API for Project Educational Tools

Andy Oram

31 January 2009

This document offers a design for the educational software described in the following article :

> http://radar.oreilly.com/archives/2008/02/developing_an_i.html
> Developing an improved online environment for educating computer users

The purpose of this environment is to help people contribute information to computing projects and to help other people find that information more easily.

Please comment on part or all of this document by writing to me at andyo@oreilly.com. You can put your comments in your email or use revision control (change tracking) to show changes and add suggestions, and send me the resulting document. Each version of the document has a unique filename containing the date of its publication, to allow references to particular versions of the document.

Rationale

Most small software projects do a poor job of educating project members. Common problems include the following:

- Most new projects lack documentation, which makes it hard for the project to spread beyond an inner circle of illuminati.
- When documentation is added, it tends to be inadequate. The first efforts are usually reference documentation without good definitions or context, and tutorials that are aimed at people who are already expert in the field and have used similar systems in the past. This may widen the base to first adopters, but still shuts out a wide range of potentially interested project members.
- On forums, the same questions get asked repeatedly. Although the answers are archived and sometimes incorporated into FAQs, there are always new features, bugs, and popular applications of the software that introduce new periods of dislocation where information is hard to find.
- Popular projects build up a large collections of documents, but they tend to rehash similar topics and leave large information gaps. Authors tend to write about topics that interest them personally, without knowing what information is more urgently needed by project members.
- Because information is contributed in small pieces by volunteers in diverse forums, it becomes fragmented and hard to combine.

A well-organized environment can increase the amount of information while decreasing information overload. People will contribute more when they see that the information is used well. The environment also provides several tools for putting information in a form that makes it easier to edit and update.

Once tools are created to support this environment, projects can bring their members up to speed faster, expose members more fully to the projects' power, and open the projects up to new audiences who had trouble understanding how to use them before.

I intend this document to be a springboard for discussion. You can read and comment on each section independently:

Use cases: This short section shows the purpose of the design and how people are expected to use it. You can offer useful comments on this material without reading the rest. More background is available in the article listed at the start of this document, along with companion pieces at *http://www.praxagora.com/community_documentation*.

Client/server API: This is the bulk of the document, suggesting a RESTful API that can support the system. It should be read by people with some knowledge of software design.

Complications: In order to keep the “Use cases” section short and focus on the positives, most of the difficulties I've encountered are dumped into this final section. This section should be a hot spot for discussions of the system's viability, and can be read by anyone who wants to look at the potential pitfalls of the system.

Use cases

Project education tools support people's desire to contribute online information on computer projects, and to find that information. Typical activities include:

- Asking questions
- Writing and editing answers
- Promoting an answer to a more comprehensive document
- Tagging questions, answers, and other documents
- Searching
- Recommending related documents (cross-references)
- Rating documents and recommendations
- Adding quizzes
- Exchanging information about document quality, as statistics about ratings and quiz results
- Deleting or combining these elements

The people who perform all these tasks are called *contributors*. It may seem inappropriate to apply the term to something as simple as asking a question, but in fact asking a question in a community-based system can be a very valuable contribution. By asking the question, a novice shows that information is missing from the system and mobilizes others to provide the information. The API in this document allows the question to be the groundwork for answers that are useful and easy to find.

Of course, many forums suffer from having novices post the same question over and over, but the system described in this document should make that less frequent.

The software consists of:

- *Clients*, the working environment for contributors. A client can be an IDE, a text editor, a web page hosting a forum or wiki, or some web page running a script.
- *Servers*, which store and display the information. Good candidates for servers include content management systems, wikis, and other documentation systems.

Simple use cases follow. Because life doesn't always remain simple, the final “Complications” section of this document lists numerous sources of failure and difficulty.

Along with use cases, the following subsections offer suggestions for the implementation of the client and server. It would be difficult to understand the operation of the protocol without a sense of how the client and server are likely to behave internally. These suggestions are also intended to allow clients and servers to offer the richest possible behavior.

Posing a question

Someone who encounters an error message while trying to install some hardware or software visits a forum devoted to that product to post a question. The web page hosting the forum is the client. The back-end program run by the forum submits the question to a server that posts it on a FAQ.

Server implementation suggestion: The server can send notifications to people on the forum who have indicated a desire to be told as soon as a new question is posed.[1]

Attaching additional information to a question

A programmer having trouble with his code may post a question along with several attachments, such as a set of lines of code containing the problem and an error message generated by running the program.

Client implementation suggestion: The client, which could be the IDE of his choice, might provide an easy graphical interface for pasting different attachments into windows. For instance, the client might allow the contributor to open two windows for the question (the *source*) and the error message or code (the *target*), select a range within the source, drag the cursor to the target, and click again in the target to select a range. In this way, someone could connect particular parts of his question to specific words or lines in the code or the error message.

Server implementation suggestion: The server can display the results with highlighting and a line.

Answering a question

A contributor is notified that a question was posted to a forum she subscribes to. After running the code from the question in the IDE of her choice and finding some answers, she posts an answer in a box provided by the IDE. The corrected code could be submitted as an attachment.

Version information

Most questions apply to specific versions of software or other products. For instance, a bug may apply to a particular version of an application and even be affected by its interaction with a particular version of the operating system, a database, etc.

Version information can be added by the person posting any document, including a question or answer. The version information can also be added or updated later by anyone. The maintainer of the site is responsible for checking the validity of version information and removing inappropriate changes, as in any system that accepts reader-generated content.

Server implementation suggestion: Forms for posting questions should include fields where people can indicate what software or other products are involved, and which version they are using.

Client implementation suggestion: Fields for submitting version information can indicate a range of versions, such as 2.0 through 2.4.5.

Editing an answer

A programmer seeing the first answer may decide to add additional information or correct an error.

1As an enhancement of this process, the web client or server might extract keywords from the question, search the FAQ or another archive, and display the results before allowing the contributor to post the question. This is not part of the system in this document, but could save a lot of time and prevent many duplicate questions. The length of the text in a question may make such searches unfeasible, however; the operation would be more of a correlation between documents than a search.

Furthermore, contributors may update the version information.

Promoting an answer to a more comprehensive document

Project leaders notice, in reports generated from their log files, that one of the answers in their FAQ is receiving a lot of visits. They change the FAQ to point to a stand-alone web page and assign an author to rewrite the answer for that web page, straightening out the document's organization[2] (which is disjointed because several people contributed to the answer independently) and adding some more background.

Tagging documents

Tags are ubiquitous on user-contributed web sites. In this system, anyone posting a question or other document can label it with a recommended audience (such as “DBA” or “Oracle DBA”) and with topic tags (“Oracle,” “Java,” and so on). Tags can be added when a question or other document is created or at any later time, and can be added by anyone, just like the version information discussed earlier. Audience tags can divide the audience in a fine-grained manner (for instance, “Java programmer new to Ruby”).

Tags can be included in lists, which can be nested. Thus a tag list for “Databases” might include a tag list for “DBA” which in turn contains individual tags for “DBA,” “Oracle DBA,” “PostgreSQL DBA,” and so on.

Server and client implementation suggestion: Tags should be consulted by search tools that access the sites. Each site can show readers the existing tags in use, and suggest a list of tags based on what the project leaders know of their site members, in order to standardize use of tags. In order to promote consistent tagging, sites can present contributors with lists of suggested tags.

If consistent sets of tags or audience descriptions appear on many recommendations, automated spidering tools (not covered in this document) could show readers pathways through documents, creating something like virtual books out of many documents by different people. For more information on cross-references, see:

http://radar.oreilly.com/2008/01/two-tools-we-need-to-improve-o.html#cross_reference

Searches

Tools for search are outside the scope of this document, but it does offer fields that help to make searches more effective:

- Versions of software covered by a document
- Audience tags, discussed in the previous section
- Topic tags, discussed in the previous sections
- Starting and ending dates, reflecting the earliest and latest date that the document should have in order to be returned by the search (each document bears the date of its most recent revision)

2 It's worth harking back to the famous article in *Nature* comparing the quality of the Encyclopedia Britannica with that of Wikipedia. There is much disagreement over the accuracy of their factual comparisons, but no one has denied one little-noted aspect of the *Nature* study: it said Britannica articles were better organized and edited, and therefore easier to read, than Wikipedia articles.

Results of the ratings and quizzes discussed later can also be used by sites to rank the documents returned.

Recommending a related document

Server and client implementation suggestion: Each question or other document should be accompanied by a form for suggesting related documents.

Recognizing that an answer may use terms and concepts that are unfamiliar to some readers, a contributor uses this form to provide a name and URL of a document that could be read before the current one, along with a list of tags and a description of the audience that would benefit, using the system described earlier under "Tagging documents."

Rating documents and recommendations

Servers can provide rating systems, although I recommend they be supplemented by quizzes, which provide more reliable feedback about quality. Documents, suggested recommendations, and answers to questions can all be rated by readers.

