Web Crawler Architecture Design

Group 32 - André Silva and Evandro Giovanini
Software Architecture @ IST 2024/25

Introduction

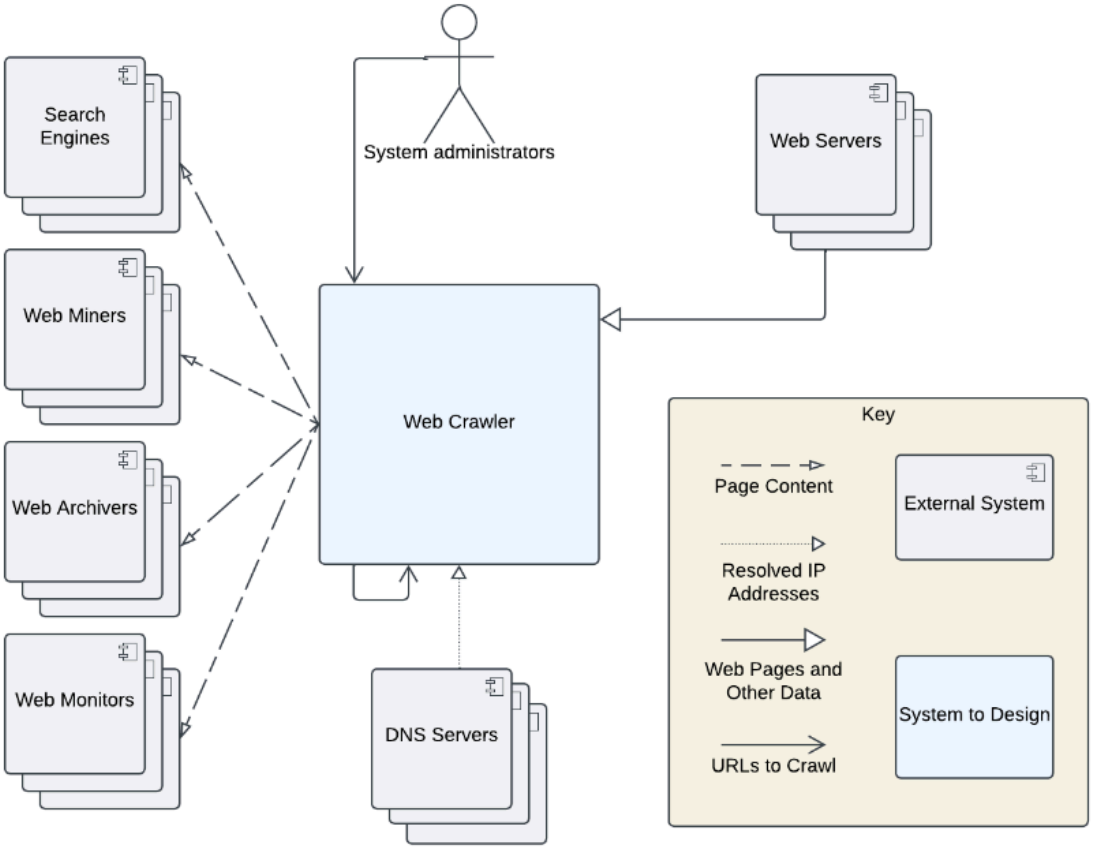
Web crawlers, also known as spiders or robots, are critical tools used to automate the discovery and retrieval of content across the internet. These systems power a wide range of applications, from search engine indexing to web monitoring, data mining, and digital archiving. The purpose of this project is to design the architecture for a scalable and efficient web crawler leveraging the Attribute-Driven Design (ADD) methodology.

The web crawler under consideration is designed to support the indexing of HTML web pages, specifically for a search engine, with the capability to store crawled data. It must handle a massive scale of operation, while maintaining key quality attributes such as modifiability, integrability (politeness and respect for other system's restrictions), and security.

This document presents the architectural design process, driven by functional requirements, quality attributes, and system constraints. Using ADD, the design is iteratively refined to address architectural drivers, incorporating scenarios for performance, availability, modifiability, and other critical qualities. Each iteration is supported by architectural views and decisions that ensure the system meets its operational goals effectively.

Through this structured approach, the resulting architecture aims to balance the complexities of large-scale web crawling with the flexibility needed to adapt to future requirements.

Context



Requirements

Functionality Requirements

FR1. The web crawler must discover new or updated content on the web

FR1.1. Given a set of URLs, the crawler must **download** the corresponding web pages/content.

FR1.1.1. The crawler must **resolve** the URL to an IP address via DNS.

FR1.1.2. The crawler must use provided seed URLs as a starting point for the crawl process.

FR1.2. The crawler must **extract links** from the downloaded web pages, following them to discover more content.

FR1.2.1. The crawler should perform Server Side Rendering of pages that use JavaScript to dynamically generate links and content.

FR1.2.2. The system must avoid targeting **repeated** URLs.

FR1.3. The system must **store** the downloaded content for up to 5 years.

FR1.3.1. The system must not store **duplicate** page content.

FR1.4. The system must **filter** found URLs and downloaded content, avoiding storing and processing advertisements, spam URLs and other irrelevant content.

FR1.5. The crawler must be able to assign **priorities** to URLs, and target URLs with higher priority first.

FR1.6. The crawler must **revisit** already seen URLs. The revisiting frequency is based on priority and update frequency.

Quality Attribute Scenarios

Performance

P1 - Performance in normal mode

Source of Stimulus	Internal - The crawler.
Stimulus	Finds 400 new URLs per second in web pages.
Environment	During normal system load.
Artifact(s)	The web crawler.
Response	The pages and content matching the new URLs are fully processed (downloaded, parsed and stored).
Response Measure	No pages are left unprocessed. Average Latency: ● 95% of URLs are processed in under 1s. ● 4% of URLs are processed in under 2s. ● 1% of URLs are processed in under 5s. Jitter: ● < 5s

P2 - Performance in peak load

Source of Stimulus	Internal - The crawler.
Stimulus	Finds 800 new URLs per second in web pages.
Environment	During peak system load.

Artifact(s)	The web crawler.
Response	The pages and content matching the new URLs are fully processed (downloaded, parsed and stored).
Response Measure	No pages are left unprocessed. Average Latency: ● 90% of URLs are processed in under 1s. ● 8% of URLs are processed in under 2s. ● 2% of URLs are processed in under 5s. Jitter: ● < 10s

P3 - Performance of Data Accesses on Unpopular Content

Source of Stimulus	The external search engine using the crawler's data.
Stimulus	Makes sporadic requests to access unpopular crawled content.
Environment	During normal system load.
Artifact(s)	The web crawler.
Response	The system responds with the requested data.
Response Measure	The average latency of the response is 500ms. The jitter may be up to 1 second. All requests for content are satisfied.

P4 - Performance of Data Accesses on Popular Content

Source of Stimulus	The external search engine using the crawler's data.
Stimulus	Makes sporadic requests to access popular crawled content.
Environment	During normal system load.
Artifact(s)	The web crawler.
Response	The system responds with the requested data.
Response Measure	The average latency of the response is 100ms.

	The jitter may be up to 300ms. All requests for content are satisfied.

Integrability

I1 - Politeness

Source of Stimulus	The web crawler.
Stimulus	Integrates with (attempts to crawl) a web server that has a `robots.txt` file.
Environment	Runtime.
Artifact(s)	The web crawler.
Response	The crawler and the website correctly exchange information. The crawler does not violate any resource access limits or allowed request rates.
Response Measure	The time to adapt to the website rules should be less than 1 second if they were not previously known. When crawling a host, the time to detect if a URL is allowed should be less than 50ms. Before knowing the website's rules, no further crawling should be made on the website's host. Crawling in other hosts is completely unaffected by the enforced limits.

I2 - Integrating a Web Monitor Client

Source of Stimulus	The system's stakeholders.
Stimulus	Want to integrate the crawler with a web monitoring software.
Environment	Development, with the crawler already operational for Search Engine Indexing.
Artifact(s)	The web crawler.

Response	The required changes are completed, tested and deployed and the crawler starts correctly exchanging information with the web monitor.
Response Measure	The changes are made, tested and deployed in less than 1 month, with no more than 5 person-months of effort.

