MediaWiki

An architectural overview

TU Wien
184.159 Software Architecture, 2014W

Group 19
Cismasiu Anca (0727280)
Gamerith Stefan (0925081)
Krapfenbauer Klaus (0926457)
Reithuber Manuel (0725031)
Schreiber Thomas (1054647)

Table of contents

Information Sources

As our main information sources, we used the chapter provided in the book, the documentation provided by the MediaWiki-project[1] as well as the source code itself. For more information about this process, please refer to the experience report.

Links to the information sources we used for recovering the architecture:

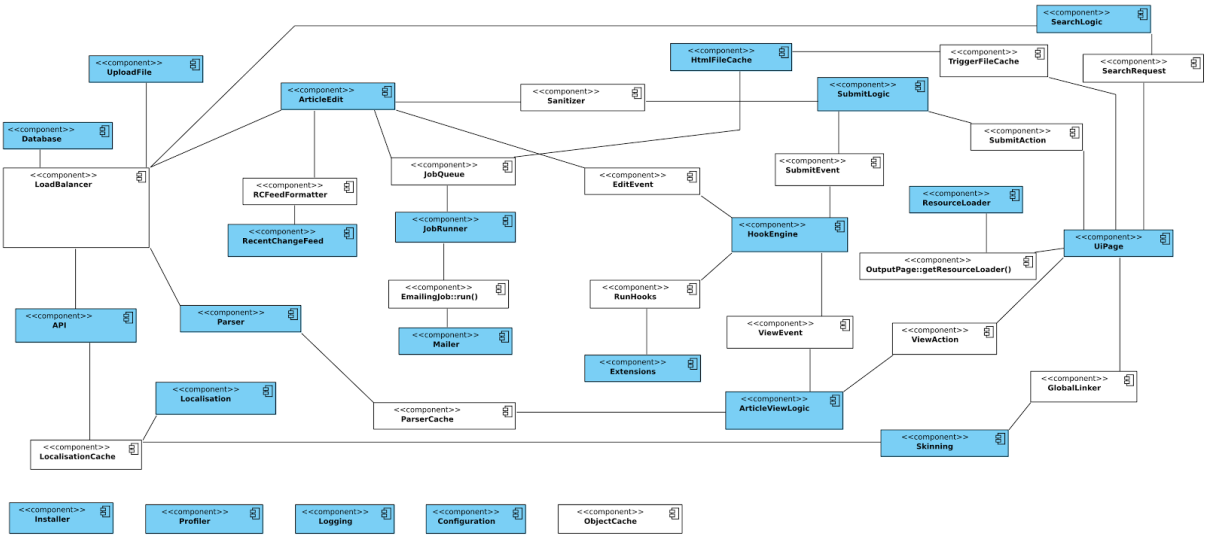
- http://www.mediawiki.org/wiki/Category:MediaWiki_components for a rough overview of MediaWiki components (which are not equal to the set of architectural components)
- Database layout
  https://www.mediawiki.org/wiki/Manual:Database_layout
- Localisation documentation
  https://www.mediawiki.org/wiki/Localisation#Overview_of_the_localisation_system
- Code documentation generated by the doxygen documentation generator
  https://doc.wikimedia.org/mediawiki-core/master/php/html/
  Class hierarchy https://doc.wikimedia.org/mediawiki-core/master/php/html/inherits.html
- The Architecture of Open Source Applications
  http://aosabook.org/en/mediawiki.html
- Source code of MediaWiki v1.24
  http://releases.wikimedia.org/mediawiki/1.24/
- The MediaWiki parser, uncovered
  http://musialek.org/?p=94
- Memcached
  http://memcached.org
- Alternative PHP Cache
  http://php.net/manual/en/book.apc.php
- XCache
  http://xcache.lighttpd.net
- PHP Caching and Acceleration with XCache
  http://www.jasonlitka.com/2006/12/20/php-caching-and-acceleration-with-xcache/
- Wikipedia - a very cheap story
  http://domasmituzas.files.wordpress.com/2011/09/velocity2008-wikipedia.pdf
- Wikipedia: site internals, configuration, code examples and management issues
  http://domasmituzas.files.wordpress.com/2011/09/mysqluc2007-wikipedia-workbook.pdf
- Book: MediaWiki by Daniel J. Barret, published by "O'Reilly Media, Inc. in October 2008, First edition; ISBN: 978-0-596-51979-7
- Redis Quickstart
  http://redis.io/topics/quickstart
- PHP XML Document Object Model
  http://php.net/manual/de/book.dom.php

[1] www.mediawiki.org/wiki/

- The overall PHP Manual including function reference documentation (especially for custom autoloading stuff and the Standard PHP Library)
  http://php.net/manual/en/

Architecture

Connector & Component view



In the above Connector & Component view, we try to give an overview over MediaWiki's architecture. The main components are featured, as well as the main connectors. Since we treat connectors as first class citizens, our connectors are given in the same notation that components are, but are recognizable by the white background.
Many of MediaWiki's components are connected by global functions or method invocations, where the class resolution is done by a class loader provided by MediaWiki (/includes/AutoLoader.php). To keep the view clean (since otherwise a large portion of the components would simply be connected to the class loader or by global functions), we omitted these portions of the architecture.
The Connector & Component diagram is provided a second time as an appendix as a vector graphic in pdf format, which allows to zoom in without losing picture quality.