In particular, document owners can use ratings to indicate whether they agree with recommendations for related documents. A recommendation that readers of document *A* also read document *B* has a neutral rating until the owner of document *A* weighs in; that owner should be able to influence the weight assigned to that recommendation. Although the owner should be able to remove a recommendation entirely (because the recommendation system will be a magnet for spam), owners should be discouraged from removing relevant recommendations; they can be downgraded through the rating system instead.

Adding a quiz

To find out better whether a document is meeting the needs of the readers, the author decides on a key concept that the reader must learn from each page, and asks another project leader (who is more likely to have the necessary distance from the document) to write a question to put on each page. After showing the questions to some representative members of the audience for review, the author adds the appropriate question at the bottom of each page. The readers receive instant feedback on whether their answers are correct, and a database collects answers to show the success of the document.

A prototype quiz development program suggests that implementation would not be difficult:

http://www.praxagora.com/quiz/

The application is a crude prototype. It creates a web page that looks like a quiz, but does not actually allow people to take the quiz. It also does not allow you to reuse a login; it allows only new logins.

For more information on quizzes, see:

http://radar.oreilly.com/2008/01/two-tools-we-need-to-improve-o.html#quiz

Exchanging statistics about ratings and quiz results

Project leaders and contributors will be interested in the popularity of material at other sites, and at the ratings assigned by users or quizzes. Sites may therefore collaborate to exchange information about

how many people rated docments or took quizzes, and the results. Central sites may accumulate such statistics from multiple projects to produce comparisons: who offered the best learning experience this year? Which documents were read the most and got the most positive ratings?

Deleting entries

Each week, a project leader goes through the FAQ to remove inappropriate questions and combine duplicates.

Client/Server API

This API is intended to be flexible enough to be implemented in many different client and server environments, and therefore in many programming languages. The design attempts to satisfy RESTful principles by exposing resources in URIs and controlling operations through HTTP methods.

The design is divided into modules. So long as the core is provided, any number of the other modules can be supported independently:

- Question
- Tag
- Search
- Recommendation
- Rating
- Subjectversion
- Quiz
- Search

Under each module is a set of methods for creating and manipulating each resource it provides (question, answer, etc.). Each method includes parameters in the body of the request, some of which may be required and some of which are optional. The descriptions in this document show the information passed in XML, but the actual format depends on the HTTP Content-Type header.

Each resource is given its own URI by the server. Although the URLs for some resources are valuable to readers—who may, for instance, link to questions and answers from other parts of the Web—some resources bear URLs simply to serve as handles for subsequent reads and deletions.

Because the URL defines a stable resource, the parameters that appear as elements in the path are those that distinguish the resource from others, notably the version of the resource.

Like the “Use cases” section, this sections offer suggestions for the implementation of the client and server.

Methods

The API uses standard HTTP methods. All four methods are available for every resource, but this document usually describes only the POST method because everything needed for PUT, GET, and DELETE can usually be derived from the information under POST.

In this document, we use the shorthand `http://entry-point` to denote the entry-point provided by the server, and the shorthand `http://entry-point/resource` to denote some resource that has already been created.

POST

Creates a new resource. Some fields are required; others are optional. Every call includes an optional field to specify the person issuing the call; this ID becomes the creator of the resource. The response contains all the information passed in the POST, along with a version number that can be

used in subsequent methods.

URIs for POST methods are simple and reflect the parent-child relationship among resources. For instance, assume that a project provides the entry point *http://project.example.com/faq_installation.* A question is posted through the URI:

http://project.example.com/faq_installation/question

Assume that the server stores the question at the URL:

http://project.example.com/questions/faq_installation/questions/#q87

An answer can then be posted to:

http://project.example.com/faq_installation/questions/#q87/answer

The answer is considered a child of the question in a tree of resources, so the URI in the answer's POST method begins with the URL of the question.

The URIs in POST commands, as is typical in RESTful web services, refer to operations on the server rather than actual documents. Thus, there is no actual document at *http://project.example.com/faq_installation/question* or *http://project.example.com/faq_installation/questions/#q87/answer*. However, *http://project.example.com/faq_installation/questions/#q87* points to a real document that any browser or other HTTP-aware program can retrieve.

PUT

Replaces an existing resource. This can be useful in many situations: editing a question to add new material, replacing a picture or other attachment, updating version information, correcting a typo in a recommendation, and so on.

The URI for a post always consists of the name of a resource followed by the current version number:

`http://entry-point/resource/versions/version`

For instance, to replace the question at *http://project.example.com/faq_installation/questions/#q87*, if the version number is 1, issue a PUT to *http://project.example.com/faq_installation/questions/#q87/versions/1.*

Unless otherwise specified in this document:

- All the fields in the body of the PUT and in the body of the response are the same as for POST.
- All the fields in the PUT are optional, because the server can reuse existing parameters for any omitted fields.
- The requester in the PUT, if specified, becomes the creator of the resource.

GET

Returns all the information stored by the server about a resource. Because many resources point to child resources—for instance, a question is the parent of an answer, audience tags and topic tags—the GET can ask the server to recursively check those resources and return information on them as well.

All GET methods take the following syntax:

`http://entry-point/resource/versions/version`

The version parameter is optional. If it is omitted, the current version of the resource is returned. The fields in the response are formatted the same as for POST.

DELETE

Removes a resource and decrements the reference counts on the children so the server can remove those as well. All DELETE methods take the following syntax:

`http://entry-point/resource/versions/version`

The version parameter is optional. If it is included, only the specific version is deleted; otherwise all versions are deleted.

If the body is included, it is used just to pass a `requester` parameter. No body is returned in the response; the Status field can indicate the result of the request.

Text, attachments, and other files

The core API, which allows clients to create files of various types on the server, is small but covers a wide range of activities: submitting articles, audio files, or videos about a project; uploading screen shots that illustrate a problem; uploading sample source code to illustrate a procedure; and many other possibilities.

To understand the relationships among resources, imagine that somebody asks a question about an error message and includes some sample code and a screen shot showing the message. The client software would post the question, then issue two separate posts to create a file of sample code and a file containing the screen shot. Two more posts would create two connections: one between the question and sample code and the other between the question and screen shot.

References and connections

Many resources in this environment have meaning in respect to other resources; this document treats them in parent-child relationships. For instance, answers are children of the questions they answer; recommendations are the children of files, questions, and answers; endorsements, audience tags, and topic tags are the children of many resources.

This API therefore includes a `reference` field in resources that can refer to others. The resource making the reference is the child and the resource to which it refers is the parent. Multiple references are allowed, although most resources will probably be the child or only one other resource.

However, another type of connection can be made between resources, usually because one file is attached to another. If a contributor points to a particular segment of code or an element of a screen shot, the client posts a connection and the server indicates the connection in some visible manner, such as a line across the screen connecting the source to the target.

Usually, the context in which the contributor creates the attachment makes it clear to the client which document it should be attached to. For instance, a contributor may pull up a new window to insert an error message or source code while writing a question; the client should create the connection behind the scenes. The client can create a Connection resource to connect the attachment to the document. It's conceivable that contributors will desire a one-to-many relationship for an attachment, and the separation in this API between creating the file and creating the connection allows one-to-many relationships.

More discussion can be found in the “Complications” under *References versus connections*.

Identifying documents and positions within documents

The URL format is quite powerful for the purposes of this API, when expanded with its fragment syntax (the part following a “#” character) and XPointer's string ranges. Of course, using URLs leads to the classic Web problem of broken links, which will be described in the “Complications” section of this document under *Missing resources*.

A typical FAQ question, posted to a FAQ at *http://project.example.com/*, might be assigned the following URL by the server:

http://project.example.com/faq_installation/questions/#q87

The question might point to a code in an attachment that the server gives the name *attachments/a24*. Suppose the server represents that code in `<pre>` HTML entities. If the contributor highlights the first instance of the string “SiteLogin” in the code, it might be associated with the following XPointer:

http://project.example.com/attachments/a24#xpointer(string-range(//pre, “SiteLogin”)[1])

Client implementation suggestion: As described in the “Attaching additional information to a question” section under “Use Cases,” client software should provide tools that allow contributors to easily select ranges and connect them with lines.

Clients often create multiple resources, and must refer to one resource when creating a reference to it in another resource. For instance, a recommendation may be created along with a reference to its parent question, and an audience tag may be created with a reference to its parent recommendation. Wherever possible, this API allows a single call to create multiple resources, but the problem still remains of how the client can refer to them before the server creates them and assigns permanent URLs.

To solve this problem, the client assigns temporary URLs to the resources in these methods so that the server can tie them together. The server recognizes that they are temporary and have no meaning outside the transaction because they are relative URLs. The server is thus free to assign any permanent URLs it wants. If a `reference` field in a client call refers to some other URL on the Web, the client passes an absolute URL that should not be changed by the server. The “Complications” section discusses the processing of related calls in more detail.

