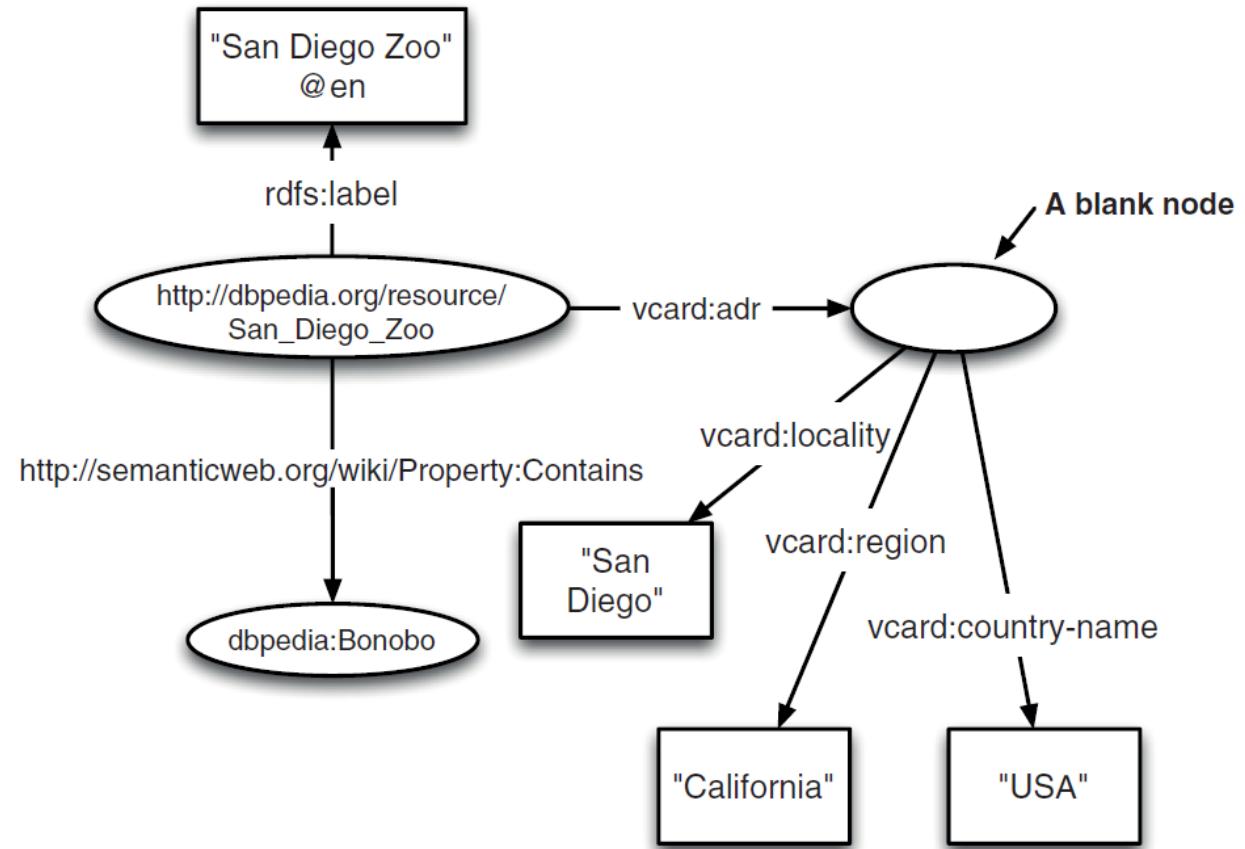


More Comprehensive RDF example



Blank Nodes

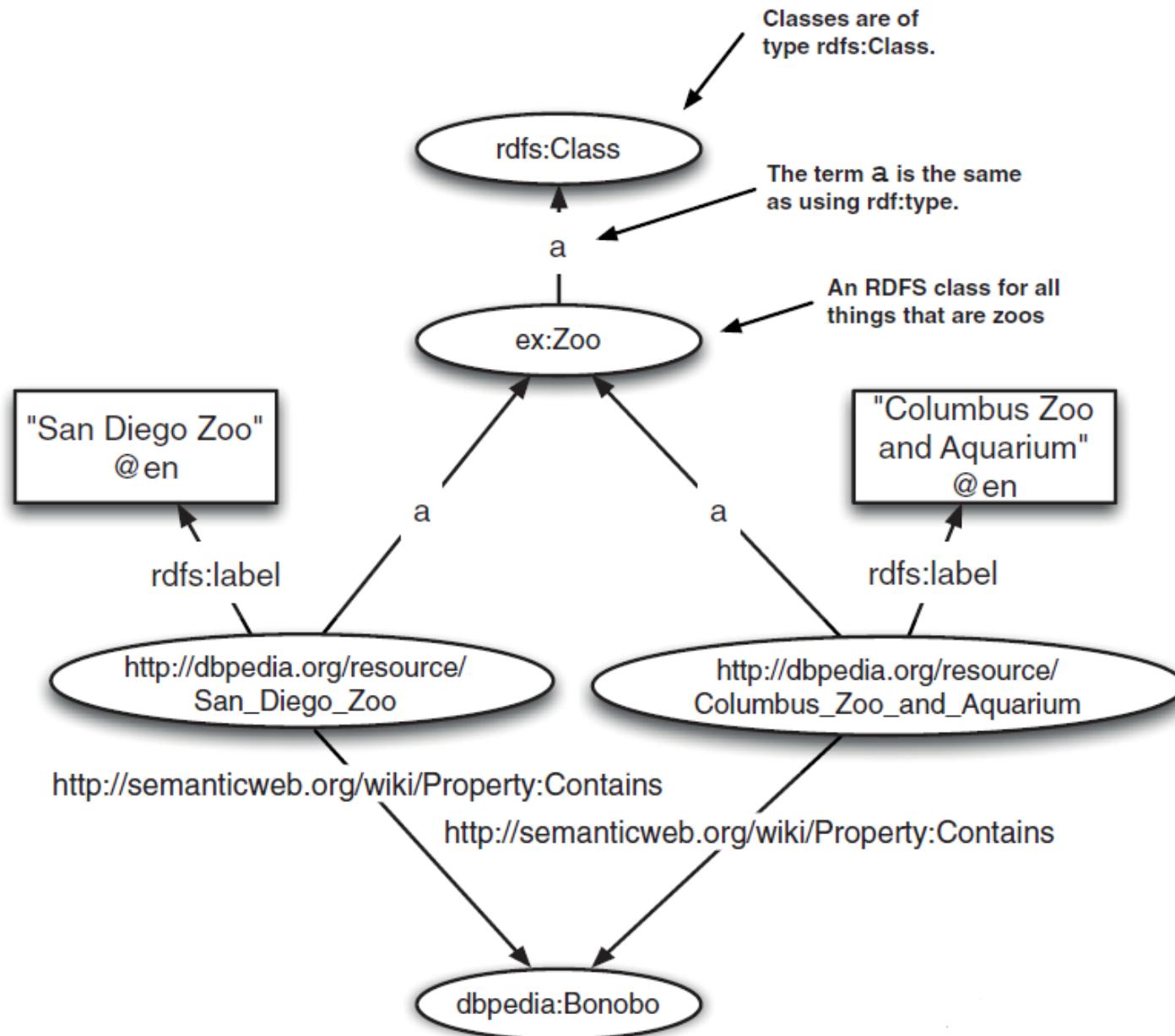
- Useful when you need to link to a collection of items but don't want to bother making up a URI for it
- Some RDF databases automatically assign URIs to blank nodes so they may be more easily operated upon. This process is known as *skolemization*
- In general, avoid using blank nodes as much as possible