Description and mapping to the source code

Components

API

Besides index.php, the API is the other main entry point for requests to MediaWiki. It provides access to MediaWiki's functionality for automatic tools (such as external software or bots). As an example, the generation of RSS feeds is powered by the API component. The API component is located in /includes/api/Api*, where /includes/api/ApiMain.php features as a gateway to the concrete API functionality.

Most API classes query the database directly (fetching a database connection using LoadBalancer) or call methods of their HTML counterparts and pretty much replicate the functionality of the normal web site (which is the reason for us not going into further detail here).

As connector to the requesting visitor, the api is connected by method invocation in /api.php ($processor->execute()).

The API supports a lot of different formats (implementations of ApiFormatBase), JSON and XML being likely the most prominent ones.

Configuration

The configuration component provides configuration values for the MediaWiki installation. The default values are stored in /includes/DefaultSettings.php. Installation-specific values are provided in /LocalSettings.php, the file which is created by the Installer component.

The Configuration connects to all other components via global variables with the prefix '$wg'. These variables are loaded in the /includes/WebStart.php-helper file.

Installer

The installer provides assistance for the set-up of a MediaWiki-installation. It is called by method invocation in /includes/WebStart.php in case that the installer was not executed yet (which is recognized by the definition of MW_NO_SETUP in the Configuration component).

The implementation of these component is located in /includes/Setup.php and /includes/installer/*. It is connected to the database and to the Configuration by creating the file /LocalSettings.php as a result of the process.

Logging

Since it is possible to run MediaWiki in environments that differ in language and style, a central logging system for debug- and error-logging is needed. This is provided by the Logging-component. The implementation of this component is provided in /includes/GlobalFunctions.php.

The Logging component is connected to all other components via the global function calls in /includes/GlobalFunctions.php, e.g. wfDebug() or wfErrorLog().

Profiler

As mentioned below in the part about the performance-quality, one of MediaWiki's main objectives is performance. The Profiler component is used to provide methods to measure the performance and find out bottlenecks in the installation. These bottlenecks can then be addressed by other components (e.g. by using a different caching system).

The implementation of the Profiler component is mainly done in the /includes/profiler-folder. There, MediaWiki offers multiple profiler implementations that mainly differ in form of their storage backend (database, text files, etc.).

The Profiler component is made accessible to other components via method invocations in /includes/profiler/Profiler.php. This file is loaded by /includes/WebStart.php. The profiler has

to be activated by creating a /StartProfiler.php-file that further specifies which profiler implementation should be used.

Database

The /includes/db folder contains files that relate to the Database component. /includes/db/Database.php defines a database interface and abstract class, which are wrapped or implemented by the different native abstraction layers for different database types(MS SQL Server, MySQL, MySQLi, Oracle, Postgres, SQLite), called Database<TYPE>.php

ChronologyProtector.php manages the LoadBalancer objects, handling the initialization and shutdown of load balancer. It ensures consistent event chronology by delaying execution on a slave until the data available on that slave's master is up to date.

SearchLogic

Basically MediaWiki does not store plain text data. Instead for compatibility reasons, PHP-Objects are serialized and persisted as BLOBs (Binary Large Objects). This has the implication that the search capabilities of the DB-Query Engine can not be used. To overcome this restriction an search index is dynamically created during each insert/update operation. The respective classes for this kind of operation are: includes/search/SearchPostgres, includes/search/SearchSqlite, includes/search/SearchMySQL, includes/search/SearchMssql and includes/search/SearchOracle. Each class provides a vendor specific search mechanism through indices. For example the MySQL implementation uses the Fulltext search capabilities of the MYIASM-Engine.

Extensions

MediaWiki was designed to be extendable from ground up. This mechanism is done through proper configuration of global variables or functions. Since each MediaWiki installation has its own LocalSettings.php file for application dependent configuration settings, installing an extension is just as simple as adding a couple of lines of code in the respective configuration file.

Besides the general extension capabilities there exists an number of other special extension mechanisms, namely Tag-Extensions, Parser-Extensions, Skin-Extensions, MagicWord-Extensions and API-Extensions. Below is a brief description of each of these extension types:

- *Tag Extensions*
  With Tag Extensions it is possible to extend the builtin wiki markup with additional tags as e.g. “<donation/>” . In order to facilitate this behavior it is necessary to register a callback function by adding the respective initialisation function to the global array “$wgHooks[‘ParserFirstCallInit’]”. The added initialization function then should register another function for processing the new tag. Again, as with any extension, registration is accomplished by inclusion in LocalSettings.php .

- *Parser Extensions*
  Parser Extensions are useful if a developer wants to trigger a function call by wiki markup language as e.g “{{ #functionname: param1 | param2 | param3 }}” . This

extension is triggered in the same manner as Tag Extensions. The only difference is the registration by the method “setFunctionHook()” instead of “setHook()”.

- *Skin Extensions*
  Although MediaWiki has a default look it is possible to provide a custom skin. There exists a number of different skins[2] on the WWW. To install a different skin just unpack the archive file for the respective skin, drop it into the skin subfolder and include it by adding a new line “include(newskin.php)” to LocalSettings.php .

