Knowledge Management
(in the Web)

Knowledge Search Engines

Duration: 3hrs

Tutorial
The ideal search engine would be able to match the search queries to the exact context and return results within that context. While Google, Yahoo and Live continue to hold sway in search, here are the engines that take a semantics (meaning) based approach, the end result being more relevant search results which are based on the semantics and meaning of the query, and not dependent upon preset keyword groupings or inbound link measurement algorithms, which make the more traditional search engines easier to game, thus including more spam oriented results.

**Popular SW Search Engines:**

- Semantic Web Search Engine (SWSE)
- Sindise
- Watson
- Falcons
- Swoogle
- Semantic Web Search
- Zitgist Search
- Hakia

**Semantic Web Search Engine (SWSE)**

SWSE (Fig. 1) is a search engine for the RDF Web on the Web, and provides the equivalent services a search engine currently provides for the HTML Web. The system explores and indexes the Semantic Web and provides an easy-to-use interface through which users can find the information they are looking for. Because of the inherent semantics of RDF and other Semantic Web languages, the search and information retrieval capabilities of SWSE are potentially much more powerful than those of current search engines. SWSE indexes RDF data from many sources, including OWL, RDF and RSS files. RSS2 is converted to RDF and they will be adding GRDDL sources soon. Developed by DERI Ireland.

![SWSE](http://swse.deri.org/)

**Fig. 1. Semantic Web Search Engine.**
From a bird’s-eye view, the architecture of the Semantic Web Search Engine resembles that of traditional keyword-based search engines. Queries are accepted and results generated based on summarized data in a central database. The basic search query form is plaintext, which supports intuitive features much like that of Google. Text may be quoted to treat multiple words as a single unit, and stop words (e.g. “is,” “the,” “of,” etc.) will—to some degree—be ignored. Starting with a plaintext query form maximizes the flexibility of future search enhancements.

Thus, in the SWSE architecture, keyword-based search still plays a role. Given a search query Q in plaintext form, phrases of Q are matched via case-insensitive string matching against the `<rdfs:label>` and `<rdfs:comment>` elements of all available RDF documents, as well as all `<rdfs:Literal>` elements, as identified by property definitions. Results of this string-matching phase are URIs of both properties and non-properties, weighted according to frequency.

Once potential properties and resources are identified, they are combined to form RDF queries against the RDF knowledge base. This is the heart of the SWSE architecture in which the various combinations of properties and resources are queried and results collected and ranked. As an example, if a given plaintext query results in potential properties $p_1$ and $p_2$, and potential non-property resources $n_1$, $n_2$, and $n_3$, a query will be generated for each combination in which either zero or one element is missing (e.g. $n_1 p_1 n_2; n_1 p_1 n_3; n_1 p_1 ?; ? p_2 n_1;$ etc.). This “fill-in-the-blank” RDF statement detection process is repeated with those new elements discovered forming a “hop” in the sense of a breadth-first search algorithm.

Given example query “wife of President before Adams,” it matches substrings against the SWSE knowledge base. From the genealogy ontology, it matches the `isWifeOf` property; from the US Presidents ontology, it matches the `presidentBefore` and `presidentAfter` properties, as well as the `JohnAdams` and `JohnQuincyAdams` resources. During the first pass, it detects statements shown below.

- GeorgeWashington presidentBefore JohnAdams.
- JamesMonroe presidentBefore JohnQuincyAdams.
- ThomasJefferson presidentAfter JohnAdams.
- AndrewJackson presidentAfter JohnQuincyAdams.
- AbigailSmith isWifeOf JohnAdams.
- LouisaJohnson isWifeOf JohnQuincyAdams.

For each new resource detected, the process repeats itself, yielding new RDF statements.

- MarthaCurtis isWifeOf GeorgeWashington.
- ElizabetKortwright isWifeOf JamesMonroe.
- MarthaSkelton isWifeOf ThomasJefferson.
- RachelRobards isWifeOf AndrewJackson.
- JamesMonroe presidentAfter JamesMadison.
- ThomasJefferson presidentBefore JamesMadison.
- etc.

Though this process may be repeated many times, it limits the number of repetitions—i.e. the maximum breadth—to a small number such as two or three, otherwise results will be flooded with irrelevant inferences.