Data storage and client reads

The server should store all the parameters passed to each POST or PUT method, along with the URL it returns. Thus, the POST method for a file causes the server to store the URL it creates for the document, the content of the document, the format (such as JPEG), the ID of the creator, and the current version. The server returns all this information in the response to a POST or PUT, and to any subsequent GET.

Server implementation suggestion: To traverse the trees of parent and child resources, the server should maintain a tree of related resources as metadata for each resource, and be prepared to return this tree to the client in response to GET calls.

The most complicated set of resources in this API is the quiz. A single call to create a quiz may contain a list of questions, and each question may contain a list of choices for the reader answering the questions. The metainformation also indicates the order of the questions, the order of the choices within

each question, and which choice is correct.

Here is an example of a possible chain of parent and child resources, starting with the earlier example of the question at *http://project.example.com/faq_installation/questions/#q87*:

1. A GET directed to *http://project.example.com/faq_installation/questions/#q87/answers* returns a list of answers (which probably contains only a single item) stored in an `<answers>` entity. If the GET requests a recursive search, the answer contains a list of recommendations for related documents in a `<recommendations>` entity. See the upcoming sections "Lists" and "A sample interaction" for the format of the response.
2. The answer itself may be at *http://project.example.com/faq_installation/questions/#q87/answers/answer#1*. Perhaps the first GET did not ask for recommendations because the reader did not show an interest in them. When the reader subsequently asks for recommendations, the client can retrieve them by issuing a GET to *http://project.example.com/faq_installation/questions/#q87/answers/answer#1* and including an empty `<recommendations>` entity in the body. The server then returns full information about each recommendation that is a child of that answer. If the client includes `<endorsements>`, `<audiences>`, or `<topics>` entities within the `<recommendations>` entity, full information on the requested resources is also returned.

In short, a client can usually get all child resources in a single GET. But it may be more efficient to ask for just the parent (top-level) resources. Thus, the client can quickly read a question and answer and display them while requesting the child resources. The client may also choose to defer requests until the reader needs the resource. If the reader doesn't ask to view recommendations, for instance, the client can avoid fetching all the related resources.

Web services can return an arbitrary number of values of different types, and can accommodate the complex data structures returned by the GET calls. Because numerous formats exist for marshaling data and exchanging it among systems (for instance, XML and JSON), this API assumes that implementations can set and get the HTTP Content-Type header and transparently support different formats.

For the sake or brevity, this document shows only one format, XML, for the body of requests and responses. Elements can appear in any order within the XML body, so long as nesting follows the rules in this document.

Creators and owners

The environment should identify contributors so they can be recognized and rewarded (which usually consists of being asked to do more work). Putting an owner on each contribution also provides some rights; the site may put items on an owner's watch list and notify the owner when the item is changed; the owner may also have final say in case of disagreements. When recommendations are added to a document, the owner of the document can elevate or downgrade the value of the reference.

I do not recommend putting locks, a simple form of "technical measure" used to enforce ownership rights on content. All too often, an author goes 404 or becomes too busy to update content, and the content metamorphoses from a generous gift to an annoying burden on project leaders.

Numerous identity systems are emerging on the Internet; OpenID currently seems to be the winner. It may be necessary to support several forms of identification, though, including a simple email address. This document assumes that any identification system used can be passed robustly among

implementations in different programming languages.

Each method in this API allows the client to pass the ID of the person requesting the information in a `requester` field. Thus, the system opens up a way for servers to limit access to some kinds of information. Because this environment is intended for use in open information exchanges, it's more likely that a server would simply log the persons making the requests for usage analysis. If someone overloads the server with too many requests, the requester field can be used to reject requests, although this is not a robust form of denial of access.

This API always makes the requester field optional in order to encourage system to allow anonymous contributions. The impacts of anonymity on the system will be described in the "Complications" section under *Login and ID management*. However, clients and servers can implement the methods to determine the requester (and the owner of a resource) from the environment in which the methods are called. Any site where contributors log in can pass their IDs to all the functions, although it can also provide a "Post anonymously" button.

Deletions and clean-up

Any system with parent and child resources such as this one faces a clean-up problem; the solution proposed here is the traditional approach of recursively searching for resources and deleting them in the reverse order from which they were created.

For instance, a set of resources could typically be built up as follows:

1. Post question
2. Post answer
3. Post recommendation

These can be added at different times by different people, or all at once by a single method. If the question is later deleted, the client should be able to simply issue a DELETE method to remove the parent question and to assume the child resources will also be deleted.

Server implementation suggestion: Upon receiving the request, the server should discover the child resources and issue calls in the following order:

1. Delete recommendation
2. Delete answer
3. Delete question

The client should not be responsible for discovering the relationships between resources, because many other clients can add more answers, attachments, recommendations, connections, and other resources. This in another reason (in addition to answering GET calls) that the server must discover the relationships by including a tree of related resources with each document. The implications will be described in the "Complications" section.

Note that the "Delete" operations are actually decrements of reference counts, because many references may exist to a single resources. This is discussed in the "Issues deserving further discussion" subsection of the "Complications" section, under *Missing resources* and *The complexities of resource management*.

Software (subject) and document versions

Because the features, bugs, and other usage issues of software and other subject matter can change radically from one version to the next, each document should contain meta-information showing which versions of the subject matter it covers. This capability is provided in this API by a resource called Subjectversion.

Client implementation suggestion: A client should provide a contributor with a box where the contributor can enter various subjects and version numbers. Simple examples of use cases include:

2.3.4	A contributor might enter a single version number to indicate the version he's struggling with. The server might display this information as the range 2.3.4–2.3.4, just to be consistent with the syntax for other ranges.
2.3.4–current	This syntax is common when a new feature was added in 2.3.4 and is expected to persist over subsequent versions.
beginning–2.3.3	If a new feature is introduced in 2.3.4, all previous versions can be denoted by this range.
2.3.4–2.3.8	This syntax will be common for bugs that appear in one version and are eventually fixed. (The fix presumably is in 2.3.9, leaving 2.3.8 as the last version to which the answer applies.)

Tracking the versions of the subject matter is entirely different from tracking different versions of documents and other resources. Each resource has a version number that the server increments after every PUT. This prevents race conditions, which are discussed in the following section.

Documents can have versions too. An author may use meaningful version numbers such as 1.1 and 2.0 to show when a document has undergone a significant edit. But that is outside the scope of this API, and is discussed in the article cited earlier:

http://radar.oreilly.com/2008/01/two-tools-we-need-to-improve-o.html#quiz

Version numbers are by no means orderly and well-formed, unfortunately. The “Complications” section looks at some of the difficulties of indicating versions, and concludes that this environment should try to provide only modest support for simple cases.

Exchanging suggested tags

Consistent tagging vastly improves searches and the productivity of spiders that create lists of recommended documents. If two people label two different documents for “Java novice,” or create a recommendation with such an audience tag, the documents can show up in the same searches and be incorporated into a diagram showing related documents. If one person uses the tag “Java novice” and the other uses “Java newbie,” the system would have trouble seeing that the documents are for the same audience. See the section *Providing consistent tag sets* under the “Complications” section for more discussion.

Server and client implementation suggestion: To promote consistent tagging, servers and clients can suggest audience and topic tags to contributors. This API provides the Taglist for that purpose. Taglists can be nested.

Exchanging statistics

As explained under “Use cases” in the section *Exchanging statistics about ratings and quiz results*, this system allows sites to exchange statistics concerning how many people have rated documents and answered quizzes, along with results. The exact calls used to exchange this information are outside the scope of this document. The collection and sharing of statistics is a huge subject with many variations. For the sake of efficiency and to protect privacy, sites may aggregate statistics or scrub information in certain ways, such as removing timestamps and leaving only date.

This document is concerned with providing data for each resource and activity to facilitate the sharing of statistics. In particular, each resource bears a timestamp that reflects the most recent change to the resource. Ratings and quizzes bear timestamps.

API conventions

This API strives for consistency through the following conventions:

- Parameters that indicate distinct resources have distinct URIs. Thus, parameters that can differentiate resources, notably the version parameter, appear in the URI. Other parameters, which don't differentiate resources, appear in the body.
- Parameters are listed as optional whenever omitting the parameter could lead to a valid result. If a parameter in the body of a client's request is optional, the descriptive comment says so. One optional parameter, the version number, can be listed in the URI of a GET or DELETE operation because it denotes a unique resource.
- A list is returned in response to GET methods that asks for a list: for instance a GET that requests all recommendations attached to a document. No other lists are used in this API, because each call normally requests and returns one of each parameter. However, in some cases, a method can pass more than one instance of a parameter.
- Consistent names are used to simplify coding. For instance, each item of a list is returned enclosed in an `<item>` entity. The contents of such entities are also consistent (for instance, a resource is always a URL).

