WikiGraph

Mark Jordan, Jeremy Lenz, Robert McClure,
Austin Nakamura, Michael Rush, Khanh Tran,
Thomas Van Doren

System Design Specification and Planning Document

Draft 1
4 February 2011

CSE 403 - CSRocks Inc.

Version	Primary Author(s)	Description of Version	Date Completed
1	Rob McClure	Initial draft with database schema	2011-02-01
2	Thomas Van Doren	UML Class View, Requests diagram	2011-02-02
3	Thomas Van Doren	XML specs, url defs, design assumptions, schedule	2011-02-03
4	Rob McClure	Services alternatives	2011-02-03
5	Mark Jordan	Sequence Diagram	2011-02-03
6	Thomas Van Doren	Test and doc plan, risk assessment	2011-02-03
7	Austin Nakamura	Flash client alternatives	2011-02-03
8	Rob McClure	Added to services architecture	2011-02-03
9	Jeremy Lenz	Second sequence diagram	2011-02-04
10	Austin Nakamura	Added to frontend architecture	2011-02-04
11			

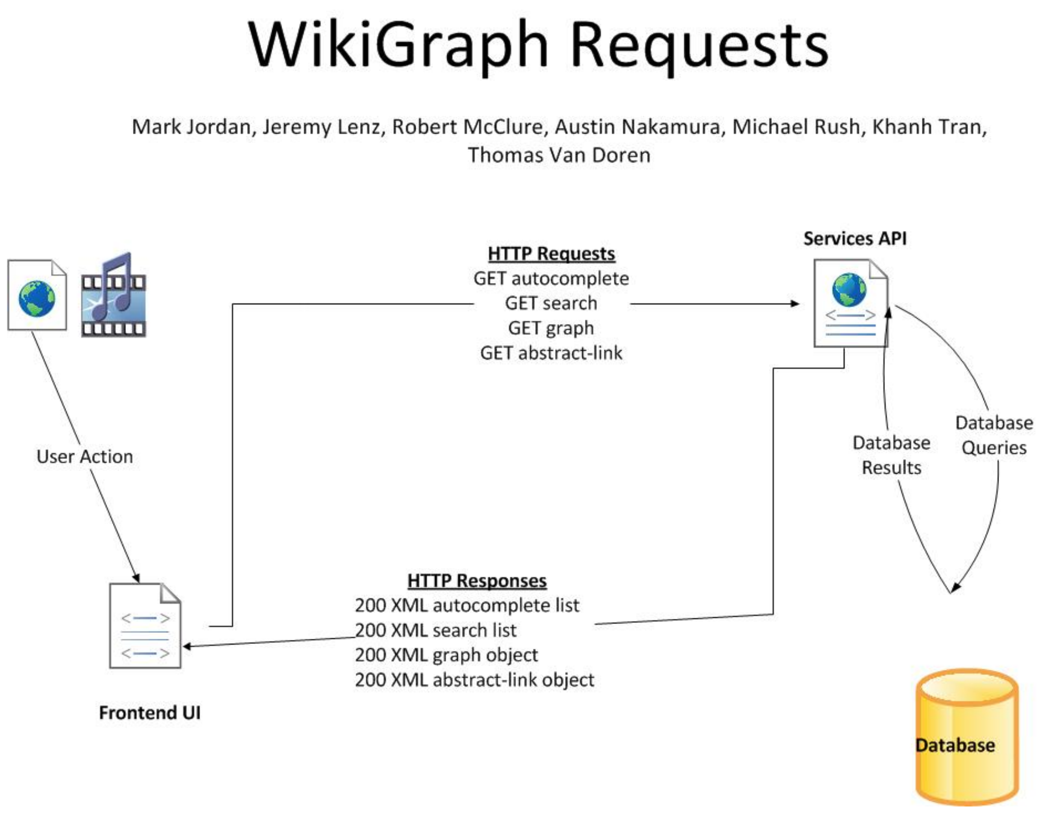
Introduction

WikiGraph is a simple tool to visualize relationships between various Wikipedia articles. It provides a visualization of how Wikipedia articles relate to each other. It is an easy tool to use and learn. It provides access to brief information about the article in the client and offers links to access to the full articles on Wikipedia. WikiGraph is intuitive and informative.

System Architecture

The overall WikiGraph architecture is modeled below with the kind of requests highlighted. The frontend flash ui makes HTTP requests to the PHP data services API. A url scheme will be used to differentiate request endpoints. Data will be passed along in the query string of these requests. Since all the requests to the services will be GET requests, the frontend does not have to worry about packaging data in different ways.