- *MagicWord Extensions*
  Magic words are like shortcuts for user applications. Common bahavior, variable names or parser functions can be accessed in an abbreviated form enclosed in double-curly braces. E.g. {{ __TOC__ }} which places the table of contents at the word’s current position. In order to provide custom magic words an arbitrary php-file has to be assigned to the global array “$wgExtensionMessagesFiles[]” .

- *API Extensions*
  API Extensions extend the core API of MediWiki. As with any extension it needs to be included in LocalSettings.php and added to the global variables “$wgAPIModules” (for Top-level modules), “$wgAPIPRopModules” (for Property modules), “$wgAPIListModules” (for List modules) and “$wgAPIMetaModules” (for Meta modules) . For example a typical use case for an API extension would be providing statistics. API Extensions are accessible through api.php and should output XML-formatted data.

Parser

The Parser is both messy and essential for translating the MediaWiki markup language into valid HTML. It started out relatively small and simple, but as new additions were made the parser got more and more complex. Nowadays no one fully understands the whole parsing process in detail, so there are hardly any changes to the core parsing logic.

Nevertheless the overall parsing process can be divided into two parts. The preprocessing and the linking. The former is mainly done in includes/parser/Preprocessor_DOM.php and includes/parser/Preprocessor_Hash.php . These two Preprocessor implementations are basically the same but the first one is faster because it takes advantage of PHPs internal DOM parser. Both preprocessors basically turn raw wiki markup into a tree structure for further processing.

The second step in the parsing process is the linking. This is done in the parse() function in includes/Parser.php . As links are probably one of the most essential items in MediaWiki, clients needs fast access to them and they need to quickly distinguish if two links relate to the same target without searching the whole link text. In order to accomplish this, all links are replaced by a unique id and effectively stored for later comparison. Then all links in the text are replaced with the respective link retrieved from a previous step and forwarded for output.

[2] http://www.mediawiki.org/wiki/Manual:Gallery_of_user_styles

UploadFile

MediaWiki provides full-blown image and file management in its UploadFile component. For each image, several versions with different resolutions are saved and can be embedded into articles.
There's also the class ImagePage to view each image's details and some Special pages to find unused images, duplicates, etc. (SpecialListDuplicatedFiles, SpecialUnusedImages)
The uplaod code itself is in includes/upload/UploadFromFile.php.

ArticleEdit

The ArticleEdit encapulates all functionality that is needed to get the edited version of an article into the database. The source of this component is located in /includes/EditPage.php as well as in /includes/page/WikiPage.php.

The ArticleEdit component is connected to the database via the LoadBalanceConnector which is invocated in /includes/page/WikiPage.php:doUpdate() via the global function call 'wfGetDB). The ArticleEdit component itself gets the new article text from the SubmitLogic over the SanitizerConnector and is called after all sanitizers in /includes/EditPage.php:internalAttemptSave() via the replaceSectionContent()-method.

Mailer

The Mailer component sends e-mails notifying users about article changes and the like. Sending mails is done asynchronously using MediaWiki's JobQueue. Most of its implementation resides in UserMailer.php which utilizes either the PEAR mailer or PHP's internal mail() function.
The asynchronous part of the mailing component is in EmailingJob.php which extends the JobQueue's Job interface.

SubmitLogic

SubmitLogic encapsulates the logic behind saving the edit of an article before it is saved to the database. The source of this component is mainly located in /includes/EditPage::edit() and /includes/EditPage/attemptSave(). In the submit logic, user permissions are checked, the sanitizers are executed and the edit is attempted to be saved. In case of an successful save as well as in the case of an error, the user is presented the corresponding message.

The SubmitLogic is connected to the UIPage via indirect method invocations. It is connected to the ArticleEdit-Logic via the SanitizerConnector via method incovations in /includes/EditPage.php:internalAttemptSave() and to the Hook Engine via registering the submit event and calling registered hooks on this event.

UiPage

UiPage is our web endpoint of MediaWiki. Most of the requests go to index.php (using URL rewriting) which includes /includes/MediaWiki.php. The MediaWiki class then executes the action requested by the user and displays the right page.
It also tries to get a pre-cached version of the file from HtmlFileCache or alternatively stores the computed pate in the file cache if enabled and applicable (not all pages are considered cacheable).
And it applies the page design (aka skin) and prepares the resource links (to javascript/css files) using the respective components.

ResourceLoader

As part of MediaWiki's performance efforts, the resources (JavaScript, CSS) are loaded centrally by the ResourceLoader. The ResourceLoader also handles minification and caching of scripts, deferred loading, inlining[3] along with many other features.

The sources of this component are contained in /includes/resourceloader, the main file being /includes/resourceloader/ResourceLoader.php. The ResourceLoader is connected to the UIPage component via method invocation in /includes/OutputPage.php.

RecentChangeFeed

The RCFeed ("Recent changes feed") provides an overview about recent changes on the MediaWiki installation.

There are two types of engines available: UDP (/includes/rcfeed/UDPRCFeedEngine.php) and Redis Publish/Subscribe (RedisPubSubFeedEngine.php) which implement the /includes/rcfeed/RCFeedEngine.php interface. In the UDP Engine implementation, the packet is simply sent using the UDP protocol, whereas the Redis Engine follows a Publish/Subscribe architecture style.

