Designing RESTful APIs
(Application Programming Interfaces)
Neal Audenaert
Welcome to DHSI 2012!

Thanks for joining the DHSI community this year – our 10\textsuperscript{th} year, but our 11\textsuperscript{th} offering!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered on the Sunday afternoon, and received your login information), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes – your instructors will let you know – or that the most current version of all DHSI-related information may be found on our website at dhsi.org. Do check in there first if you need anything that’s not in this coursepak.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
Daily Schedule

Sunday, 3 June 2012 [DHSI Registration]

DHSI Registration
At UVic Housing / Residence Services Office (Craigdarroch Building)
See the University of Victoria @ Google Maps

After registration, many will wander to Cadboro Bay and the pub at Smuggler's Cove.

Monday, 4 June 2012

8:00 to 8:30
Last-minute Registration
MacLaurin Building, Room A100
See the University of Victoria @ Google Maps

8:30 to 9:30
Welcome
MacLaurin A144

Classes in Session (Class locations for the week are as listed below)

1. Text Encoding Fundamentals and their Application (Clearihue A102)
2. Digitisation Fundamentals and their Application (Clearihue A015)
3. Introduction to XSLT for Digital Humanists (Clearihue A103)
5. Geographical Information Systems in the Digital Humanities (Human and Social Development A170)
6. Physical Computing and Desktop Fabrication for Humanists (MacLaurin D016)
7. Digital Pedagogy in the Humanities (MacLaurin D110)
8. Creating Digital Humanities Projects for the Mobile Environment (Human and Social Development A270)
9. Designing RESTful APIs (Application Programming Interfaces) (MacLaurin D115)
10. Digital Humanities Databases (MacLaurin D114)
11. Augmented Reality: An Introduction (MacLaurin D109)
12. Issues in Large Project Planning and Management (Hickman 120)
13. Digital Editions (Clearihue A012)
14. Out-of-the-Box Text Analysis for the Digital Humanities (Clearihue A105)
15. Understanding the Pre-Digital Book (McPherson Library A003, A130)
16. Online Tools for Literary Analysis (MacLaurin D010 (M-W), D111 (Th-F))
17. SEASR Analytics (MacLaurin D107)

Noon to 1:15
Lunch break / Unconference
MacLaurin A144
(Unconference discussions through the week are coordinated by Deb Raftus; discussion topics, scheduling, and room assignments from among all DHSI rooms will be handled at this meeting)

1:15 to 3:50
Classes in Session

4:00 to 5:00
Institute Lecture: Laura Mandel (Texas A&M)
MacLaurin A144

5:00 to 6:00
Light Reception
University Club

Tuesday, 5 June 2012

DHSI Colloquium: Textual Analysis
MacLaurin A144


8:00 to 9:20
Classes in Session

9:30 to Noon
Classes in Session
Wednesday, 6 June 2012

8:00 to 9:20
DHSI Colloquium: Digital Pedagogy  
MacLaurin A144  
Almond Aguila, "The Pedagogy of Facebook"; Kathi Inman Berens, "Failure is Frictive: Coding and Pedagogy"; Eugenie Duthoit, "Re-thinking the Use of Digital Tools to Assist the Pedagogical Translation from Latin"; Chris Friend, "Bringing Technology to Student Writing: How DH Practices Can Enhance Composition Pedagogy"; Peggy Jubien, "Reexamining Our Tools: Linking Educational Technology to the Socio-Political Dimensions"

9:30 to Noon
Classes in Session

Noon to 1:15
Lunch Discussion: James Cummings (Oxford), Elizabeth Burr (Leipzig), Jennifer Guiliano (Maryland), Rebecca Niles (Toronto), Sebastian Rahtz (Oxford), and Ray Siemens (Victoria), "Training Institutes in the Digital Humanities"  
MacLaurin A144

1:15 to 3:50
Classes in Session

4:00 to 5:30/6.00
DHSI Colloquium: Archives and Databases  
MacLaurin A144  
Orhan Elmaz, "How and Why to Create a Frequency Dictionary of Media Arabic"; Christopher Laxer, "Designing a Literary Labels Database"; Mike Nutt and Markus West, "Omeka and MicroTiles: Building Library Exhibits for Enormous Displays"; Anne Salsich, "The Archive and Digital Humanities: 'Shansi: Oberlin and Asia'"; Charles Shirley, "Can Putting Troilus and Cressyde into a Database Aid Critical Study?"; PANEL of Tara Thomson, J. Matthew Haculak, Katie Tanigawa and Stephen Ross, "The Modernist Versions Project"

Thursday, 7 June 2012

8:00 to 9:20
DHSI Colloquium: Gaming, Gamification, and Media Studies  
MacLaurin A144  

9:30 to Noon
Classes in Session

Noon to 1:15
Lunch break / Unconference, various locations  
Lunch Discussion: Andrew Stauffer (U Virginia) and Laura Mandell (Texas A&M), "Peer-Review, Publication, and the Academic Evaluation of Digital Scholarship: An Open Discussion"  
MacLaurin A144

1:15 to 3:50
Classes in Session

4:00 to 5:30/6.00
DHSI Colloquium: Mapping and Visualization  
MacLaurin A144  

Friday, 8 June 2012 [DHSI + Beyond Accessibility]

8:00 to 9:20
DHSI Colloquium: The Way Forward
### Beyond Accessibility: Textual Studies in the 21st Century

**Saturday, 9 June 2012**

**[Beyond Accessibility]**

**9.00-12.45**

- **Beyond Accessibility: Textual Studies in the 21st Century**
  - Please visit the conference website
  - Room 1 (MacLaurin D101)
  - Room 2 (MacLaurin D103)
  - Plenary Session (Hickman 105)
  - Breakout / Meeting Room (Hickman 120)

**6.30-8.30**

- **TBA, Reception / Dinner**

**Sunday, 10 June 2012**

**[Beyond Accessibility]**

**9.00-12.45**

- **Beyond Accessibility: Textual Studies in the 21st Century**
  - Please visit the conference website
  - Room 1 (MacLaurin D101)
  - Room 2 (MacLaurin D103)
  - Plenary Session (Hickman 105)
  - Breakout / Meeting Room (Hickman 120)
Connect to UVic: Windows 7 and Vista

NEW UVic wireless configuration utility

The UVic wireless configuration utility will automatically configure the "UVic" wireless network on your Windows XP SP3, Windows Vista, or Windows 7 computer.

Download now

Note: The UVic wireless configuration utility is still experimental; use this application at your own risk. UVic is not responsible for any damage caused by the use of the wireless configuration utility. Please report any problems to the Computer Help Desk.

If the above doesn’t work, please follow the manual instructions listed below. After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are on campus.

1. Before you start this procedure, ensure the following:
   - Your wireless card and its drivers have been installed and you have rebooted your laptop since the installation.
   - Your laptop is powered on and booted up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.
   - You are using Windows to manage your wireless connections. If you are using a third-party application (sometimes network adaptors come with their own applications), you may experience problems during the configuration process.

2. Temporarily connect to the Internet using UVic Open, an Ethernet port, or your home network. Download the security certificate by right clicking thawte Primary Root CA and saving the thawte.cer file to your computer. Once the file is saved to your computer, locate the file, double click on it, select Install Certificate..., and follow the Certificate Import Wizard instructions.
3. Once you have successfully installed the certificate, open your **Start** menu and click on **Control Panel**.

4. Click on **Network and Internet** or **Network and Sharing Center**.
5. Click on **Network and Sharing Center**.

6. Click on **Manage wireless networks**, located on the left menu.

7. Click **Add**.
8. Click **Manually create a network profile**.

9. Enter the following information:
   - Network name: **UVic** (case sensitive).
   - Security type: select **WPA2-Enterprise**.
   - Encryption type: automatically sets to **AES**.
   - Security Key/Passphrase: (leave blank).
Ensure both checkboxes are selected (by default, the second box is not). Click Next.

10. Click Change connection settings. For now, ignore the pop-up window in the bottom-right corner.

On the Connection tab, ensure the Connect to a more preferred network if available checkbox is not checked.

11. Click the Security tab. Ensure the authentication method is PEAP. Then click Settings.
12. Check the box beside *thawte Primary Root CA* in the list of *Trusted Root Certification Authorities*.

If you cannot find the correct certificate listed, please return to step 2 to download the certificate.

At the bottom of the dialogue, ensure that the **Authentication Method** is **Secured**.
13. Deselect the checkbox for **Automatically use my Windows logon**... and click **OK**.

![Configure](image)

14. Close the remaining windows. In the bottom-right corner of your screen, you should see a small window pop-up informing you that **Additional information is required to connect to UVic**. Click on it to provide additional information.

![Additional information](image)

15. Enter your personal **NetLink ID** followed by **@uvic.ca** in the **User name** field, and your **NetLink ID password** in the **Password** field. Click **OK**.

![Login](image)

You should now be connected to the **UVic** secure wireless network.
Connect to UVic: Mac OS X 10.5 and newer

After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are using UVic's wireless network.

1. Before you start this procedure, ensure the following:
   - Your wireless card and its drivers have been installed and you have rebooted your laptop since the installation.
   - Your laptop is powered on and booting up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.

2. At the top-right corner of your screen there should be the Airport icon (a semi-circle). If you do not see this icon, your AirPort card or AirPort software may not have been installed properly.

3. Click on the AirPort icon (it may be partially darkened) to reveal a menu. Ensure your AirPort is On.

4. Scroll down the AirPort menu and select Join Other Network ...

5. In the window that opens, enter the following information:
   - Network Name: UVic (case sensitive)
   - Security: WPA2-Enterprise
   - User Name: your NetLink ID
   - Password: your NetLink ID password
   - 802.1X: Automatic

Click Join.
6. If you see a message about Mac OS X wanting to access your Keychain, click **Always Allow**.

7. A **Verify Certificate** window will open saying that the certificate is not trusted.
   - Click **Show Certificate**.
   - Check the box that says **Always trust "sac1cled050..."** (the exact name may vary) and click **Continue**.
   - If you are prompted for your computer password, enter it and click **OK**.
You should now be connected to the UVic secure wireless network. To disconnect from the wireless network, click on the AirPort icon and click Turn Airport Off. Next time you connect to UVic, you should not need to enter any additional credentials.
Connect to UVic: iPhone or iPod Touch

Before starting, ensure:

- Your device is using firmware version 4.0 or higher.
- Your device is powered on and booted up.
- You are in an area with wireless coverage.
- You have a NetLink ID and password.

1. From the Home screen, press the Settings button.  
   2. Press the Wi-Fi option.
   3. Under the Choose a Network... heading, select UVic.

After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are using UVic’s wireless network.
5. Enter your personal NetLink ID followed by @uvic.ca in the Username field. Enter your NetLink ID password in the Password field. Press Join.

6. If prompted, press Accept to verify the thawte Primary Root CA certificate.
Your device should now be connected to the **UVic** secure wireless network.
Designing RESTful APIs

Neal Audenaert
Institute for Digital Christian Heritage
neal@idch.org

prepared for
Digital Humanities Summer Institute
University of Victoria
June 4-8, 2012

date: 18 April 2012
About the Instructor:
Neal Audenaert is passionate about working with humanities scholars to design and build software that helps them work better and share their work with others. Neal received his Ph.D. from Texas A&M University in 2011 where he was an active participant in several Digital Humanities projects including the Cervantes Project, the Nautical Archaeology Digital Library, Digital Donne and the Picasso Project. His dissertation research focused on supporting creativity and knowledge formation during the early stages of humanities research. Neal is broadly interested in how people gather, organize and use information in the course of their work and designing interactive applications that support them.

In 2009 he founded the Institute for Digital Christian Heritage (IDCH) to provide technical consulting and software development support to researchers who find it difficult to obtain this support at their host institutions. IDCH focuses religiously oriented projects while its counterpart, Digital Archives, Research & Technology Services (DARTS), supports DH projects in general.

Neal lives with his wife Michelle in Bryan, TX. Their daughter, Amelia Ruth, was born March 2, 2012. They enjoy traveling, gardening and cooking.