The services will return valid XML for each request. Standard HTTP response codes will be returned with data. Client errors will have a 4xx error code, server errors 5xx error code, and successful requests will have 2xx error codes. All other HTTP codes are not pertinent to the scope of the WikiGraph system. The specific structure of the XML responses is designated below.



Frontend Flash Client

The WikiGraph frontend user interface is an Adobe Shockwave Flash player. It is developed using Adobe's opensource flash compiler and framework, Flex. Specifically, MXML is used to design and build the document body. ActionScript is used simultaneously to implement the document behavior: rendering the graph, animating transitions, making requests to the data services api, handling responses from the services api, and taking action after a user event. The flash client will reside at `/graph/` within the WikiGraph website.

As the user types in information to the search field, a one to two second pause in typing will cause an asynchronous auto-complete request to be sent to the server, and eventually a list of closest matches will be sent back. When the user does enter a search term, the client sends out a search request to the server, but from there, the returned information depends on whether the search was successful. By chance, if the requested search term does not specifically match any articles on the database, the client will receive a list of search results of similar articles; upon the user's selection of one, a request for the related graph information will made. However, if the search term produces an exact match, the client will be given the graph information needed to construct it, forgoing the search list. Upon receiving the graph object, the client will automatically parse the information and call a client side function which draws the interactive graph, using the information provided by the graph object. While interacting with the graph, clicking any of the nodes will cause a graph request for the article identified on the node to be asynchronously sent to the server. Hovering over a node will result in an abstract and a link request being sent to the server; upon receiving and parsing the returned object, a tooltip will be displayed near the relevant node, displaying both a short portion of the abstract, while clicking it will open up the actual article in another webpage.

PHP Data Services API

The WikiGraph Data Services API is the interface in which a WikiGraph client must use to request and receive the data to render graphs, search for article nodes, and generally access the WikiGraph data. The specific data structures returned by the api are listed below. The api only accepts HTTP GET requests. This, semantically, makes sense because no data will change as a result of a client request. Indeed, this is a read only api.

The url endpoints at which specific data may be requested are as follows:

URL Endpoint with Query String	**Description**
`/api/autocomplete/?q=<partial-phrase>`	Returns a list of all article titles which match the partial phrase.
`/api/search/?q=<search-phrase>`	Returns a list of all articles which match the phrase.
`/api/graph/[<node-id>/][?t=<title>]`	Returns a graph for the given node-id, if specified. If, for some reason, a title is specified instead of a node id, the article with the best matching title is returned. A HTTP 4xx response is returned if the specified node cannot be found or neither a node id or article title was specified.
`/api/abstract/<node-id>/`	Returns the title, abstract, and hyper link to wikipedia for the specified article with uid, node-id.

Each service is implemented in PHP. The data is returned as XML, as detailed in the following section. In order to fetch the data, the services connect to a common database, form and run queries, and use the results to create the XML objects.

In order for the services to be used with different databases with different schemas, each database will have an associated database wrapper. These wrappers will know how to fetch the data for each of the services for its specific database. As a result, the services will only have to use a different wrapper in order to use a different database, so code in those services will not have to be rewritten.

XML Data Structures

Each of the following structures represent the body of an XML document. They will be wrapped in valid XML tags and specifiers. Those details were left out since they bear no real insight into the design or implementation of the system.

The autocomplete and search responses are identical. The difference is the root tag. The graph structure is similar to both the search and autocomplete structures in that it is a list of nodes (at the high level). The nodes for a graph are more complex as they indicate connections.

Graph XML Response Structure

The graph data structure is at the heart of the WikiGraph ui. It is intended to ease the frontend flash client's responsibility of rendering by designating all of the relations within the structure. Unique identifiers (uid) are given to each node such that the flash client can easily track nodes across several requests without in-memory data structures. The uid will allow the flash client to quickly find a node on an already drawn map (either partially or fully). This is not immediately useful, but will reduce the complexity of drawing larger (higher degree) maps in the future.

```
<graph center=”[uid1]”>
      <source id=”[uid1]” title=”[node-title]”>
            <dest id=”[uid2]”/>
            <dest id=”[uid3]”/>
            ...
      </source>
      <source id=”[uid2]” title=”[node-title]”></source>
      …
</graph>
```

Search XML Response Structure

```
<search query=”[phrase that was searched on]”>
      <item id=”[uid1]” title=”[node-title]” />
      <item id=”[uid2]” title=”[node-title]” />
      …
</search>
```