The RDF statements discovered via the aforementioned querying process form a semantic webgraph. In an attempt to match multiple RDF statements to a given plaintext query, it numbers each word of the plaintext query, as shown below.

```
wife of President before Adams
1 2 3 4 5
```

Then it counts the number of potential properties, $np$, detected for the plaintext query string, deduplicating based on location. For the given example, the word “wife” in location 1 matches the `isWifeOf` property, whereas the word “President” in location 3 matches both the `presidentAfter` and `presidentBefore` properties. Counting location 3 only once,
np is two, indicating that it results should combine at most two RDF statements. More specifically, it traverses only two properties in its semantic webgraph.

**Sindise**

Sindise (Fig. 2) is a lookup index for Semantic Web documents built on data intensive cluster computing techniques. Sindise indexes the Semantic Web and can tell you which sources mention a resource URI, IFP, or keyword, but it does not answer triple queries. Sindise currently indexes over 20 million RDF documents. Developed by DERI Ireland.

![Fig. 2. Sindise.](image)

Sindise allows its users to find documents with statements about particular resources. It is in the first place not an end-user application, but a service to be used by any decentralised Semantic Web client application to locate relevant data sources. As an application service Sindise can be accessed through its Web API, for human testing and debugging it is also offered an HTML front-end.

Thus, searching e.g. for the URI of Tim Berners-Lee can be displayed on the HTML interface for humans. However, the application interface returns the same results but in various machine-processable formats such as RDF, XML, JSON and plain text, an example is shown below. In this example, several documents are returned, each of which mentions Tim Berners-Lee’s URI. The results are ranked in order of general relevance and some further information is given to enable users to choose their preferred source.