Classes

- RDF resources may be divided into groups called classes using the property `rdf:type` in the RDF Schema (RDFS) standard
- The members of a class are known as instances of the class, just as they are in object-oriented programming
- RDFS classes are themselves RDF resources and are of type `rdfs:Class`.
- The `rdfs:subClassOf` property may be used to state that one class is a subclass of another.

Class Example



RDF Vocabulary

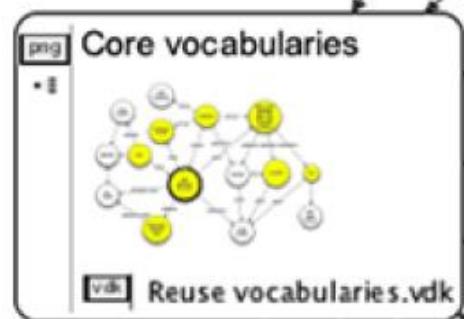
- RDF vocabularies provide definitions of the terms used to make relationships between data elements
- RDF vocabularies are distributed over the Web, are developed by people all over the world, and only come into common use in Linked Data if a lot of people choose to use them.
- We have already seen a number of terms used in Linked Data such as foaf:name, rdfs:label, and vcard:locality.
- These terms are grouped together to form RDF vocabularies, examples:
 - The terms rdfs:label, rdfs:comment, and rdfs:seeAlso are all defined in the RDF Schema vocabulary (<http://www.w3.org/2000/01/rdf-schema#>)
 - The terms vcard:locality, vcard:region, and vcard:country-name all come from the vCard vocabulary associated with the vcard prefix (<http://www.w3.org/2006/vcard/ns#>)

Who Mints RDF Vocabularies

- Anyone can make an RDF vocabulary, and many people do.
- This would seem to be a recipe for disaster. How can anyone reuse Linked Data if it contains terms that you've never seen before?
- There are two ways to make this problem tractable:
 - Make certain that the URIs defining Linked Data vocabularies themselves follow the Linked Data principles
 - Reuse existing vocabularies whenever possible.

Core RDF Vocabularies

Use terms from these core vocabularies to describe commonly understood data.



- ? Naming things? ← Use rdfs:label, foaf:name, skos:prefLabel.
- ? Describing people? ← Use FOAF, vCard. **FOAF: Friend of a Friend**
- ? Describing addresses? ← Use vCard. **vCard: Virtual Contact File**
- ? Describing projects? ← Use Description of a Project (DOAP).
- ? Describing web pages and other publications? ← Use dc:creator and dc:description. **dc: Dublin Core**
- ? Describing an RDF vocabulary? ← Use a VoID description.
- ? Describing existing taxonomies? ← Use SKOS. **SKOS: Simple Knowledge Organization System**

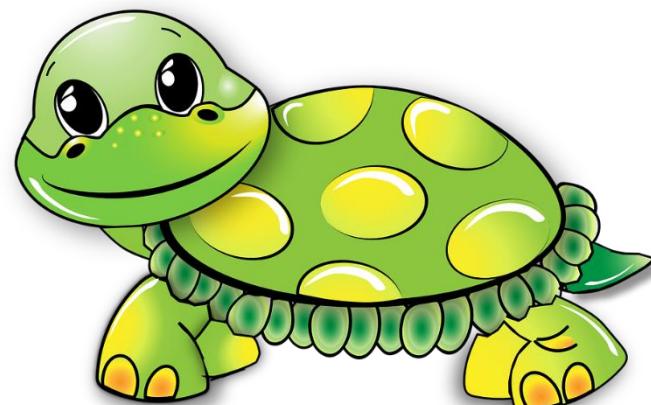
• See also
vdk Authoritative vocabularies.vdk