Editing as a two-stage process

Statelessness is seen as a good policy for this system, but it does leave a window for conflicting edits to a document. Clients should offer an Edit operation for contributors' convenience, but it will be implemented as two separate operations in this API: a GET followed by a PUT. In order to prevent conflicts when two people try to update a resource at the same time, the server assigns a new version number each time a resource changes. The version number can be arbitrary and should be treated by the client as opaque, see the section *Updates and version numbers* under the “Complications” section for more discussion.

Every POST, PUT, and GET method returns the current version. Although a GET can request a particular version, most clients will omit the `version` field in order to get the server's current version by default.

A PUT method must specify the current version of the resource as a field in the URI. The server must reject the PUT if it specifies a different version from the current one. The outcome will be that

someone who issues a GET, changes the resource, and then issues a PUT will have the change rejected if anyone retrieved and has changed the resource in the interim.

A misbehaving client could request a PUT on a higher version number than the one it received in the GET and try to undermine the version control system, but this is not a unique security problem because it would be no different from any other destructive operation on an existing resource.

If this discipline is followed, the result will be similar to how wikis handle conflicts by simply rejecting an edit if someone has changed the resource during the edit. Source control systems are slightly gentler: they report conflicts and put the burden on the later contributor to resolve them.

Server and client implementation suggestion: Servers might be advised to offer information about conflicts, and clients could then offer editing tools that make it easier to view and resolve the conflicts.

The problem of conflicts is thus a well-known issue, and is not addressed futher in this document.

If documents are stored in XML, a client can request an arbitrarily small part of the file. Most likely, the server will allow a section to be retrieved, or perhaps individual paragraphs. See “Complications” for further discussion.

It might be worth adding a callback parameter to each “Edit” function so that the client can warn a contributor when someone else tries to edit the resource. This would beg the question of how to handle the conflict proactively. Postponing the problem until check-in time is much simpler and probably just as effective.

URI fields

The following fields appear in the paths of resources in this API. Fields that the client fills in with values are in *`italic`* font to distinguish them from fields that are typed exactly as shown here. Thus, in the following URI for manipulating a question:

`http://entry-point/question/versions/version`

the client fills in the base URI of the server's entry point (*`http://entry-point`*) and the version requested.

Fields in body

Many fields in the bodies of POST and PUT methods are simply resource names (`files`, `questions`, `answers`, etc.). Some GET methods return resources that are children of the resource passed in the URI: for instance, a GET on a question can return a list of answers, a list of recommendations, and a list of connections attached to the question.

Relative versus absolute URLs

All resources get URLs on the server. The client passes a relative URL to refer to other resources that the client is sending before or after the current resource. The server assigns the permanent URLs after storing the resource to which it refers, and assigns a new URL for the reference on the server. An absolute URL is used by the client to refer to refers to some other document on the Web. The server should not change an absolute URL. See the earlier section “Identifying documents and positions within documents” for more background.

Special request fields

Certain fields can appear in the bodies of multiple requests, in addition to fields that have particular meanings for particular resources:

`reference`

Parent resource to which this resource refers (for instance, an answer contains a reference to the question it's answering)

`requester`

The ID of the person on whose behalf the client issues the request

`source`

Originating end of a connection

`target`

Terminating end of a connection

`timestamp`

Timestamp in the standard ISO 8601 format *(YYYY-MM-DD*T*hh:mm:ss)* reflecting the most recent change to the resource. Searches can request starting and ending timestamps, so that searchers can put limits on the age of resources returned.

`version`

Specifies a particular version of the resource; generally used by the client on PUT methods

The first field in every method (and the only field in a DELETE) is the optional `requester` parameter. See *Login and ID management* under the “Issues outside this document's scope” subsection of the “Complications” section for more information.

Special result fields

The following fields can turn up in results, in addition to fields that have particular meanings for particular resources:

`creator`

ID of the requester who created or last changed a resource

`item`

Information on a single resource in a list

`resource`

The URL of the resource being described in the response

`tags`

Encloses a list of tags in Audience and Topic resources

`taglist`

Encloses a list of suggested tags for the server to return to the client; can be nested

`timestamp`

Timestamp in the standard ISO 8601 format *(YYYY-MM-DD*T*hh:mm:ss)* reflecting the most recent change to the resource.

`value`

For text resources, the content of the resource. For ratings, a floating-point value from 0 to 1 inclusive

`version`

A version number that changes with each change to a resource. Submitted by the client on subsequent PUT methods

Lists

Lists of child resources are returned to GET methods. If no fields are requested in the body of the GET, it returns just the URL of each child resource. For instance, a GET for a question may return:

```
<answers>
  <item>
  <!-- URL of an answer to the question -->
  </item>
   ...
</answers>
<recommendations>
  <item>
  <!-- URL of a recommendation attached to the question -->
  </item>
   ...
</recommendations>
...
```

Only the URL for each resource is returned, but full information on particular types of resources can be requested in a GET method by including an entity named after the resource in the body. For instance, if the GET passes a `<recommendations>` entity in the body, the server returns all the metainformation about each recommendation. This in turn will include the minimal information that recommendation includes on each embedded child resource (audience tags, endorsements, and topic tags). As just shown, this minimal information consists of the child's URL.

Furthermore, the body of the GET may request children in nested fashion. This is shown in the following section

Searching

Searches are sent from the client to the server. This API therefore covers only the format for the information exchanged between client and server. Search tools are outside the scope of the API.

Currently, the API supports only a flat list of text terms, not Boolean operators. The API could be extended to allow Boolean operators if the servers can take advantage of them.

Server implementation suggestion: Until the API is extended, servers can do what most search sites do and interpret certain terms (OR, NOT) as Boolean terms to control the search.

The API does allow version information, audience tags, and topic tags to be collected by the client and passed to the server.

Client implementation suggestion: The client will probably require the searcher to specify the URLs of servers supporting this API. Perhaps when someone downloads software from the server, the client can suggest adding its URL to a list of search sites, so that searches can easily be directed to the server later. Thus, the client may build up a list of servers from which the person has downloaded software and present a list of the servers as part of the form for a search.

Server implementation suggestion: It is hoped that the server can use the results of ratings and quizzes to rank search results.

A sample interaction

This section shows what the bodies of some client requests and server responses look like. It uses XML for the content type.

First, a very simple request. A contributor with the ID *joe@example.com* creates a question "Is there any special handling I need for Python to handle RDF?" The body of the POST is:

```
<requester>
joe@example.com
</requester>
<value>
Is there any special handling I need for Python to handle RDF?
</value>
```

The server creates the URL *http://project.example.com/faq_installation/questions/#q87* to display the question and returns the following response:

```
<resource>
http://project.example.com/faq_installation/questions/#q87
</resource>
<version>
1
</version>
<creator>
joe@example.com
</creator>
<timestamp>
2009-01-15T07:14:02
```

```
</timestamp>
<value>
Is there any special handling I need for Python to handle RDF?
</value>
```

A contributor with the ID *sally@example.org* posts a recommendation to the document “Building Metadata Applications with RDF” located at *http://www.xml.com/pub/a/2003/02/12/rdflib.html*. The client posts to *http://project.example.com/faq_installation/questions/#q87/recommendation* and includes the followng body:

```
<requester>
sally@example.org
</requester>
 <reference>
http://project.example.com/faq_installation/questions/#q87
</reference>
<value>
http://www.xml.com/pub/a/2003/02/12/rdflib.html
</value>
<title>
Building Metadata Applications with RDF
</title>
<when>
any
</when>
```

The server creates a place with an arbitrary URL for the recommendation; let's suppose it's *http://project.example.com/faq_installation/recommendations/#rec1*. The server returns:

```
<resource>
http://project.example.com/faq_installation/recommendations/#rec1
</resource>
<version>
1
</version>
<creator>
sally@example.org
</creator>
<timestamp>
2009-01-15T09:42:55
```

```
</timestamp>
<reference>
http://project.example.com/faq_installation/questions/#q87
</reference>
<value>
http://www.xml.com/pub/a/2003/02/12/rdflib.html
</value>
<title>
Building Metadata Applications with RDF
</title>
<when>
any
</when>
```

Although Sally could have included an endorsement, audience tags, and topic tags in this post, let's suppose that she waits and adds the topic tags `Python` and `RDF` later. This requires a post to *http://project.example.com/faq_installation/recommendations/#rec1/topic* with the following body:

```
<requester>
sally@example.org
</requester>
<reference>
http://project.example.com/faq_installation/recommendations/#rec1
</reference>
<tags>
  <value>
    Python
  </value>
  <value>
    RDF
  </value>
</tags>
```

The response is similar to other responses and won't be shown here in order to save space. We'll assume the server creates a topic list at *http://project.example.com/faq_installation/topics/#topic14*.