I3 - Integrating a Web Archiver Client

Source of Stimulus	The system’s stakeholders.
Stimulus	Want to integrate the crawler with web archiving software.
Environment	Development, with the crawler already operational for Search Engine Indexing.
Artifact(s)	The web crawler.
Response	The required changes are completed, tested and deployed and the crawler starts correctly exchanging information with the web archiver.
Response Measure	The changes are made, tested and deployed in less than 1 month, with no more than 1 person-month of effort.

Security

S1 - Dealing with Spider Traps

Source of Stimulus	An external malicious website.
Stimulus	Attempts to reduce crawler availability by creating an infinite recursion of links.
Environment	The system is fully operational.
Artifact(s)	The web crawler.
Response	The attempt is detected and logged. The crawler blacklists this host and stops any tasks related to it.

Response Measure	The attempt is detected in under 5 minutes. The throughput lost should be less than 5% during the attack. After the blacklisting, the system should return to operate under normal conditions.

S2 - Malicious Content

Source of Stimulus	An external malicious website.
Stimulus	Attempts to reduce availability, access or modify system data/behaviour by hosting malicious content (e.g harmful JavaScript).
Environment	The system is fully operational.
Artifact(s)	The web crawler.
Response	The attempt is detected and logged. The crawler blacklists this host and stops any tasks related to it. Data and services are protected from any external access.
Response Measure	99,9% of attacks are detected and resisted. The few successful attacks can not access any stored data. Recovering from a successful attack must take less than 1 hour.

Modifiability

M1 - Variable Content Types

Source of Stimulus	The project manager.
Stimulus	Wants the system to be able to crawl and store a new content type (e.g PDFs).
Environment	Design time, with the crawler already running for HTML content.
Artifact(s)	The web crawler.
Response	The change is made and the system is updated, tested and deployed supporting the new content type.

Response Measure	The change is implemented and tested in less than 2 months. The financial cost of introducing the change is very low. The system suffers no downtime from the introduction of the change.

M2 - Auto-scaling the crawling process

Source of Stimulus	The web crawler.
Stimulus	An increased demand for processing capacity.
Environment	Runtime.
Artifact(s)	The web crawler.
Response	The system self-modifies to use more computational resources.
Response Measure	The new resources are available in less than 5 minutes. The system suffers no downtime, latency and throughput are unaffected during the time taken by the modification.

M3 - Scaling the content storage

Source of Stimulus	The system administrators/stakeholders.
Stimulus	Find a steady increase in demand for content storage.
Environment	Design time, with the crawler already running.
Artifact(s)	The web crawler.
Response	The new storage configuration is updated and deployed.
Response Measure	The change is made in less than 24 hours. The storage scaling occurs without any data loss. The system suffers no downtime due to the change. The cost of the actual modification is zero - only the price of actually maintaining the storage will increase.

Availability

A1 - Hardware failure

Source of Stimulus	The system's hardware.
Stimulus	Crashes unexpectedly.
Environment	During normal operations.
Artifact(s)	The web crawler.
Response	The failure is detected and logged. The system recovers and continues to operate.
Response Measure	The average time to detect the failure is < 30s. The average time from the crash until recovery is < 2m. The system is available 99,9% of the time.

A2 - Unresponsive external web server

Source of Stimulus	An external web server.
Stimulus	Fails to respond to a request in time.
Environment	Normal operations.
Artifact(s)	The web crawler.
Response	The fault is logged. The host is temporarily excluded from future downloads. The system continues to operate normally.
Response Measure	The system detects the fault in under 10 seconds. The system recovers in under 30 seconds. The system maintains its availability despite the external fault.

A3 - Handling Malformed URLs

Source of Stimulus	A downloaded web page.
Stimulus	Contains an invalid or malformed URL.

Environment	Normal operations.
Artifact(s)	The web crawler.
Response	The invalid URL is detected, logged and discarded. The system continues to operate normally.
Response Measure	Identifying a malformed URL should take less than 20ms. The system must still be fully available despite the fault.

Attribute-Driven Design Rounds

Round 1 - Fully Functional System

Iteration 1 - Basic Page Downloading and Storage

In this iteration, the goal is to enable the system to initialize with a list of seed URLs and download the matching pages, using DNS to resolve URLs into IP addresses.
After downloading these pages, the system will be able to store them persistently, avoiding the storage of duplicate content.

Architectural Drivers:

FR1.1, FR1.3.

Elements of the system to refine:

The whole system;

Design concepts:

The *Split module* tactic is applied to separate the crawler functionalities into their own modules. We also utilise the Client-Server pattern to represent the interaction between the crawler system (the client) and the external websites (servers), using DNS as the discovery mechanism.

Architectural Elements, Responsibilities and Interfaces:

These two requirements entail a list of responsibilities and each of them will be satisfied by an appropriate module.

Starting with the SeedURL initialization, we create the **SeedLoader** module. This module will be called to obtain the URLs in the seed.

Next, we instantiate the **URLScheduler module**. This module will expose an interface that can be called to obtain the next URL to be crawled. For this, it will use the aforementioned SeedLoader.

For the actual downloads, we create the **Fetch module**. This module has three key responsibilities:

1. First, it needs to query DNS servers and obtain IP addresses from the URLs. This is performed by the **DNSResolver sub-module**.
2. Then, it sends HTTP requests to the resolved IP addresses and receives HTML and other content types as responses. This responsibility is handled by the **NetworkClient sub-module**.
3. Finally, it needs to handle these responses and call the part of the system responsible for storing the content (discussed next). This is handled in the **ResponseHandler sub-module**.

We also instantiate the **FetchCoordinator** as a **sub-module** of Fetch. This module handles the coordination of the downloading, using the other three submodules to perform the operations. The parent, Fetch, exposes only FetchCoordinator's interface to other modules.

Then, for the storage of the content, a **WebCrawlerContentStorage component** is created. This is a persistent repository for web page content that can be accessed by the crawler (to write downloaded content) or by the external client systems (to read and interpret the stored content). This content is stored as an entity with relevant information (the content itself, source url, hashed content…). A good choice for this type of data is a NoSQL database, such as MongoDB.
Additionally, we need some code to perform operations on this repository. We create the **ContentStore module**, which will handle all the CRUD operations (create a new content entity, read content…). We then create the **Content module**, with a **ContentSeen sub-module**, which uses the ContentStore to verify if a certain piece of content has already been stored by the crawler. To fit the requirement of storing content up to 5 years (while keeping costs low), we add the **ExpiredContentHandler sub-module**. This module is called to remove content older than 5 years from the repository.
The content module exposes all of its submodules' interfaces.

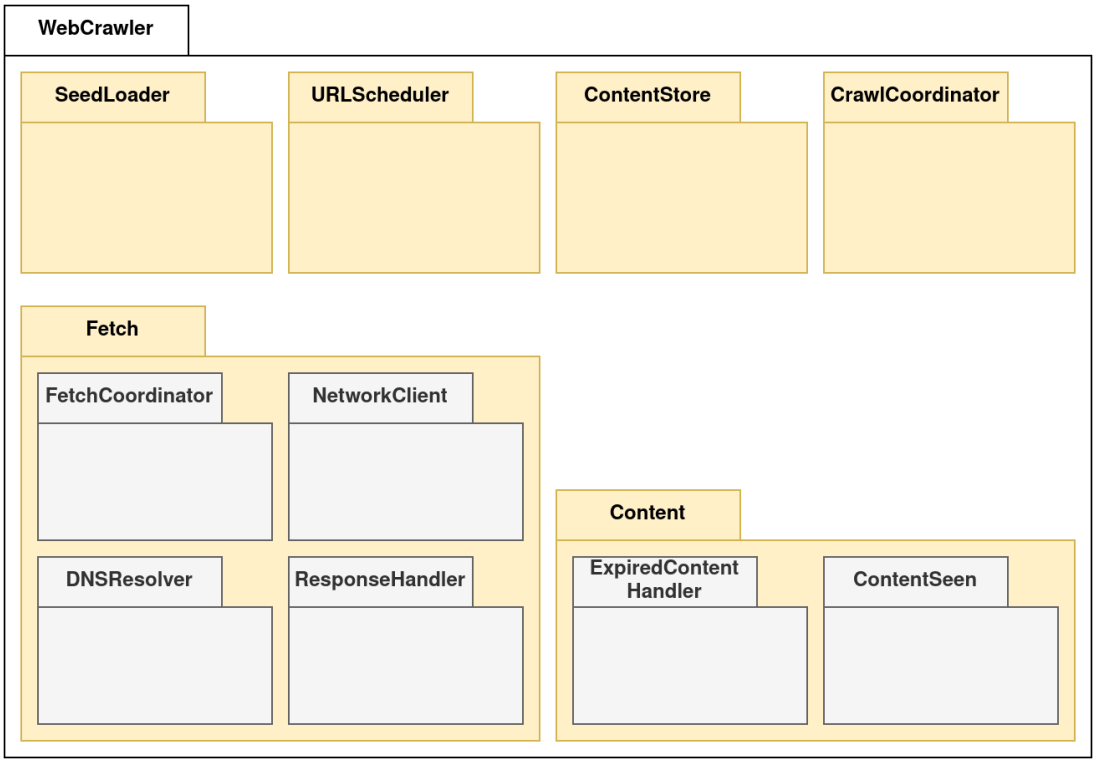
Finally, we need an implementation unit to manage the overall process. We call it the **CrawlCoordinator module**. This module uses the others and performs the download and store operation sequentially - it reads a URL from the URLScheduler and calls the