Autocomplete XML List Structure

```
<list phrase=”[partial phrase that was searched on]”>
      <item id=”[uid1]” title=”[title]” />
      <item id=”[uid2]” title=”[title]” />
      <item id=”[uid3]” title=”[title]” />
      ...
</list>
```

Abstract, Link XML Object Structure

This object contains more details about a specific article which are not necessarily pertinent to rendering the graph, but may be interesting.

```
<info id=”[uid1]”>
      <title>[title]</title>
      <abstract>[abstract of 50 words]</abstract>
      <link>[wikipedia url</link>
</info>
```

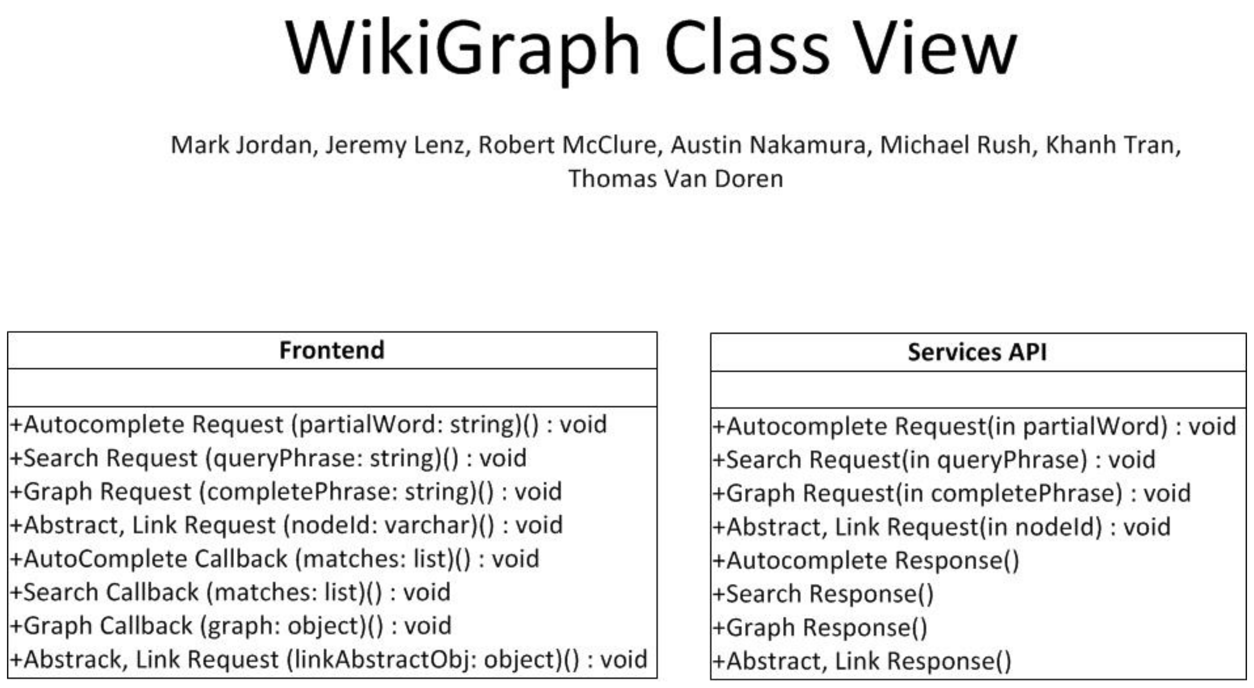
URL Specification

WikiGraph will be deployed to `http://wikigraph.cs.washington.edu/`. The endpoints are listed below.

URL Endpoint with Query String	**Description**
`/graph/`	Access the flash client
`/`	Redirects to /graph/
`/test/`	Access the test flash client. This version may include bugs and is not intended to represent a released version of the WikiGraph product. It may contain features which are under development. It is intended for developer use **only**.
`/api/autocomplete/?q=<partial-phrase>`	Returns a list of all article titles which match the partial phrase.
`/api/search/?q=<search-phrase>`	Returns a list of all articles which match the phrase.
`/api/graph/[<node-id>/][?t=<title>]`	Returns a graph for the given node-id, if specificied. If, for some reason, a title is specified instead of a node id, the article with the best matching title is returned. A HTTP 4xx response is returned if the specified node cannot be found or neither a node id or article title was specified.
`/api/abstract/<node-id>/`	Returns the title, abstract, and hyper link to wikipedia for the specified article with uid, node-id.
`/test-api/*`	Identical purpose to the `/api/` endpoint, however new features, bugs, and inconsistent behavior may exist. This is a place to find bugs and try new features before a deploying to the production version. It has the same endpionts as `/api/`. It is intended for developer use **only**.