Course Objectives
This course will provide an introduction to the fundamental theory and practice of designing distributed systems for the Web using the REST (Representational State Transfer) architectural style. It is intended for humanities scholars who will act as domain experts, project managers or decision makers on large medium to scale projects. After completing the course, participants should be able to make informed decisions about whether investing in a REST API is appropriate for their project and to engage actively in designing and evaluating an API to ensure that it meets the scholarly needs of their project.
# Contents

What is REST: An Overview.......................................................................................................................... 5

Motivations for building a RESTful API ........................................................................................................... 7

Key Features of the REST Architectural Style.................................................................................................. 8

Resources ......................................................................................................................................................... 8

Representations of Resources ........................................................................................................................... 9

Addressability .................................................................................................................................................. 10

Connectedness ............................................................................................................................................... 12

Statelessness ................................................................................................................................................... 13

Summary ............................................................................................................................................................ 14

Understanding HTTP ....................................................................................................................................... 15

Anatomy of HTTP Requests and Responses .................................................................................................... 15

HTTP Methods ............................................................................................................................................... 17

HTTP Headers ............................................................................................................................................... 18

HTTP Response Codes ..................................................................................................................................... 20

Representations ............................................................................................................................................... 21

JSON ............................................................................................................................................................... 21

XML ................................................................................................................................................................. 23

Designing RESTful Architectures ..................................................................................................................... 24

Decision Process ............................................................................................................................................. 26

Requirements Design ...................................................................................................................................... 26

Designing Resources ...................................................................................................................................... 28

Figure Out the Data Set .................................................................................................................................. 28

Split the data into resources .............................................................................................................................. 28

Name the resources with URIs .......................................................................................................................... 29

Expose a subset of the uniform interface .......................................................................................................... 30

Design the representation(s) served to the client ............................................................................................ 30

Design the representation(s) accepted from the client .................................................................................... 32

Integrate this resource into existing resources ................................................................................................ 33

Consider the typical course of events: what is supposed to happen? .............................................................. 34

Consider error conditions: what might go wrong? ........................................................................................... 34

Summary ............................................................................................................................................................ 34
What is REST: An Overview

Over the past two decades or so, the Web has emerged from military and academic research labs to become a ubiquitous and powerful tool to publish information. Global access to a network of information on a scale that was nearly unimaginable when I was growing up (not all that long ago) is taken for granted by students entering college today. Beyond providing a means to disseminate information, the so-called Web 2.0 acts as a vehicle to connect people and promote individual engagement (for better or worse) with that content. Social media allows ideas and memes to move from obscurity to global prominence in the span of days.

We routinely interact with information on the Web directly, reading documents, viewing images, watching movies and listening to music. The information that we consume is lightly structured: easy for people to understand and use, but difficult for computers to manipulate beyond basic support for displaying and linking content.

Increasingly, there is a focus on creating a programmable web. On the programmable web, content is published in ways that enable programs, in addition or instead of human user, to access, understand, use, reshape, and combine information from different sources in order to create new resources. Innovators can design new ways to present information, develop algorithms for analyzing data, or build services to help people interact in new ways because they are able to write software that interacts with the content of the Web in a formal, structured manner.

Modern software applications are large and complex systems. They provide lots of features, interact with rich data sources and operate in complex environments. No one person can write one of these programs from scratch. Fortunately, no one has to. Programs are divided into sub components (commonly called libraries) that can be built by separate teams of developers and shared across a wide variety of applications. These libraries may themselves be quite large and complex and may in turn depend on other libraries, however, they typically hide this complexity behind a well-defined, organized set of functions called an application programming interface, or API. A programming language typically contains a set of core APIs and different software development projects will produce additional modules to support specific needs. Examples of APIs include packages for working with databases, communicating with a file system or over a network, image processing, managing an address book, etc.

An API simply encapsulates some core, commonly needed functionality and makes it available to other applications. As software systems grow, it is often helpful to write programs that run on different computers. These distributed systems allow an application to take advantage of data, physical resources like servers, complex algorithms and other services without the need to download, install, or integrate different software packages. Distributed applications take the principles of software reuse and data sharing to a new level.

The Internet provides one standard networking system (among many different options) for applications to use to communicate between different machines. The World Wide Web provides the architectural framework to passing messages between machines (HTTP) and for uniquely identifying resources on a
global network (URI). These technologies that power the human web, also form a substrate that we can use to write distributed programs.

The key question that REST addresses is how those programs should be organized. Many of the applications that have been designed to run on the web adopted a traditional programming model based on invoking blocks of code called methods or procedures. In this approach, the Web is simply a framework that is used to pass messages about what procedures to call and what data to use with those procedures. The REST architectural style adopts a very different approach that I’ll call a Resource Oriented Architecture. In this software design style, instead of defining different methods, applications describe a set of resources and use methods from a very small “uniform interface” to access and manipulate those resources.

REST is the very foundation of how the Web was designed to operate. Consequently, building RESTful applications for the Web tends to be easier than trying to build a corresponding traditional application that simply uses the web to pass message between machines. The term REST, which stands for Representational State Transfer, was first used to describe the architecture of the Web (or rather “of how the World Wide Web should work”) by Roy Fielding in his dissertation and related conference and journal articles. Other works by Fielding include the HTTP/1.1 and the URI standards, so he has a pretty good idea of what he’s talking about.

This class will focus on how to design these applications. We won’t get into the details of how to implement specific components (that would require significant programming skills and that everyone in the class could use the same programming language). Instead, we’ll focus on the fundamentals of how RESTful APIs work, what they are useful for, and how to translate the domain specific needs of a digital humanities project into a well-designed API. My goal is for you to be able to make informed decisions about when a project you are involved in should make the investment needed to create a RESTful API and to be able to work closely with software developers to design an API that really meets the specific scholarly needs of your project as well as to offer constructive criticism on APIs that others have built.

Rather unsurprisingly, a lot of information about the fundamental architectural style of the Web, and about how to build resources using that style, is available on the Web. Most of these are adequate and I’ve included references to a number of links in the Resources appendix and throughout the text of this course packet. Wikipedia’s usually a good place to start. Google is a good second option.

In general, I’ve found that the available web resources are often good, but typically present only a small portion of the overall story and vary significantly in quality. While there are often definitive Web-based resources for programming languages or other technologies, I haven’t come across a great web site that I can point you to that neatly summaries the fundamentals of building RESTful Web APIs.

---

1 This isn’t my term, lots of people have used it. But lots of people use lots of different terms to describe a range of overlapping concepts. Resource Oriented Architecture is the one that I think fits best.
2 Fielding encouraged people to cite his dissertation rather than the conference and/or journal papers because his dissertation was available via the Web for free (see his blog post [http://roy.gbiv.com/untangled/2010/icse-most-influential-paper-award](http://roy.gbiv.com/untangled/2010/icse-most-influential-paper-award) for an interesting story about this).
It is nice to have a systematic treatment of a topic to make sure that I’ve understood the core ideas well and that I have them organized in the right place in my mind. To that end, I’ve used three books in pulling this class together, most notably, *RESTful Web Services* by Leonard Richardson and Sam Ruby. Much of this course material is derived to some extent from their work (although, very little of it is directly from them). The downside here is that the book is geared toward programmers, whereas my target audience for this class isn’t developers *per se*, but domain experts who will be working with programmers to develop APIs and project managers who need to make informed decisions about whether an API is the right investment for their projects.

Finally, a sizable portion of this coursepak consists of editorializing. I’ve been fairly free in terms of offering my thoughts, perspectives and recommendations. These are offered, not as firm and dogmatic rules but rather as thoughtful and educated, but possibly flawed, reflection on designing RESTful systems in the context of Digital Humanities projects. They are offered as such and my hope is that they are useful starting points for conversation. I am very much looking forward to hearing your thoughts and ideas during the course and refining the editorial and advice portions of this coursepak.

**Motivations for building a RESTful API**

The primary reason to create a RESTful API is because you or your institution has some set of resources that it would like to make broadly available for others to use. Those resources might include unique data collections such as digitized special collections, a library of TEI encoded material, an authoritative catalog of historical people, place and events, or computational tools like image processing, mapping tools, or natural language processing. The range of resources that might make for an interesting and useful API is nearly limitless. Many (but not all) projects that are valuable for people to access, are also valuable for programs to access.

Below is a list of five big advantages/motivations for building an API in general and a RESTful API in particular. These are offered in no particular order and are by no means exhaustive.

**Encapsulate a complex system:** One of the most important uses of an API is to encapsulate the implementation details of a complex system and provide a simple interface to access that system. Google Maps, for example, is a fairly complex system for managing a massive, deep-zoom mapping system, geospatial data, markers, route-finding algorithms and other services. It requires significant computational resources and complex sub-systems. Despite this complexity, it is possible to learn the API (Google Maps uses a JavaScript API rather than a REST API) and incorporate maps into your own web pages within a day. Software complexity isn’t the only factor here. Many systems require resource intensive computational support that may be expensive and difficult to configure. An API can allow users to access these advanced resources remotely from Web or desktop applications.

**Allow multiple implementations:** Often, there are multiple different ways to solve the same problem. These different implementations might use different programming languages; implement algorithms tailored to different scenarios; take advantage of special hardware like OpenGL for graphics, if present; or interface with different legacy systems. An API presents a general way to solve a class of problems that hides the details of specific implementations. This also eases long-term maintenance of an
application and promotes stability since developers can change the implementation as needed without breaking third party applications that rely on the API.

**Support multiple presentations:** APIs allow for a separation of concerns. One team, for example, can focus on its area of expertise in providing access to widely useful data, while other teams can develop interactive mobile apps, data visualization tools that run on a desktop, or a web site for public education. The ability to build a system component by component allows different groups of stakeholders to invest in creating value that meets their specific needs while leveraging the work of others and making their work available in turn.

**Integrate disparate resources** At times, a large project will require data and/or other components that cannot be provided by a single institution. In the field of New Testament textual studies, for instance, New Testament manuscripts are held by many different institutions who may not be willing to commit their digital facsimiles to a common repository. By implementing a common API, these different institutions can each retain ownership and control over their collections, while enabling applications to be used seamlessly across different institutions. Aggregator applications can (assuming they have permission to do so) present a unified interface over these diverse resources. The Open Archives Initiative’s Protocol for Metadata Harvesting (OAI-PMH) and the Atom syndication format are two widely adopted mechanism for doing this.

**Centralize access to a common resource** On the other hand, there is often much to be gained by providing a centralized service that can be accessed by many different client applications. Websites like Facebook, Twitter, and Del.icio.us rely on bringing the personal collections of many different users together into a single location in order to facilitate social connections and to leverage the so-called “wisdom of the crowd”. These services provide APIs that make their centralized resources available to a wide range of end-user applications. Zotero’s new Web API provides a similar service to support notetaking and building scholarly bibliographic collections.

**Key Features of the REST Architectural Style**

**Resources**

Resources are at the heart of a RESTful API. A resource is, very simply, something you can reference. In the words of the W3C:

> We do not limit the scope of what might be a resource. The term "resource" is used in a general sense for whatever might be identified by a URI. It is conventional on the hypertext Web to describe Web pages, images, product catalogs, etc. as “resources”. ... However, our use of the term resource is intentionally more broad. Other things, such as cars and dogs (and, if you've

---

3 Although it should be noted that there are significant technical challenges depending on the nature of services to be provided by the aggregator.


5 Atom Enabled. [http://www.atomenabled.org](http://www.atomenabled.org) [April 2012]
Let’s be a bit more concrete. I’ve split things out into a several categories to highlight the breadth of possible resources and to help you start thinking about how you might organize some domain into resources. This isn’t a standard vocabulary and my divisions aren’t authoritative, complete, or distinct. It is merely intended as a place to start thinking about how different things can be thought of as resources.

**Normal Resources:** Things like html pages, images, videos, audio, etc. These might be grouped into folders or directories for organizational purposes. This is what we traditionally think about when we think about accessing content on the web. These are often

**Dynamic Resources:** Things like the latest school closings, the current status of a flight, the most recent blog entries. These are concrete concepts, but the information they represent may change regularly or be transient (the status of my flight into UVic this week isn’t going matter much in six years).