• Links to core vocabularies

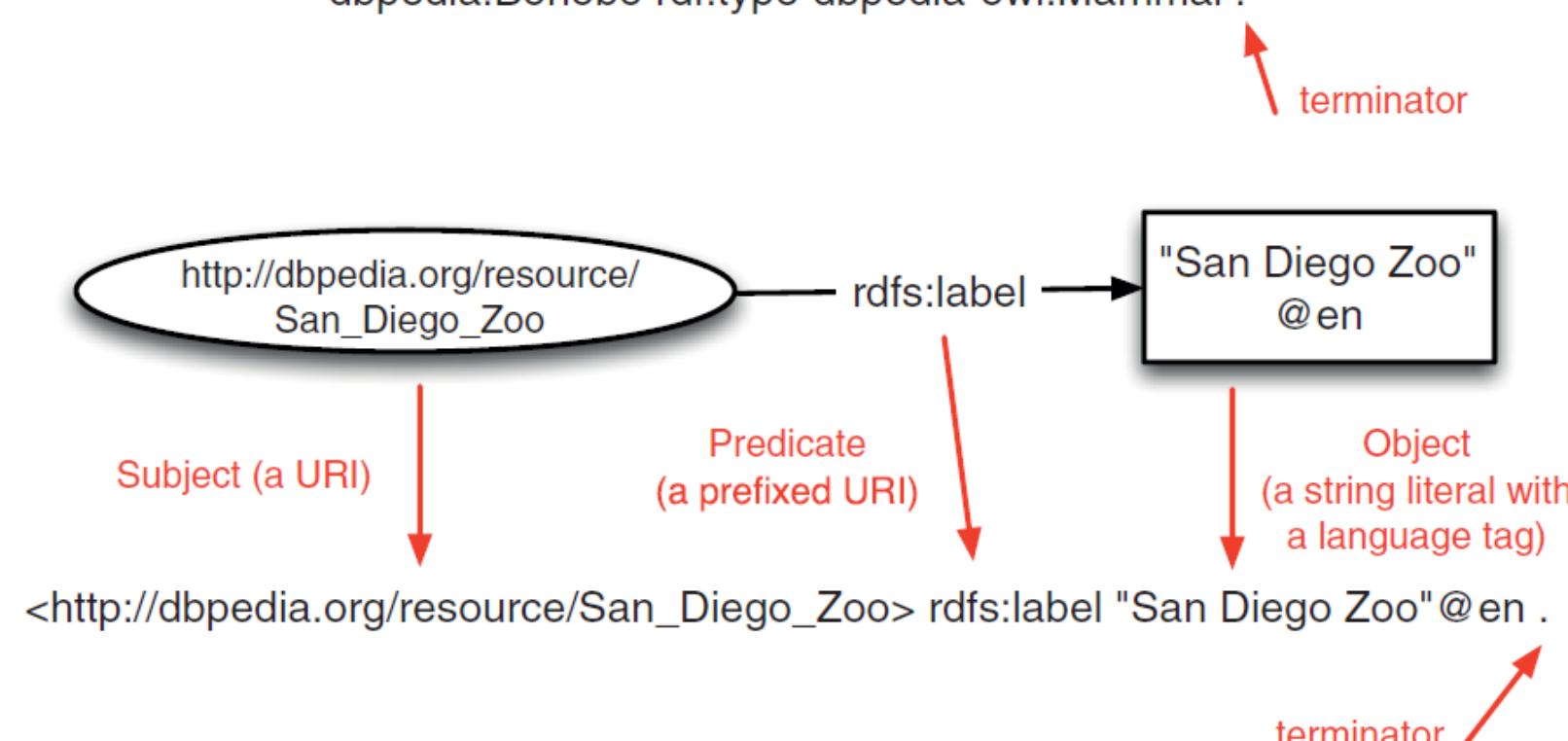
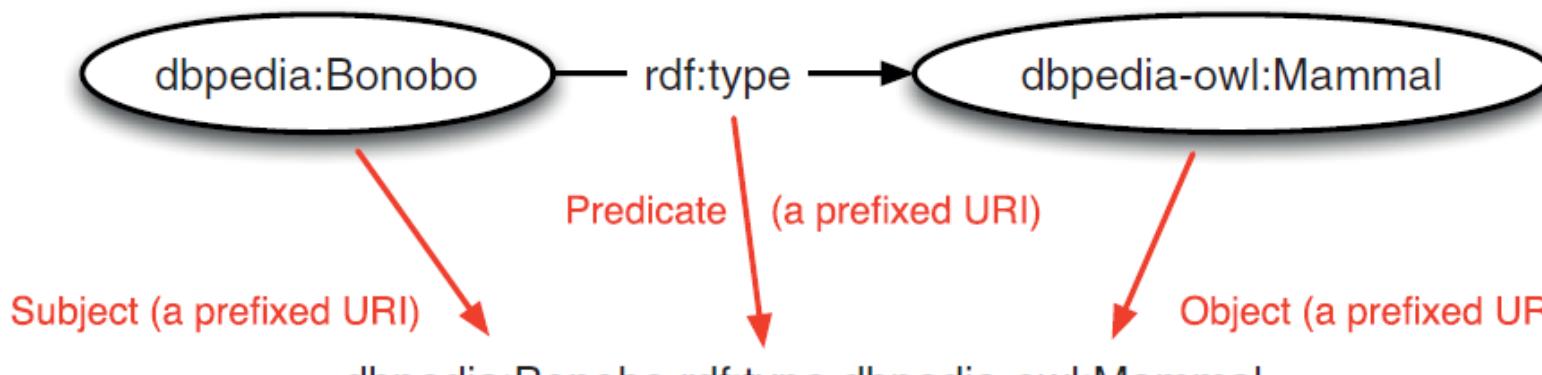
- DOAP
- Dublin Core
- FOAF
- SKOS
- vCard
- VoID

RDF formats for Linked Data

- RDF is a data model, not a format.
- Different formats can be used to serialize (express) RDF data
- Some of the most commonly used formats are:
 - Turtle—A simple, human-readable format
 - RDF/XML—The original RDF format in XML
 - RDFa—RDF embedded in HTML attributes
 - JSON-LD—A newer format aimed at web developers