Now a client performs a GET on the original question, leaving the body empty. The server returns all the metadata associated with the question, which includes just the URL of the child recommendation. The GET does not descend further to check the recommendation's metadata, which would include the topic information.

```
<resource>
```

```
http://project.example.com/faq_installation/questions/#q87
</resource>
<version>
1
</version>
<creator>
joe@example.com
</creator>
<timestamp>
2009-01-15T07:14:02
</timestamp>
<value>
Is there any special handling I need for Python to handle RDF?
</value>
<recommendations>
  <item>
  http://project.example.com/faq_installation/recommendations/#rec1
  </item>
</recommendations>
```

Suppose that a client now issues a GET with a body specifically asking for all the information on the recommendation and child tags. The body of the GET looks like:

```
<requester>
joe@example.com
</requester>
<recommendations>
  <topics>
  </topics>
</recommendations>
```

The response contains all the information returned to the former GET, plus information on recommendations and topics within those recommendations.

```
<resource>
http://project.example.com/faq_installation/questions/#q87
</resource>
<version>
1
</version>
```

```
  <creator>
  joe@example.com
  </creator>
  <timestamp>
  2009-01-15T07:14:02
  </timestamp>
  <value>
  Is there any special handling I need for Python to handle RDF?
  </value>
  <recommendations>
    <item>
  ____<resource>
  ____http://project.example.com/faq_installation/recommendations/#rec1
  ____</resource>
  ____<version>
  ____1
  ____</version>
  ____<creator>
  ____sally@example.org
  ____</creator>
      <timestamp>
      2009-01-15T09:42:55
      </timestamp>
  ____<reference>
  ____http://project.example.com/faq_installation/questions/#q87
  ____</reference>
  ____<value>
      http://www/.xml.com/pub/a/2003/02/12/rdflib.html
  ____</value>
  ____<title>
  ____Building Metadata Applications with RDF
  ____</title>
  ____<when>
  ____any
  ____</when>
```

```
    <topics>
      <item>
        <resource>
        http://project.example.com/faq_installation/topics/#topic14
        </resource>
        <version>
        1
        </version>
        <creator>
        sally@example.org
        </creator>
        <timestamp>
        2009-01-16T015:10:23
        </timestamp>
        <reference>
        http://project.example.com/faq_installation/recommendations
/#rec1
        </reference>
        <tags>
          <value>
            Python
          </value>
          <value>
            RDF
          </value>
        </tags>
      </item>
    </topics>
  </item>
</recommendations>
```

API for core resources

File

POST

http://entry-point/file

```
<metadata>
  <requester>
    <!-- Optional field: ID of the person making the request -->
  </requester>
  <type>
    <!-- Required field: Type of file (JPEG, MP3, XML, etc.) -->
  </type>
  <title>
    <!-- Optional field: Title to display and
         return in searches -->
  </title>
  <recommendations>
    <!-- Optional field: list of item tags
         containing recommendation information -->
  <recommendations>
  <ratings>
    <!-- Optional field: list of item tags containing ratings -->
  <ratings>
  <subjectversions>
    <!-- Optional field: list of item tags
          containing subjectversion information -->
  <subjectversions>
  <audiences>
  <!-- Optional fields containing audience tags, in the format
       shown under Audience in this API -->
  </audiences>
  <topics>
  <!-- Optional fields containing topic tags, in the format
       shown under Topic in this API -->
  </topics>
</metadata>
...
```

This method can be used to pass many types of content to the server: the contents of a text box filled in by the contributor, a stand-alone file created by the contributor, an excerpt of source code, a stand-alone file such as a graphic, audio recording, or video, etc. The content is passed in the body as a BLOB after the initial fields.

Server implementation suggestion: The server can implement this method by creating a new file or by adding material to an existing web page and assigning it a URL.

Returned fields:

```
<metadata>
  <resource>
    <!-- URL of the created file -->
  </resource>
  <version>
    <!-- Current version of the resource -->
  </version>
  <creator>
    <!-- Requester who uploaded the resource -->
  </creator>
  <timestamp>
    <!-- Date and time that the resource was created,
         in ISO 8601 format -->
  </timestamp>
  <type>
    <!-- Type of file: JPEG, MP3, XML, etc. -->
  </type>
  <connections>
    <!-- List of connections to other resources -->
  </connections>
  <recommendations>
    <!-- List of item tags containing
         recommendation information -->
  </recommendations>
  <ratings>
    <!-- List of item tags containing ratings -->
  </ratings>
  <subjectversions>
    <!-- List of item tags containing
         subjectversion information -->
  </subjectversions>
  <audiences>
  <!-- Information about audience tags for this file -->
```

```
    </audiences>
    <topics>
    <!-- Information about topic tags for this file -->
    </topics>
  </metadata>
  ...
```

The content is passed in the body as a BLOB after the initial fields.

PUT

http://entry-point/file/version

If the PUT is successful, the server discards all references and connections and reduces the reference counts on child resources, so the client must preserve them from the GET operation and specify them again if it desires to keep them. The file type, however, defaults to the current type if the `type` field is omitted.

Connection

POST

http://entry-point/connection

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<source>
  <!-- Required field: URL of source -->
</source>
<target>
  <!-- Required field: URL of target -->
</target>
```

This method creates a connection between a source and a target. For instance, the source may be a question about programming and the target may be an excerpt of source code illustrating the problem. XPointer can be used to create anchor points or ranges within the source and target. The server creates the connection using any graphical or text indicator consistent with its style and interface. The server must also add metadata to the source and target (if they exist on the server) so the connection can be found while traversing them.

Server implementation suggestion: When a connection is deleted, the server should also delete the information about the connection on the source and target if they are on the server.

Returned fields:

```
<resource>
  <!-- URL of the created connection -->
```

```
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
  <!-- Date and time that the resource was created,
       in ISO 8601 format -->
</timestamp>
<source>
  <!-- URL of the source -->
</source>
<target>
  <!-- URL of the target -->
</target>
```

API for question

Question

POST

http://entry-point/question

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<value>
  <!-- Required field: text of question -->
</value>
<recommendations>
  <!-- Optional field: list of item tags containing recommendation
       information -->
</recommendations>
<ratings>
  <!-- Optional field: list of item tags containing ratings -->
</ratings>
```

```
<subjectversions>
  <!-- Optional field: list of item tags containing subjectversion
       information -->
</subjectversions>
<audiences>
<!-- Optional fields containing audience tags, in the format
     shown under Audience in this API -->
</audiences>
<topics>
<!-- Optional fields containing topic tags, in the format
     shown under Topic in this API -->
</topics>
```

Server implementation suggestion: The server will probably add the question to an existing web page, such as an FAQ, instead of creating a separate web page. And a server will probably display will a “Write an answer” button on each question.

Returned fields:

```
<resource>
  <!-- URL of the created question -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
    in ISO 8601 format -->
</timestamp>
<value>
  <!-- Text of question -->
</value>
<connections>
  <!-- List of connections to other resources -->
</connections>
<recommendations>
```

```
<!-- List of item tags containing recommendation information -->
</recommendations>
<ratings>
  <!-- List of item tags containing ratings -->
</ratings>
<subjectversions>
  <!-- List of item tags containing subjectversion information -->
</subjectversions>
<audiences>
<!-- Information about audience tags for this question -->
</audiences>
<topics>
<!-- Information about topic tags for this question -->
</topics>
```

Answer

POST

http://entry-point/question/answer

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<reference>
  <!-- Required field: URL of question being answered -->
</reference>
<value>
  <!-- Required field: text of answer -->
</value>
<recommendations>
  <!-- Optional field: list of item tags containing recommendation
       information -->
</recommendations>
<ratings>
  <!-- Optional field: list of item tags containing ratings -->
</ratings>
<subjectversions>
  <!-- Optional field: list of item tags containing subjectversion
```

```
      information -->
</subjectversions>
<audiences>
<!-- Optional fields containing audience tags, in the format
     shown under Audience in this API -->
</audiences>
<topics>
<!-- Optional fields containing topic tags, in the format
     shown under Topic in this API -->
</topics>
```

Adds an answer to the question. It is usually best to have only one answer (which many people can augment and edit) to each question, but there may be a good reason for multiple answers; for instance, if the answer is substantially different for different subjectversions, there can be different answers bearing different child subversions.

Returned fields:

```
<resource>
  <!-- URL of the created answer -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of the question being answered -->
</reference>
<value>
  <!-- Required field: text of answer -->
</value>
<connections>
  <!-- List of connections to other resources -->
```

```
</connections>
<recommendations>
  <!-- List of item tags containing recommendation information -->
</recommendations>
<ratings>
  <!-- List of item tags containing ratings -->
</ratings>
<subjectversions>
  <!-- List of item tags containing subjectversion information -->
</subjectversions>
<audiences>
<!-- Information about audience tags for this answer -->
</audiences>
<topics>
<!-- Information about topic tags for this answer -->
</topics>
```

API for tag

Audience

POST

```
  http://entry-point/audience
  <requester>
    <!-- Optional field: ID of the person making the request -->
  </requester>
  <reference>
    <!-- Required field, may appear multiple times:
         URL of resource to which this audience tag applies -->
  </reference>
  <tags>
    <value>
      <!-- Text describing the audience -->
    </value>
    ...
  </tags>
```

Returned fields:

```
<resource>
  <!-- URL of the created audience tag -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of resource to which this audience tag applies -->
</reference>
<tags>
  <value>
    <!-- May appear multiple times:
         Text describing the audience -->
  </value>
  ...
</tags>
```

Taglist

POST

```
http://entry-point/taglist
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<reference>
  <!-- Required field, may appear multiple times:
       URL of resource to which this taglist applies -->
</reference>
<taglist>
  <!-- May appear multiple times, and contain nested taglists -->
```

```
    <value>
      <!-- Text describing the taglist -->
    </value>
    <tags>
      <value>
        <!-- May appear multiple times: Text of the tag -->
      </value>
      <timestamp>
      <!-- Date and time of most recent change to resource,
           in ISO 8601 format -->
      </timestamp>
      ...
    </tags>
    ...
  </taglist>
```

Tag lists are more likely to be created on servers than by POSTs from clients. The GET call is the important one for this resource, because clients need to retrieve servers' suggestions for tags in order to display them to contributors. It is not certain that anyone will use the POST, PUT, and DELETE methods, but they are allowed for the sake of completeness. Clients can suggest tags to servers, but it is likely that clients will simply include new tags when submitting other resources such as answers and recommendations. The server will cull the tags and add them to taglists on its own, with intervention by human administrators.

Returned fields:

```
  <resource>
    <!-- URL of the created taglist -->
  </resource>
  <version>
    <!-- Current version of the resource -->
  </version>
  <creator>
    <!-- Requester who uploaded the resource -->
  </creator>
  <timestamp>
  <!-- Date and time that the resource was created,
       in ISO 8601 format -->
  </timestamp>
  <reference>
```

```
    <!-- URL of resource to which this taglist applies -->
  </reference>
  <taglist>
    <value>
      <!-- Text describing the taglist -->
    </value>
    <tags>
      <value>
        <!-- May appear multiple times: Text of the tag -->
      </value>
      <timestamp>
      <!-- Date and time of the most recent change to resource,
           in ISO 8601 format -->
      </timestamp>
      ...
    </tags>
    ...
  </taglist>
```

Topic

POST

```
  http://entry-point/topic
  <requester>
    <!-- Optional field: ID of the person making the request -->
  </requester>
  <reference>
    <!-- Required field, may appear multiple times:
         URL of resource to which this topic tag applies -->
  </reference>
  <tags>
    <value>
      <!-- May appear multiple times: Text describing the topic -->
    </value>
    ...
  </tags>
```

Returned fields:

```
<resource>
  <!-- URL of the created topic tag -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of resource to which this topic tag applies -->
</reference>
<tags>
  <value>
    <!-- May appear multiple times: Text describing the topic -->
  </value>
  <timestamp>
    <!-- Date and time of most recent change to resource,
         in ISO 8601 format -->
  </timestamp>
  ...
</tags>
```

API for search

Search

POST

http://entry-point/search

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<textlist>
```

```
  <!-- List of text strings to search for -->
  <value>
    <!-- Text string -->
  </value>
</textlist>
<subjectversions>
  <!-- Optional field: list of item tags containing subjectversion
       information -->
</subjectversions>
<audiences>
<!-- Optional fields containing audience tags, in the format
     shown under Audience in this API -->
</audiences>
<timestamp_start>
  <!-- Earliest date and time (reflecting most recent change)
       that a resource can have in order to be returned,
       in ISO 8601 format -->
</timestamp_start>
<timestamp_end>
  <!-- Latest date and time (reflecting most recent change)
       that a resource can have in order to be returned,
       in ISO 8601 format -->
</timestamp_end>
<topics>
<!-- Optional fields containing topic tags, in the format
     shown under Topic in this API -->
</topics>
```

Returned fields:

```
<searchresultlist>
  <!-- List of resources found by server -->
  <searchresult>
    <!-- Each field is included only if the server has
         information in that field for the resource -->
    <resource>
      <!-- Required field: URL of the resource -->
    </resource>
```

```
    <version>
      <!-- Current version of the resource -->
    </version>
    <creator>
      <!-- Contributor who uploaded the resource -->
    </creator>
    <timestamp>
    <!-- Date and time of most recent change to resource,
       in ISO 8601 format -->
    </timestamp>
    <title>
      <!-- Title of resource -->
    </title>
    <subjectversions>
      <!-- List of item tags containing
           subjectversion information -->
    </subjectversions>
    <audiences>
      <!-- Audience tags associated with the resource -->
    </audiences>
    <topics>
      <!-- Topic tags associated with the resource -->
    </topics>
</searchresult>
</searchresultlist>
```

API for recommendation

Recommendation

POST

http://entry-point/recommendation

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<reference>
  <!-- Required field, may appear multiple times:
```

```
        URL of document to which this recommendation should
        be attached -->
  </reference>
  <value>
    <!-- Required field: URL of the recommended document-->
  </value>
  <title>
  <!-- Optional field: title of the recommended document -->
  </title>
  <when>
    <!-- Required field: time when recommendation applies -->
  </when>
  <audiences>
  <!-- Optional fields containing audience tags, in the format
       shown under Audience in this API -->
  </audiences>
  <topics>
  <!-- Optional fields containing topic tags, in the format
       shown under Topic in this API -->
  </topics>
  <endorsements>
  <!-- Optional fields containing endorsements, in the format
       shown under Endorsement in this API -->
  </endorsements>
```

It is important to understand the roles of the `reference` and `value` fields. Suppose someone has posted a question at *http://project.example.com/faq_installation/questions/#q87*. Someone who thinks the document *http://cs.example.edu/howto.html* has useful supplementary information adds a recommendation. The client sets the `reference` field to *http://project.example.com/faq_installation/questions/#q87* and the `value` field to *http://cs.example.edu/howto.html*. Note that this distinction maps neatly to all the other uses of `reference` and `value` fields in this API. The `reference` field usually refers to a parent resource of the one being created, whereas the `value` field contains the text or other content of the resource being created.

The `when` field can take one of three values:

`before`: recommendation is best read before the document in the reference

`after`: recommendation is best read after the document in the reference

`any`: recommendation is useful at any time in conjunction with the reference

Returned fields:

```
<resource>
  <!-- URL of the created recommendation -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of document to which this recommendation is attached -->
</reference>
<value>
  <!-- URL of recommended document -->
</value>
<title>
<!-- Title of the recommended document -->
</title>
<when>
  <!-- Time when recommendation applies -->
</when>
<audiences>
<!-- Information about audience tags in this recommendation -->
</audiences>
<topics>
<!-- Information about topic tags in this recommendation -->
</topics>
<endorsements>
<!-- Information about endorsements in this recommendation -->
</endorsements>
```

Endorsement

POST

http://entry-point/recommendation/endorsement

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<reference>
  <!-- Required field, may appear multiple times:
       URL of recommendation to which this endorsement applies -->
</reference>
<value>
  <!-- Required field: text of the endorsement -->
</value>
```

Returned fields:

```
<resource>
  <!-- URL of the created endorsement -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of recommendation to which this endorsement applies -->
</reference>
<value>
  <!-- Text of the endorsement -->
</value>
```

API for Rating

Rating

POST

```
  http://entry-point/rating
  <requester>
    <!-- Optional field: ID of the person making the request -->
  </requester>
  <reference>
    <!-- Required field: URL of document to which
         this rating applies -->
  </reference>
  <value>
    <!-- Required field: floating-point value from 0.0 to 1.0 -->
  </value>
  <audiences>
  <!-- Optional fields containing audience tags, in the format
       shown under Audience in this API -->
  </audiences>
  <topics>
  <!-- Optional fields containing topic tags, in the format
       shown under Topic in this API -->
  </topics>
```

Adds a rating to the specified resource. The rating can range from 0.0 (lowest rating) to 1.0 (highest rating).

Server implementation suggestion: The server should store the original value of the rating to use in calculations, but can display the rating in any desired format. For instance, the owner of the server may choose to display five-star ratings, in which case 0.5 would be represented as two and half stars.

Returned fields:

```
  <resource>
    <!-- URL of the created rating -->
  </resource>
  <version>
    <!-- Current version of the resource -->
  </version>
  <creator>
```

```
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<reference>
  <!-- URL of document to which this recommendation applies -->
</reference>
<value>
  <!-- Current rating -->
</value>
<audiences>
<!-- Information about audience tags for this rating -->
</audiences>
<topics>
<!-- Information about topic tags for this rating -->
</topics>
```

API for Subjectversion

Subjectversion

POST

```
  http://entry-point/subjectversion
  <requester>
    <!-- Optional field: ID of the person making the request -->
  </requester>
  <reference>
    <!-- Required field: URL of document to which this one points -->
  </reference>
  <versionmin>
    <!-- Required field: beginning version
         to which reference applies -->
  </versionmin>
  </versionmax>
    <!-- Required field: ending version
```

```
        to which reference applies -->
</versionmax>
```

This creates version information to go with the resource. The subject is the software or other subject matter that has multiple versions. The `versionmin` and `versionmax` fields provide a range. Either can be passed as `beginning` or `current`. If a client indicates that a resource applies to the current version of the subject, the server assumes that the resource still applies even when new versions appear of the subject.