FetchCoordinator module, which will perform the necessary work to obtain and store the page. Periodically, this module can also call the ExpiredContentHandler to clean the storage of old content.

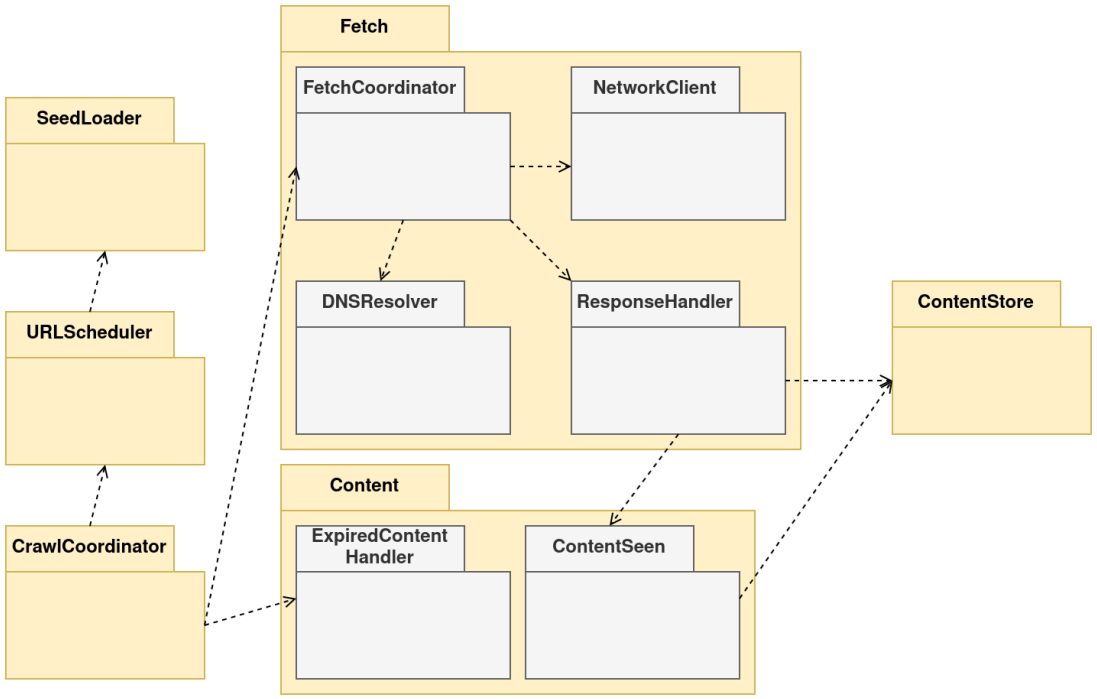
All of these modules are packaged in the **WebCrawler module** and will run in a single component - the **WebCrawler component**.

View Sketches:

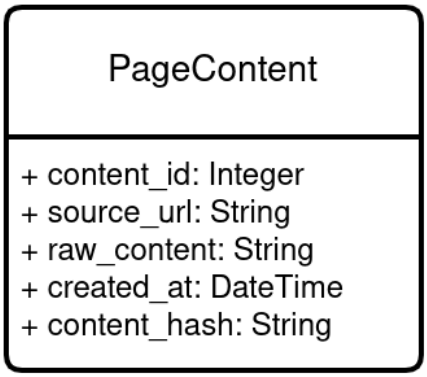
Web crawler Module Decomposition View:



Web Crawler Module Uses View:

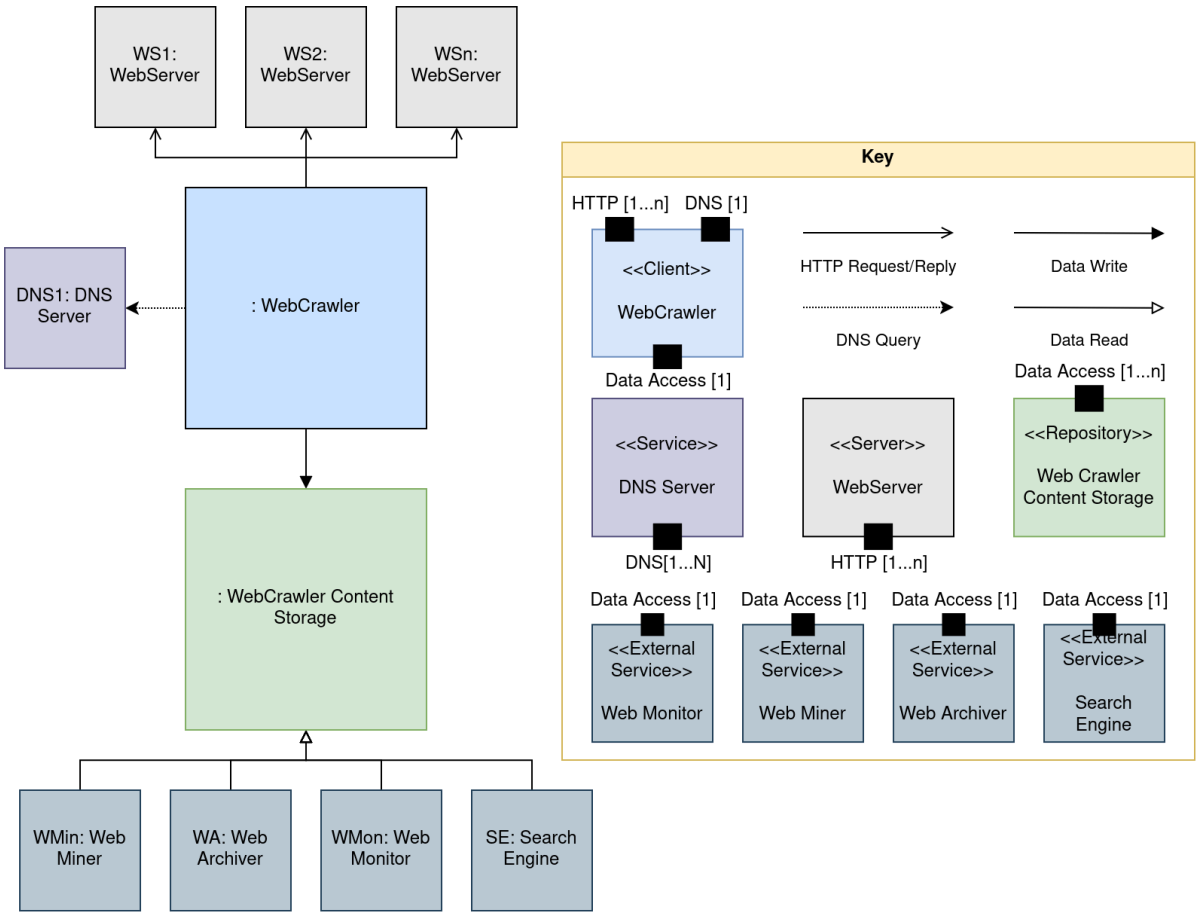


Content Storage Data Model View:



System C&C View:

This is based on the Context Diagram, with more explicit interactions between the components. It follows the client-server style for the interactions between the web crawler and the external web servers, and the shared-data style in the interactions between the crawler, the web crawler storage and the client applications.