**Business Processes:** Things like the peer-review process of a journal article or an application to include a new site in a federation like NINES can be modeled as resources. These will often have multiple sub-resources. A peer-review process might have multiple drafts of the review, an assigned editor and multiple reviewers, the individual reviews, and final and/or intermediate decisions about the paper.

**Data Resources:** Access structured data is a common use of RESTful APIs. This might include demographic data, geographical information, summaries of collections, databases, etc. This access may be parameterized by search queries, filters, aggregation, sorting, grouping, etc.

**Services:** A resource can point to a computational and/or physical service such as printing a book on demand, or performing a complex analysis of a document image. Search engines are (in my mind) a very common class of algorithmic web services. Services may be long-running resources and generate secondary resources that allow a client to monitor the current status of the resource.

**Representations of Resources**

Resources are abstract concepts. Depending on your domain, it might make perfect sense to say that a dog or a building is a resource, but you clearly aren’t going to package up a dog and send it over to the wire (not with current technology, at any rate). The “representational” part of representational state transfer means that a server sends a representation of a resource to the client.

---

6 [http://www.w3.org/TR/2004/REC-webarch-20041215/#id-resources](http://www.w3.org/TR/2004/REC-webarch-20041215/#id-resources) [April 2012]
This representation might change based on a user’s permission (an administrator might see more data that an ordinary user), the user’s preferred format (HTML vs. PDF, JSON, XML, or HTML), a language or national local, or the type of device making the request.

HTTP provides mechanisms, called header fields, to help the client and server determine the appropriate representation to send back to the client. The process of picking the best representation of data is called content negotiation (and is beyond the scope of this course). Another strategy for content negotiation is to tack the desired file format onto the end of the URI. For example, you might see /reports/2009/Q3/sales.html or .../sales.pdf. Similarly, you might tack on a language code, .../sales.en.html or .../sales.es.html.

Many people feel that extensions are in appropriate (going so far as to say you shouldn’t serve web pages that end in .html. Different URIs technically indicate different resources, not different representations of the same resource. Instead, they argue, you should use the content negotiation methods provided by HTTP. I’m a bit less strict. The advantage of using filename extensions is twofold. First (and least importantly), it’s the default behavior for most file systems, so you end up fighting a bit with your tools to remove them when you are serving files. Second, most people don’t understand the HTTP headers and/or don’t have the tools/time/patience/inclination to use them to specify the representation they want. Putting this information into the URI makes different representations accessible to a wider audience. I think the tradeoff between theoretical purity and convenience is justifiable in many situations. You can always provide an extension free version to satisfy the purists.

A brief introduction to some of the different representations can be found in the appendices.

**Addressability**

The other major requirement for resources in a RESTful application is that they are addressable. There is at least one URI pointing to each resource. That may sound rather unexciting, but the ability to uniquely reference each and every think in the system is important. As a counter example, consider the job search application at one major employer shown in figure 1. Business units may post jobs on their individual web-sites and advertise them through email and other means by providing people with the “NOV” number.
Searching for a particular job (e.g., 10527) takes you to a generic search results page who’s URI (/applicants/jsp/shared/search/SearchResults.jsp?time=1334347138405) that supplies no information about what was searched for. The link for the only job in the result (why not just take me to that job in the first place?) goes to /applicants/jsp/shared/position/JobDetails.jsp. There is no URI that references this particular job, so business units can’t provide potential employees with a link for the job they want to apply to. They have to give applicants written instructions and a NOV number to copy and paste into a search form. A link would be much easier for everyone involved.

That is inconvenient for the people involved, but on the programmable web, there is no option to provide a client program with human readable instructions to follow. This isn’t a resource that can be accessed programmatically and third parties can’t build services that enhance, extend or improve the capabilities of this system. Perhaps this employer doesn’t want other sites finding its job openings and making them visible to a wider audience, but that runs 100% counter to our purposes of building resources for the DH community that other people can build on and re-use in innovative ways.

There are some other instructive problems with the URIs that this site uses that are worth looking at. First, URI’s should be descriptive, short and easy to understand. The URI /applicants/jsp/shared/position/JobDetails.jsp fails on most of those counts. It is easy enough to understand that this points to job details for a position once you’ve spent some time looking but there is quite a bit of clutter that makes this harder to find than it should be. It clearly isn’t short. It also isn’t descriptive in the proper sense. In a URI the forward slash implies a sub-resource. The position isn’t part of an applicant; this is a representation of the position that is suitable for an applicant.

This URI also leaks information about how this resource is implemented in served. This is being processed by a Java application using Java Server Pages. One key motivation for creating an API is to hide implementation details so that you can change the details (say, use Java Server Faces, C#) without breaking the people who use your API.

---

7 It’s a bit worse than that because the search form is hidden within frames so the user has to go to the main employment page and find the search tool, but we’ll stick with the REST problems and leave the usability issues for a different course.
This URI should have been something like /positions/10527/details. That’s short, easy to understand, descriptive, and linkable. You can extend this to provide access to related resources, for example, /positions/10527/applicants/292347/resume. That would be the résumé for the applicant whose ID is 292347, but you probably figured that out.

Which leads us to another feature of URIs as they relate to resources: URIs point to exactly one resource, but multiple URIs can point to the same resource. /applicants/292347 and /positions/10527/applicants/292347 both presumably point to the same applicant.

Finally, according to the formal REST model, URIs should be opaque. That is to say, the client should not have to guess (or be told) exactly how to construct a URI. The server should provide links (somehow) that allow the client to take any appropriate actions relevant to a particular resource.

There are some good reasons for this; it makes clients easier to implement, it makes systems more robust to change, the system becomes self-documenting, etc. While these are good guidelines, I have a couple of concerns. First, if you are using well designed names people will be able to guess how to construct URIs for themselves. In the scheme above, you could probably figure out how to find job #12534. If people can figure out your API, chances are high that they will take advantage of that fact. Second, there are some types of queries that need to be constructed dynamically. User supplied search terms are one example. These queries will involve URI templates where the client needs to plug the correct value into the correct slot. And sometimes, while it might be possible to enumerate all possible URIs that a client would need, it’s just more convenient (and efficient) to let the client construct them from a template (for example, retrieving the correct tiles for a deep zoom image).

My suggestion then would be that URIs shouldn’t (really) be opaque. They should be short, understandable names for your resources. At the same time, you should strive to provide relevant links to clients whenever practical so that clients can be implemented as if the URIs were opaque. When this isn’t possible, you should provide URI templates so that clients can easily construct links without starting from scratch.

**Connectedness**
Fielding refers to this concept as Hypertext as the Engine of Application State (HATEOAS) and you’ll see that terminology widely used. Richardson and Ruby (96) suggest that this is a rather presumptuous (and intimidating) name for a very simple concept: stuff on the Web is linked together. They prefer the term connectedness.

On the human-accessible Web, links are everywhere. With the exception of those few times that you type a URL into your browser, links are how you get from one document (the application state in which your browser is displaying a particular document) to another. Links are what makes the Web, the Web.

In RESTful APIs, links play a similarly important role, but with a bit of a twist (in my mind at least). For humans, links are identified (and their destination hinted at) by the human readable anchor text or some other visible page element (the stuff you click on). That linking mechanism has a lot of built-in ambiguity and imprecision, at least in terms of in describing the content, nature and relationship of the
link. That ambiguity and imprecision is fine for humans. In fact, it is probably better than trying to be excessively precise. For the programmable Web, on the other hand, ambiguity and imprecision are rarely desirable traits. Web API clients will rarely be programmed to go off clicking around in the hopes that they might find something interesting today (at least, not the ones we are interested in here). In this case, links serve a rather different purpose.

In the programmable web, links provide a formal mechanism for connecting and structuring the intrinsic relationships within our data and for identifying the different options to retrieve further information. While it is possible to build some of this information directly into a client (for example, if a server sends a bibliographic record indicating that a document was authored by the person whose id is 42, the client could “just know” that it needs to send a GET request to /authors/42), providing explicit links to related information in the document representation makes implementing the client easier. It is also more robust to future changes since the client can be automatically upgraded to a new version of the API simply by sending a different link.

**Statelessness**

Each request between a client and a server should be independent of all previous and future requests. When a client makes a request to a server it includes all of the information the server needs to respond to that request. Consider this requirement in contrast to a file system, for example, that remembers your current working director and interprets your commands relative to that directory, or a video game that keeps track of how many bad guys you’ve shot.

As a concrete example, consider the task of navigating from one page to the next—a common requirement when filling out multi-page forms or navigating through pages of search results. One possible (and sadly common) way to get to the next result is to provide the user with a link to 
/<user_id>/application?page=next. In this case the server needs to know what page the user is currently viewing in order to deliver a different result based on that current state. Alternatively (and correctly), the server could provide a link to 
/<user_id>/application?page=2.

Another way of looking at this is to consider statelessness in terms of addressability (Richardson and Ruby 89). In RESTful systems, every bit of interesting information is addressable. In the previous example, the nextPage link included an implicit reference to information that isn’t addressable: the page that the user is currently viewing. The server must only use state information that is explicitly provided in the URI (or maybe in the message body).

What about scenarios when you need to restrict the set of possible actions based on what the user has already done? For example, what if the server needs information from page 2 in order to decide how to display page 3? This is resource state instead of application state. In RESTful applications, resources can (and should) have state that changes over time. It is perfectly reasonable for a request to fail because of the state of a resource on the server.

For example, a user may be required to fill in page two of an application before moving onto page three. In this case, if the user decides to PUT data to /<user_id>/application?page=3 without having filled in /<user_id>/application?page=2, the server can send a response of 409 Conflict to
indicate that the requested action failed because of “a conflict with the current state of the resource,” namely page 2 hasn’t been filled out.

The key here is that all of this state is addressable, you can reference it with a URI, not stored in memory in some session manager provided by your programming language of choice.

Statelessness carries some significant advantages. It makes things simple for the server. Since each request is independent, the server doesn’t have to keep connections to a client open any longer than is needed to return a response to the immediate request. Scalability is improved because the server only needs to keep track of resources for the current request/response cycle, not for all of the clients who might be currently interacting with it. Requests can be routed to different servers without the need to keep track of which client interacted with which server previously (there are some caveats here). Intermediate resources like caches and proxies can intercept and modify requests as needed without worrying about other requests that client might have made.

There are two noteworthy violations of statelessness that are common. First, many applications maintain session state on the server and supply the client with a cookie or else engage in URL re-writing in order to make sure that this state can be retrieved on subsequent requests. This might be used, for example, to keep track of an authenticated users credentials and status as he uses an application. Other uses involve storing, in memory, an object such as an application that the user is filling in. These systems violate the principles of a RESTful architecture. There are (usually) better approaches to implementing these requirements (although ease of development sometimes should trump architectural purity).

The second violation, so far as I can tell, is unavoidable. Many RESTful Web API’s track the client’s use of a resource in order to prevent excessive requests (Google, for example, restricts clients to 25,000 free requests to its Map API per day) and to bill for usage. This is not, strictly speaking, RESTful, but it is a minimal intervention and necessary, so we’ll let it slide.

Summary
That summarizes the fundamentals of a RESTful architecture. We design systems that access resources rather than procedures (blocks of code). These resources can have multiple different representations and are each uniquely addressable through a universal naming scheme (URIs). Links between resources are used to guide the client to related information and to control its flow through the different steps in a multi-step process. Together, these features allow (and REST requires) each interaction between a client and server to happen in isolation (statelessness).

There is one more principle that we haven’t addressed. That is the “universal interface”. This basically means that clients interact with resources using a universal set of operations and that all clients use the same operations in all systems. The specific operations vary in different RESTful systems, but for Web services, the universal interface is defined by the HTTP standard. That is the subject of the next chapter.
Understanding HTTP

One key principle of building RESTful APIs is to make full use of the standard protocols (the uniform interface) for interacting with resources. In the case of Web APIs, that protocol is HTTP. HTTP provides a rich vocabulary for interacting with resources (identified by URIs), that includes request methods, header fields and response codes. Effective use of these existing mechanisms lets us build complex applications whose general operation is well-understood by the host of intermediary services provided by the Web (caches, proxies, clients, etc.)