Turtle: Terse RDF Triple Language



Bonobo example data in Turtle format



```
@prefix dbpedia: <http://dbpedia.org/resource/> .  
@prefix dbpedia-owl: <http://dbpedia.org/ontology/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
@prefix ex: <http://example.com/> .  
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

Prefixes are defined first

Every statement must end with a period

dbpedia:Bonobo

 rdf:type dbpedia-owl:Eukaryote , dbpedia-owl:Mammal ,

 dbpedia-owl:Animal ;

 rdfs:comment "The bonobo, Pan paniscus, previously called the pygmy chimpanzee and less often, the dwarf or gracile chimpanzee, is a great ape and one of the two species making up the genus Pan; the other is Pan troglodytes, or the common chimpanzee. Although the name \"chimpanzee\" is sometimes used to refer to both species together, it is usually understood as referring to the common chimpanzee, while Pan paniscus is usually referred to as the bonobo."@en ;

 foaf:depiction <http://upload.wikimedia.org/wikipedia/commons/a/a6/Bonobo-04.jpg> ;

 foaf:name "Bonobo"@en ;

 rdfs:seeAlso <http://eol.org/pages/326448/overview>

Commas separate multiple objects of the same predicate

Semicolons separate multiple predicates of the same subject

That's it Folks



Further Reading

Chapter 2: Linked Data Structured Data on the Web, David Wood et al.

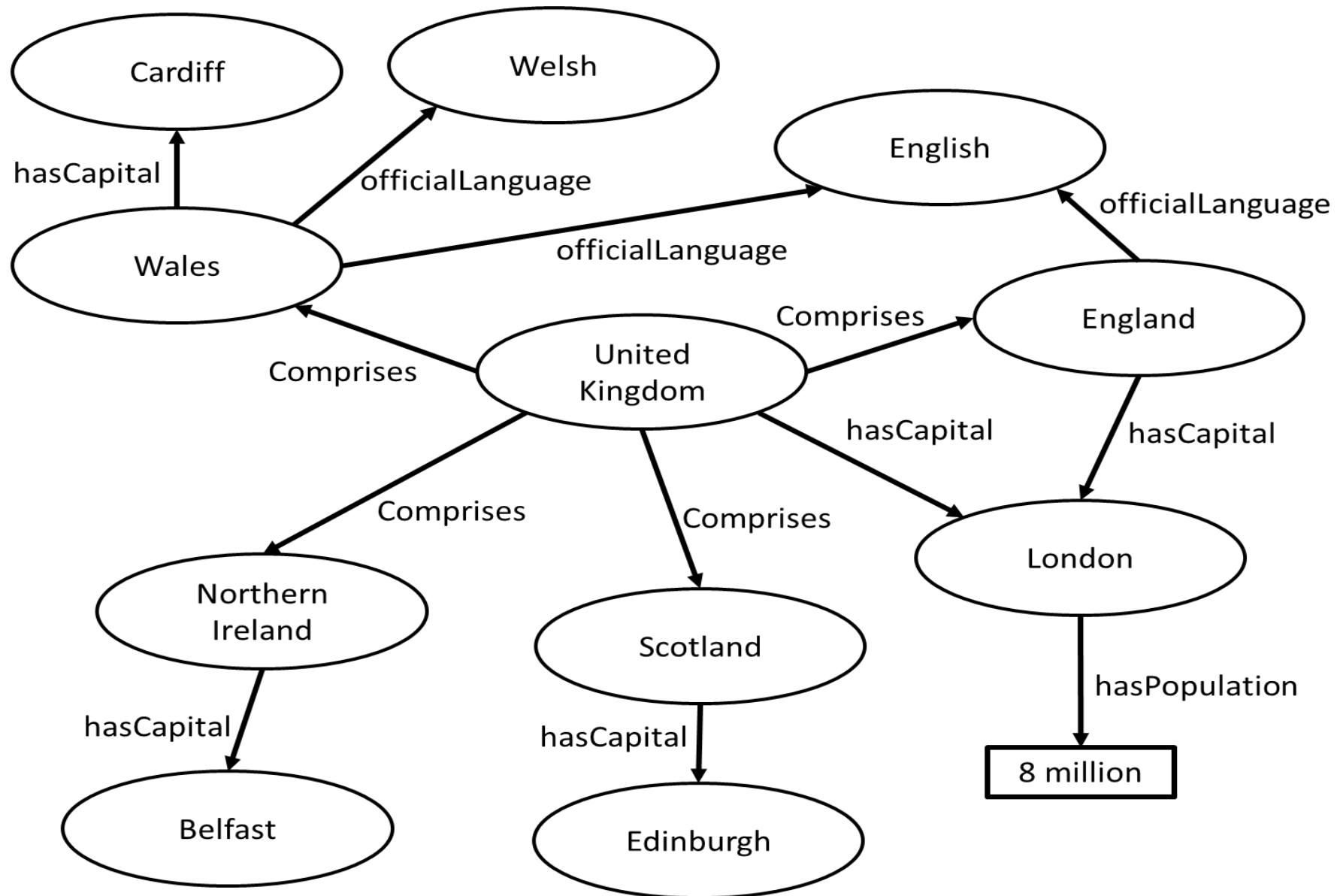
Web Services and Web Data

XJCO 3011



Session 16 – RDF Exercises

- Draw an RDF diagram to represent the concepts in the following statement:
- “The United Kingdom comprises four countries: England (whose capital is London), Wales, Scotland, and Northern Ireland. The capitals of Wales and Scotland are Cardiff and Edinburgh, respectively. Belfast is the capital of Northern Ireland. London is also the capital of the United Kingdom and it has a population of 8 million people. English is the official language of England, while in Wales both English and Welsh are official languages.”
- Then use the Linked Open Vocabularies (LOV) website to try to find suitable vocabularies for the predicates/properties of the graph.



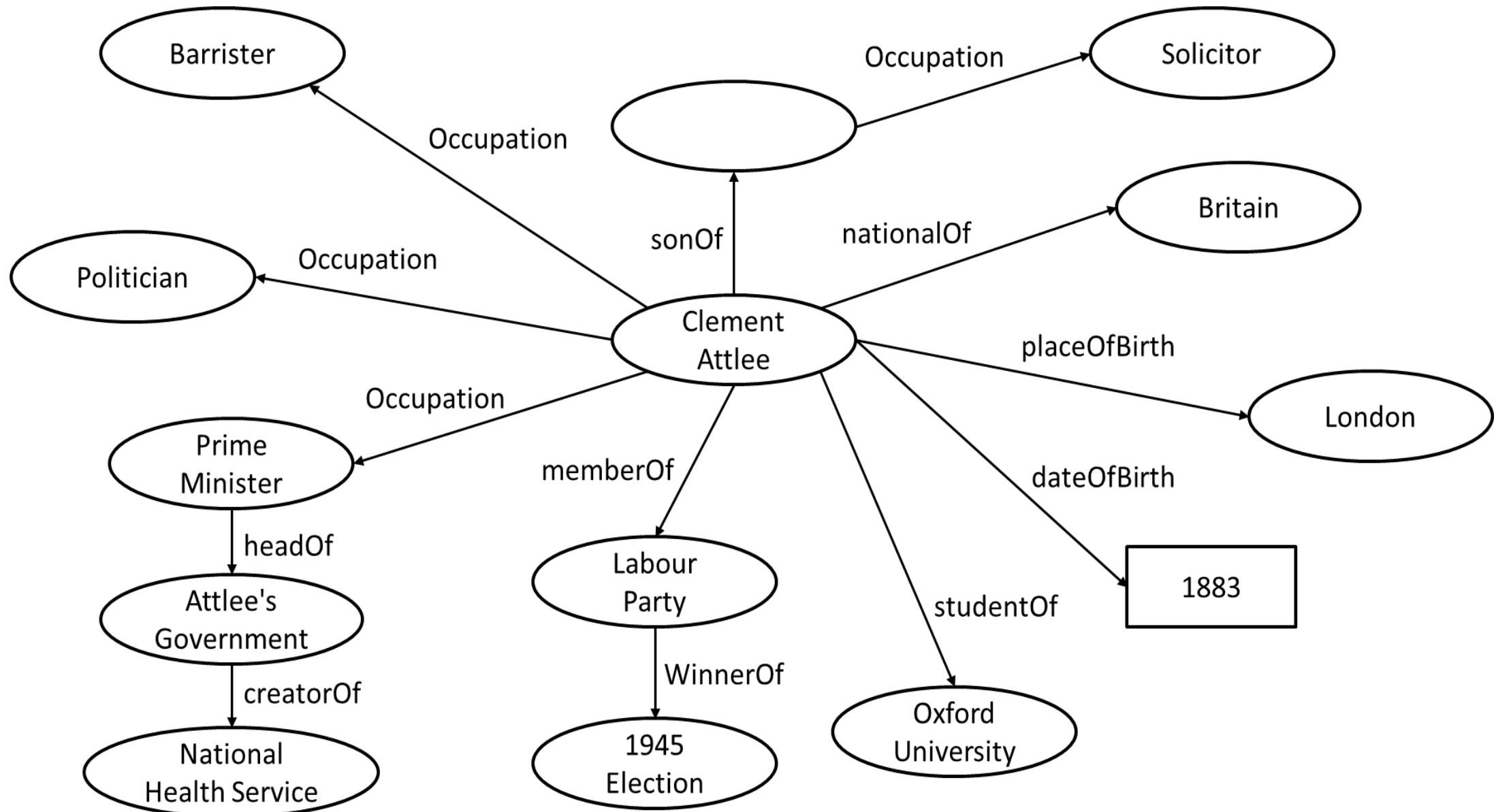
- Linked Open Vocabularies (LOV) website: find suitable vocabularies for the predicates/properties of the graph. <https://lov.linkeddata.es/dataset/lov>

- Draw an RDF diagram to represent the concepts in the following statement:
- “Clement Attlee was a British politician. He was born in London in 1883. His father was a solicitor and, after studying at Oxford University, Attlee became a barrister. He became Prime Minister when the Labour Party won the 1945 election. Attlee’s government created the National Health Service.”

Solicitor: A British lawyer who gives legal advice and prepares legal documents

Barrister: A British lawyer who speaks in the higher courts of law

- Then use the Linked Open Vocabularies (LOV) website to try to find suitable vocabularies for the predicates/properties of the graph.



Web Services and Web Data

XJCO3011



Session 17 - Querying Linked Data with SPARQL

SPARQL (*SPARQL Protocol and RDF Query Language*)

- SPARQL is a **recursive acronym** for *SPARQL Protocol and RDF Query Language*
- SPARQL is to RDF data as SQL is to relational databases. SPARQL is the **query language** for structured data on the Web, i.e., data accessible in RDF formats or representable as such.
- With SPARQL, we can query the Web of Data as if it were a **database—a big, highly distributed database on the internet**. SPARQL can query local files containing RDF data, or RDF files accessible on the Web. It is also able to query multiple data sources at once and thus **dynamically build a large, virtual RDF graph** from those multiple data sources.
- Because many people are already familiar with SQL, SPARQL was designed to look and act as much like SQL as possible, even though the **traditional relational data model differs significantly from the graph data model of RDF**.
- Like SQL, SPARQL is based on a widely implemented standard, but various vendors have extended the language.



Querying Flat RDF Files with SPARQL

- The select query in the following listing looks for people that the owner of a FOAF document knows.
- This query will return some number of people and their URLs.
- Anyone listed in the FOAF file **with an rdfs:seeAlso URL and a foaf:name** will be returned.

Listing 5.2 SPARQL query to find FOAF friends

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?name ?url
where {
    ?person rdfs:seeAlso ?url ;
              foaf:name ?name .
}
```