Design View

The class view below describes the requests, responses, and callbacks each of our two modules handle.



WikiGraph Class View

Mark Jordan, Jeremy Lenz, Robert McClure, Austin Nakamura, Michael Rush, Khanh Tran, Thomas Van Doren

Frontend

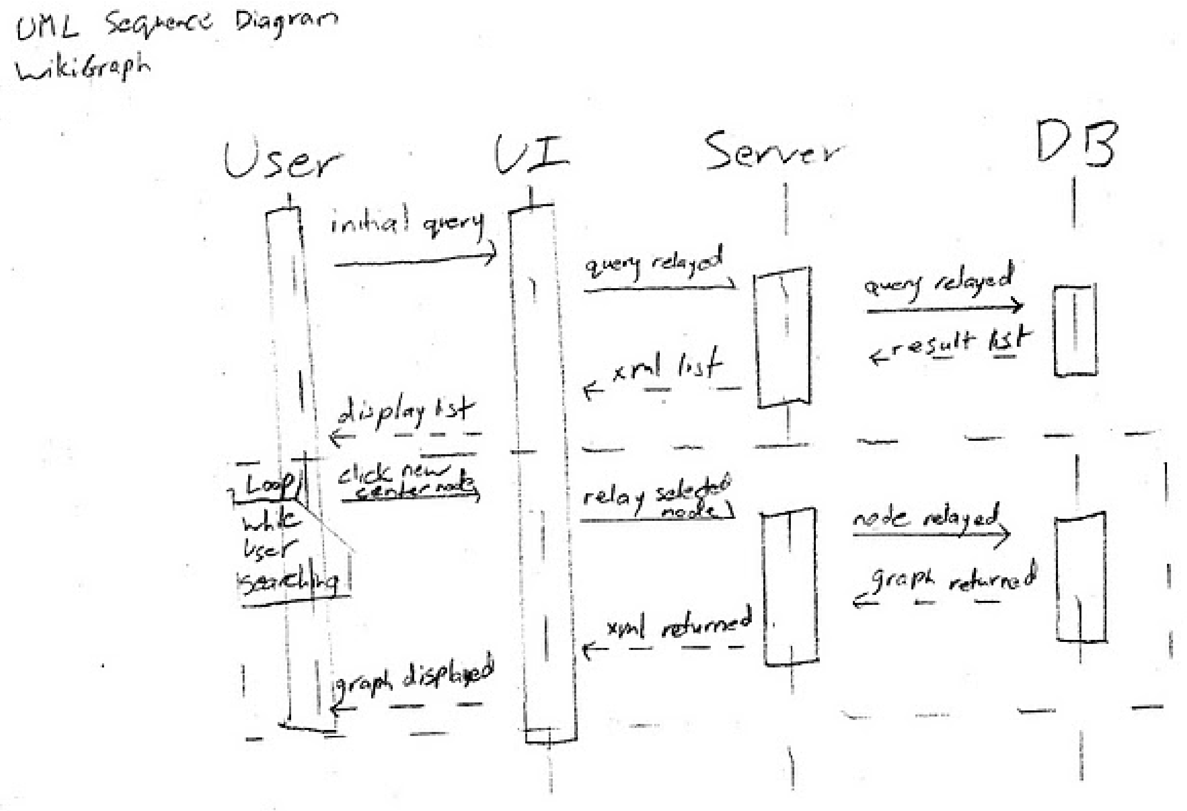
+Autocomplete Request (partialWord: string)() : void
+Search Request (queryPhrase: string)() : void
+Graph Request (completePhrase: string)() : void
+Abstract, Link Request (nodeId: varchar)() : void
+AutoComplete Callback (matches: list)() : void
+Search Callback (matches: list)() : void
+Graph Callback (graph: object)() : void
+Abstrack, Link Request (linkAbstractObj: object)() : void

Services API

+Autocomplete Request(in partialWord) : void
+Search Request(in queryPhrase) : void
+Graph Request(in completePhrase) : void
+Abstract, Link Request(in nodeId) : void
+Autocomplete Response()
+Search Response()
+Graph Response()
+Abstract, Link Response()

WikiGraph
Mark Jordan, Jeremy Lenz, Robert McClure, Austin Nakamura, Michael Rush, Khanh Tran, Thomas Van Doren