Unfortunately, many applications make use of only two request methods (GET and POST, and often incorrectly), a very few response codes (OK, Not Found, and Internal Server Error), and none of the headers. Instead of understanding and using HTTP, they reinvent the wheel and pipe proprietary, opaque and complex requests over HTTP using RPC style GET and POST requests.

To create well designed RESTful APIs, you’ll need to have a solid understanding of the standard HTTP methods (aka, verbs) and be familiar with the different headers and response codes that can be used to add semantics to HTTP responses and requests.

Due to time limitations, we’ll focus on the HTTP methods in class and only briefly touch on headers and response codes. RESTful Web Services provides a good discussion of these topics and the RESTful Web Services Cookbooks offers numerous recipes for how to use headers and response codes to meet specific design challenges.

Anatomy of HTTP Requests and Responses

Let’s take a quick look at what gets sent over the wire every time you make an HTTP request. There’s a lot going on here, but fortunately, we only care about a few parts.

```
GET /collection/authors?last=Abbo&limit=45 HTTP/1.1
Host: lha.idch.org
Connection: keep-alive
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) […] omitted […]
Accept: */*
Referer: http://lha.idch.org/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US, en; q=0.8
Accept-Charset: ISO-8859-1, utf-8; q=0.7,*; q=0.3

[MESSAGE BODY (IF ANY) GOES HERE]
```

Listing 1: An HTTP Request

The first line contains the HTTP method (GET in this case). Methods are discussed in detail below, but in general they describe the different types of actions that can be taken on any given resource. Briefly, GET
retrieves a representation of a resource, **PUT** updates a resource, **DELETE** removes the resource from the server, and **POST** appends to a resource or creates a sub-resource.

The next item on the first line is the resource we are looking for. In this the resource we are retrieving is the first 45 authors whose last name begins with ‘Abbo’. The final item is the protocol that the client is using to send this request, **HTTP/1.1**. The second line identifies the host where this resource is located.

Next come the headers. **HTTP** is used to pass messages around the internet. These messages are wrapped in address information (the first two lines) along with a number of different headers that describe how the server should process the request. These headers can identify the language the client would like to receive, the preferred format of the response, and a variety of other information. There are lots of different header fields and we’ll touch on a few of them briefly later on.

After the headers, a blank line separates the envelope from the message body. Some requests require a message, others don’t. In particular, **GET** and **DELETE** don’t use a message boy, **PUT** and **POST** do.

Here’s the response to this request.

```
HTTP/1.1 200 OK
Server: nginx/0.8.54
Date: Mon, 09 Apr 2012 20:47:25 GMT
Content-Type: application/json
Connection: keep-alive
Vary: Accept-Encoding
Content-Length: 384
Content-Encoding: gzip

{
  "header": [...],
  "authors": [
    {
      "id": "4g6du",
      "name": "Abbot, Ezra",
      "uri": "http://lha.idch.org/collection/authors/4g6du"
    },
    ... More stuff here ...
  ]
}
```

**Listing 2: An HTTP Response**

The first line states the protocol that the server is using (**HTTP/1.1** again) and a status code that describes what type of response this is. Common status codes are described in detail below and include **200 OK**, **404 Not Found**, **301 Moved Permanently**, and **500 Internal Server Error**. After the first line come the message headers, followed by an empty line, which is in turn followed by the representation of the resource. In this case, we have a JSON document containing an array of authors.
The specific content of the headers and response body should be interpreted in light of the response code. For example, if the response was 404 Not Found, the message body might contain a description of other things the client might do to find the resource, but it won’t contain a representation of the resource. If the response is 301 Moved Permanently, the Location header should contain the URL for where the resource is now.

**HTTP Methods**

The HTTP protocol defines a number of methods. Typical Web development focuses on only two of these GET (what happens when you type a URL in a browser, follow a link or reference an image in a <src href="..."> tag) and PUT (what happens when you submit a form: <form action="url" method="put">). Unfortunately, the ubiquity of these methods on the human web can mean that we forget about the other methods and their proper use.

The four most common methods (GET, PUT, DELETE and POST) are described below.

- **GET** Retrieve some representation of a resource. In a GET request, there is no content to the request, all of the information is supplied in the URL and the header fields. GET is both safe and idempotent (see below).

- **PUT** Updates the state of a resource. In general, the body of this request should contain a representation of the resource that is in the same format as was supplied by the server. In other words, once a client has received a representation of a resource, it should be able to modify that resource and PUT it back to the server to update the resource’s state (assuming of course that it has permission to do so). It may make sense to only PUT a portion of a resource, for example, excluding hyperlinks and other server generated components or (less desirably) only including changed data. PUT is idempotent.

- **DELETE** Deletes a resource from the server. DELETE is idempotent.

- **POST** Sends data to the server. POST is the most flexible (and most easily abused) of the HTTP methods. The canonical RESTful use of POST is to create a new resource on the server when the resource’s URL is not known in advance (e.g., posting a message to a bulletin board /messages that will create a new resource /messages/<msg_id>). Other uses include sending data to be service oriented resource (e.g., sending text to process to a NLP part of speech tagger) or POST is neither idempotent nor safe.

There are others (HEAD, OPTIONS, CONNECT and TRACE) but these are less interesting for our purposes. It is also possible to extend the set of methods if absolutely necessary. The most notable extension is WebDAV which adds a number of methods to support file-system style interactions with web resources (e.g., COPY, MOVE, LOCK).

**Safe Methods** A method is “safe” if it is guaranteed not to change the underlying representation of the resource or to produce side effects. Retrieving a resource is, logically, a safe operation – you aren’t
Idempotent production exist. too been you if required, content-Language (including browsers) can take advantage of this feature of a method by caching responses.

One caveat to this rule is that some side effects such as updating server logs are typically allowed. I have been known to implement resources that are generated on demand (such as scaled versions of images) and then stored on the server in response to GET requests. While caching responses and resources is well within the bounds of “safe”, generating on demand resources that may require significant computation (100’s of milliseconds) requires careful implementation. It is probably better practice in a production system to pre-generate derived resources.

**Idempotent Methods** A method is idempotent if repeated identical calls will produce identical results regardless of the current state of that resource or other resources on the server. For example, if you were to PUT an image to a website (e.g. PUT http://www.example.com/neal/profilepic) three different times, the result should be the same as if you’d only done it once. Web infrastructure (including browsers) can take advantage of this by automatically re-issuing a request if the server takes too long to return a response. GET, PUT, DELETE are idempotent.

**HTTP Headers**
HTTP headers function like stickers on the outside of an envelope that provide details about how it should be processed. For the most part, we’re going to gloss over headers, except to point out that they exist. The reason isn’t that headers aren’t important, but that they are fairly detailed and I want to keep our focus at a bit higher level, given the amount of time that we have.

I do, however, want to highlight four important functions that headers provide.

**Content Description** First, they describe the content of a request or response. Among other things, this includes the format of the response (an internet mime-type), the language and character encodings used, the content length, data and times, and an MD5 hash for error checking. Not all of these are required, but most are pretty common.

The header files that support content description include: Content-Length, Content-Encoding, Content-Language, Content-MD5, Content-Type, Date.

**Caching** One of the easiest ways to speed things up on the web is to simply not retrieve content that you already have. Most Web browsers will keep documents like images, JavaScript and CSS files, and entire Web pages in a local cache on your hard drive so that they don’t have to download the same content each time they visit a page.

The web architecture also supports intermediate caching servers to help control information flow over the web. A service provider might provide a cache as part of their internal architecture. This way, when you make a request, the “server” (really a network of machines working together) will first check to see if someone else has requested that same resource recently, and if so, serve up a copy stored in a cache.
If not, then your request gets passed along to a server that isn’t currently busy which performs all of the need database lookups, etc. to process your request. The response is then stored in the cache before it is sent back to you. Other intermediate servers (e.g., your ISP) might perform a similar function. The caching headers provide a way to control what content gets cached, what doesn’t and when cached representations expire.

The other set of headers that I’m lumping into this category allow clients to make conditional requests. For example, to get a resource only if the version on the server has changed, or to update a resource only if the version on the server hasn’t changed.

The header files that support caching include: Cache-Control, If-Match, If-Modified-Since, If-None-Match, ETag, Expires, Last-Modified, Vary

**Content Negotiation:** REST focuses on providing a representation of a resource. But what representation should you return? Content negotiation is the mechanism by which a client tells the server the types of resources that it would prefer, those which it will accept, and, by extension, those that it won’t accept. This can be used to specify media types (I’d prefer HTML, but I’ll accept XML, and PDF only if I have to), what language or other local specific information to use, how text should be encoded (UTF-8, for example), and whether or not compressed data is acceptable.

Content negotiation it the preferred way to indicate things like whether or not the server should provide an HTML, XML or JSON based representation of a document (as opposed to using an extension such as data.html, data.xml, data.json). Of course, people who access your API may or may not be able to (or understand) use HTTP headers correctly, so I would recommend a both/and approach: allow clients to specify representation formats using HTTP headers or file extensions, with file extensions taking precedence over the headers. Alternatively, you could just ignore the client’s preferences and provide whatever representation you want. That’s not a very nice way to play with others, but it does make life easier on you.

The header files that support content negotiation include: Accept, Accept-Charset, Accept-Encoding, Accept-Language, Content-Encoding, Content-Language, Content-Type, Transfer-Encoding

**Authentication** Finally, header fields also provide a couple of built-in methods (basic⁸ and digest⁹) to authenticate the person who is sending a request and a mechanism for extending this for other authentication methods (e.g. WWSE¹⁰). Most sites these days choose to use HTML forms¹¹ to handle authentication in the request/response messages rather than in the headers. Within the context of web

---

¹⁰ Wikipedia, “HTTP+HTML Form-Based Authentication” [April 2012].
¹¹ Wikipedia, “HTTP+HTML Form-Based Authentication” [April 2012].
applications, however, we usually aren’t authenticating users directly, we’re authenticating applications. That’s where API keys come in. Of course, we often need to authenticate the actual user who’s making a request via a third-party client to an API. For that, we’ll most likely use OAuth. As a general rule of thumb, if security is important to you, you need to talk to someone who knows what they are doing.

The header files that support authentication include: Authorization, WWW-Authenticate

**HTTP Response Codes**
Each HTTP response is identified by a response code; a numeric code with a short status message. This provides a formal mechanism for the client to know whether or not a message succeeded or not and how to interpret the response message and headers.

Response codes are grouped into four main categories: the 2XX series indicating successful requests, the 3XX series indicating redirections to other resources, the 4XX series for failures that are the client’s fault, and the 5XX series for failures that arise from an error on the server. The 1XX series is rarely used and includes informational response headers. I’ve provided brief descriptions of some of the most common response codes in an appendix.

When designing a RESTful API you’ll need to think through the types of responses that you might want to send for different requests. Could a request to update a resource cause a conflict with another resource on the server (for example, registering an account with a username that is already in use)? Will you need to start a long running process that may or may not complete? Are you going to track resources that move, such as merging two bibliographic records for the same item?

There are HTTP response codes for each of these scenarios (and many more) and being familiar with the response codes may help you think through some of the different scenarios people might encounter as they interact with your system.

---

Representations

REST focuses on sending representations back and forth between clients and servers, so we need to spend some time talking about what different representations are available and when to use them. Many representations are straightforward: images in different formats and sizes, PDF documents, metadata records that follow established standards, media formats, etc. You’ll need to think about these (for example, will you be providing audio in MP3 or WAV formats or both?) but the questions are pretty straightforward.

The bigger questions come up when you design representations for information that is unique to your project when there aren’t established standards or when those standards don’t meet your needs. Then you’ll need to give some specific thought to what format to use (typically JSON, XML or both), and how to actually represent the data.