Localisation

The languages/Language.php is the parent class that contains a factory method to create a subclass for each language, defined in languages/classes. The Message objects do most of the internationalization, but the parent class contains the localizable message strings found in languages/messages and important language-specific settings and language-specific functions, like uppercasing, lowercasing, printing dates, formatting numbers, direction, custom grammar rules, construct lists, etc.

Article View Logic

In the Article View Logic the business logic in creating the view for a single MediaWiki article is contained. The main logic is contained in /includes/page/Article.php.

The Article View Logic is connected to the Parser (and indirectly to the database) via the Parser Cache component that is invoked in /includes/page/Article.php:view(), to the Hook Engine via triggered View events and to the UI Page component via global variables (GlobalLinker in our view).

JobRunner

The JobRunner component allows to run the Jobs that are currently waiting for deferred execution in the JobQueue (for definition of Jobs and the JobQueue please refer to this chapter). The JobRunner component is located in /includes/jobqueue/JobRunner.php.

After each client request there's a certain possibility that MediaWiki::triggerJobs() sends an asynchronous HTTP request to the unlisted special page SpecialRunJobs. Therefore background processing is achieved by triggering a new external request to the Site[4].

The JobRunner component is (of course) tightly connected to the JobQueue. As an example for a job we picked the Mailer component that puts EmailingJobs in the JobQueue. The

[3] https://www.mediawiki.org/wiki/ResourceLoader/Features
[4] http://www.mediawiki.org/wiki/Manual:Job_queue#Changes_introduced_in_MediaWiki_1.23

run()-method of these jobs is then in turn called by the JubRunner component via direct method invocation.

HtmlFileCache

The HtmlFileCache handles the saving to and loading from a object file cache on the file system of the rendered HTML view pages. The source is located in /includes/cache/HTMLFileCache.

HookEngine

The HookEngine consists of the functionality for registering and running hooks between all the other components. There are hooks everywhere in the project which can be registered in this central component. For example hooks are run before and after there is a new page submitted (SubmitEvent), a page is changed (EditEvent) and before and after pages are viewed (ViewEvent). The methods which are registered as hooks typically are implemented by an extension which gets the control flow with these methods.

The class includes/Hooks.php is the main class and central point for the HookEngine and contains the static methods for registering and running hooks. There is also a method wfRunHooks in the includes/GlobalFunctions.php exposed to the other modules which calls the Hooks class' run() method.

Skinning

Skinning is probably one of the big advantages of MediaWiki. New skins can be easily integrated into the runtime configuration. There exists a number of different skins[5] on the WWW. To install a different skin just unpack the archive file for the respective skin, drop it into the skin subfolder and include it by adding a new line "include(newskin.php)" to LocalSettings.php .

Connectors

Job Queue

The Job Queue-component provides functionality to enqueue long-running tasks without the need for cronjobs/crontabs. Other components, like ArticleEdit, are allowed to put their own jobs in the global Job Queue which will then ensure that these jobs are executed by the Job Runner Component.

The implementation of the Job Queue is provided in the /includes/jobqueue folder, where /includes/jobqueue/JobQueue.php features as a main entry point. Jobs implement the Job interface and are being executed by the JobRunner component (for details please refer to this chapter).

The Job queue is connected to other components by a Singleton JobQueueGroup which allows to put jobs in the queue and execute them. Jobs by other components have to be a subclass of /includes/jobqueue/Job.php and their classes are registered by the global variable '$wgJobClasses'.

[5] http://www.mediawiki.org/wiki/Manual:Gallery_of_user_styles

EmailingJob::run()

The JobRunner is connected to the Mailer component through the run function in EmailingJob.

LoadBalancer

The LoadBalancer is connected to the Database component through the getConnection function in /include/db/LoadBalancer.php
The global function wfGetDB in GlobalFunctions.php gets a database object through the load balancer, through the LBFactory singleton. The database type can be chosen according to query type: the master database for write queries, or a slave for potentially lagged read queries.

This global function is the endpoint of the connection with the other components. The /includes/ApiBase.php getDB() function calls wfGetDB.
Similarly, the Parser component calls it in fetchScaryTemplateMaybeFromCache($url) in /include/parser/Parser.php to get the master for write queries and a slave for read queries.

RCFeedFormatter

On every recent Change save, the /includes/changes/RecentChange.php function notifyRCFeeds(array $feeds = null) notifies all feeds of the change. It formats the data accordingly for each feed. The available formatters are: IRCColourful, JSON, MachineReadable, XML.

ObjectCache

Since MediaWiki has strict performance requirements, an ObjectCache was introduced to store arbitrary PHP-Objects for faster access. MediaWiki provides a number of different ways to to store cached objects. These places for storage are the database (default), the file system, external systems or even custom storage solutions usually provided as an extension. Custom ObjectCache implementations are registered by the global variable '$wgObjectCaches'. Custom configurations should be done in a file named LocalSettings.php.