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<rdf:RDF xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://www.w3.org/People/Berners-Lee/card#i">
    <rdfs:seeAlso rdf:resource='http://www.w3.org/People/Berners-Lee/card#i' />
    <rdfs:seeAlso rdf:resource='http://danbri.org/foaf.rdf' />
    <rdfs:seeAlso rdf:resource='http://heddley.com/edd/foaf.rdf' />
    <rdfs:seeAlso rdf:resource='http://www.eyaloren.org/foaf.rdf' />
  </rdf:Description>
</rdf:RDF>
```
Moreover, Sindice enables Semantic Web clients such as Piggy Bank or Tabulator to find documents with information about a given resource, identified through an explicit URI, an inverse functional property or a keyword search. This capability fits well on top of many existing Semantic Web clients. The immediate use for Sindice inside such clients is to enable a “find out more” button, to be shown next to the available information about a resource.

Upon pressing that button, the client would contact Sindice for a ranked list of documents with more information about the resource. The user would be presented with a ranked list of these documents including a human-readable source description. The user could then choose the sources of interest (or those considered trustworthy), after which the client application could import the information from these documents. The user could maybe also select to “always consider these domains as providers of good information” to allow fully automated information import during subsequent lookups.

For clients that implement the linked data principles, integration with Sindice is trivial. Sindice behaves as a “good citizen” of the linked dataWeb: it provides all results as RDF that themselves follow the “linked data” principles. While Sindice supports, uses, and promotes the linked data model (namely in its crawling and ranking), it also supports locating information about URIs that are not URLs and cannot be de-referenced such as telephone numbers or ISBN numbers. But most importantly, Sindice helps locating statements about resources made outside their “authoritative” source.

**Watson**

Watson (Fig. 3) allows searching through ontologies and semantic documents using keywords. At the moment, you can enter a set of keywords (e.g. "cat dog old_lady"), and obtain a list of URIs of semantic documents in which the keywords appear as identifiers or in literals of classes, properties, and individuals. You can also use wildcards in the keywords (e.g., "ca? dog"). Developed by KMi, UK.

![Watson](image)

**Fig. 3. Watson.**

Even if the first goal of Watson is to support semantic applications, it is important for it to provide web interfaces that facilitate the access to ontologies for human users. Users may have different requirements and different levels of expertise concerning semantic technologies. For this reason, Watson provides different “perspectives”, from the most simple keyword search, to sophisticated queries using SPARQL. These interfaces are implemented in Javascript, using the principles of AJAX, thanks to the DWR Library.
The keyword search feature of Watson is similar in its use with usual web or desktop search systems. The set of keywords entered by the user is matched to the local names, labels, comments, or literals of entities occurring in semantic documents. A list of matching ontologies is then displayed with, for each ontology, some information about it (language, size, etc.) and the list of entities matching each keyword. The search can also be restricted to consider only certain types of entities (classes, properties, individuals) or certain descriptors (labels, comments, local names, literals).

At a technical level, this functionality relies on the Apache Lucene indexing system. Different indexes – concerning semantic documents, entities, and relations between entities – are built from the metadata extracted during validation.

One principle applied to the Watson interface is that every URI is clickable. A URI displayed in the result of the search is a link to a page giving the details of either the corresponding ontology or a particular entity. Since these descriptions also show relations to other elements, this allows the user to navigate among entities and ontologies. For example, with the query “university researcher student”, we obtain 19 matching semantic documents. Among them, http://www.aktors.org/ontology/portal.daml contains the entity http://www.aktors.org/ontology/portal#Researcher. Clicking on this URI, we can see that this entity is described (sometimes we different descriptors) in several ontologies. In particular, it is shown to be a subclass of http://www.aktors.org/ontology/portal#Working-Person. Following the link corresponding to this URI also shows its description in each of the semantic documents it belongs to. Then, the metadata corresponding to one of these documents can be retrieved following the appropriate link, e.g. http://www.aktors.org/ontology/portal, to find out about its languages, locations, etc. Finally, a page describing a semantic document provides a link to the SPARQL interface for this semantic document, as described in the next paragraph.

A SPARQL endpoint has been deployed on the Watson server and is customizable to the semantic document to be queried. This endpoint is implemented thanks to the Joseki SPARQL server for Jena. A simple interface allows to enter a SPARQL query and to execute it on the selected semantic document. This feature can be seen as the last step of a chain of selection and access task using the Watson web interface. Indeed, keyword search and ontology exploring allow the user to select the appropriate semantic document to be queried. The developers plan to extend this feature to be able to query not only one semantic document at a time, but also to automatically retrieve the semantic data useful for answering the query. This kind of feature, querying a whole repository instead of a single document, has been implemented in the OntoSearch2 system.

**Falcons**

Falcons (Fig. 4) is a keyword-based search engine for the Semantic Web, equipped with browsing capability. Falcons provides keyword-based search for URIs identifying objects and concepts (classes and properties) on the Semantic Web. Falcons also provides a summarization for each entity (object, class, property) for rapid understanding. Falcons currently indexes 7 million RDF documents and allows you to search through 34,566,728 objects. Developed by IWS China.