There are a few general recommendations to follow when designing your representations. First, don’t reinvent the wheel. Wherever possible use existing standards. Chances are the community that developed the standard has done a better job of thinking through the issues and potential problems than you have. Following standards will save you the trouble of going through that work again and ensure that your work can be more easily understood and used by others.

Second, avoid religious wars. Lots of people (myself included) have strong preferences between competing standards. That’s fine, but you should strive to provide representations that meet the preferences of any and all of the communities that will use your API. That often means providing both JSON and XML representations.

Third, keep things as simple as possible from the perspective of the person using your service. Notably, that means that you should consider the common use cases and bundle up enough information that the client doesn’t have to make several different requests to get a set of information that will be routinely needed.

JSON

I prefer JSON for passing data across a Web API. JSON has three main advantages. First it is simple. I often hear people talk about how simple XML is, but it isn’t (try programming with XML in Java someday). The JSON standard fits on the back of a business card. Second, because the standard is simple, parsing and processing JSON data is fast. All major browsers include built in JSON parsers, so it’s really fast on the web, which is a lot of what I use my APIs for anyway. Third, there are easy to use implementations in most programming languages. These implementations often include automatic translations between JSON and the native data structures, so there’s no need to mess with the DOM or other XML driven programming tools.\(^{13}\)

\(^{13}\) In fairness, there are utilities that translate XML into native structures too.
I haven’t found a tutorial online that I’m really happy with. The canonical resource is the http://json.org. It has the formal (and short) specification for the language and offers a good description, but I find that the images get in the way a bit and the formality could benefit from a few examples. So what follows is my slightly re-written version of that.

JSON is just a highly restricted subset of the JavaScript language that is suitable for representing data. It provides two basic structures

- A collection of name/value pairs (we’ll call these objects)
- An ordered list of values (we’ll call these arrays)

Before we look at those two structures, let’s start off by defining what a value is. A value can be one of the following:

- a string: a sequence of letters surrounded by quotation marks. For example, “neal”, “string”, “”، حرف and "" are all strings.
- a number: just what it sounds like. 42, 0, -4.2, 3.0e12
- a boolean: true, or false
- null

In addition to these values, objects and arrays can also be values.

An object is an unordered set of name/value pairs. An object begins with a left brace ( { ) and ends with a right brace ( } ). Names are simply strings (see above). Note that strings must be surrounded by quotation marks. Each name is followed by colon (:) and the name/value pairs are separated by a comma (, ). There should not be a comma after the last name/value pair. Note that the name/value pairs are not ordered, so if you display an object on one system (the server, for example) and then again on another (the client) the contents may be re-arranged.

{ 
  "name" : "value"
}

{ "first" : "Neal",
  "last" : "Audenaert",
  "age" : 33,
  "married" : true,
  "address" : { "street" : "1700 George Bush Dr. E. STE 220",
    "city" : "College Station",
    "state" : "TX",
    "zip" : "77840" }
}

Listing 3: Example Objects
An array is an ordered collection of values. An array begins with a left bracket ([) and ends with a right bracket (]). Values are separated by a comma (,). The values can be of any type, so you can mix numbers, strings, Booleans, objects and arrays as the values of an array.

```
[]
```

```
[“neal”, “joe”, “fred”, true, false, 4.323e98, “george”]
```

```
```

Listing 4: Example Arrays

There are a number of good examples online at [http://json.org/example.html](http://json.org/example.html). Doug Crockford who “discovered” JSON has a nice video (50 min) describing the history and development of the standard that I’d highly recommend. [http://www.youtube.com/watch?v=C-JoyNuQJs](http://www.youtube.com/watch?v=C-JoyNuQJs).

JSON is a simple, lightweight tool for representing data that is adequate in a wide variety of circumstances. There are, however, a couple of noteworthy shortcomings. First, it doesn't provide support to validate that a particular JSON object conforms to a standard format like DTD, Schema, or RelaxNG provide for XML documents. Second, it doesn't provide tools like XSLT to support descriptive transformation of a JSON object or array into another type of document or data structure.

In general, I haven’t found these shortcomings problematic. Typically, I trust the person providing data to me to be providing it in the correct format. In my applications, that causes a runtime error, but so does trying to validate an XML document and finding that it isn’t valid. The biggest drawback that I see is that there is no support when I’m writing JSON, but these structures are almost always being generated programmatically. As for XSLT, that seems a small loss. I find it much, much easier to process a data structure programmatically with the support of a template tool like YUI Substitute than to create and debug XSLT code.

**XML**

XML is widely used to transfer data on the Web. It predates JSON, so naturally, there are quite a few older Web services that use it either because they too predate JSON or wanted to use a technology that was widely understood and well-supported. However, some people also prefer XML and it typically makes sense to send data in both formats.

There are two notable uses for which XML is very well suited. The first is when you need to send data for which there are widely used XML based standards such as METS or Dublin Core metadata records. If there are already widely adopted XML metadata standards, you should be using them (perhaps in addition to friendlier JSON formats). Second, XML is a useful document markup format and Digital Humanities projects will often involve documents marked up according to the TEI guidelines. JSON is a tool for data, and doesn’t make sense as a document markup language.
Designing RESTful Architectures

You’ll need to pick a design process that works for you and your development team, but here are some helpful suggestions. I’ve taken them from a very helpful presentation by Josh Bloch, who wrote the APIs for some of the major Java libraries\(^{14}\).

**Gather Requirements** You need to figure out what you want your API to do and write it down somewhere that everyone on the team can find it. This sounds like a no-brainer, but this step is often forgotten, overlooked, or done poorly. Unless your team is very small (one to three people) and probably even then, you should have a written list of your requirements, a way of prioritizing your requirements and a way of assigning requirements to different versions of your API. If you don’t, you will build things that no one wants or needs and you will get bogged down trying to add “just one more thing,” a problem formally known as requirements creep and informally as living hell.

The reason people don’t do this is because they hate requirements gathering. It can (but doesn’t have to) be a tedious, mind-numbingly boring process that results in equally massive, tedious, and mind-numbingly boring documents that aren’t read and don’t do a bit of good. I’d recommend against that approach. I prefer to write down simple, short “stories” about what people want should be able to do with your system. They should be reasonably specific (concrete examples are always easier to read and understand than high-level, nitpickingly precise statements), clear English (or pick your favorite language) prose. You can gather them on a wiki (or whatever happens to be convenient) and change them as needed up to the point you start to actually write code to implement them. In general, the more simple you make this process, the more likely people are to actually write (and read) the requirements. I wouldn’t design a nuclear power plant using this process, but for most systems that we’re likely to work on, this will probably work better.

To gather your requirements, start off by taking a look at other systems and identifying what works and what doesn’t. Also, get to know your data. Presumably you or someone else on the team is an expert in the subject you are dealing with. That person should be able to provide you with pretty good descriptions of what they want to do. Finally, talk to the people who will be using your API. This is your opportunity to reach out to your (future) user base and get their involvement from the start. Your API needs to respond to real-world demands and people are more likely to use your system if they’ve been involved in the process of building it.

**Keep it Simple** An API should do one thing and do it well. The interface should be a small as possible. It is easier to add features over time than to take them away. An API should also be as easy to understand as possible. There are a couple key guidelines here. First, names (in our case URIs) should make sense and should point to things that are easy to understand. If you are having a hard time naming something or explaining something, chances are you’ve got a bad design. Second, conceptual weight is more

\(^{14}\) Bloch, Joshua. “How to design a good API and why it matters” in *Proc. 21st ACM SIGPLAN Conference (OOPSLA)*, 2006, pp. 506-507. See a related video here: [http://www.youtube.com/watch?v=aAb7h5CtvGw](http://www.youtube.com/watch?v=aAb7h5CtvGw) [April 2012]. Slides and other resources are available here: [http://research.google.com/pubs/author32.html](http://research.google.com/pubs/author32.html) [April 2012].
important than the sheer number of resources in your design. It should be possible to interact with similar resources in similar ways. Query parameters that accomplish the same thing should have the same name across resources (and, by extension, different names if they do different things).

**Documentation Matters** A lot. If you don’t document your system well, people won’t use it. The first step in documentation, of course, is to design a clean, simple API with meaningful names for things. Minimally, your API should do more or less what it says it does and HTTP methods should behave properly. Clear, easy to read documentation with good useful examples is a critical component of your API. Just as you would test your API or a new user interface that you build, you should test your documentation. Show it to a few people (outside your core project team). Does it make sense to them? Chances are your first effort isn’t nearly as clear or easy to understand as you think it is (case in point, this course packet?).

The folks at Zotero\(^\text{15}\) have done a good job of this. I’m also really fond of the documentation for YAHOO User Interface\(^\text{16}\). That’s a JavaScript library, so there isn’t a one to one correspondence, but it is a good model for how libraries should be documented. As a side note, on their job postings they clearly indicate that they are looking for developers who are passionate about documentation. Good documentation is built into their software development culture. On the other hand, Flikr’s API documentation is terrible\(^\text{17}\).

**Consider Performance Implications** Think through how people will work use your APIs and make sure that you’ve designed your resource representations accordingly. For example, if you are building a document management system, and people need to display author names alongside author titles with a link to details about the author, then you document representation should include author names and links, not just author ids. At the same time, you probably don’t want to send all the details about a particular author over the wire for each document that you retrieve. Clients that interact with RESTful APIs will need to make HTTP requests every time they need to retrieve information. These requests take time and you will need to think carefully about how to balance the number of requests, the time it takes to respond to request, and the size of the response representation in order to meet specific performance goals. Your API needs to reflect the needs of the clients who will be using it, not the details of the data structures you use to implement it.

**Handle Exceptional Conditions** Things go wrong: clients make bad requests, resources aren’t available as expected, servers break, networks fail. Fortunately, the HTTP standard provides a rich set of high-level mechanisms for reporting exceptional conditions (aka errors). Your design should think through what can go wrong and use these mechanisms appropriately. You will also need to consider what sort of specific information your application needs to provide in order to help clients understand and respond to these errors when they happen.

---
\(^\text{15}\) Zotero. “Zotero Server Read API” [http://www.zotero.org/support/dev/server_api/read_api] [April 2012]
Plan for Intermediate Actors One key component of the Web infrastructure is the presence of intermediate actors that can, for example, cache responses and support cross-cutting concerns like authentication and authorization.

Eat Your Own Cooking An API needs time to mature. That means people need to write applications that use your interface in order to hammer out the initial design flaws. If you release an API that no one has used to build real applications, chances are you will find out very quickly and very publicly that you forgot something important in your design. Feed it to a few close friends as well. You need more than one implementation. With just one, you might have designed features that are tailored to a specific need. With two or three different implementations, you’ll find the places where you need to provide a more general solution. Then you’re ready to share it.

Decision Process
Building a REST based API for your project is not a step to be taken lightly. Doing this correctly will require careful up-front design, buy-in from the people who will use your API, and significant programming effort. An API is designed for others to use, which implies a level of commitment to document and maintain both the API and the RESTful Web-service.

Of course, you can always cobble together something that works and meets your own local needs and then see if things catch on from there. That is to say, you can create a rapid prototype RESTful API. Sometimes that makes sense as a method for structuring your initial design or supporting your own internal development. I tend to create systems in which rich JavaScript/AJAX driven UI’s interact with semi-RESTful API’s on the back end as a way to make my development process easier.

If you want your API to be broadly useful to others, however, it is going to take a bit more effort. Before you invest the time, effort and money in this process, it is critical that you make sure you have a good reason to do so. As a rule of thumb, if you don’t need to go down this road, don’t. For a bit more detail on some of the factors that I think are important when considering whether a public API is a good investment, see the decision support questionnaire in the appendices.

Requirements Design
People will use your API. At least that’s the goal. There are two implications to this. First, you need to spend some time figuring out what these people want and need. They won’t come just because you built it. Second, once you build it, any changes you make will break things that people have built and are using. Breaking things makes people upset.