The ObjectCache itself is located in a class named ObjectCache which is located in the file includes/ObjectCache.php . This class provides a generic factory method to invoke different ObjectCache implementations depending on the above mentioned configuration variable '$wgObjectCaches'. Each ObjectCache implementation extends the class BagOStuff located in includes/objectcache/BagOStuff.php as includes/objectcache/APCBagOStuff.php (for APC[6]), includes/objectcache/HashBagOStuff.php (in-memory hash table implementation), includes/objectcache/MemcachedBagOStuff.php (for Memcached[7]), includes/objectcache/SqlBagOStuff.php (default database caching implementation), /includes/objectcache/XCacheBagOStuff.php (for XCache[8]), /includes/objectcache/RedisBagOStuff.php (for Redis[9]) , /includes/objectcache/MultiwriteBagOStuff.php (an ObjectCache container class for multiple child ObjectCaches) and /includes/objectcache/EmptyBagOStuff.php (for no cache).

[6] http://php.net/manual/en/book.apc.php
[7] http://memcached.org/
[8] http://xcache.lighttpd.net/
[9] http://redis.io/

LocalisationCache

Connection to Localisation Component happens in languages/Language::getLocalisationCache(), by using the global variable $wgLocalisationCacheConf. The Language::getMessage function calls the LocalizationCache which handles the lazy loading of the messages from the back end cache, recaching of messages and language fallback sequence.

The API, Parser and Skin components use the global function wfGetLangObj from /include/GlobalFunctions.php for localization, which accesses the LocalisationCache through the Language component's factory method.

RunHooks

The RunHooks Connector comprises the wfRunHooks global function. When an event has occurred, the functions defined in the global variable wgHooks are run. This is how extensions are incorporated in the system.

OutputPage::getResourceLoader

The getResourceLoader function in OutputPage connects the UIPage component with the ResourceLoader Component.

Sanitizer

The Sanitizer connects the SubmitLogic component with the ArticleEdit component. It is called by the SubmitLogic in /includes/EditPage.php:internalAttemptSave() where the text submitted by the user is run through various sanitizers (XSS, Spam protection), a failure in a single one is resulting in a failure for the whole edit by the user.

The main Sanitizer class itself is located in /includes/Sanitizer.php.

EditEvent, SubmitEvent, ViewEvent

These are events which occur when submitting, changing and viewing a Wiki page. These events trigger the HookEngine to run the specified hooks before and after the event occurs. In the source code these events occur as calls to the wfRunHooks method. Below is an (incomplete) list of the hooks that are run for these events.

Hook method name	Source file	Line
ArticleAfterFetchContentObject	includes/Article.php	431
ArticleFromTitle	includes/Article.php	123
ArticleViewRedirect	includes/Article.php	998
EditPageBeforeEditChecks	includes/EditPage.php	3762
EditPage::attemptSave	includes/EditPage.php	1517
PageContentSaveComplete	includes/page/WikiPage.php	2019

GlobalLinker

MediaWiki has a dedicated class for registering and retrieving different kinds of skins, but only one skin can be set as the default skin. This class is called Skinfactory[10] . The default skin is retrieved through the configuration option “DefaultSkin” which is internally used[11] as the key to retrieve the default skin for display. The latter is accomplished by a call to the static method SkinFactory::makeSkin($skinName)[12].

SearchRequest

A search request is sent from the UIPage component through a method call in SpecialSearch::ShowResults to searchText($query) or searchTitle($query) in /includes/search/SearchEngine.php.

ParserCache

The ParserCache[13] connector obviously connects the Parser with the ArticleViewLogic. The connector can be setup using any of the mentioned caching mechanisms as e.g. Memcached or Database Caching. The main method for retrieving the content of the ParserCache is get()[14]. The counterpart this method is save()[15] which obviously saves the output produced by the parser for faster retrieval.

TriggerFileCache

The HTMLFileCache class exposes a couple of functions that make up this connector: HTMLFileCache::clearFileCache is used when the page is deleted or edited to trigger a cache clear and HTMLFileCache::newFromTitle is used on view actions.

SubmitAction

The SubmitAction connector connects the UIPage component with the SubmitLogic component. The connector is located in /includes/MediaWiki.php:performRequest() where in the performAction()-method in the same file, the SubmitLogic component is called by the call of the show()-method of the abstract class Action, where the EditAction has been initialized by the ActionFactory-class.