Iteration 2 - Content Parsing, Link Extraction and URL/Content Filtering

In this iteration, we wish to design the functionality for discovering new pages and content from the downloaded pages by extracting links from them and using these as the next targets of the crawl.
By the end of the iteration, the crawler will also be able to filter the found URLs and downloaded content, avoiding storing and processing advertisements, spam and other irrelevant content.

Architectural Drivers:

FR1.2, FR1.4

Elements of the system to refine:

The Content Module; The URLScheduler Module; The Web Crawler Module;
The whole system as a component;

Design concepts:

The *Split module* tactic is applied to separate the new functionalities into their own modules. The *Redistribute Responsibilities* and *Encapsulate* tactics are also applied, to better encapsulate the URL scheduling, filtering and discovery.

Architectural Elements, Responsibilities and Interfaces:

The functionality of processing a HTML document or other downloaded content can be split into a few smaller responsibilities.
Firstly, we should determine if the content is valid and correctly formatted - this is accomplished by a new **HTMLValidator** module. Then, we check if we have seen it before (already accomplished in the last round). If it's a new piece of HTML content, we first need to execute and obtain the rendered output of any JavaScript in the page (new **JSRenderer** module). Next, we validate the content using the new **ContentFilter** module, which discards pieces of content that are an advertisement, spam or irrelevant. To abstract the implementation of handling the new content, the **NewContentHandler** module is created. Its interface allows the Fetch module to call it and not be concerned about the actual processing of the content after this call.

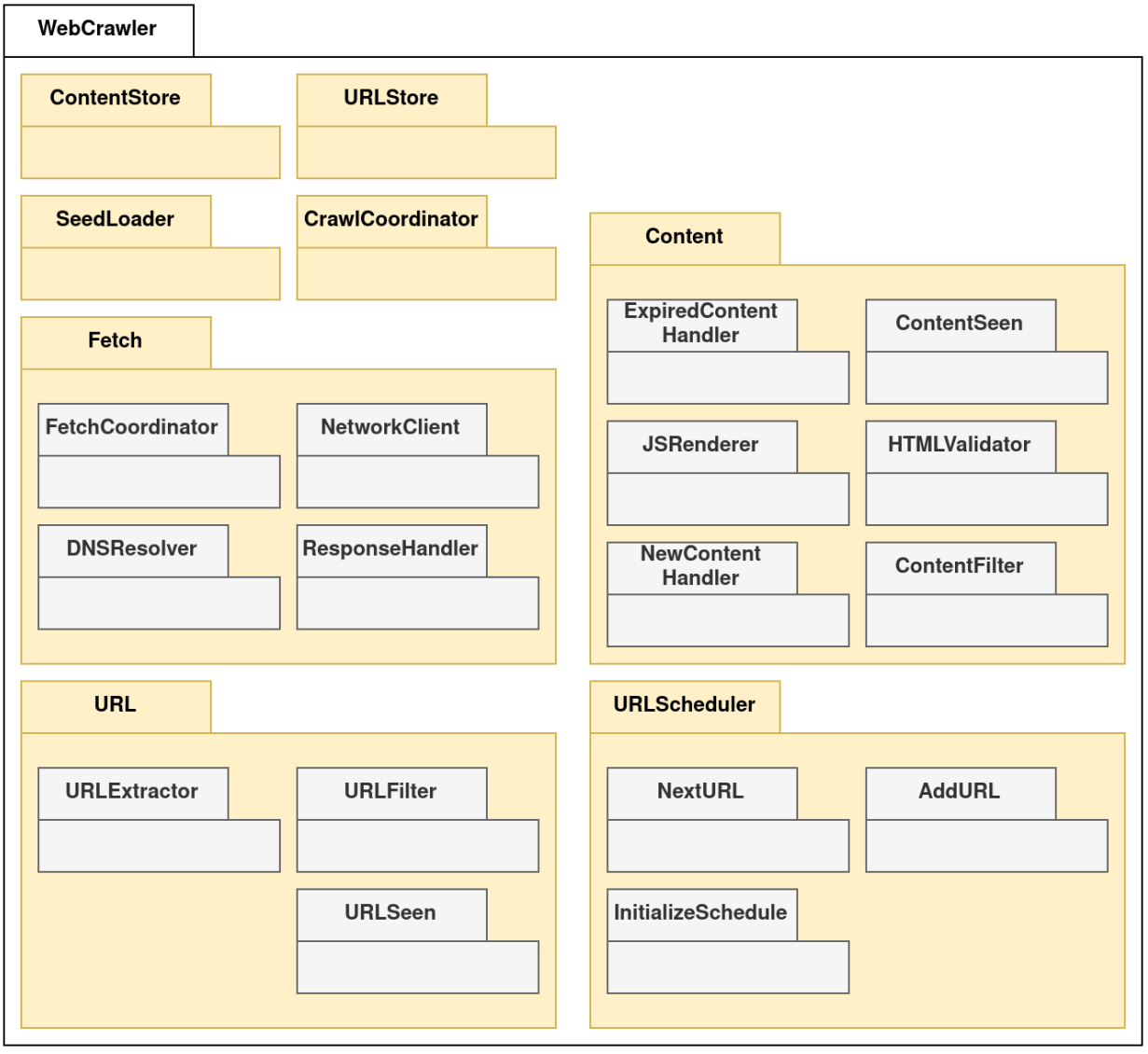
The web crawler system also has the functionality of extracting links from the HTML content. This process is accomplished by the new **URLExtractor** module, that, upon finding a link, calls the also new **URLFilter** module, which discards the link if it corresponds to irrelevant content (i.e, spam, advertisements…).

If the found link is a duplicate, it is also filtered out. For this, we require a new repository component (the **WebCrawlerURLStorage component**, which can also be a NoSQL database), and modules for interacting with it (**URLStore** and **URLSeen**). All of these modules, except the **URLStore**, are placed as a submodule of **URL**, a new module for handling all the responsibilities related to URLs.

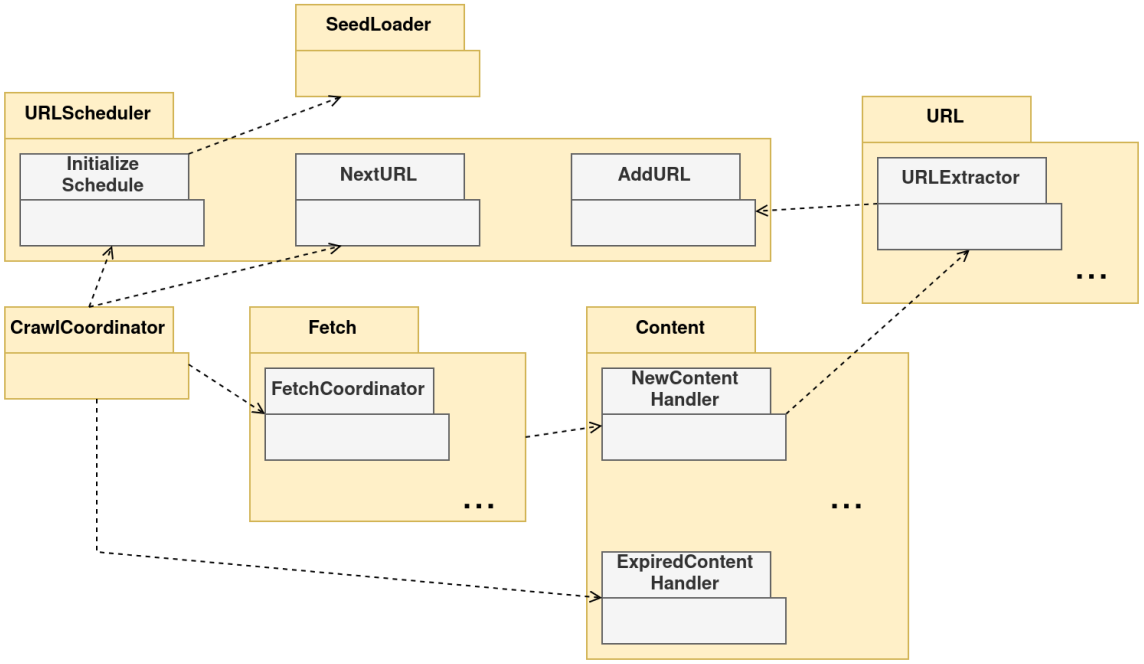
Finally, we need to modify the scheduling logic to accommodate the new URLs found in the crawled pages. The URLScheduler module is split, and three new modules are created: **InitializeScheduler**, for getting the seed URLs into the scheduler, **AddURL** for adding a newly found URL to the scheduler, and **NextURL**, for getting the next URL in the scheduler.