You need to do some up front planning for your API project so that you get (more or less) the right overall architecture. You won’t get everything right at first and you will certainly want to add features over time. If your project is big enough and lasts long enough, you may even need a strategy for deprecating older parts of the system—marking features that are no longer supported and allowing people time to migrate to new features before you pull the plug.

At the same time, building a good API isn’t like other ‘agile’ software projects where you can go back and refactor things that you don’t get right the first time around. Other people will write programs that
use your API (that’s the point) and any changes you make might break their programs. You need to put some serious thought into your up-front design. If you get it wrong, you might be stuck with it for a long time. Of course, that thought might come in the form of rapid prototyping of an API (for internal use) that you fully intend to revise heavily before releasing to the public.

A formal discussion of requirements analysis and design is beyond the scope of this course, but I don’t think that this needs to be rocket science. Here are a few general guidelines that I think make sense.

**Know your data.** You are building an API to provide access to your data. To start with, forget about the fact that there is a computer involved and focus on providing a clear, concise description of your data and/or the domain you are modeling. As much as possible, your system design should reflect your data. In general, when I work with humanities scholars, I come in with a detailed knowledge of the technology but only a basic understanding of their data. My first job is to learn the terms they use and how different components fit together. The biggest challenge is often to get people to forget the technology and focus on what they actually do on a daily basis.

One person I worked with in the field of textual criticism kept referring to things as articles and paragraphs, but the actual data he was showing me looked nothing like articles and paragraphs. I finally had to stop him and ask if that was really the terms he used to describe this stuff. It turned out that the software vendor he’d been working with was trying to fit his data into a system that recognized books, articles and paragraphs, so he’d adopted that terminology when talking with tech people. What he really wanted were variation units and variant readings.

You might need, at some point to abstract from the specific terminology of your discipline into more abstract concepts if those concepts really are appropriate, but don’t start there. Start by focusing on what you do with the words you use when you talk to colleagues.

**Steal ruthlessly.** You aren’t the first person to do this. Chances are there are other systems out there in some form or another who have dealt with many of the same types of requirements you have. It might just be in terms of a small component. Maybe they address the same problem you are working on and did a really bad job. Whatever the case, you should learn from others as much as possible and deviate from any established norms only when necessary. Your API will be easier to learn if people are already familiar with parts of it. And you don’t need to steal only from other Web APIs. What are the challenges and solutions that people have dealt with in other digital projects. What have people done in print?

**Ask.** I’ve covered this before, but the best way to gather requirements is to ask the people who will be using the system. Keep in mind, of course, that you don’t have to do everything that people want and that your role as a designer is to synthesize people’s responses into high-level commonalities. I’m not advocating design by committee. That goes against the goal of getting a simple, clean interface. Your API needs a strong, cohesive design to it. But you do need to have a deep understanding of what people want and that process begins by gathering specific feedback.

**Prototype and alpha test.** You won’t get things right the first time, no matter how hard you try. There is a tension between the “design it all up front” approach and the “tweak and refine it as you go”
approach. I’m going to suggest a third way that I’ll call the “big rocks” approach. You need to do enough work up front so that you have the “big rocks” in place—the core architectural components. If you get those wrong, it’s hard to go back and fix things. But once you’ve got that hammered out pretty well, you can start to prototype some of the components you don’t yet understand and use that experience to gain clarity on requirements that you might have missed. And before you release (even a beta) you need people to write code that interacts with your API. That will bring up the more problems that you haven’t thought of yet. So long as everyone knows that you’re not ready yet, no one will get hurt if you need to change things. Once you go to a Beta release, the basic API should be in place and you are just working of the last few bugs.

**Designing Resources**

Richardson and Ruby (148) offer a helpful outline for transitioning a high-level description of your requirements into resources.

1. Figure out the data set
2. Split the data into resources

For each kind of resource

3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) served to the client
6. Design the representation(s) accepted from the client
7. Integrate this resource into existing resources
8. Consider the typical course of events: what is supposed to happen?
9. Consider error conditions: what might go wrong?

**Figure Out the Data Set**

In this step you will assemble and organize the requirements for your system. This can be a formal requirement document (if required) or something simpler like an authoritative prose description of your system or a cleaned up list of use cases or user stories. The level of formality that you need depends on your project’s needs. My general opinion is that you don’t need to get everything exactly right here because things are likely to change as we go through this process. Once you get done, you will have a very specific set of requirements.

**Split the data into resources**

To do this, you first want to identify all of the main nouns in the description of the data set. Most likely these nouns will be the resources you need to create (some will be attributes of resources). When looking at nouns, make it a resource if you ever need to refer to that thing specifically.

Once you’ve identified the concrete objects that you will make available through your API, it is time to think about collections of those objects. So if you need to access an author, will you also need to access authors? These collections are likely candidates for resources in their own right. Notably, collection resources can provide a natural resource to use to search, filter, and sort its member objects.
Finally, look through your data set for algorithmic components and processes. Do your users need to register for something? If so, you probably need a ‘registration’ process. Keep in mind, we are trying to identify things, so whenever possible we want to use nouns rather than verbs to describe our resources (e.g., registration rather than register). Sometimes, however, we will end up with verb. Perhaps you need a search engine? That will likely be named ‘search’ despite our preference for nouns. Perhaps you need to merge records of, for instance, two names in an authority file that actually refer to the same person. I’ll think of this as a ‘merge service’ (a noun), but I would probably name it ‘merge’. If we start seeing a lot of verbs in our resources list, that’s probably a good sign that we need to focus more on things rather than functions, but at the same time, we shouldn’t jump through hoops in order to avoid verbs at all costs.

**Name the resources with URIs**

Now that we’ve identified all of the resources with English names, it is time to start assigning proper RESTful names: URIs. Rather than specifying the full URI, we’ll assume that our Web service all starts from a common end point ([http://www.example.com/api/](http://www.example.com/api/)) and we’ll just specify the part of the URI after that, beginning with a forward slash (/). Our URI names should reflect the English names of the resources that we’ve identified. I’ll use brackets to identify parts of the URL such as identifiers that will change in predictable ways. So for example, if we want to describe a collection of documents we might have resources like the following

- `/documents` The collection of documents
- `/document/{docId}` A document with the indicated id
- `/authors/` The collection of authors
- `/authors/{authId}` The author with the indicated id

We’ll use the forward slash to indicate sub-resources that are directly related to a parent resource. So, if we want to represent chapters explicitly, we might use this URI:

```
/documents/{docId}/{chId}
```

One naming option worth mentioning is matrix parameters because they aren’t commonly used, but can be very useful. Query parameters are the name value pairs that come at the end of a URI after a question mark (?) and are used to restrict or otherwise clarify what information a particular resource should provide. Matrix parameters, on the other hand, can be attached to any path segment (the bits between forward slashes) and come after a semi-colon (;). For example, if we wanted to represent different editions of the same book, we might chose to do something like the following:

```
/documents/{docId};edition={ed}/{chId}
```

In this case, the URI without the matrix parameter would represent some default edition, or perhaps a higher level construct such the book in all of its editions. If present, the matrix parameter refines which specific edition of the book we should use for interpreting the rest of bibliographic record.
Matrix parameters strike me as very useful concepts but they aren’t very well known and hence aren’t used very often (case in point, I’ve never used them before). It is probably worth your while to do some quick prototyping to make sure that you can use them in your programming language of choice and that your prospective users can use them as well.

Exposé a subset of the uniform interface
Now that you have names for your resources, you need to define what actions are possible for each of those names. Since you only have four options (the uniform interface: GET, PUT, DELETE, POST), you have a lot of structure already built in. You don’t need to use every method with every name (for example, it probably doesn’t make sense to delete or put to the collection-level resource /authors).

As a quick example, here’s one possibility for our document collection

<table>
<thead>
<tr>
<th>Resources Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>URL Template</strong></td>
</tr>
<tr>
<td>/documents</td>
</tr>
<tr>
<td>/documents/{docId}</td>
</tr>
<tr>
<td>/authors</td>
</tr>
<tr>
<td>/authors/{authId}</td>
</tr>
</tbody>
</table>

I’ve included a simple table like this in the Appendix D. Note that we are still at a pretty high level of abstraction here. I probably want to be able to search for documents by their title and other properties (something like /documents?title=wind) and I haven’t specified any of those details yet. Second, there is a lot of symmetry here—/documents and /authors behave in almost identically. I could have copied and pasted the two rows and just changed documents to authors (which I did). That is intentional (and because this is a simple example). Designing parallel structures different parts of the system helps make the overall API easier to learn and understand.

Design the representation(s) served to the client
Next we need to decide on and design the representations that our server will send out to the client. This is where we need to go back through our requirements and find out what sort of properties we need for each of our different resources. This is also the step where I’m going to add in additional detail about the query parameters, and really flesh out the behavior of the resources a bit. I’ve got a different table that I’m going to use for that (I’ll just work with one resource for simplicity sake)
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Version</th>
<th>Values</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>0.1</td>
<td>string</td>
<td>none</td>
<td>Query by title. This will perform a ranked search of documents. If supplied, the result list will be sorted by the highest ranked results by default.</td>
</tr>
<tr>
<td>author</td>
<td>0.1</td>
<td>string</td>
<td>none</td>
<td>Query by author’s name. This will match any prefix of the author’s last name.</td>
</tr>
<tr>
<td>before</td>
<td>0.1</td>
<td>4 digit year</td>
<td>none</td>
<td>Restrict search to documents published before this year.</td>
</tr>
<tr>
<td>after</td>
<td>0.1</td>
<td>4 digit year</td>
<td>none</td>
<td>Restrict search to documents published after this year.</td>
</tr>
<tr>
<td>order</td>
<td>0.1</td>
<td>author year ranked</td>
<td>author</td>
<td>A named sorting strategy. By default, documents are sorted by author’s last name then by the year of publication. To sort by year of publication, then last name, set this value to ‘year’.</td>
</tr>
<tr>
<td>pgSize</td>
<td>0.1</td>
<td>number</td>
<td>50</td>
<td>The number of documents to return per page. This will return a maximum of 1000 items even if the value specified here is larger than 1000.</td>
</tr>
<tr>
<td>offset</td>
<td>0.1</td>
<td>number</td>
<td>0</td>
<td>The index of the first document to return.</td>
</tr>
</tbody>
</table>

**Representation (JSON)**

```
{
   "header": {
   
   },

   "documents": [
   {
   "id": 47,
   "title": "Gone with the Wind",
   "authors": [
   {
   "id": 23,
   "name": "Margaret Mitchell"
   }
   ],
   "publicationDate": 1936
   }
   ]
}
```
In this table, I’ve added a lot of documentation. I’ve also tried to include a few examples. These examples will be useful for documenting this interface to end users later on, but they also provide me with a way to match up the description of what my system would do that I pulled together in the requirements gathering process and how those different requirements got implemented. When I’m done, if I go back through my requirements, and I don’t find an example of something, I know that I might have missed it. Finally, these examples give me a way of sorting through what query parameters I need to include.

I’m not going to return all the results every time (there could easily be millions of them). Instead, I’m returning a slice (or page) of the results. I’ve included a couple of parameters for slicing large results into pages. In a larger project, this is likely to be something that I need to do for most collections, so I’ll just document these parameters one.

A proper representation would need to supply a lot more information and probably other formats. I’ve tried to keep things simple. I’ve added a header to provide a bit of meta-information. Since I’m sending over a number of different query parameters that have default values, I’m going to use this to describe exactly how I interpreted the query that was received. That will likely be useful for debugging purposes. Also, I’m not returning all results (there could easily be millions of them). I’ll also need to tell the client how many documents I returned out of the total in this slice and where in the complete result set the documents that I returned are.

I’m returning a simplified representation of the author. An author will have information broken out into first and last name, and might also include details like a date of birth and death, a brief biography, etc. For the purposes of this resource however, all I need to do is to be able to display the name of the authors for the document and create a link to the author’s profile, so I won’t include any extraneous information in order to keep my representation as small as possible. If I find that I need more info, I can add it in later (that’s what alpha prototyping is for).

I haven’t bothered with any links yet because that will come up in a later step. You might decide to design the entire representation at once, but I like the fact that Richardson and Ruby break out the tasks of defining the main representation and links into two separate tasks.