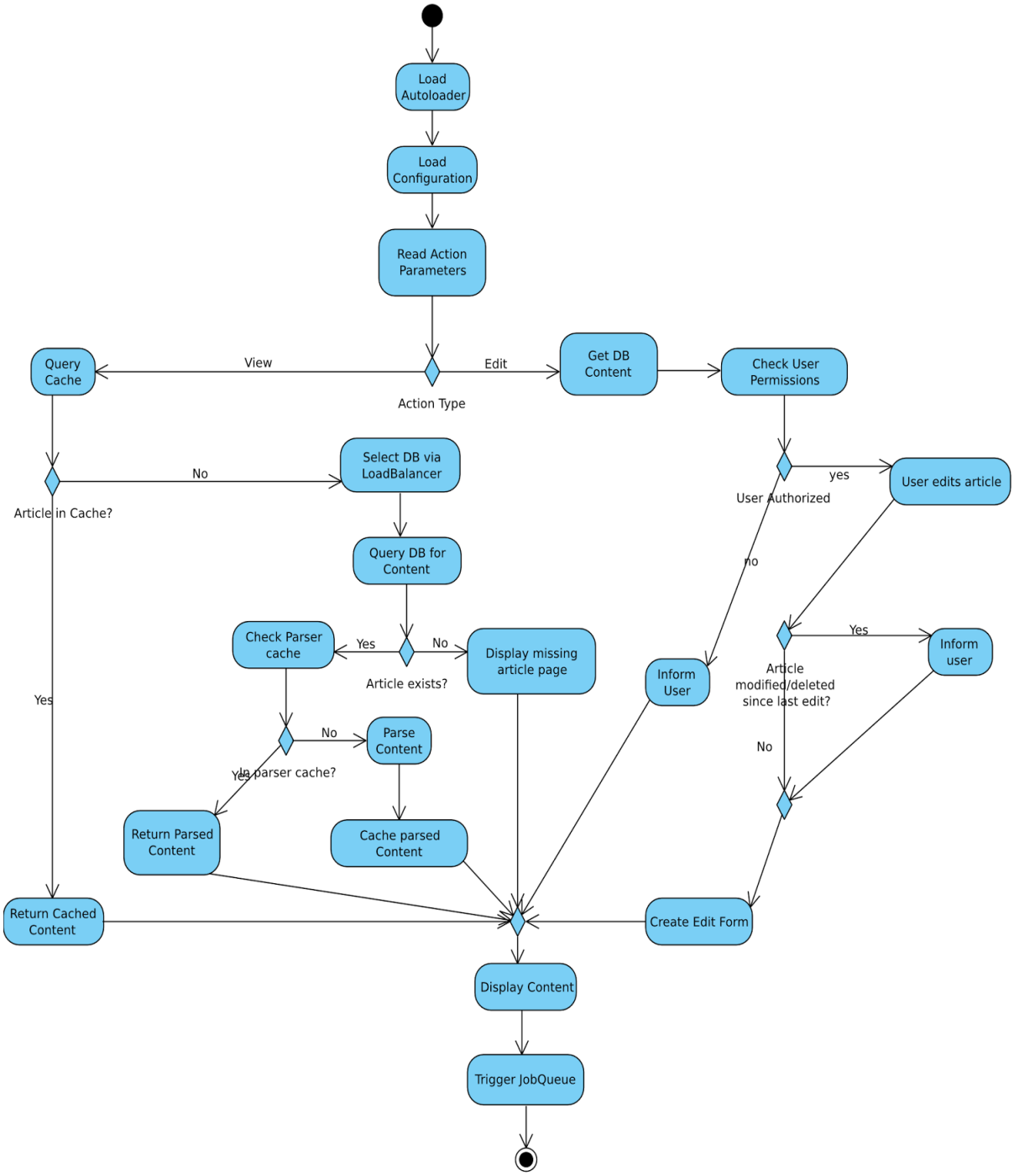
ViewAction

The ViewAction connector connects the UIPage component with the ArticleViewLogic component. The connector is located in /includes/MediaWiki.php:performRequest() where in the performAction()-method in the same file, the ArticleViewLogic component is called by the call of the show()-method of the abstract class Action, where the ViewAction has been initialized by the ActionFactory-class.

[10] This File is located in includes/skins/Skinfactory.php
[11] Line 386 in includes/context/RequestContext.php
[12] Line 395 in includes/context/RequestContext.php
[13] This File is located in includes/ParserCache.php
[14] Line 185ff in includes/ParserCache.php
[15] Line 247ff in includes/ParserCache.php

Complementary view: Activity diagram

Graphic representation



We saw an overview of many artifacts in the architecture of MediaWiki in the Component & Connectors diagram. They all seem quite loosely connected, so we chose an Activity diagram as our complementary view to demonstrate, how all these components interact. In our activity diagram, we show how the use cases “Show Article” and “Edit Article” are handled by MediaWiki’s architecture, which components and connectors are used when handling such a request and how this all interplays with MediaWikis strong focus on Caching.

A request is handled as follows: index.php (as the other main entry point besides API.php) is called by the visitor's webbrowser (e.g. "index.php?action=view&title=Main_Page"). First, the Autoloader registers the class loader function with php. After that, the configuration of MediaWiki is loaded, and the parameters of the visitor's request are parsed (in this case, "view" as action, "Main Page" as the title).

Already after that, the first cache is loaded. If there is a hit, for this article, the article is returned from the cache immediately, no further processing is necessary on the MediaWiki servers, keeping the process load low. If there is no cache hit for the page, the loadbalancer selects a suitable database.

Only after the database connection is returned by the load balancer, the database request looking for the article is made. In case that the article does not exist, the user is confronted with missing-article page. If the article is found in the database, it is queried a second time to look up a cached version of this article. If one is found, the cached version is returned, only otherwise the article is actually parsed and cached.

After the content is returned to the visitor, MediaWiki stats processing jobs that are currently in the JobQueue. The visitor's requests are used here like a crontab.

As shown in the Activity diagram, MediaWiki really takes a focus on caching. In the standard configuration, a cache is used even before a database query is made. By altering the configuration, these caches can be made even more prominent.