In many cases, people have something in mind and want to learn more about it. For example, to attend ISWC 2008, many researchers come to Karlsruhe for the first time. They want to know more about this city, maybe anything or maybe something in particular. Actually, such requirements exist widely in traditional Web search, covering over 60% of Web queries. Traditional Web search engines provide webpages which contain the keywords in a query, e.g., “Karlsruhe”. The user obtains a series of webpages, but he/she lacks ways to get these webpages organized, or in other words, it is not easy for the user to specify a particular dimension of knowledge about the subject, except for resubmitting queries with different combinations of keywords to try his/her luck.

The Semantic Web brings possibilities of classifying knowledge without using third-party algorithms. Generally, objects on the Semantic Web are with typing information, which can be naturally utilized to organize knowledge. In Falcons, after submitting a query “Karlsruhe”, the user is served with a list of objects as well as several types such as “Event”, “Landmark”, and “Organization”. Then the user can specify a type to focus on a particular dimension of knowledge. For example, organizations in Karlsruhe attract the user’s interest, and thus “Organization” is selected. Thus, the results are filtered to include only the objects of the type “Organization”, such as “University of Karlsruhe”. Moreover, the type panel is updated to include “Club”, “Company”, “University”, etc., which are all subclasses of “Organization”. So actually, Falcons enables the user to navigate a class hierarchy to shift the focus from one type of knowledge to another.

In some cases, people seek objects with one or more particular properties, e.g., “find out people that know Peter Mika”. The user has some knowledge about the targets but wants to know more. In traditional Web search, it is called a resource query, which covers over 20% of Web queries. In this case, the user knows that the target people know Peter Mika, and wants to obtain their names. In traditional Web search engines, the user may submit a query with the phrases “knows” and “Peter Mika”, but he/she is likely to find a lot of webpages that contain both phrases but these pages do not answer the query well. Even some useful webpages are returned, the user still has to open those pages and mine related knowledge from them, which costs a lot of time.

On the Semantic Web, objects are defined with RDF triples, which are naturally property-value pairs. It creates conditions to improve the precision of retrieval. In Falcons, after submitting a query with the phrases “knows” and “Peter Mika”, the user obtains a list of objects. The user can immediately find that Michael Hausenblas and Frank van Harmelen know Peter Mika, and the demand is satisfied, even before clicking on any links in the results.
page. It is because, for each object in the results, we provide a snippet of knowledge of this object, as the form of property-value pairs. It is important that the snippet is query-dependent, which may directly answer the user’s question in mind. If the user wants to know more about each resulting object, he/she can click on it to obtain a comprehensive knowledge of this object, which is integrated from all over the Semantic Web.

On traditional Web search engines, in order to seek relations between objects, e.g., between Peter Mika and Jim Hendler, the user submits a query with their names. A webpage is returned only because its content contains both names, although in many cases it is too difficult to find out any relations between these two phrases in the page due to the long distance in text.

Comparatively, the RDF model used by the Semantic Web exactly describes relations between resources. The scenario in the previous subsection has already showed how Falcons enables to seek direct relations. Falcons also enables to seek indirect relations. Thus, the user submits a query with the phrases “Peter Mika” and “Jim Hendler”, and obtains a list of objects. The first one is a person that knows both Peter Mika and Jim Hendler, and the second one is a conference of which both Peter Mika and Jim Hendler are organization committee members. The user immediately gets answers and does not have to spend any more time in other activities like reading webpages.

Swoogle

Swoogle (Fig. 5) searches through over 10,000 ontologies. 2.3 million RDF documents indexed, currently including those written in RDF/XML, N-Triples, N3(RDF) and some documents that embed RDF/XML fragments. Currently, it allows you to search through ontologies, instance data, and terms (i.e., URIs that have been defined as classes and properties). Not only that, it provides metadata for Semantic Web documents and supports browsing the Semantic Web. Swoogle also archives different versions of Semantic Web documents. Developed by the Ebiquity Group of UMBC.

Swoogle is an ongoing project undergoing constant development. In Swoogle a general user can query with keywords and the SWDs (Semantic Web Documents) matching those keywords will be returned in ranked order. Swoogle ranks SWOs (Semantic Web Ontologies) higher than SWDBs (Semantic Web Databases); thus, SWOs will be returned as query results before SWDBs. The highest ranked SWDs typically are the base ontologies that define the
Semantic Web languages, such as the RDF and OWL language definitions, which are used by almost all SWDs and are always imported by SWOs.

For advanced users, an advanced search interface is provided, which essentially allows them to fill in the constraints to a general SQL query on the underlying database. The user can query using keywords, content based constraints (e.g. type of SWD, OWL syntax, number of defined classes/properties/individuals), language and encoding based constraints (N3 vs XML), and/or the Rank of the document.

The metadata are stored in a database, and it has already indexed over 3.000.000 SWDs. Currently, Swoogle has evolved to Swoogle2 (version 2), which is featured by three components: Swoogle Search, Ontology Dictionary and statistical measures of the collection of SWDs.