Returned fields:

```
<resource>
  <!-- URL of the created subjectversion information -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<versionmin>
  <!-- Beginning version to which this reference applies -->
</versionmin>
<versionmax>
  <!-- Ending version to which this reference applies -->
</versionmax>
<reference>
  <!-- URL of the reference -->
</reference>
```

API for Quiz

This part of the API starts with metadata about the document to which a quiz is attached. The metadata is currently intended for use only with quizzes and is assigned to a URL that only the associated quizzes refer to.

The use of quizzes is not meant to be automated. Quizzes are inherently interactive, requiring the user to read the text and click on checkboxes. In fact, automated access would suggest that someone is

gaming the quiz by stuffing the answer box. Thus, there is no API for submitting quiz choices.

Document

POST

```
http://entry-point/document
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<url>
  <!-- Required field: URL of document -->
</url>
<title>
  <!-- Optional field: title of document -->
</title>
```

Creates information about the document that the quiz is testing. The `url` field may contain a fragment (# syntax) or XPointer specification to refer to part of a larger document, so multiple quizzes can be attached to different parts of a document.

Returned fields:

```
<resource>
  <!-- URL of the created document information -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<url>
  <!-- URL where document can be found -->
</url>
<title>
  <!-- Title of document -->
```

```
</title>
```

The URL returned for the resource (not the URL where the document can be found) appears in subsequent quiz-related methods manipulating the document information.

Quiz

POST

```
http://entry-point/quiz
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<document>
  <!-- Required field: URL of document information -->
</document>
<contactinformation>
  <!-- Optional field: where to send updates about the quiz -->
</contactinformation>
<questions>
<!-- Optional fields containing information about questions
     in the quiz, and optionally information about choices
     in each question -->
</questions>
```

Creates basic information about the quiz. The `document` field passes the previously created URL that points to document information.

Server implementation suggestion: The contact information is interpreted by the server; for instance, it may be a list of email addresses that should be notified when someone takes the quiz. The server may use the contact information or requester information to control access to quiz results.

Returned fields:

```
<resource>
  <!-- URL of the created quiz information -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
```

```
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<document>
  <!-- URL where information about the document is stored -->
</document>
 <contactinformation>
  <!-- Where to send updates about the quiz -->
</contactinformation>
<questions>
  <!-- Information about questions in this quiz-->
</questions>
```

Question

POST

```
http://entry-point/quiz/question
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<number>
  <!-- Required field: integer great than zero, referring to
       number of question within the quiz -->
</number>
<correct>
  <!-- Required field: integer greater than zero, indicating
       which choice is correct -->
</correct>
<value>
  <!-- Required field: text of question -->
</value>
<choices>
  <!-- Optional fields containing information about choices
       in this question -->
</choices>
```

Creates a question. It is the responsibility of the server to refuse to create a quiz whose questions are

out of order.

Each question offers multiple choices. The number of the correct answer is passed in `correct`.

Server implementation suggestion: The server keeps a running total of the number of choices, which makes it easier for the quiz software to catch invalid choices passed by the client.

Returned fields:

```
<resource>
  <!-- URL of the created question information -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<quiz>
  <!-- URL of the quiz -->
</quiz>
<number>
  <!-- Number of this question within the quiz -->
</number>
<correct>
  <!-- Which choice is correct -->
</correct>
<value>
  <!-- Text of question -->
</value>
<choices>
  <!-- Information about choices in this question -->
</choices>
```

Choice

POST

```
http://entry-point/question/choice
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<number>
  <!-- Required field: integer great than zero, referring to
       number of choice within the question -->
</number>
<value>
  <!-- Required field: text of choice -->
</value>
```

Creates a choice for the quiz-taker to select for the question.

Server implementation suggestion: It is the responsibility of the server to refuse to create a question whose choices are out of order.

Returned fields:

```
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
<timestamp>
<!-- Date and time that the resource was created,
     in ISO 8601 format -->
</timestamp>
<number>
  <!-- Required field: integer great than zero, referring to
       number of choice within the question -->
</number>
<value>
  <!-- Required field: text of choice -->
</value>
```

Docversion

POST

```
http://entry-point/document/docversion
<requester>
  <!-- Optional field: ID of the person making the request -->
</requester>
```

```
<docversion>
  <!-- Required field: a version of the document -->
</docversion>
<creation_time>
  <!-- Required field: When the version was created -->
</creation_time>
```

A quiz can appear on many different versions of a document in order to let the author see which version is most effective at teaching readers. This method creates information concerning the version of the document on which the quiz appears. The `docversion` is a version number meaningful to humans, such as "2.1." The `time` is the date and time the version was created, which is useful for historical tracking. The `time` is not meant to be time time the call was issued or handled by the server.

Returned fields:

```
<resource>
  <!-- URL of the created docversion information -->
</resource>
<version>
  <!-- Current version of the resource -->
</version>
<creator>
  <!-- Requester who uploaded the resource -->
</creator>
<timestamp>
<!-- Date and time of most recent change to resource,
     in ISO 8601 format -->
</timestamp>
<docversion>
  <!-- Version of the document -->
</docversion>
<creation_time>
<!-- Date and time that the creator of this version wants to record
     as its creation, in ISO 8601 format -->
</creation_time>
```

Errors

Different types of errors will be discovered during implementation of this system, but a few that are easy to anticipate include:

- Missing resource
- Resource is of wrong type
- Parameter is of wrong type
- Format not supported
- Malformed URI
- XPointer error
- Malformed requester ID
- Access denied to resource
- Version mismatch during attempted PUT
- Data request too large

The system should also have a robust way to pass through errors occurring at lower levels, such as “Network connection severed” or “Filesystem full.”

Complications

This section lists the difficulties inherent in the design, and the reasons for the choices I made in the face of the difficulties.

Issues outside this document's scope

I begin with a task common to most specifications: excluding certain issues from the document's scope. I am proud to say that this lists differs from the lists in many specifications, which draw up their scope to leave out the hard questions and postpone them till the unspecified future, thus raising the question in readers' minds as to whether the specification is not only unrealizable but perhaps irrelevant. In contrast, the issues excluded in the following list are those that have already been solved and can safely be left to other parts of the environment.

Login and ID management

It may be useful to think of this API as part of a session between client and server, and to add “Start session” and “End session” functions in order to handle session-wide parameters such as identification, access rights, and general communication decisions such as the use of encryption or compression. But currently it seems that those parameters can be handled by the hosting system.

Many servers store information about users with arbitrarily chosen IDs internally. When storing data on each resource, the server can store the creator ID in the internal format, so long as the server can generate the original, meaningful ID passed by the client when the ID is requested by a “Read” function.

Missing resources

It was noted from the beginning of the World Wide Web that its links are one-directional and that missing resources are hard to deal with. This design decision made the Web as a whole more robust (and probably made its implementation possible in the first place), but ironically made each connection on the Web less robust.

The problem is effectively impossible to solve on the Web anyway, because many flaws in the system are created at lower levels when servers move or are taken down. In other words, even if we eliminated 404 errors, we'd still get DNS resolution errors and “server not found” messages.

This API doesn't try to solve the general problem, but makes its own references and connections bidirectional, when possible, in order to keep its own corner of the Web as tidy as possible.

Servers should store the URLs created by references and connections with each resource, so that the client or server can navigate from a question to its answer, from an answer back to its question, from a recommendation to its audience and topic tags, and so on. Each resource should also store a reference count that is incremented when a new reference or connection is made, and decremented when it is deleted.

Not all deletions will be reported, however, because of the problem noted earlier about servers that disappear. We should assume that some resources will have dangling reference counts and never be removed. On modern systems with cheap disk space, this problem can be ignored, unlike the corresponding problem of memory leaks when programs fail to clean up their data.

Another problem is that references can be created outside this API, because URLs are simple strings

that anybody can copy and paste. It will probably be commonplace for a contributor to type or paste a URL into the text of a document, and the system will not know about it. The system will then not be notified if the reference goes away, and we'll be left with the traditional 404 problem.

One common change made to resources in this system is for a contributor to create a new, stand-alone document from the answer to a question. The site coordinator who arranges this change should probably leave the question in place and change the answer to a pointer to the stand-alone document. This solution avoids the deletion of resources.

Questions will often be consolidated, however, because contributors will post duplicates of pre-existing questions. It may be a good idea for the site administrator to make the deleted question's URL point to the pre-existing question.

Network problems and transactions

The gap between theory and practice is always greater in networks than in theory. Because many use cases can be represented in this API by a set of multiple function calls, the system effectively operates as a set of transactions and is subject to network outages, delays, and the confusion introduced by caching (which allows different contributors to see different data for a single object at the same time). Even if all the operations are requested in a single method, the server has to implement them sequentially and may fail at some point, leaving the system in an inconsistent state.