Mapping to the source code

In the following table, the mapping from the Activity view to the actual code in the current MediaWiki (v1.24.0)

Type	**Label**	**Code**
Start Node	-	/index.php:1
Activity	Load Configuration	/includes/WebStart.php:80-105
Activity	Read Action Parameters	/index.php:46 > /includes/MediaWiki.php:run():435> main():508 > getAction():139 > /includes/actions/Action:getActionName()
Decision	Action Type 'view' or 'edit'	/includes/MediaWiki.php:main():584 > :performRequest():282 > :performAction():414 (done via call on abstract class 'Action')
following branch 'view'		
Activity	Query Cache	/includes/MediaWiki.php:performAction():414 > /includes/actions/ViewAction:show():44 > /includes/page/Article:view():540ff
Decision	Article in Cache?	/includes/page/Article:view():569,575

Activity	Return Cached Content	/includes/page/Article:view():573,582
Activity	Select DB via LoadBalancer	/includes/page/Article:view():609 > /includes/page/WikiPage:exists():462 > :loadPageData():379 > (AutoLoader) > /includes/db/LoadBalancer:getConnection()
Activity	Query DB for Content	/includes/page/WikiPage:loadPageData():395 > :loadFromRow()
Decision	Article exists?	/includes/page/Article:view():609
Activity	Display missing article page	/includes/page/Article:view():611 > :showMissingArticle()
Activity	Check Parser Cache	/includes/page/Article:view():621
Decision	In Parser Cache?	/includes/page/Article:view():621
Activity	Return Parsed Content	/includes/page/Article:view():628
Activity	Parse Content	/includes/page/Article:view():688 > /includes/poolcounter/PoolCounterWork:execute():114 > /includes/poolcounter/PoolWorkArticleView:doWork():135 > /includes/content/AbstractContent:getParserOutput():490 > /includes/content/WikitextContent:fillParserOutput():338 > /includes/parser/Parser:parse()
Activity	Cache parsed Content	/includes/poolcounter/PoolWorkArticleView:doWork():149 > /includes/parser/ParserCache:save()
following branch 'edit'		
Activity	Get DB Content	/includes/MediaWiki.php:performAction():414 > /includes/actions/EditAction:show():56 > /includes/EditPage:edit():459 (> analog via LoadBalancer, see 'view'-branch)
Activity	Check User Permissions	/includes/EditPage:edit():495 > :getEditPermissionErrors()
Decision	User authorized?	/includes/EditPage:edit():496
Activity	Inform User	/includes/EditPage:edit():501 > :displayPermissionError():623 > /includes/OutputPage:formatPermissionErrorPage()
Activity	User edits article	various sources, all triggered by the post

		data in /includes/EditPage:edit():528
Decision	Article modified since last edit?	/includes/EditPage:edit():528 > :attemptSave():1292 > :internalAttemptSave():1693 > load article from DB, see 'view'-branch > :internalAttemptSave():1754
Activity	Inform user	/includes/EditPage:edit():553 > :showEditForm():2498 > :showConflict()
Continuing for both branches		
Activity	Display Content	/includes/MediaWiki:main():594 > /includes/OutputPage:output():2204
Action	Trigger JobQueue	/includes/MediaWiki:run():446 > :triggerJobs():644 > /includes/jobqueue/JobRunner:run():74 /includes/jobqueue/JobQueueGroup:execute ReadyPeriodicTasks()

Architectural styles

Looking at the source code, it becomes evident that MediaWiki's development process did not focus on strictly following textbook architectural styles and patterns.
As an example, the name of the interface IDBAccessObject (and its implementing classes) suggests a DAO pattern, but it's mostly business logic classes implementing it (and therefore accessing the database directly).
The following is a list of architectural styles that have been followed at least to a certain extent:

Client-Server Style

Looking at MediaWiki as a whole, we see a web application with the user's web browser (as well as API clients) being the client and the web server being the server. Also, the Three-Tiered Pattern comes to mind.

Object-Oriented Style

MediaWiki follows a Java-style class hierarchy. Almost each class has its own .php file (with the same file name as the class itself. Those .php files are split up into a number of directories quite like packages in Java (see: MediaWiki file naming conventions[16]).
There are however some global functions (prefixed with 'wf') and variables ('$wg...') which break with that style.

Blackboard Style

MediaWiki uses a lot of global variables (prefixed with '$wg') to provide core functionality. As these objects provide a global state, we can speak of the Blackboard style in that regard.

[16] http://www.mediawiki.org/wiki/Manual:Coding_conventions#File_naming

Most of the global variables are there for legacy reasons though. Newer code uses singletons instead[17].

Pipe and Filter Style

The Parser component (which translates wiki markup to (X)HTML) somewhat resembles the Pipe and Filter style. It takes text input and applies transformations to it (replacing magic words and other placeholders, translating wiki markup style formatting to HTML tags, etc.).

Publish-Subscribe

One of MediaWiki's core concepts regarding modularity are hooks. They are used extensively throughout the application and provide an easy way to react to any kind of in-app event synchronously.
The event handlers are registered in the global $wgHooks variable and are called by the emitting function using wfRunHooks()

Again, there is a slight deviation from the style: Event handler functions can influence the further distribution of the hook event by returning 'false' or an error string (which aborts further distribution of that event).