Process View



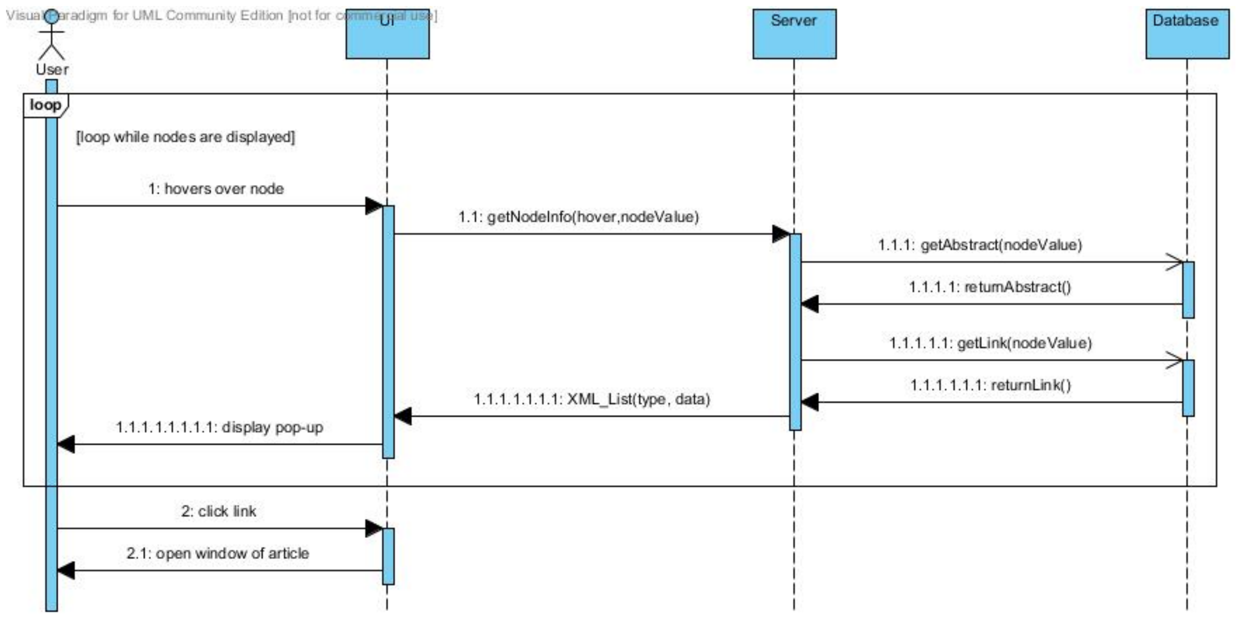
The User and UI are always active, but the Server and DB only need to be active when there is a request to process. The UI's calls will always be asynchronous, so the UI does not freeze.

The User queries the UI with the intial query. The UI sends the request to the server, which queries the database. An XML result is returned with the possible autocomplete title pages of the user's search, which are displayed.

The User can then loop, selecting wiki pages, represented as nodes, as the center of the graph. Each time, the request is relayed to the database, the server returns an xml result to be parsed by the UI, and is displayed as a graph for the user.

WikiGraph

Mark Jordan, Jeremy Lenz, Robert McClure, Austin Nakamura, Michael Rush, Khanh Tran, Thomas Van Doren


Visual Paradigm for UML Community Edition [not for commercial use]
User
UI
Server
Database
loop
[loop while nodes are displayed]
1: hovers over node
1.1: getNodeInfo(hover,nodeValue)
1.1.1: getAbstract(nodeValue)
1.1.1.1: returnAbstract()
1.1.1.1.1: getLink(nodeValue)
1.1.1.1.1.1: returnLink()
1.1.1.1.1.1.1: XML_List(type, data)
1.1.1.1.1.1.1.1: display pop-up
2: click link
2.1: open window of article