The API follows the current trend of trying to keep each operation stateless. However, many operations must be executed together in order to have a practical effect, and some of the problems that arise are discussed in the following section, “Issues deserving further discussion.”

Character sets and locales

Although this API deals largely with text documents, it says nothing about character sets and other issues related to text, such as internationalization and localization. These are document-level issues and should be handled as each document is viewed and edited. I would strongly resist suggestions to handle these document-level issues in the API, but it could be done by adding character set and locale parameters to the individual calls or to a “Start session” call.

Link spam

The system described in this document will certainly be abused. Site administrators and contributors will have to set up watch lists, monitor their submissions for link spam, and set up filters to catch common abuses, just as in other existing systems that accept outside contributions.

Search

Search is critical to the value of any large, interconnected system. This API addresses only the information exchanged between client and server, not the implementation of search on the server. Note also that the recommendation system in this document allows an alternative way to find resources that should make it easier for people to find information on particular topics, concerning particular versions of software, and information aimed at particular audiences.

Providing consistent tag sets

Consistency is definitely a virtue in the tags used for audiences and topics. In particular, consistent audience tags allow the system to generate suggested paths through documents by different authors in different places. Searches would also be easier. My only contribution to consistency is to provide a `taglist` for servers to send suggested tags to clients. Like most community-based projects, I

await further results from the sophisticated research going on currently into folksonomies and other technologies to improve metadata.

Issues deserving further discussion

Several difficulties are inherent in this system, and continued discussion is useful to seek better solutions.

The complexities of resource management

In theory, the system could be designed so it doesn't have to maintain resource trees or reference counts. It could check the content of each resource before deleting it and recursively discover all the references. But this is error-prone, particularly because contributors can type or paste in the URLs of resources that have already been referenced elsewhere. The system could end up deleting a resource to which some other document still refers.

Operations consisting of multiple calls and multiple steps

As mentioned previously, a single operation by a contributor can result in multiple calls (transactions, loosely defined) with resources that refer to one another. Notably, a question or answer may be created along with an attachment, which also entails creating one or more connections. A recommendation often has an endorsement, one or more audience tags, and one or more topic tags. This API allows the client to send the information in any order, but processing is easier if the higher-level resources are created first. The server should maintain a information for a limited period of time if it receives a resource that refers to another resource (by a relative URL) that has not yet been requested.

Thus, if the server receives the calls for endorsements, audience tags, or topic tags, but times out while waiting for the recommendation, it can throw away the information from other calls and return error messages so the client can try again.

An alternative implementation would be for the client to work from the bottom up: issue the calls for the audience tag and topic tags, collect the URLs returned by the server, and then create the endorsement. This approach seems very disciplined and tidy, but it threatens to turn into an implementation nightmare. The server now has to hold on to information twice as long while its response returns to the client and the client sends the endorsement. The client has to maintain time-outs too. How do these time-outs interact, and does each side lengthen its time-out in recognition that the other side is waiting for a time-out? Furthermore, the server can never create anything until all resources are received, during which time it may throw away information received early in the transaction.

To retrieve metadata on a child resource, a client must either slurp up all the data on all related resources, or issue multiple calls in order to find the URL of the child resource. A shortcut could be to allow some resources to appear on the requested URL. For instance, to get subjectversion data about Python, we could let a client issue a GET on:

`http://entry-point/subjectversion/subject=Python`

This syntax would be useful for only a few resources that tend to have repetitive contents (subjectversion, audience, and topic) and would require extra complexity in the implementation, so I don't believe it's worth the occasional saved operation.

Lack of consistency in subjectversion information

The versions of the software being described in each document are a fundamental piece of metadata that should be stored with each document. I see no way to provide and maintain this information except the manual efforts of contributors. The problem is compounded by the interaction of multiple parts of the system on each question: for instance, a problem could very well be specific to a particular combination of operating system, database, and application.

Clients can encourage contributors to document the versions by providing text boxes for that purpose and interrupting them with deliberately annoying prompts if they try to submit material without filling in the boxes. Contributors can also apply peer pressure to provide subjectversion information, because it's often necessary to answer questions anyway.

As new versions are released, contributors who care enough about documents to keep them up-to-date can check whether they apply to the new versions and alter the subjectversion metadata. The "current" keyword allows a contributor to say that a document continues to apply until further notice.

The format of the version information as a range (for instance, 2.3.4–2.3.8) allows the system to satisfy searches by version number. Someone searching for information on subjectversion 2.3.6 of the software will presumably be directed to a document covering 2.3.4–2.3.8, whereas the document won't be displayed in searches for 2.3.1 or 2.3.12.

Unfortunately, version numbers are rarely so orderly. A lot of software use numbers such as 2.3a or 2.3alpha. And how does the system handle versions what when a bug exists in both 2.3 and 2.4, but is fixed in 2.3.1 and 2.5? Even worse, suppose the bug returns in 2.5.1?

We have to accept that no automated system can handle the messy trajectory of feature improvements and bug fixes. If we handle searches and the display of subjectversion information gracefully for the simple use cases listed in the "Software (subject) and document versions" section of this document, we will immensely enhance the usability of information.

References versus connections

The Connection resource could be used to add references, eliminating the need for the Reference field in other methods. But I believe this would be a foolish and deceptive simplification, based on a superficial resemblance between references and connections.

References are intrinsic to many resources: for instance, an answer, recommendation, or endorsement would have no meaning without its parent resource. This kind of link is quite different from a connection, which is meant to allow contributors to show very localized relationships. These are different uses of the system, and their semantics may well diverge even though they seem to be satisfied by the same function syntax right now.

Furthermore, because references are so intrinsic to many resources, it is more robust to indicate them directly in the functions that create those resources. It would keep me up at night to worry that a client or server might create the document but forget to issue the call creating the reference. Even if the two are created on the server in sequence, it is safer to keep them in a single call.

Editing portions of documents

To minimize bandwidth for data transfers as well as conflicts among multiple editors, clients and servers should divide large documents into smaller chunks that are convenient to edit. Many wikis place an edit button on each paragraph; this is a convention worth considering. XPath and XPointer enable URLs for an arbitrary chunk of edited text.

When an entity such as a paragraph is added or removed, the larger entity should be specified, because the numbering of the entities within XPath and XPointer changes and old URLs become invalid. The versioning mechanism in GET and PUT methods allows the client and server to make sure that the client does not replace the wrong paragraph because it tried to edit paragraph 5 and someone else removed it before the PUT call.

The standard problem with unstructured HTML appears in trying to edit portions of documents. XHTML is not sufficiently structured because heading tags can be inserted arbitrarily. The classic work-around is to convert between XHTML (for displaying results) and some more structured format such as DocBook (for storage) using XSLT. However, a server could also simply break up a document along XHTML header tags and assume the results will be reasonable.

Reuse of quizzes, questions, and choices

The API design assumes that a quiz applies to only one document, that a question is used in only one quiz, and that a choice appears in only one question. This is a safe assumption because the context for each of these resources has an enormous impact on the wording of the questions and choices; it is likely that the wording, order, and other parameters will change even if the same idea is reused in another quiz.

Multiple ratings

Ratings can reflect a variety of considerations: whether a document is accurate, whether it's easy to read, whether it's relevant to the reader's need, and so on. Some sites offer multiple ratings in order to reflect different criteria, but I believe this requires an investment of time and thought on the part of the reader that few will have the patience to invest. The question of relevance is somewhat addressed by the Recommendation system. In any case, a rating is a very subjective response to a document, which is why I recommend that projects try to use quizzes instead of ratings, or in addition to them. If ratings are used, a single rating is already enough of a commitment to ask of a reader.

Because it would be cheating for a single person to assign multiple ratings, the system could consider making the Requester field a required one, but it doesn't seem worthwhile introducing a variation to the model in this particular case because anyone wanting to cheat could find other ways to assign multiple ratings.

Updates and version numbers

Because resources are linked to others in cascading trees (questions, answers, audience and topic tags, recommendations, connections, etc.) the information associated with each resource goes beyond its text content. A client that requests a resource gets metadata about the child resources as well, so the Last-Modified and Etag headers sent by the server with a resource should reflect changes in the metadata that the server would send as well as the text content. On the other hand, it would be wasteful to resend a large body of text if only one piece of metadata has changed. Therefore, it might be useful to find a way to distinguish changes in content from changes in metadata so that a client can download only what it needs.

One possible mechanism for the distinction is this API's versioning system for resources (the numbers under the `/versions` element of the resource path). The examples in this design document assume a simple incrementing version number, but developers could consider generating version numbers from the content, such as by using a Pearson hash function. The client and server could compare hashes to determine whether the text has changed since the client's previous GET.

Another advantage of a hash is that someone who restores a document to its previous content would end up generating the previous version number as well, which would allow clients to make changes without having to make an unnecessary download.