Names information to return in the results

Namespace declarations

Triple patterns used to match RDF statements

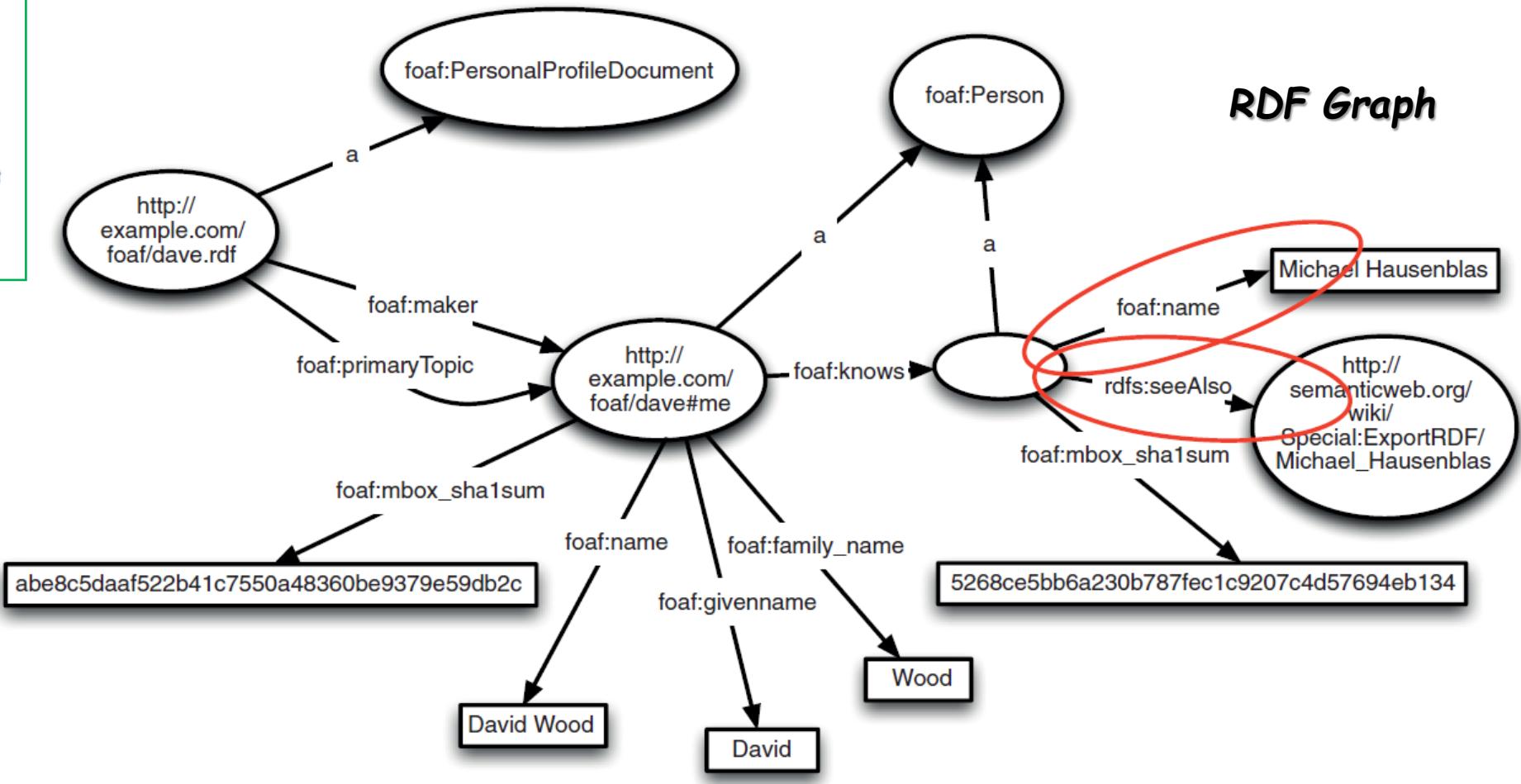
Defines patterns to match and filters to perform on the data