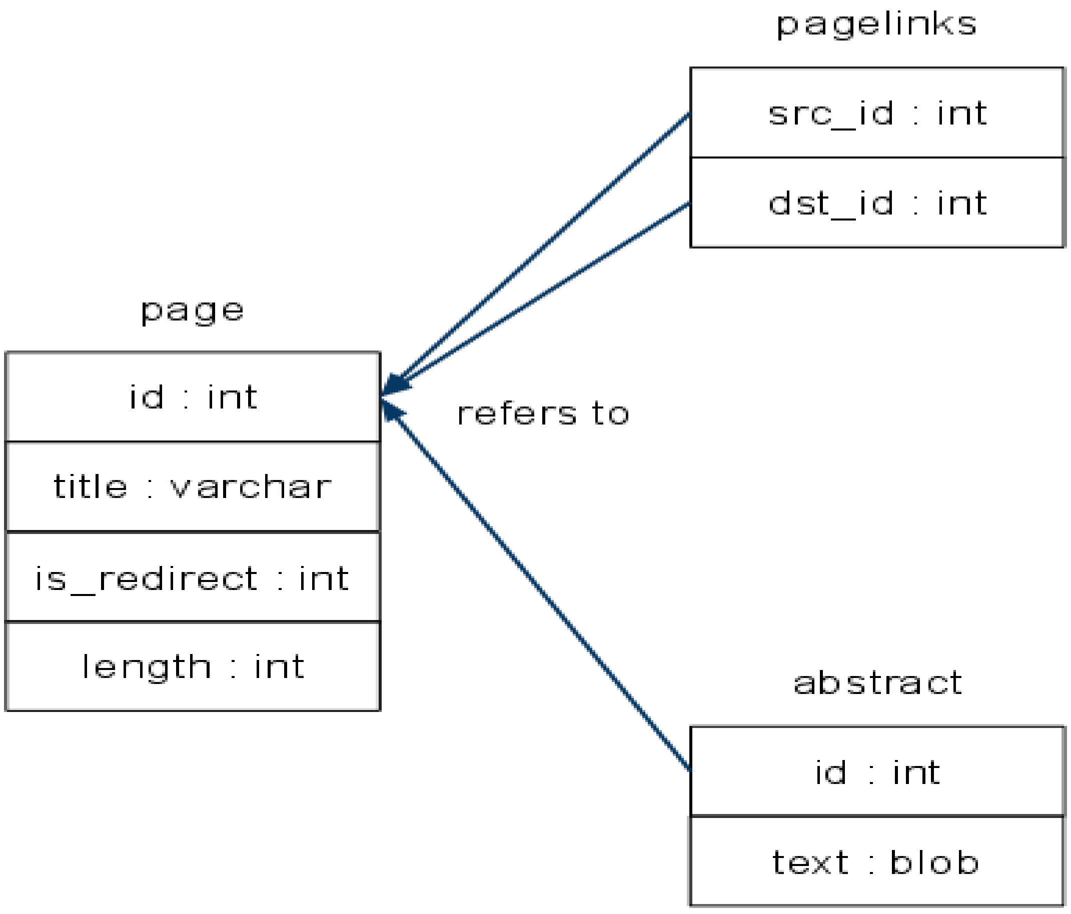
The above diagram details the process of the mouse-over or hover action on nodes, which creates a pop-up with finer detail about the node. The user is naturally always attentive as contrasted to the ui, server, and database which will process requests when queried.

The user initiates the ui with a hover over action, in which the ui sends a request to the server for the given node information. The server then sends multiple requests to the database acquiring data like the url to the wikipedia page and the abstract of the wikipedia article. the server then relays this back to the ui in an XML format, which is processed by the ui and displayed in the pop-up.

If a user then clicks on the wikipedia link provided in the pop-up, the ui will then request the browser to open a new window/tab to the nodes wikipedia article.

Database Schema

The database schema is a subset of the MediaWiki schema, which is then modified to better suit our application. A diagram of the schema is given below:



The page table holds basic information about each page. The pagelinks table holds an entry for each link in a page, including which page the link points to. The abstract table holds the first section of each page, which is usually its abstract.

Build System

WikiGraph is continuously built using the Hudson Continuous Integration service. The most recent builds for both the dev and release branches may be found at:

http://wikigraph.cs.washington.edu/build

A 'one step' build script is used in Hudson to clone the repository, checkout the appropriate branch, build the various binaries, and run unit tests. The pertinent executables and resources are then packaged and stored on the server where they can later be deployed.

A developer is never responsible for 'building' WikiGraph. Whenever Hudson recognizes a change in the repository, it builds a new version of WikiGraph. If errors are encountered during the build, the developer(s) responsible for the changes are notified via e-mail that their changes caused a build failure. The build supervisor is also notified. New commits will kick the build and hopefully the developer(s) have fixed the issues.

The advantage to this system is that the most recent build for various branches is available in the build artifacts. This makes it easy to deploy the application for testing and releases.

Design Alternatives and Assumptions

Services Alternative

For the backend, three typical choices (PHP, Django, and Ruby on Rails) were considered for our implementation. Django and Rails are both fully-featured web development frameworks that use a Model-View-Controller layout to build the web application. For many applications, using a framework aids in the cleanliness and speed of application development, especially if the project is quite large. However, it quickly became apparent to our team that our backend service provider was not going to be complex. In particular, we were going to use an essentially external database, and we were not going to be serving large number of web pages. Large portions of the web frameworks deal with serving web pages and building its own database, so it did not make sense to go through all of the extra work to use a framework which provided much more than we needed.