View Sketches:

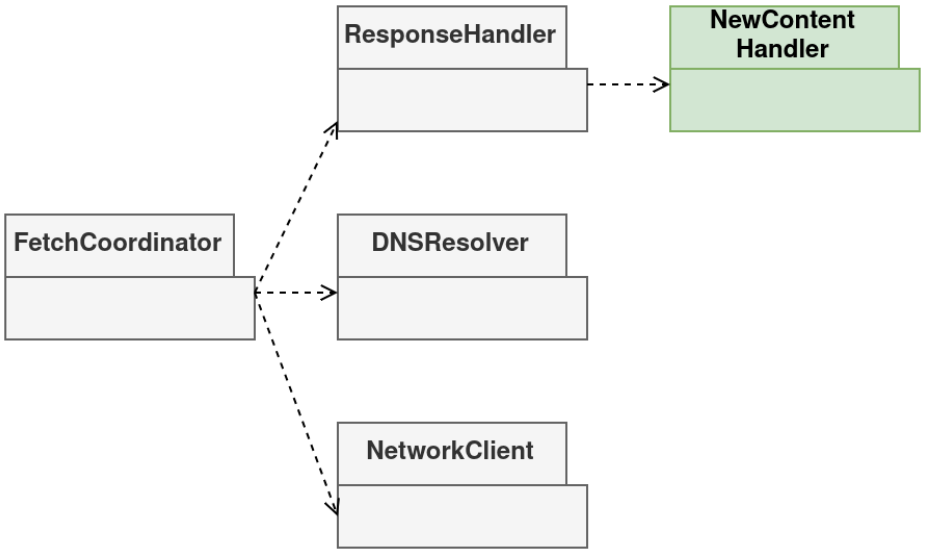
Web crawler module decomposition view:



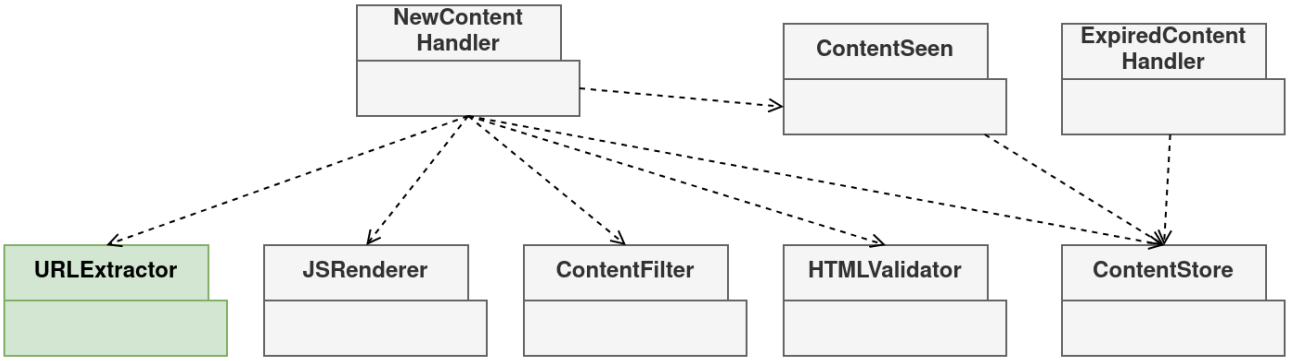
Web crawler module uses view:



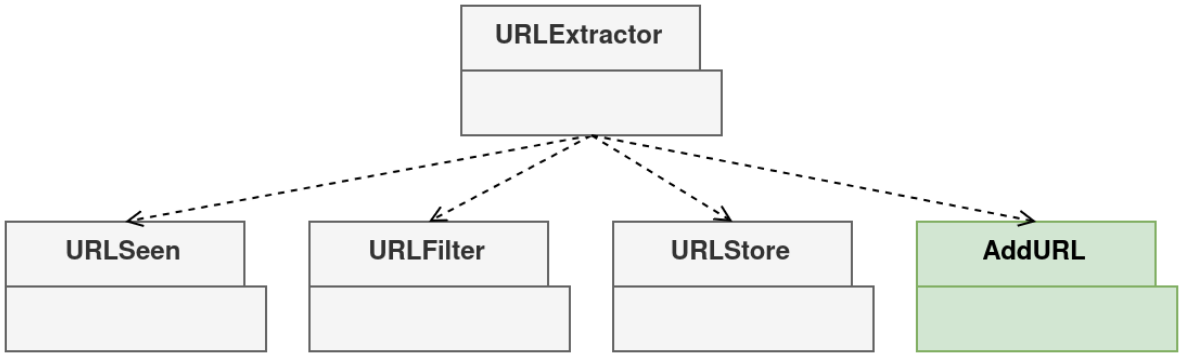
Fetch module uses view:



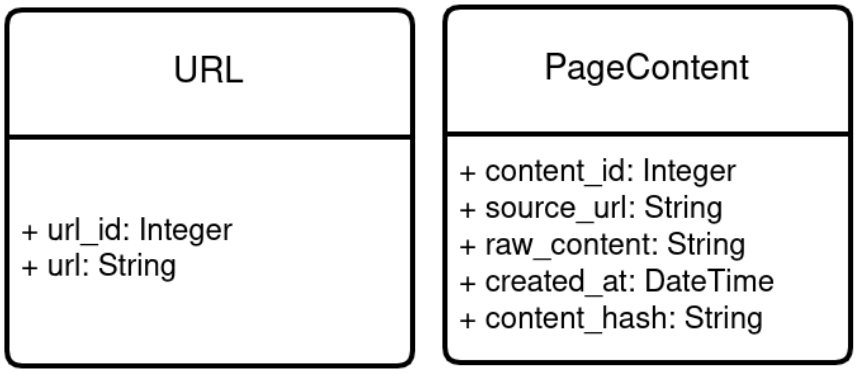
Content module uses view:



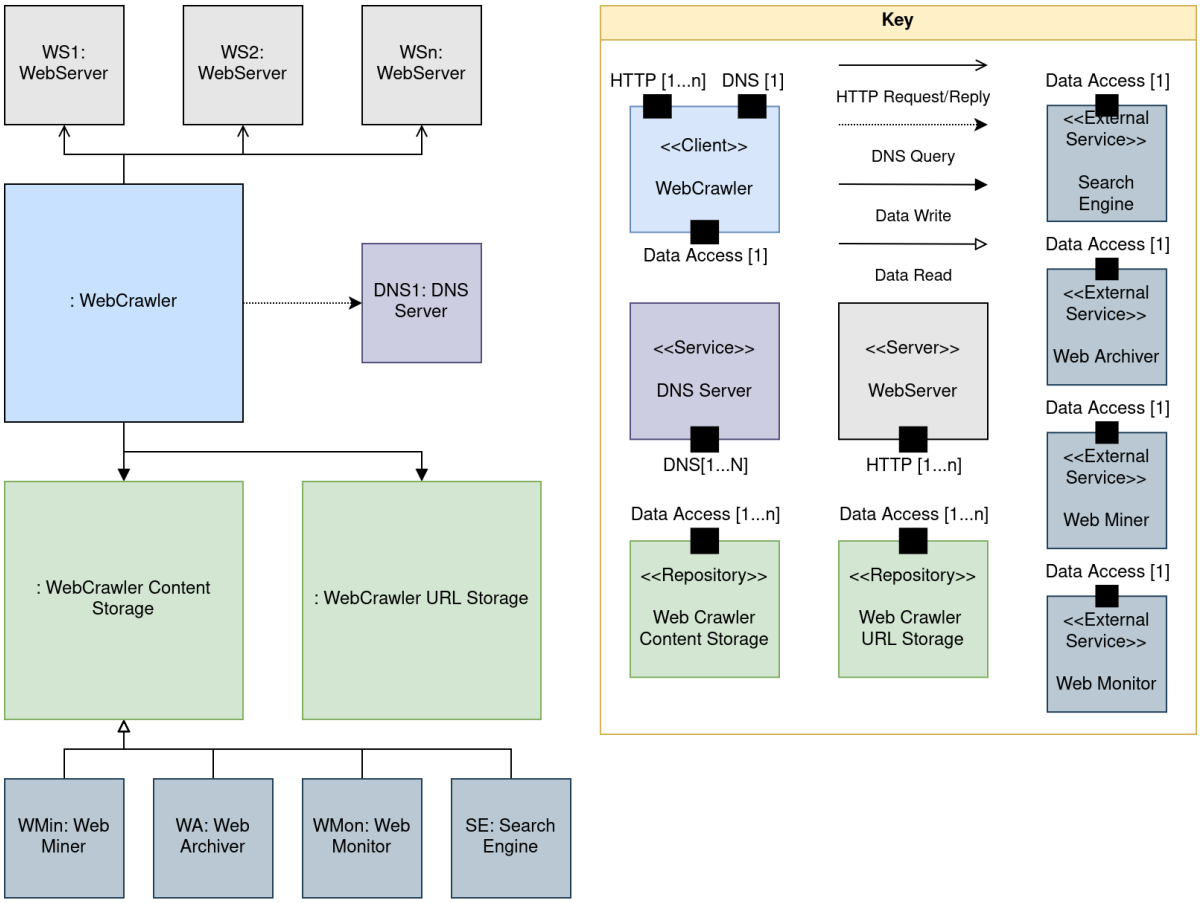
URL Module uses view:



System data model view:



Top level C&C view:



Iteration 3 - Prioritization

After this iteration, the system will be able to assign priorities to the URLs it finds, and crawl the pages assigned to the higher priority links before the others.

Architectural Drivers:

FR1.5

Elements of the system to refine:

The URLScheduler module;

Design concepts:

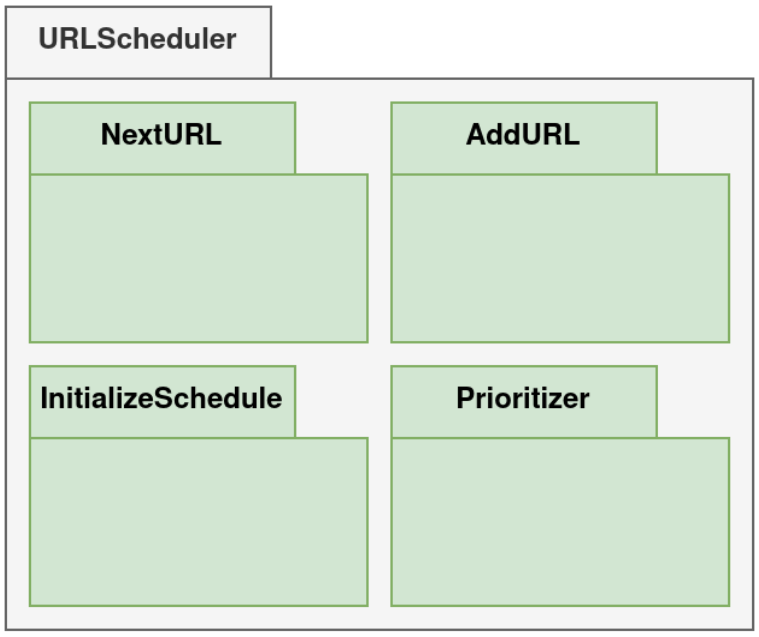
The *Split module* tactic is applied to separate the new functionality into its own module.

Architectural Elements, Responsibilities and Interfaces:

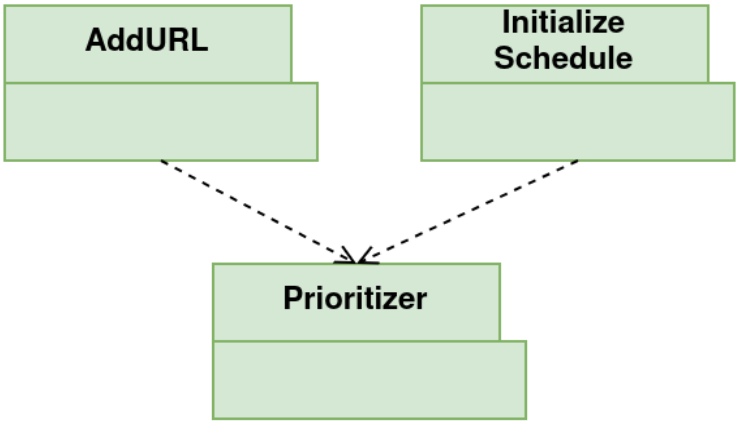
For the prioritization, it would make sense that, when a URL is added to the future download collection, its priority is computed, and it would be inserted in the correct order for selection. For the responsibility of computing the priority we instantiate the **Prioritizer Module.**

View Sketches:

URLScheduler module decomposition view:



URLScheduler module uses view:



Iteration 4 - Periodic Re-crawling

After this iteration, the system will be able to periodically revisit previously crawled web pages. The revisiting frequency should be based on their update frequency and priority (refer to last iteration).

Architectural Drivers:

FR1.6

Elements of the system to refine:

The URLScheduler module, The URL repository entity;

Design concepts:

The *Split module* tactic is applied to separate the new functionality into its own module.

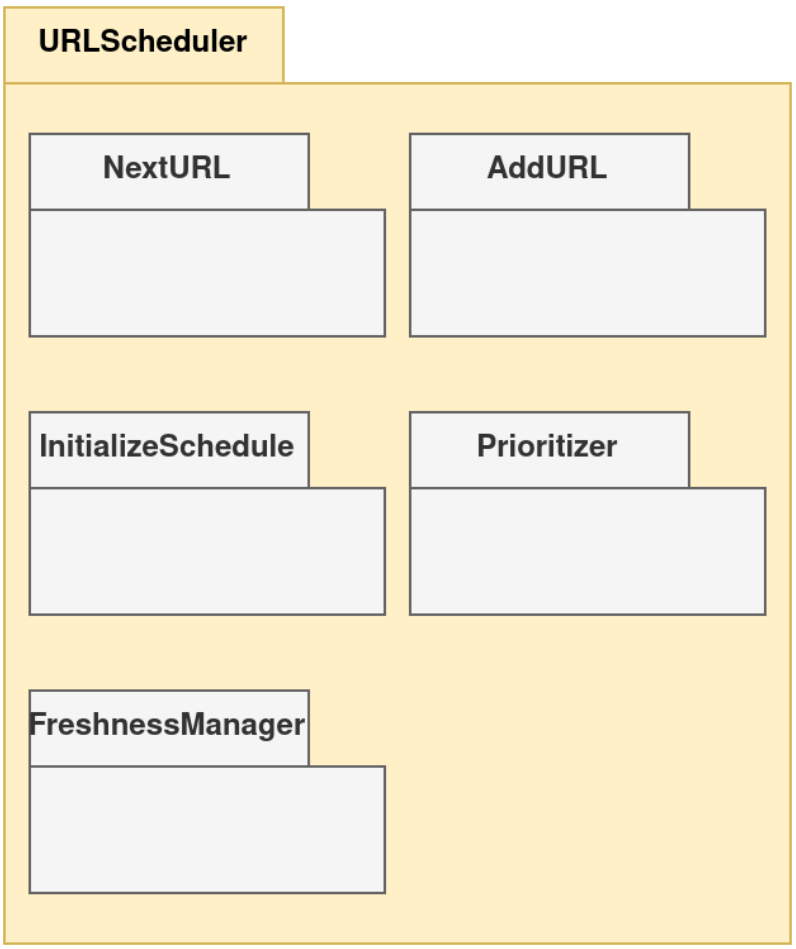
Architectural Elements, Responsibilities and Interfaces:

In this case, we are modifying the selection of the next URL to be crawled to include possible re-crawls. These possible re-crawls are calculated based on the URL's priority and update frequency. For this to be possible, the system must be changed so that the URL priority is stored with the URL. Therefore, the previously instantiated prioritizer module needs to use the URLStore.
Additionally, we must modify the entity storing a URL to include the priority, the number of updates it has seen and the time it was first seen.