Select Query in Action

Query

```
select ?name ?url  
where {  
  ?person rdfs:seeAlso ?url ;  
    foaf:name ?name .  
}
```

RDF Graph



Result

name	url
"Michael Hausenblas"	<http://semanticweb.org/wiki/Special:ExportRDF/Michael_Hausenblas>

Querying Multiple RDF Files

- Unlike SQL, SPARQL isn't limited to querying a single data source. You can use SPARQL to query multiple files, web resources, databases, or a combination thereof.
- For example, we can see the personal information in a FOAF profile can be extended with address information (which can be represented in RDF via the vCard vocabulary).
- This shows that RDF files may be combined, just like any other RDF graphs. Graphs of information combine well (unlike tables and trees). The magic is in the reuse of identifiers. Both files refer to the same URI identifying a person.

Listing 5.6 A SPARQL query that combines FOAF and vCard data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix vcard: <http://www.w3.org/2006/vcard/ns#>

SELECT ?name ?city ?state
where {
  ?person foaf:name ?name ;
           vcard:adr ?address .
  ?address vcard:locality ?city ;
           vcard:region ?state .
}
```

The SELECT clause, showing three variable bindings to be returned in the results

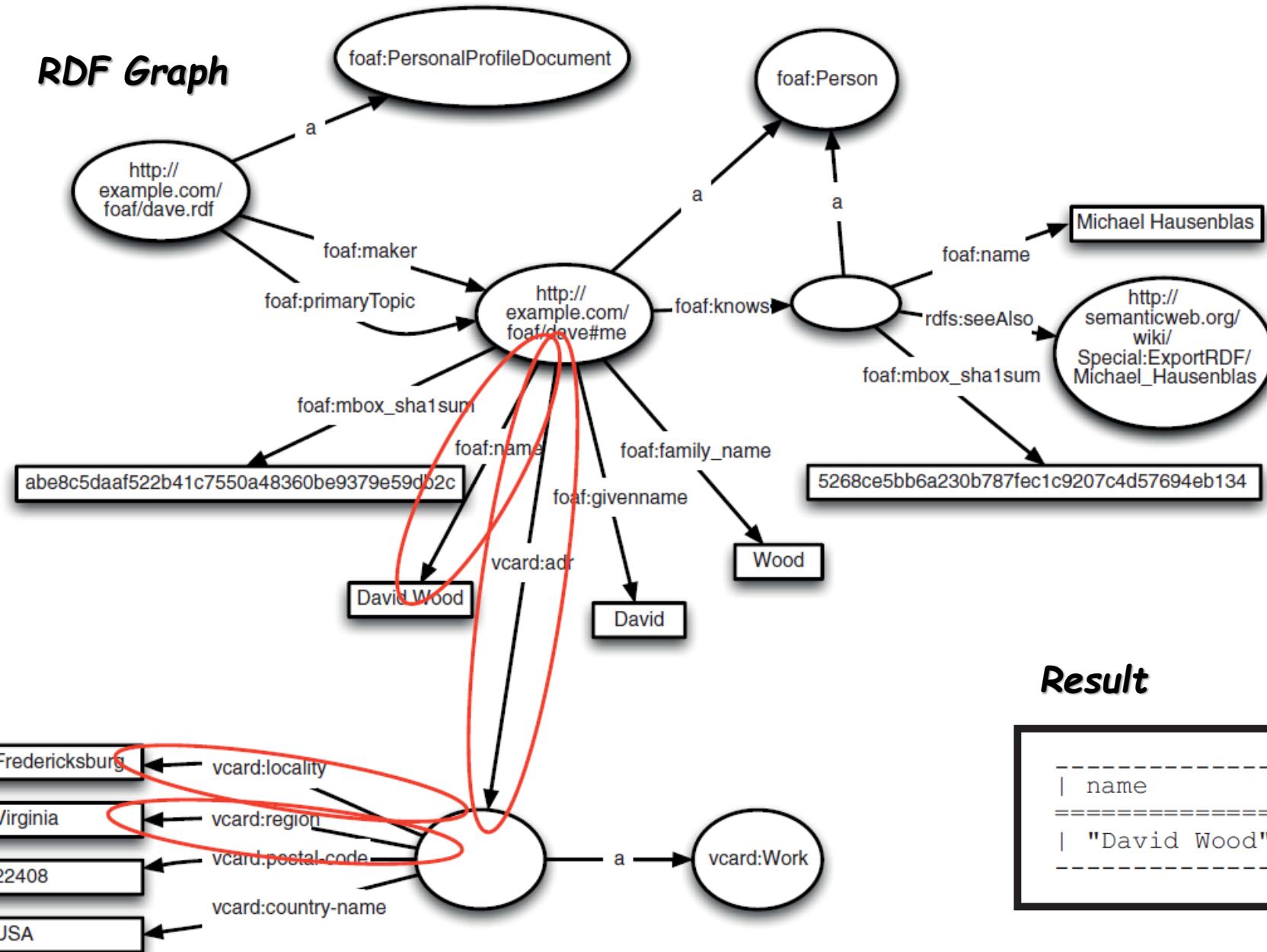
Triple patterns mapping the address to a city and state

A triple pattern to find the blank node representing an address

NOTE One of the primary assumptions of Linked Data is that two people using the same identifier are talking about the same thing. Reusing identifiers for resources allows data to be combined.

Querying Multiple RDF Files in Action

RDF Graph



Result

name	city	state
=====		
"David Wood"	"Fredericksburg"	"Virginia"

SQL vs SPARQL

Developers used to SQL might note that variable names in SPARQL's SELECT clause don't name variables to query from the database; they determine **which variables used in the WHERE clause's triple patterns get returned in the output**. That's confusing for some new users, but it makes sense once you wrap your mind around the concept of matching triple patterns against an RDF graph.

Listing 5.6 A SPARQL query that combines FOAF and vCard data

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix vcard: <http://www.w3.org/2006/vcard/ns#>

SELECT ?name ?city ?state
where {
    ?person foaf:name ?name ;
              vcard:adr ?address .
    ?address vcard:locality ?city ;
              vcard:region ?state .
}
```

The SELECT clause, showing three variable bindings to be returned in the results

A triple pattern to find the blank node representing an address

Triple patterns mapping the address to a city and state

Example of Querying Multiple RDF Files

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
```

Namespace prefixes.