**Semantic Web Search**

Powered by RDF Gateway, Intellidimension's proprietary platform for Semantic Web applications and agents. Developed by Intellidimension Inc (Fig. 6).

The standard search engine interface is quite simple, the user can just type one or more of keywords describing the information he/she is trying to locate. This is no more complicated than a traditional Web search engine. However like a traditional Web search engine this can lead to a large number of irrelevant results. To narrow the search users can restrict it to the specific type of resource that they are trying to locate such as a person (FOAF Person) or news article (RSS Item). If the search is still producing a large number of irrelevant results than they can refine it further by specifying one or more specific property values that the resource must have. For example if a user is trying to locate a person with a last name of 'Smith' and a first name of 'John' he/she would enter the search string '[[foaf:surname]~smith [foaf:firstName]~john'.

Thus, the best search strategy to narrow the search is to increasingly narrow it in each step. For instance, firstly, you can enter the keywords that best describe what you are looking for (e.g. web browser). Then select the type of resource you are looking for from the drop list (e.g. RSS Item). Finally, enter one or more properly values of the resource (e.g. [rss:title]~web browser).
**Zitgist Search**

The Zitgist Query Service (Fig. 7) simplifies the Semantic Data Web Query construction process with an end-user friendly interface. The user need not conceive of all relevant characteristics - appropriate options are presented based on the current shape of the query. Search results are displayed through an interface that enables further discovery of additional related data, information, and knowledge. Users describe characteristics of their search target, instead of relying entirely on content keywords.

![Zitgist Search](image)

Zitgist's goal is not to be a replacement to traditional search engines such as Google. In short and middle term its goal is to be complementary to traditional search engines; to be another tool in Web users’ toolkit. The semantic web and semantic web search engines like Zitgist will be quite useful to make some order, classify and search in all the data that has been created so far and that is yet to be created on the Web.

The difference between a traditional search engine like Google and a semantic web search engine like Zitgist is that the aggregated, indexed and queried data is different. Google mostly use text files such as HTML, PDF, DOC, Etc., and Zitgist use RDF files from genuine or converted data sources. This difference has a big impact on how users build queries to answer to their questions. Google users use keywords to try to define what they are searching for. Then the search engine will check in their database to find these keywords into the texts they aggregated and indexed.

The first step is to choose which type of subject a user is searching for. In the first version of Zitgist, users are allowed to choose among some type of subjects: musical things such as artists, bands, albums, tracks, performances, or people, groups, projects, geographical locations, documents and discussion forums. Once the user choose what he was searching for he has to describe the characteristics of that subject.

For instance a user tries to find a person. So, there are some characteristics describing a person that can be defined by the user. Depending on the user interface (basic or advanced) more or less characteristic will be available for the description of that subject. So the user chooses to search for a person that has the name “Chris” and that is interested in the
“Semantic Web”. The search engine will then return results matching subjects know by Zitgist having these two characteristics.

Using Google, the user would have use the query string “chris semantic web” that has three distinct keywords: (1) “chris” (2) “semantic” and (3) “web”. The problem is that there is no relation between these keywords. Is he searching for someone named Chris that is working in the semantic web domain or that is interested in the semantic web? Is the user searching for something else? There is no way to know. The best Google can do, is putting their algorithmic magic into action to try to find what the user is searching for, and hoping it is really what he wants. But for Zitgist, if the person [the subject of the search] defined himself as having the name Chris and having an interest in the Semantic Web (defining himself using RDF) than the results are definitely what the user is searching for.

The above example was quite simple. How a user could easily describe a subject, with its own set of characteristics, that knows another subject, also with its own set of characteristics? In this example, the user searches for a person known as “Alice”. But he doesn’t search for any person named “Alice”. This user wants to find a person knows as “Alice” that know another person named “Bob”. This query is quite easy to do using Zitgist. The user described the subject he wants to search for: “Alice”. This subject is a person with the name “Alice” that “knows” a person called “Bob”. Note that the user interface changed its colour when a new subject is introduced into the query ["Bob"]. That way, users can easily see which subject they are describing. After that, the user could always add new characteristics to Bob. He could say that Bob is interesting in writing and that he lives near London for example. In fact, the possibilities are endless.

What is interesting with the semantic web is that anybody can describe anything. One of these interesting example is when we start to think about Document. In fact, what are documents? What describe a Document? Etc. A document can be described with an author, a creation data, a publication date, an editor, a publisher, its medium, etc. But its content can also be described such as its topics. If someone describes one of the documents he created and that explicited the topic(s) of that document, Zitgist could easily find it that way: Thus, a user can search for a “Document” that as a “Person” named “Alice” that “knows” another person named “Bob” as the “Topic” of the “Document”.