**Design the representation(s) accepted from the client**

I’m going to do the same thing here that I did for the **GET** method. I’m actually going to create a new detailed description table for each HTTP method that I have defined in my master table. So here it is:

<table>
<thead>
<tr>
<th>List Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>
This turns out to be much simpler. There are no parameters (in this case). For the representation, I don’t need to supply a list of documents, just one. I also don’t need a header since that was used to convey information to the client. I’ve removed the id from the document representation (since I don’t know it) and I’ve also changed the list of authors to be just the id of the author. I’ve designed this format to be fairly tolerant of the data that goes in this representation, but I’ve described in the documentation what exactly I’ll accept. If I’m going to be flexible about the data I get from the client, then that flexibility becomes part of my contract.

Integrate this resource into existing resources
Now that I know what data I’m sending back and forth over the wires, I need to design the connecting between different resources. This is a pretty simple system, so I only need a few pieces of information.

- A link to self. This link will include any supplied default parameters such as the sorting profile and the result slice details. That way, the client (and whoever else ends up with this representation will know who to repeat this query.
- Paging details. I need to provide links to retrieve the next and previous results for this query (if relevant).
- Links to the full (and canonical) reference for the paper’s authors.

I’m going to add the link to self and the paging details to the header and the links to the authors to the author objects. Here is the snippet for the header links:

```
“header” : {
    “links” : [ 
    ],
},
```

And the author URI gets added like this:
“authors” : [ 
  { “id” : 23,
    “uri” : “http://.../authors/23”,
    “name” : “Margaret Mitchell” }
],

Consider the typical course of events: what is supposed to happen?
In our last two steps, we need to think through how a client will interact with this resource and what sort of response codes we will need to return.

First, for the GET method, we’ll expect that the user provides us with a valid request that will match zero or more documents. Matching zero requests is a valid search result (as opposed to a 404 Not Found error). In these scenarios, we’ll return a 200 OK response with the results representation in the message body.

For the POST method, if all goes according to plan, we’ll create a new resource on the server. In this case, we need to return a 201 Created response code. We’ll also need to set the Location header to the URI of the document we just created and return a representation of that document in the message body.

Consider error conditions: what might go wrong?
For the GET request, there isn’t a whole lot that can go wrong. The two things that I see are if the client specifies an invalid date range (for example, before 1900 and after 1901) or if it specifies an invalid slice (an offset greater than the number of documents in the result or less than zero). In both of these instances we’ll send a 400 Bad Request with a description of what went wrong in message body.

For the POST request, things are a bit more complicated. The client/user might not be logged in (401 Unauthorized) or he might not have permission to create documents (403 Forbidden). The document might specify an author id that doesn’t exist (409 Conflict). The client might send XML in which case we could be nice and explain that we don’t accept XML just yet (415 Unsupported Media Type). Alternatively, for that and any other permutation of just junk data being sent to us, we’ll respond with the catch all 400 Bad Request. Of course, in all of these situations, we’ll try to provide a helpful message to explain what went wrong in more detail.

Of course, there’s a whole host of other problems that could go wrong such as the database being down. Those errors are our fault will all get grouped up under 500 Internal Server Error. We’ll log these exceptions and return a page with a stack trace and a link to report this error to our bug tracking system.

I think that covers the errors, but in all likelihood I’ve missed something. We’ll figure it out when we start writing code that uses this API.

Summary
Once you’ve gone through this process, you should have a pretty good start on your API design. I emphasize that this is a good start, because the design process isn’t done. You, or your programming team, rather, needs to build a reference implementation of the API to make sure that you can actually
implement all of those nifty features that you designed. You’ll probably want to prioritize the most important features and decide what you will put off until the next version. It is a lot easier to describe how things should work than to actually implement them. There are still a bunch of technical details that we haven’t addressed like headers for cache control, responding to conditional request, and authentication.

You also need to spend some time writing clients for your API. The point of an API is that it lets people do innovative things that you’ll never dream of, but you need some good examples to get them started. You also need the exercise of actually using your API in order to discover some of the glaring flaws in your design (and trust me, there will be flaws). It might also make sense to have an independent review just to get a different perspective from someone who doesn’t have a vested interest in your design.

All of those things aside, however, at this point, you have completed the core task of translating your domain specific needs into RESTful API.
Appendix A: Useful Resources

Books
These are the books that I’ve used in preparing the class (along with some notes). There are LOTS of resources available on the Web, but it is often useful to have a more in-depth, structured presentation of the topic.

This book is packed with short, easy to understand solutions to REST design challenges. This is the only one of these three books that I would definitely recommend buying. If you are going to be designing RESTful API, you should have this book. Skim through it once to get a feel for what it has to offer and then turn to it as a reference to address specific questions during your design work.

As far as I can tell, this is the definitive resource on building RESTful Web services. It is definitely oriented toward developers with a focus on Ruby on Rails. I like the example-driven method to introducing RESTful architectures (including the pitfalls of RPC and REST-RPC hybrid architectures) and he gives an excellent overview of how to design REST architectures (I’ve taken the procedure for the course directly from this book).

Burke, Bill. RESTful Java with JAX-RS. O’Reilly (2009)
Another great book, but definitely aimed at Java developers. That said, the first two chapters provide an excellent over of REST with the added advantage of being short. Past that, this is primarily useful for people writing code.

First, a giant caveat: This is a brand new book and I haven’t read it (yet) so I can’t personally recommend it. This book aims to provide a concise guide that highlights the business potential of APIs for decision makers, which should make it very relevant to this class, assuming it is well written. There is reason to hope the authors include the Director of Engineering at Netflix (Jacobson) and the CTO of Apigee (Brail), a site that I’ve found very helpful in preparing this course.

Web Resources
There are far more web resources than I care to look at or you care to read. Here are some that I’ve found useful and/or interesting. Please search around for more and suggest any that seem particularly informative.

Overview and Introduction

Rodriguez, Alex. “RESTful Web services: The basics”, IBM Developer Works: (Nov 2008)  

Wikipedia. “Representational State Transfer”  

**URL Design**


http://www.w3.org/TR/webarch/#identification [April 2012]


Théreaux, Olivier. “Understanding URIs” in “Common HTTP Implementation Problems”, WC3 (Jan 2003)  
http://www.w3.org/TR/chips/#uri [April 2012]

Appendix B: HTTP Headers

Below is a description of some of the HTTP headers that I think are worth knowing about. See REST Patterns\(^\text{18}\) of Wikipedia\(^\text{19}\) for more detail. This is by no means a comprehensive list. This is intended to offer a high-level perspective on some of the different header fields that are available and the types of client-server interactions they govern. For a better understanding of how these fields are actually used, I would recommend the REST Patterns web site for details of each field and the RESTful Web Services Cookbook for practical examples and advice.

**General Headers**

These are used in both the request and the response.

**Cache-Control**

Used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain. For example: `Cache-Control: no-cache`

**Content-**

This family of headers specifies details about the format of the message body. In includes `Content-Encoding` (what form of compression was used), `Content-Language`, `Content-Length`, and `Content-Type` (the MIME type of the representation) among others.

**Request Headers**

These are sent by a client to the server.

**Accept-**

This family of headers specified details about the type of response a client will accept from the server. These fields are used for content negotiation (e.g., deciding whether to send XML or JSON or to send the French language version of a resource instead of the English version). Headers in this family include: `Accept` (the media type to accept), `Accept-Charset`, `Accept-Encoding` (what forms of compression are acceptable), and `Accept-Language` among others. These headings can be used to distinguish between preferred formats and otherwise acceptable format with qualifiers for deprecation. For example, it might express that the GIF version of an image is preferable unless it represents less than 80% of the quality of the JPG version.

**Authorization**

Specifies the HTTP authorization credentials of the client. For example: `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`

---

\(^\text{18}\) REST Pattens. “HTTP Headers” [http://restpatterns.org/HTTP_Headers] [April 2012]

Host
The host and port number of the server from which the resource is being requested (obtained from the original URI).

If->*</p>
This family of headers is used in conjunction with conditional requests. It includes If, If-Match, If-Modified-Since, If-None-Match, If-Unmodified-Since among others.

Referer
Used to specify (for the server’s benefit) the URI of the resource from which the request URI was returned. This header field was misspelled (referrer) in the original standard.

User-Agent
The user agent of the client making this request. For example, User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0)

**Response Headers**
These are the headers sent by the server to the client.

ETag
Used for supporting conditional requests to indicate the version of a resource returned in this response. The implementation of the ETag header is implementation dependent with the caveat that must be different for different states of the same resource. Consequently, a message digest (such as MD5) is often used.

Expires
Gives the date and time when this resource should be considered stale and expunged from any caches. For example: Expires: Thu, 01 Dec 1994 16:00:00 GMT

Last-Modified
The date this resource was last modified.

Location
Used to specify the URI of a resource in redirection or resource creation.

Vary
Tells downstream proxies how to match future request headers to decide whether the cached response can be used rather than requesting a fresh one from the origin server.

WWW-Authenticate
Indicates the authentication scheme that should be used. For example, WWW-Authenticate: Basic
Appendix C: HTTP Response Codes

Below is a description of some of the most common HTTP response codes that are used to identify the result of the request. These response codes are grouped into four main categories: the 2XX series indicating successful requests, the 3XX series indicating redirections to other resources, the 4XX series for failures that are the client’s fault, and the 5XX series for failures that arise from an error on the server.

When designing a RESTful API you’ll need to think through the types of responses that you might want to send for different requests.

I have included only some of the most useful and common responses. For a complete and authoritative list, see Section 10 of the HTTP/1.1 RFC or (more usefully) the related Wikipedia article. The descriptions below are edited versions of those found in Section 10 of the HTTP/1.1 RFC.

2XX Successful Responses

200 OK
Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request the response will contain an entity describing or containing the result of the action.

201 Created
The request has been fulfilled and resulted in a new resource being created. The response should provide a URI that references the newly created resource in the Location header field and include an entity that describes the created resource. The server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

202 Accepted
The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. The entity returned with this response SHOULD include an indication of the request’s current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

---


204 No Content
The server has fulfilled the request but does not need to return an entity-body, but might want to return updated metainformation. The 204 response MUST NOT include a message-body.

3XX Redirection

301 Moved Permanently
This and all future requests should be directed to the given URI. The new permanent URI should be given by the Location field in the response and the entity of the response should contain a short hypertext note with a hyperlink to the new URI(s).

303 See Other
The response to the request can be found under a different URI and should be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The different URI should be given by the Location field in the response and the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

304 Not Modified
If the client has performed a conditional GET and the requested document has not been modified, the server should respond with this status code. The 304 response must not contain a message-body, but should include the following headers: Date, ETag and/or Content-Location, Expires, Cache-Control and/or Vary.

307 - Temporary Redirect
The requested resource resides temporarily under a different URI, but the redirection may be altered on occasion so the client SHOULD continue to use the Request-URI for future requests. The temporary URI should be given by the Location field in the response and the entity of the response should contain a short note with a hyperlink to the new URI(s).

4XX Client Errors

400 Bad Request
This is a catch all response indicating that the request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications.

401 Unauthorized
The request requires user authentication. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials.
403 Forbidden
The server understood the request, but is refusing to fulfill it. Authorization will not help and the request should not be repeated. The reason for the failure may be described in the entity. Since 403 implies that the resource does exist, the server may wish to return status code 404 in order to conceal the existence of this resource from the client.

404 Not Found
The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code should be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is also commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

409 Conflict
The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body should include enough information for the user to recognize the source of the conflict. Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the entity being PUT included changes to a resource which conflict with those made by an earlier (third-party) request, the server might use the 409 response to indicate that it can't complete the request. In this case, the response entity might contain a list of the differences between the two versions in a format defined by the response Content-Type.

410 Gone
The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer working at the server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time -- that is left to the discretion of the server owner.