```
select ?name ?latitude ?longitude      ← Requesting three fields be retrieved.
from <http://3roundstones.com/dave/me.rdf>
from <http://semanticweb.org/wiki/Special:ExportRDF/Michael_Hausenblas>
where {
    ?person foaf:name ?name ;
              foaf:based_near ?near .
    ?near pos:lat ?latitude ;
           pos:long ?longitude .
}
LIMIT 10
```

Criteria described in the form of a triple pattern.

Items preceded by ? represent variables in the results.

Results will be retrieved from two sources.¹

Only the first 10 results will be returned.

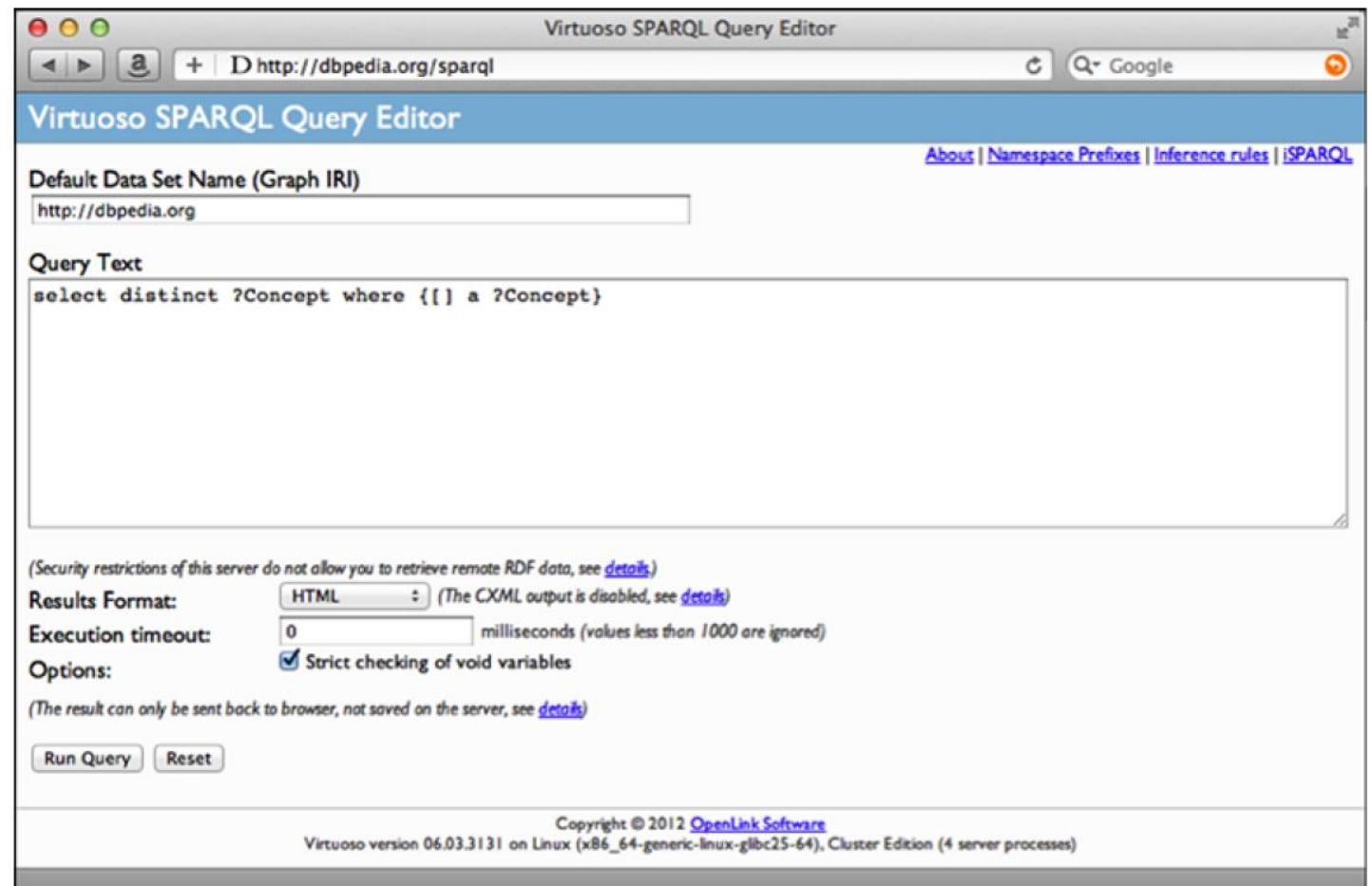
Querying an RDF file on the Web

- This can be done by adding a FROM clause after the SELECT clause, e.g.
`FROM http://3roundstones.com/dave/me.rdf`
- One of the things that makes structured data on the Web interesting is that it's distributed, unlike a relational database where the data exists in a single system.
- SPARQL allows you to have multiple FROM clauses in a single query.
- You can add multiple FROM clauses with different URLs to a query

Querying SPARQL Endpoints

- Linked Data sites on the Web often expose a SPARQL endpoint.
- A SPARQL endpoint is a web-accessible query service that accepts the SPARQL query language.
- An **HTTP GET on a SPARQL endpoint generally returns an HTML query form.**

NOTE As is true in Turtle, the syntactical convenience `a`, when used as a property, is the same as saying `rdf:type`, which is used to say that an RDF resource is an instance of a particular RDF class. The term `[]` is a blank node and will therefore match any subject.



SPARQL Endpoints

- The growing convention used by datasets on the Linked Open Data cloud is exposing SPARQL endpoints on the path `/sparql`. You can generally determine whether a given Linked Data site has a SPARQL endpoint by constructing a URL like `http://{hostname here}/sparql`. This is just a convention, but it's a useful one. Of course, you can put a SPARQL endpoint on any URL.
- DBpedia's default query gives a hint to new users on how they can discover what information the service holds. You can rewrite DBpedia's default query with more whitespace to make it more readable, as shown in the following listing.

Listing 5.8 Query the `rdf:types` a server holds

```
select DISTINCT ?Concept
WHERE {
  [] a ?Concept
}
```



The **DISTINCT** keyword ensures that duplicate results are filtered out; only unique matching results are returned.

That's it Folks



Further Reading

Chapter 5: Linked Data Structured Data on the Web, David Wood et al.

Web Services and Web Data

XJC03011



Session 19 -SPARQL Examples

Querying Flat RDF Files with SPARQL

- This query will return some number of people and their URLs.
- Anyone listed in the FOAF file **with an rdfs:seeAlso URL and a foaf:name** will be returned.

Listing 5.2 SPARQL query to find FOAF friends

**Names information
to return in the
results**

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix foaf: <http://xmlns.com/foaf/0.1/>
```