Event-Based Style

The JobQueue (which is responsible for executing long-running tasks asynchronously) follows an event based style.
The event bus is represented by the classes extending the JobQueue class (such as JobQueueDB or JobQueueRedis) and running the jobs is triggered with a certain chance after each incoming HTTP request (to achieve asynchronism, an HTTP request is issued to the unlisted special page SpecialRunJobs, which then processes all the jobs in the queue).

REST

MediaWiki offers an API (using the api.php endpoint) to external applications. It supports several different output formats and provides access to most of MediaWiki's functionality.

[17] see docs/globals.txt in the project's repository for details

Model Consistency

Consistent models are model with the absence of any contradiction. Consistency of course does not imply completeness as often misunderstood. Consistent models can be either complete or incomplete and the other way round. In our case there are some parts which are missing in the component & connector diagram and other parts are not covered in the activity diagram. For example the Autoloading part of the activity diagram is intentionally omitted in the component & connector diagram. On the other hand the Mailer component is intentionally not covered in the activity diagram because it just reflects the use cases “Show Article” and “Edit Article” .

In terms of showing model consistency it is sufficient to show that the models do not contain contradictions. In other words a mapping between these two components is adequate to prove model consistency.

Activity Model	Component & Connector Model
Load Configuration	Configuration
Decision “view” or “edit”	EditEvent, ViewEvent
Select DB via LoadBalancer	LoadBalancerConnector
Query DB for Content, Get DB Content	Database
Check Parser Cache, In Parser Cache?, Return Parsed Content, Cache parsed Content	ParserCache
Parse Content	Parser
User edits article	ArticleEdit
Display Content	ArticleEdit, UIPage
Trigger JobQueue	JobRunner, JobQueue

Table: Mapping between Activity Diagram and Component & Connector Diagram

Quality attributes

MediaWiki is a very big and commonly used project and as such it has to provide several non-functional qualities like portability, availability, modifiability, installability and performance to name just a few. For our analysis during this review we chose the two qualities *performance* and *modifiability* which are described below.

Performance

Since MediaWiki (more precisely, its predecessors) were run on one single or a just a few servers for a long time, performance has always been an issue that forced the developers to tweak out the best of their project. One of the reasons of this issue was and still is that the whole project has to succeed with a low budget because the Wikimedia foundation is funded by donations only. Another reason why performance became an issue very soon is, because of the fast growth in popularity of Wikipedia. These facts may not be of interest for the evaluation of the architecture of MediaWiki as it is now, but it is important for understanding why it is the way it is and how it became this way. This historical view also inspired us, for choosing performance as one of the reviewed qualities of MediaWiki.

Caching

Since the project is primarily based on a Client-Server architecture and since PHP does not save a state of objects per se, one of the tactics to achieve more performance was increasing the resource efficiency through different layers and methods of *caching*.
The idea of the overall caching orchestration was to save every state in between the architectural components in a cache by directly saving serialized instances of PHP objects. The benefit of this layered caching can be shown in a use case: If a user request for a page comes in, the caches are queried from top (UI components) to the bottom (database) and if a cached version of the page is found it is immediately returned if it is still valid. Thus the page might not have to get queried from the database, get parsed again or even regenerated into HTML.

Load balancer

Another architectural design decision for improving performance has been to distribute the database and maintain multiple copies of the data onto different servers to reduce the server load. This also introduces the possibility of concurrent database access. To enable the efficient use of these increased resources the component LoadBalancer (see the description in LoadBalancerConnector) has been introduced for managing the connected data stores.

Profiler

For getting an overview of where the performance can further be increased the developers implemented an own *profiler* for analyzing the project. This allows for showing where the most computation time is lost and afterwards optimizing the PHP code in an iterative process.

Reverse proxy

MediaWiki encourages site operators to place the web server behind a proxy to provide an additional layer of caching (and also a means for load balancing).
For the popular Squid proxy, there are even some helper classes to trigger cache invalidation (namely SquidUpdate, SquidPurgeClient).

Modifiability

Because of the relative long life of the project (nearly 14 years) *modifiability* had to be considered very often already.

Code modifiability

Since the original code was written in a procedural style, because at that time there were no classes in PHP, several modules had to be split over and over again to achieve a more object oriented style and increase modifiability. One example is the /includes/Title.php file which had more than 8000 lines of code previously and had been reduced to half of that size. Although the code is still not very well split into coherent modules due to the typical programming style in PHP and the historical reasons, one benefit from this style is, that the different components of the system are very loosely coupled because of the heavy use of global functions, global variables and hooks for decoupling the control flow.

Extensions

The core tactic for maintaining modifiability of the MediaWiki project is to provide mechanisms for extending the functionality without the need to modify the code of the project. This is achieved through deferring the binding of those *extensions* to the runtime. The classes are dynamically loaded at runtime with the help of the PHP autoloader. These dynamically loaded extensions can communicate with the rest of the system (and get the control flow) through globally registered hooks in the code of the core components. These hooks enable the development of different kinds of extensions reaching from simple ones like *skins* which modify the look & feel of the UI to more advanced ones like extended *parsers* which enhance the available concepts of the MediaWiki markup language. There are already many ready-to-use extensions which makes it possible to extend the functionality of the system even for non-programming system admins.