5XX Server Errors

500 Internal Server Error
The server encountered an unexpected condition which prevented it from fulfilling the request. This is frequently due to a programming error or some other server malfunction (e.g., a database is not available). Depending on the nature of the error, the response should provide detail to support error reporting and bug tracking.
Appendix D: Requirements & Resource Description Worksheets

There are a variety of different ways to capture user requirements. My goal in the requirements gathering stage is always to keep things a light weight and easy as possible. No one likes reading through pages and pages of detailed requirements documents. By making your requirements documentation as story-like and concrete as possible, you is, the more likely people are to read and implement it.

User Stories

User stories are a simple way of capturing stakeholder needs. These are commonly used in designing user-oriented applications, but I think they have a place in API design as well. The basic idea is that you (the designer) work with a stakeholder in order to solicit descriptions of what they want to accomplish. These may be captured on 3x5 cards (because it’s easier to write on note cards when you’re talking with someone than it is to type on a computer) but often time you will be gathering these by email, on a wiki or in some other non-conversational form. At any rate, you’re probably going to want a digital version at some point.

Depending on how you approach this, you’ll collect these stories either from end users (the people who will use the applications build on your design), domain experts (people who understand the scholarship behind what your API) or application developers (people who will be creating the software that actually uses your API). Of course, it’s probably a good idea to talk to all three groups, assuming there’s time and the project’s complexity warrants it.

There are a few points worth remembering. First, remember that you are the designer (assuming that you are). Users will say that they want all sorts of things, but it’s is your job to interpret their needs. You may need to adapt what they say in light technological opportunities (or restrictions) that they aren’t aware of. You may need to take a very concrete desire and abstract it to represent a wider range of scenarios. You will also need to recognize when a user’s request for a specific technological feature indicates a goal that could be better achieved through a different strategy.

Second, different people have different goals. It’s usually a good idea to talk with more than one person. This will help you make sure that you get a broader perspective on what users need and also give you a chance to reflect on what one person has said and ask follow up questions to verify, elaborate and revise their answers.

Finally, you’ll likely want to take your raw data and develop a synthesized list that captures the major user requirements in a more general form. The result should be user stories that sound like a real user said them, but may not reflect the actual words of any individual. Of course, it’s always worthwhile to show these stories to actual users to make sure you aren’t misrepresenting their needs accidentally.

---

22 Wikipedia. “User Stories” [April 2012]. There are some good links to related material here as well. This article approaches this from the semi-formalized perspective of agile software development methodologies. The guidelines here are useful (e.g., “Every user story must at some point have one or more acceptance tests attached”) but I’m envisioning these much less formally: it’s a good basic idea, use the formalisms that help in your context.
User stories are expressed in the general form of “As a <ROLE> I want <GOAL> so that <BENEFIT>”. Or you may choose to shorten it by leaving off the benefit portion. I find the benefit helpful because knowing why someone wants things a certain way often clarifies its importance and helps me think through alternative strategies to achieve the same goal. That said, the benefit clause doesn’t make sense for all stories, so there’s no need to force it.

Table 1: User stories worksheet

<table>
<thead>
<tr>
<th>User Story</th>
</tr>
</thead>
<tbody>
<tr>
<td>As a &lt;role&gt; so that &lt;benefit&gt; I want to &lt;goal</td>
</tr>
<tr>
<td>As a &lt;role&gt; so that &lt;benefit&gt; I want to &lt;goal</td>
</tr>
<tr>
<td>As a &lt;role&gt; so that &lt;benefit&gt; I want to &lt;goal</td>
</tr>
<tr>
<td>As a &lt;role&gt; so that &lt;benefit&gt; I want to &lt;goal</td>
</tr>
</tbody>
</table>

Use Cases

Use cases23 mean different things to different people. At a basic level, it is simply a statement of some particular use that someone might want to do with your system. Given this high-level description, I think use cases are among the most important (and easiest) tools in defining what a system should do. If you don’t know what people actually want to use your system for, you will end up designing and building lots of things people don’t want without providing support for the things that they need to do. Defining the use cases that you want to support up front will go a long way to keep your project from running off the rails or delivering something no one wants.

Use cases can (and should) be as informal as you like, so long as they meet your needs. When use cases are used more formally, they generally conform to some specific template that captures additional information about who the main actors in this use case are, what other scenarios they extends and a variety of other details. The term originated in a highly formal software design process supported by a visual modeling language.

I’m not a fan of the formalisms. I think that, for most projects that we work on in an academic setting, a use case should be just that: a simple statement of something I want to use the system for. We usually work in small teams where we can (and should) talk with each other frequently. If there are questions and ambiguities in the way a use case is written, we can ask for clarification. Use cases serve as a way to get key ideas out of peoples’ heads and onto the table where everyone can interact with them. They don’t need to be a replacement for a deep, internalized understanding of the project that comes from working closely with skilled colleagues.

23 Wikipedia. “Use Case” [April 2012]
GatherSpace.com “Writing effective Use Case and User Story examples” [April 2012]
That said, I have found it useful to have a bit of structure in order to capture related information that I think is important and to make notes of how I think a particular use case might actually work in the system. The following table captures a structure that I've used in the past. I offer here as a starting point for how you might think about capturing and organizing use cases in your project.

Table 2: Use case worksheet

<table>
<thead>
<tr>
<th>USE CASE NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synopsis</strong></td>
</tr>
<tr>
<td><strong>Domain</strong></td>
</tr>
<tr>
<td><strong>Depends On</strong></td>
</tr>
<tr>
<td><strong>See Also</strong></td>
</tr>
</tbody>
</table>

**Background**

A description of the problem, need and or business case for this use case along with concrete examples that might motivate it.

**Implementation**

Any implementation notes that might discuss how this particular problem should be addressed.

**Tracking Details**

<table>
<thead>
<tr>
<th>Status</th>
<th>Planned For</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Suggested By</strong></td>
<td>Person Name <a href="mailto:email@address.org">email@address.org</a></td>
</tr>
<tr>
<td><strong>Responsible</strong></td>
<td>Staff Member <a href="mailto:email@project.edu">email@project.edu</a></td>
</tr>
</tbody>
</table>
Describing Resources (High Level)
Here is the table that I used in the main document to assign URIs to resources and to describe the actions associated with the different verbs in HTTP’s uniform interface. This table is useful for steps three and four of the resource design process: “name the resources with URIs” and “expose a subset of the uniform interface”. I’ve broken the table into sub-sections so that you can deal with different components of a large API separately.

One advantage to this table is that it forces you to keep your descriptions of each resource short. That conforms to one of the key best practices in designing an API. Chances are if you have a concept that is difficult to explain, it will be difficult to understand and you need to go back and see if there is a more straightforward way of approaching that particular problem.

Table 3: High level resource description matrix

<table>
<thead>
<tr>
<th>Resources Summary</th>
<th>Resource Category</th>
<th>URI Template</th>
<th>GET</th>
<th>PUT</th>
<th>DELETE</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource Category</th>
<th>URI Template</th>
<th>GET</th>
<th>PUT</th>
<th>DELETE</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Describing Resources (Detail)
Below is a template for capturing the details of a particular resource including any applicable query parameters, usage examples and the different representations you will have for that resource. You should have one of these (or something similar) for every applicable HTTP method of every resource that you defined in the resources summary.

Table 4: Resources details template

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Description Examples</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Version</th>
<th>Values</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Representation (JSON)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Representation (XML)</th>
</tr>
</thead>
</table>
Appendix E: Decision Support Questionnaire

I’ve put together the following decision support questionnaire as a set of initial questions for decision makers contemplating the development of a RESTful API. I’m offering this as a tentative starting point that you can (should) adapt to fit the context of your project. It isn’t exhaustive and I haven’t had the opportunity to vet this with people actually working on API projects, so your feedback and suggestions would be very much appreciated.

Do I have data that other people will want to use?

Making data available to others so that they can access it programmatically is a great reason to build an API (and the foundation of movements like Linked Data). One key advantage of making data publically/programmatically accessible is that other people can do amazing things you’ve never imagined. That said, you should make a realistic assessment up front as to whether other people really do want your data and how you expect them to use it. Will they have the technical expertise that they need to use your API?

If you can’t name some specific stakeholders who will use your data and what they will use it for, you should probably re-evaluate whether it is worth investing in an API at this time. If you can identify stakeholders, are there alternative approaches to sharing your data that might meet your user’s needs?

Do I have services that involve key technologies or resources that other people will want to use?

Maybe instead of (or in addition to) data, you have services that you want to provide access to (e.g., complex image analysis/OCR technology or special purpose hardware like high-performance computers). The IMPACT project uses web API’s to enable people to use software components that they aren’t willing or able to release as open source software. Alternatively, providing web API access might enable smaller institutions to avoid installing and maintaining complex software packages (such as those that need to run on clustered computers). Amazon Web Services provides access to special hardware via a Web API.

As with data-based APIs, it is important to evaluate realistically who will use these services and why. You should also consider alternative solutions, such as releasing your software as open source.

Do you have users?

Realistically, are people going to use this? Are you sure? Publishing data in an open format is great in principle: you never know what great ideas people might come up with. But publishing data or services as a speculative exercise only makes sense if the costs are pretty low. You can always wait until you get flooded with emails begging for access to your stuff (and then respond by begging for money).

How will the external stakeholders be involved in the API design process?
Stakeholder involvement is critical to making sure that the services you build will be useful to the community that you intend to serve. How will you inform your stakeholders of your project and solicit feedback, feature requests, etc. from your stakeholders?

**Are there other similar systems that I can learn from and/or use?**
Reinventing the wheel is smart, if you are trying to learn about wheels. Otherwise, it is best to look around first and see what other people have done and what you can learn from their work (good and bad). If possible, combining efforts, reusing open source (or otherwise permissible) prior work, adopting existing API components is a great way to lower the amount of work that you have to do yourself. The point of the programmable Web is that you can reuse and add value to work that other people have done. Keep in mind, of course, that anytime you rely on others their actions can potentially break your work. Be sure to make smart, informed choices when introducing external dependencies.

**Do I have the software development support needed to create a secure, reliable API?**
You are going to be exposing your data (possibly including methods to update your data) on a World Wide Web full of bad people. You are also going to be offering a service to other people who will be depending on you for their needs. Do you realistically have the technical expertise that you need to make sure that your system is reasonably safe from secure from malicious (or accidental) abuse, that any data you store is safe, that privacy concerns can be addressed and that you can meet your stakeholders needs for reliability, availability and performance? Have you communicated the level of reliability, performance, security, privacy, etc. to the people who will be using your service? Will you have access to this technical expertise as long as you need it to maintain your system?

**Do I have an adequate hosting environment for hosting a reliable API, given my expected user base?**
Web API’s are available to the world. While many (probably most) will be lightly used, it is possible that your service will attract significant attention. Do you have the server hardware and internet bandwidth that you will need to respond to realistic use scenarios? Can you add more resources later if needed? Can you upgrade your hardware over time?

**Do I have adequate funding to develop this API?**
Based in part on the previous questions, it is time to sit down with a budget. API development can be more expense and require more technical expertise than most other forms of development. Does your project have (or can you justify and obtain) funding for this work?

**Do I have adequate funding to maintain this system?**
Your service will require ongoing resources in the form of hosting, bandwidth, maintenance and (possibly) feature enhancements. Have you realistically assessed these ongoing costs and found a funding model that will cover them?

**Can I (and/or my institution) support this API (hosting, code maintenance, bug fixes, new features, etc.) over the useful lifetime of the project?**
This combines several of the previous questions (technical expertise, hosting, funding) and adds in the complexity of institutional priorities and commitments. If all goes well, people will use your stuff but can they trust you to be there when they need you? Is your institution going to change its priorities and pull support (funding, staffing, permission) from you? As people (e.g. server admins) leave the project, can you hire and train new staff?

**Have you received approval of made agreements with the appropriate stakeholders?**

As always, it is best to get these things in writing to make sure that everyone is on the same page. This is especially true for anyone whose work will be on the critical path of your development process. This is also a good time for an external review. For example, if you think you need 2 servers (4x1.8Ghz CPU, 16 GB RAM) to host this, has the IT staff looked over this requirement?