**Namespace
declarations**

**Triple patterns
used to match
RDF statements**

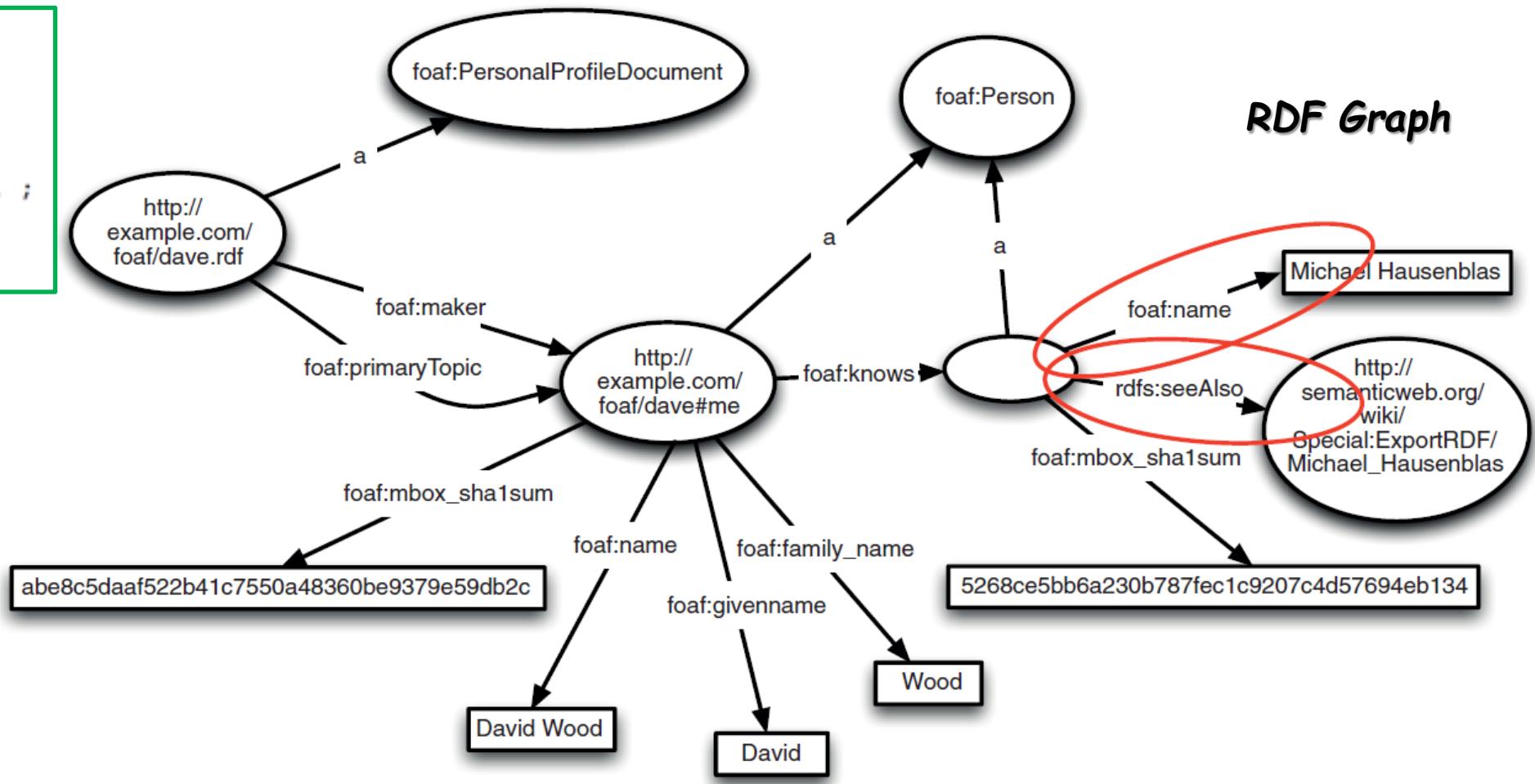
```
select ?name ?url
where {
    ?person rdfs:seeAlso ?url ;
            foaf:name ?name .
}
```

**Defines patterns to
match and filters to
perform on the data**

Select Query in Action

Query

```
select ?name ?url  
where {  
  ?person rdfs:seeAlso ?url ;  
    foaf:name ?name .  
}
```



Result

name	url	
"Michael Hausenblas"	http://semanticweb.org/wiki/Special:ExportRDF/Michael_Hausenblas	

We will use SPARQL Explorer for DBpedia
<http://dbpedia.org/snorql/>

Example 1

Find any entity which has a 'foaf:name' predicate

```
SELECT ?entity WHERE {  
    ?entity foaf:name ?name  
}
```

Example 2

Find in DBpedia any 'subject'(entity) related to Leeds

```
SELECT ?sub WHERE {  
    ?sub ?pred :Leeds.  
}
```

Example 3

Find in DBpedia any 'subject' related to "Leeds"

```
SELECT ?sub WHERE {  
    ?sub ?pred "Leeds"@en.  
}
```

Example 4

Find the name of any person born in Leeds

```
SELECT ?name ?entity WHERE {  
    ?entity foaf:name ?name.  
    ?entity dbo:birthPlace :Leeds.  
}
```

Example 5

Find the name and spouse of any person born in Leeds

```
SELECT ?name ?spouse WHERE {  
    ?entity foaf:name ?name.  
    ?entity dbo:birthPlace :Leeds.  
    ?entity dbp:spouse ?spouse.  
}
```

Example 6

Find any female person born in Leeds before 1/1/1900

```
SELECT ?name ?birth ?entity WHERE {  
    ?entity foaf:name ?name.  
    ?entity dbo:birthPlace dbr:Leeds.  
    ?entity foaf:gender "female"@en.  
    ?entity dbo:birthDate ?birth .  
    FILTER (?birth < "1900-01-01"^^xsd:date).  
}
```

Example 7

People who were born in Berlin ordered by name

```
SELECT ?person
WHERE
{
    ?person dbo:birthPlace dbr:Berlin .
    ?person foaf:name ?name .
} ORDER BY ?name
```

Example 8

The capital city of France

```
SELECT ?city {  
    :France dbo:capital ?city.  
}
```

Example 9

The capital cities of the world

```
SELECT ?city ?country {  
    ?country dbo:capital ?city.  
}
```

For you to do

- The population of France
- The population of world countries
- The depiction (picture) of any male person born in Leeds