As a result, we decided to use a simple PHP services approach. PHP let us easily associate each action that we wanted to perform (search, autocomplete, graph, etc.) with a URL. The frontend could then fetch the data it needed from the appropriate URL, passing in parameters as needed. Since this type of functionality was all that we needed, PHP seemed the most appropriate solution.

Frontend Client Alternative

For the front end, we considered a wide variety of languages, including JavaScript, HTML, CSS, Canviz and Flash. We wanted a to use a language that would be easy to learn, because of the time constraints of the project, and so we could focus on making the program accessible to users. Additionally, because our program is visual in nature, we also wanted a language capable of efficient graphical manipulation.

Eventually, we decided to use Flash, since its widespread use also guaranteed a large amount of documentation, which would significantly speed up of the learning process. Flash’s emphasis on visual presentation also gave it a significant edge over languages which were intended less for heavy graphic manipulation. Finally, since Flash is so widely used, most users will immediately be able to access and use the client, with little to no problems.

Development Plan

Team Structure

Each team (frontend and services) will meet at their own convenience. The manager will try to attend all of these meeting, but the meetings will not cater to the manager's schedule. There is a meeting of the teams scheduled for Saturday at 1pm. The manager and one representative from each team is required to attend this meeting. Topics of discussion for the Saturday meeting include: weekly status report, goals for the following week, benign blocking issues across teams (urgent issues are addressed immediately), upcoming deadlines, required assets and resources, and anything that is pertinent to the entire team.

The entire team may be reached at wiki-map-uw-cse@googlegroups.com. The individual teams are not aliased specifically. If necessary, e-mail the manager to have messages forward to a specific team.

The developer and customer wiki is available through the Google Code project at:

http://code.google.com/p/wiki-map-uw-cse/

The team structure:

Manager

- Thomas Van Doren, thomas.vandoren@gmail.com

Services Team

- Jeremy Lenz
- Mark Jordan
- Rob McClure

Frontend Client Team

- Austin Nakamura
- Michael Rush
- Khanh Tran

Project Schedule

The following schedule represents a subset of the continuously changing schedule the WikiGraph developers use to track progress. The following table is accurate up through the alpha release. After that, items are likely to change.

Task/Milestone	Effort (days)	By (date)	Owner(s)
Setup google code project	0.5	Jan-18	Thomas
Research wikipedia dumps	0.5	Jan-22	Mark, Rob
Acquire and configure wikigraph.cs	4	Jan-24	Thomas
Get wikipedia dumps	0.5	Jan-29	Mark
Setup small test db on cubist	1	Jan-29	Rob
Install reviewboard on cubist	2	Jan-29	Thomas
Install Hudson on cubist	1	Jan-29	Thomas
Configure reviewboard on cubist	1	Feb-5	Thomas
Configure Hudson	1	Feb-5	Thomas
Write simple build script for producing SWF from MXML	1	Feb-5	Thomas
Write portable script to reorganize db dumps	1	Feb-5	Mark
Get bug tracking configured, working, and going in google code	1	Feb-5	Thomas
Parse wikipedia abstracts from XML and put into db	3	Feb-5	Rob
PHP implementation for alpha: recognizes a title from a GET request, queries db, outputs simple graph data for title. Only consider happy-path	5	Feb-5	Jeremy
Research S3 capabilities as a db	1	Feb-5	Rob
Research PHPUnit vs. homemade PHP unit tests	1	Feb-5	Jeremy
Draw simple UI containing search bar/button	1	Feb-5	Michael
Ping the data services, and receive an XML object	3	Feb-5	Michael
Parse node list from XML	4	Feb-5	Khanh
Draw nodes for each node in list	4	Feb-5	Austin
Test alpha release services	1	Feb-6	Frontend Team
Test alpha release client	1	Feb-6	Services Team
Build and deploy alpha release	1	Feb-7	Thomas
Alpha Release Total:	**39.5**	**Feb-7**	
Get XML specific to search term in flash client	4	Feb-14	Michael
Recenter node functionality in flash client	4	Feb-14	Austin
Setup AWS test database	1	Feb-14	
Write complete build script including unit testing, correct package output, and source hierarchy for easy deployment	5	Feb-14	Thomas
Autocomplete in flash client works	4	Feb-19	Michael
Draw tooltips with abstract and links in flash client	4	Feb-19	Austin