Another example would be a user to search for a geographical location. Ultimately, a geographical location on Earth is defined by a longitude and latitude. So, Zigtist shows a map widget to the user. The only thing he will have to do is to click on the map to choose the location. That is it. The user interface widget is intuitive for users, and he doesn’t have to bother about how to describe the location.

Finally, if a user tries to find a “Music Artist” that “composed” “Albums” between “1980” and “1990”, the Zigtist popups a small widget that will assists the user in the creation of its search query.

Hakia

The brainchild of Dr. Riza C. Berkan (Fig. 8), tries to anticipate the questions that could be asked relating to a document and uses them as the gateways to the content. The search queries are mapped to the results and ranked using an algorithm that scores them on sentence analysis and how closely they match the concept related to the query. Hakia performs pure analysis of content irrespective of links or clickthroughs among the documents (they are opposed to statistical models for determining relevance). The engine has also started using the Yahoo BOSS service and also presents results in a “gallery” with categories for different content matching the query. Users can also request to try out the incremental changes that are being tried at Hakia’s Lab.
Hakia is a relatively new search engine that wants to find and present search results in a new way. The future of search is understanding information, not merely finding it.

For instance, a user is curious about the renaissance scientist Johannes Kepler. If he searches Hakia for Johannes Kepler, he gets a presentation page from the Hakia Galleries, containing a picture of Kepler along with search results grouped in categories like Biography and Timeline, Awards and Accomplishments, and Speeches and Quotes. This is a very convenient way to get the search results presented.

Currently, the Hakia Galleries answer around 600,000 popular queries in various topics of interest, expanding the coverage every day. Few of these are: piano, Hillary Clinton, coffee, India, breast cancer, red sox, Paris Hilton, Pokemon. Hakia galleries are distilled in a semi-automated process, a mixture of meaning-based technology and editorial process. Editors are involved in the automated gallery generation process as administrators. Their role ranges from checking, correcting, and removing items that are inappropriate. Note that humans are not involved in acquiring search results, it is all automated.

However, if the user doesn’t need a gallery of information, but something more specific, like to know more about drugs to remedy his headache. He could ask Hakia: “Which drug treats headache?”. In the search results sentences that contain an answer to the question are highlighted, like “aspirin has been used to treat migraine and other headaches” and “Nurofen is indicated for the relief of headache and back pain of musculo-skeletal origin”. So even before the user clicks on the search results, he has some of suggestions.

Notice how the sentences in the search results don’t contain the exact same words as in the query. Either of the sentences contain the word ‘drug’ and one talks about relieving headache instead of treating headache. This is because Hakia uses fuzzy logic to expand my query. Fuzzy logic means a flexible algorithm. The flexibility is used to take the original query and create its equivalent and enriched versions on the fly. The principles used in this process come from ontological semantics. The reason Hakia is doing this is to bring search results from a variety of equivalent articulations of the search query and related concepts. For example, the word headache is related to migraine or the word treatment is related to cure. Without such enrichment, a search algorithm will stick to the word used, and will not be able to retrieve results from equally relevant material.
So if the user searches for “headache treatment” a search on Yahoo or Google will not return results regarding a cure for migraine, unless this phrase “headache treatment” is present on the relevant page, or someone has used this word when linking to it. Notice that Hakia is the first search engine to introduce this ability to users, although the current beta version is not fully equipped with this capability yet.

Natural language search is one of Hakia’s advantages. This simply means that the user can pose a question to the search engine (e.g. “When was Abraham Lincoln born?”) instead of breaking the question down into keywords (e.g. “abraham lincoln birth”). Natural language search also means that the user can expect an answer to his question right on the search results page. Hakia will present search results which contain an answer to the question, not a list of web pages that might or might not contain an answer. This is made possible by research in the intersection between the scientific disciplines of philosophy of language, mathematical logic, and cognitive science. This is called ontological semantics, a formal and comprehensive linguistic theory of meaning in natural language.

Hakia’s SemanticRank algorithm differs from popularity algorithms like Google’s PageRank in that it determines a site’s relevancy not by its popularity, but by the relevancy of the query to the content of the page, which is determined by popular vote (via link referrals) in Google. The critical breaking point in this equation is when the user queries start to become longer than usual, unique, complex, and personal. When this happens, the ‘popularity’ reference point disappears. Queries like these are called long-tail queries, and there are zillions of them.

Moreover, Hakia has invented a new system called Qdexing, which is specifically designed for meaning representation. Qdex means query detection and extraction. This entails analyzing the entire content of a webpage, then extracting all possible queries that can be asked to this content, at various lengths and forms. These queries become gateways to the originating documents, paragraphs and sentences during the retrieval mode. Note that this is done off-line before any actual query is received from a user.