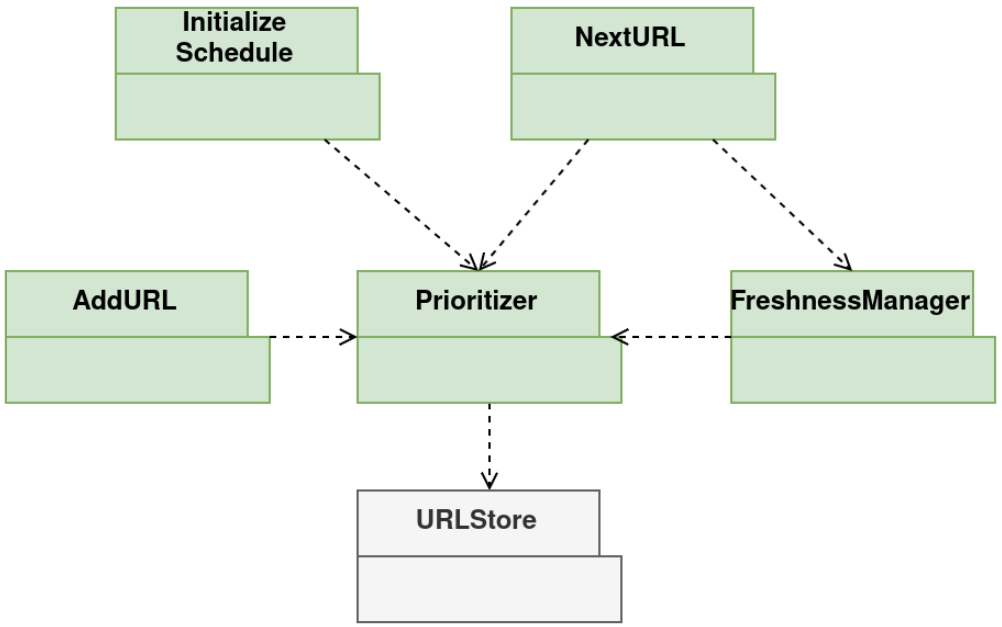
With these modifications in place, we can instantiate the **FreshnessManager module**. It is contained within the URLScheduler and uses the URLStore to obtain the priorities and update frequency of URLs. With this information, it determines whether there should be a re-crawl. It is called by the NextURL module, to compute the next target of the crawler.

View Sketches:

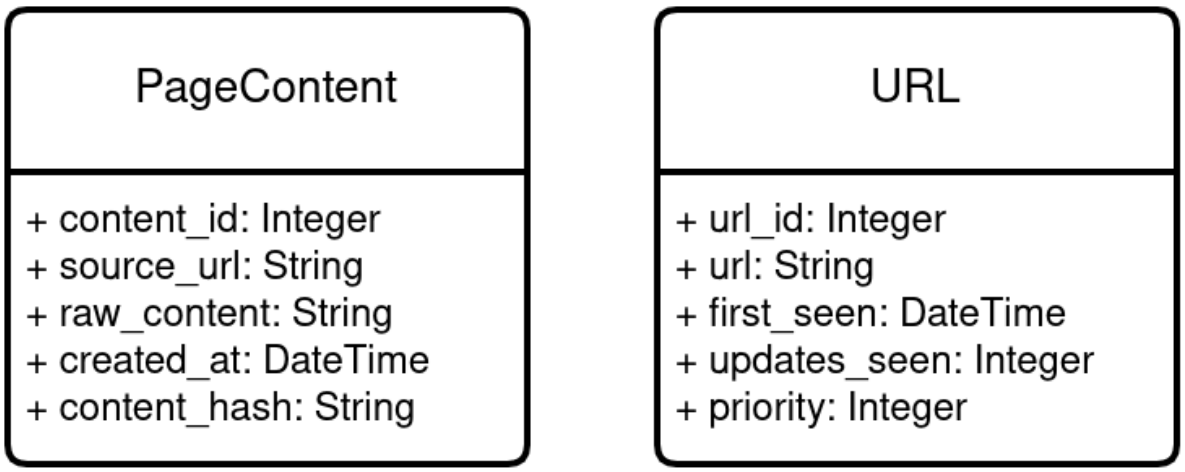
URLScheduler Module Decomposition View:



URLScheduler Module Uses View:



System Data Model View:



Round 2 - Performance in normal and peak load

With the previous round completed, the system accomplishes its basic functionality - crawling web pages, extracting and following links, storing content, prioritizing URLs, and maintaining freshness. However, the current sequential architecture cannot handle the required throughput and latency. In this round, we address performance the following performance bottlenecks, aiming to meet our performance quality attribute scenarios:

PB1. Sequential Processing Bottleneck

- Current design processes URLs sequentially
- The system remains idle during network I/O (downloading), which takes up most of the page processing time.
- Current approach cannot achieve required throughput

PB2. DNS Resolution Bottleneck

- Synchronous DNS lookups block crawler threads (10-200 ms per request)
- DNS requests for the same domains are redundantly performed
- DNS resolution time significantly impacts download performance

PB3. URLs to Download Storage Bottleneck

- Must handle hundreds of millions of URLs in the queue
- Memory-only storage is insufficient and risky
- Disk-only storage creates I/O bottleneck
- Need for efficient prioritization and access patterns

PB4. Geographical Bottleneck

- The current design's latency is heavily influenced by the distance to the crawler's target web server. If we are crawling pages hosted across the world, the speed at which we can do this is very slow.

Iteration 1 - Concurrent crawling

In this iteration, we wish for the system to increase the page processing latency and throughput by crawling web pages in a concurrent manner.

Architectural Drivers:

P1, P2, PB1.

Elements of the system to refine:

The Web Crawler module; The URLScheduler module; The Web Crawler component.

Design concepts:

To achieve the best crawling performance, we need to parallelize most of the tasks the crawler performs. We utilize the *Introduce Concurrency* performance tactic to perform this parallelization.
Additionally, we apply the *Maintain multiple copies of computations* performance tactic, to increase throughput and benefit from the parallelization of the crawler tasks.
Finally, we apply the *Use an intermediary* tactic to reduce coupling between the new components and enable easier communication between them.
For the new and altered responsibilities associated with this iteration, we apply the *Split Module* and *Redistribute Responsibilities* tactics.

Architectural Elements, Responsibilities and Interfaces:

To increase the crawling latency and throughput, we need to run multiple coordinated instances of our system's sub-modules.
When describing the crawling process, we see that the responsibilities are quite independent from each other. Therefore, we can *introduce concurrency* to the process of crawling.

One component - the **FetchManager thread** - will only be in charge of downloading pages from URLs. Another thread will only do validation, checks and storage of raw downloaded content. We can name it **ContentManager.** A different thread - the **URLManager** component - will extract URLs from valid content, proceeding to validate, filter and register these as the next targets of the FetchManager component.

We now need to solve the communication and coordination problems associated with introducing concurrency, since synchronous method calls no longer suffice. For example, the downloader thread can no longer just call the Content module and wait for a response, as that would defeat the purpose of splitting the tasks. Instead, the threads now need to

communicate asynchronously. The FetchManager will, at the end of the download, place the content in a queue (*producer*), which will then be processed and removed from the queue by the ContentManager thread (*consumer*). A good way to manage this type of asynchronous communication is with a popular open-source piece of software - Apache Kafka. Kafka runs as an external service and allows us to create **Topics**, which are fault-tolerant and consistent FIFO queues. These topics are contained in a **Cluster** component. We instantiate a Kafka Topic per communication path we require so far:

1. **RawContentQueue** - The FetchManager thread writes fresh downloaded content to this queue. The ContentManager thread consumes the raw content and then processes it.
2. **ParsedContentQueue** - The ContentManager thread places the filtered and valid pieces of content it sees into this queue. Then, the URLManager thread consumes from it.

We are missing a key component - the one that will manage the crawler's scheduling. We introduce the **URLFrontier component**, running the URLScheduler module as a simple grouping of threads. This component stores the URLs extracted by the URLManager thread in another Kafka Topic. This means that FetchManager now has somewhere to obtain valid URLs from, and the basic crawling cycle is complete again.