"More Result..." functionality works in flash client	8	Feb-19	Khanh
Setup database on AWS	3	Feb-19	
Switch services to AWS databases	2	Feb-19	
Test beta release services	1	Feb-20	Frontend Team
Test beta release client	1	Feb-20	Services Team
Build and deploy beta release	1	Feb-21	Thomas
Beta Release Total:	**38**	**Feb-22**	
Implement animation in flash client	4	Feb-27	Khanh
Search history and caching to view previous searches in flash client	8	Mar-6	
Option view to designate degrees, persistent nodes, zoom, colors in flash client	8	Mar-6	
Finalize services	8	Mar-6	
Finalize databases	8	Mar-6	
Test final release services	1	Mar-7	Frontend Team
Test final release client	1	Mar-7	Services Team
Build and deploy final release	1	Mar-8	Thomas
Final Release Total:	**39**	**Mar-10**	

Risk Assessment

The top WikiGraph risks at this point:

Risk	Chance	Impact	Mitigation Steps	Contingency Plan
Flash Client is too difficult to implement	L	H	Research Flex, MXML, and AS	Use JavaScript (lib: CanViz), HTML, CSS to implement frontend client.
Database Performance too slow for user interaction	M	M	Shard database, minimize columns, use efficient and optimized queries when accessing db	Use a smaller database dump from wikipedia or a smaller wiki-database from another source.
Storing entire WP database	M	M	Looking into AWS with RDS or S3	Use a smaller database on cubist.
Using AWS S3	M	M	Looking into PHP Plugin for storing db on S3	Use AWS RDS and pay the higher rate.
Making the graph look nice	M	M	Styling the graph to maximize information	Limit the number of nodes displayed.

Test and Documentation Plan

Test Plan

Unit Test Strategy

Both the flash client and the services api will include unit tests to keep code consistent and functional. The goal is to ensure that code remains usable and without bugs as development progresses. Developers will create new unit tests as new modules, objects, and functionality are added.

These tests, and the code, will be reviewed by fellow developers to ensure consistency. PHPUnit will be used to execute the services tests. FlexUnit will be used to test the flash client. Unit tests will be run by developers before committing code to the source repository. These tests will also be run during each build.

System Test Strategy

Note: the team does not yet know what a system test means...

An end-to-end test will be created whereby a complete request from the frontend client can be tested as it travels through the entire service, hitting the database, and then handling the results. Ideally, this test will be run at build time.

Usability Test Strategy

Each team will test the other teams progress. The flash client will consistently test the services, since it is dependent upon them for a lot of its functionality. The services team will, at least once a week, test the flash client to make sure things are intuitive and that expectations are being upheld. This allows basic testing from non-expert users on a regular schedule.

WikiGraph customers will be asked to conduct usability tests prior to and post release. A specification for what features to try will be included in the request to maximize the coverage. Ideally, the customers will thoroughly test all features and functionality. This will provide a fresh perspective of the product, one which the developers cannot provide.

Summary of Test Strategy

The three aforementioned tests, unit, system, and usability, will provide a broad and deep understanding of the product as it is developed. They provide confidence that the product will not have serious errors when it is deployed to the public. The team can be certain that its code does not contain serious bugs. Ultimately this will lead to high uptime, consistent user experience, and a strong reliable product.

Bug Tracking Mechanism and Plan to Use

WikiGraph will track bugs with the Google Code issue tracker. All customers have been granted permission to file issues.

New issues will be created under the following circumstances:

1. A customer or developer will find a bug during a usability test and file a case.
2. Developers will file cases as they find bugs. If it is a bug that the developer can fix, a case does not need to be filed. However, if a flash client dev finds a services bug, a new case should be filed.
3. Code reviews: if a code review reveals an issue with some existing code, a new case should be filed by the reviewer and assigned to the appropriate team or developer.

Documentation Plan

WikiGraph is designed such that users will intuitively adapt to the UI based on standard layouts and simple flow. However, the home (splash) page will contain instructions on how to use WikiGraph. The home page is accessible via a link in the upper left corner of all pages in the flash client, thus allowing users to re-read the instructions at anytime with a single click.

The specifications for the XML response coming from the data services API will be kept up-to-date such that new clients may easily start making requests to the WikiGraph data API. These specs will be kept on the Google Code wiki so that developers may easily find them.

An administrator's guide to downloading the wikipedia database dumps, reorganizing tables with the WikiGraph scripts, and then updating the existing tables will be available. The dumps for Wikipedia occur about once a month, so documenting this process is important.