This URLFrontier's current architecture has a problem: prioritization can't be managed in a single Kafka Topic. We need a more robust design for handling this responsibility.
For simplicity, let's assume the URLs can have a priority between 1 and 10. We can then create a Topic per possible priority value (i.e, we would create 10 queues). The URLs would be inserted in the correct queue according to their priority by a new sub-module of URLScheduler, the **PriorityRouter module**, that is running in a thread component with the same name. Then, a thread called the **PrioritySelector**, which, again, runs a new module with the same name, consumes a random URL from the queues, with a bias towards the higher priorities. It then writes it to a new **DownloadQueue**, from which the FetchManager consumes URLs from. The URLManager does not need to be aware of these complexities. All it does is write the found URLs to a **FoundQueue**, another Topic. Then, the URLs are consumed by the PriorityRouter, given a priority and sent to the proper queue.

Finally, to address content freshness, the previously instantiated **FreshnessManager** module can now also run as a separate thread (we give it the same name), which will periodically find old URLs to re-crawl and add them to the correct queue. To address cleaning up bits of content older than 5 years, we add the **ExpirationManager** component, a thread, running the code in the ExpiredContentHandler module.

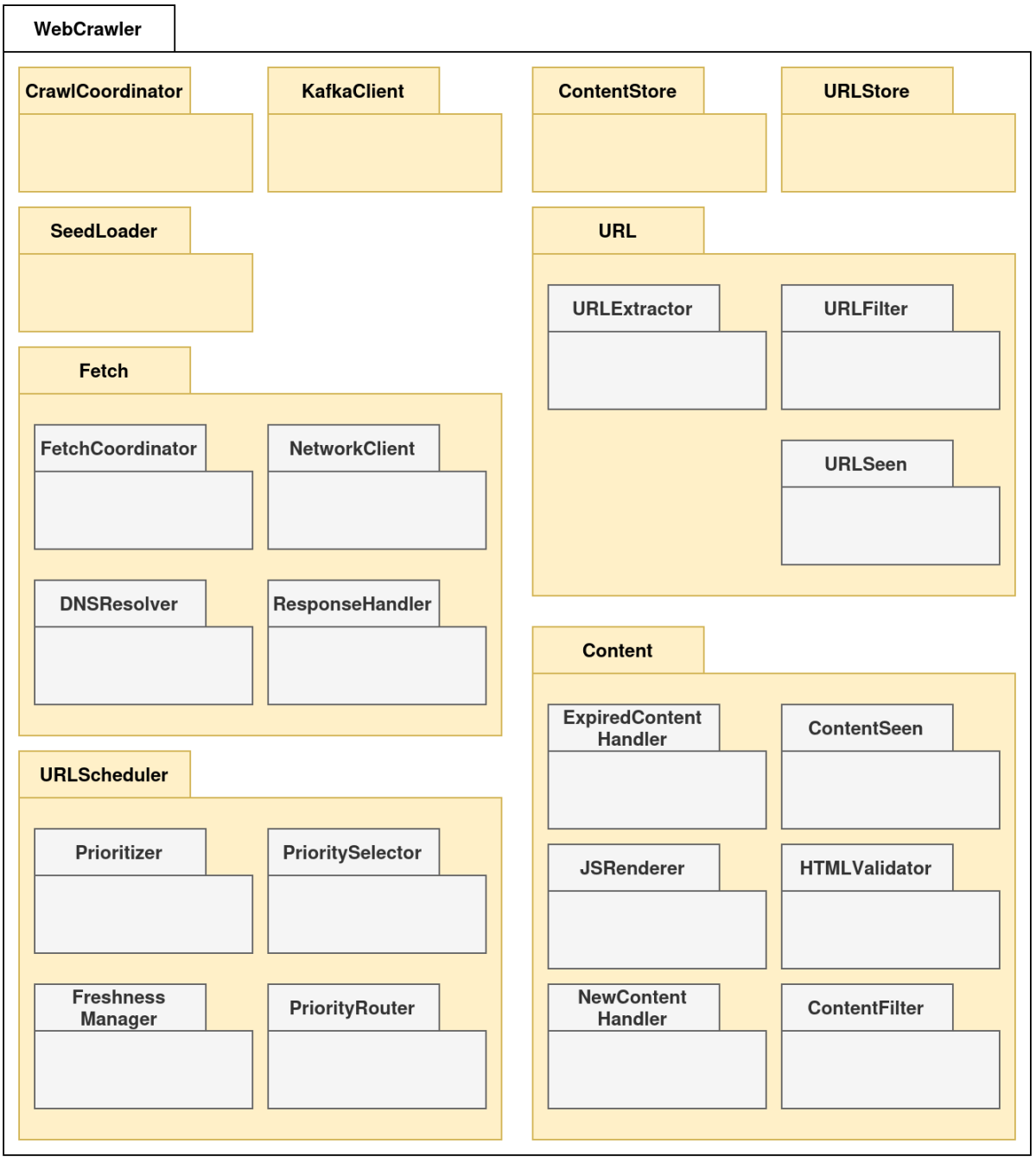
This communication setup will allow us to run multiple copies of each thread, which results in a much higher throughput. If we assume a single FetchManager thread can download an average of 20 pages per second, we could just create 20 of them and, hopefully, achieve the required download throughput. The problem with this is - increasing the number of threads in the same machine doesn't exactly mean we obtain a linear growth in throughput. There are memory, CPU and network limits that a single server can't overcome. This is a new **bottleneck** that will be addressed in a subsequent iteration. For now, we opt for a sensible default - 6 threads for downloading. With the slight throughput penalty, we can aim to achieve around 100 pages per second. Of course, this would have to be prototyped or tested in a live environment in order to obtain actual measures and adjust from there.

The remaining crawler tasks are much less time consuming than the downloads, so we don't require as many replicas to achieve the performance requirements. If we want to have 6 FetchManager threads, to obtain 100 pages per second, we most likely only need 2 ContentManagers (more than 1 because of the repository interactions), 1 LinkExtractor, 1 PriorityRouter, 1 PrioritySelector and 1 FreshnessManager.

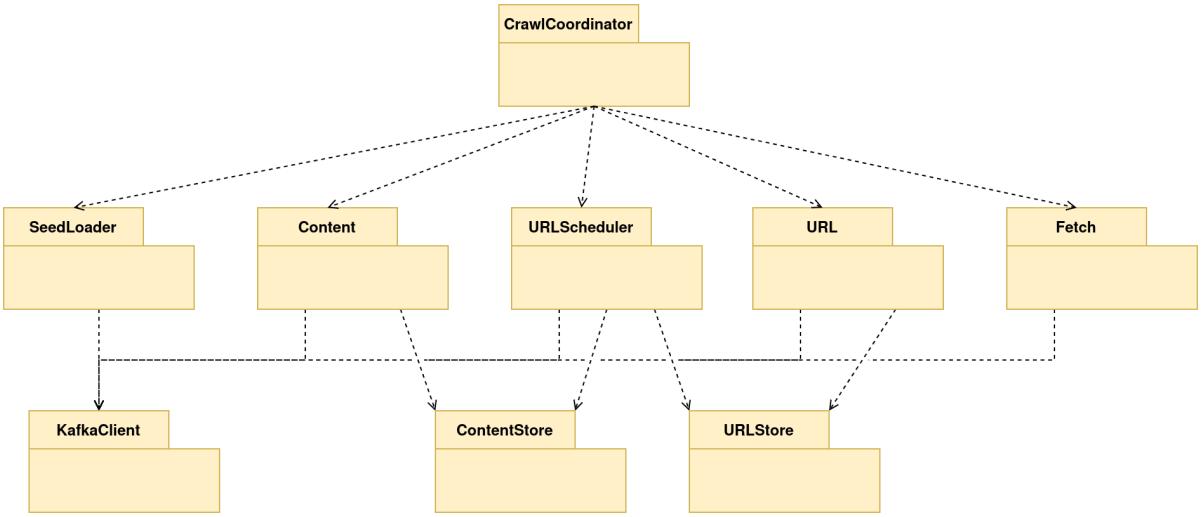
Regarding the code, besides the new modules (PrioritySelector, PriorityRouter) that have already been mentioned, we need to refactor some of the current structure. Firstly, we need a module for interacting with the Kafka Topics, let's call it KafkaClient. This is a top level module that can be used by all the other modules. The process starts at the CrawlCoordinator. It starts up the threads and loads the SeedURLs. For this, the SeedLoader also needs the KafkaClient - it inserts the seed URLs into the FoundQueue. After initializing, the threads now "run forever".
Within the URLScheduler module, we no longer require the AddURL and NextURL modules. These responsibilities are now assigned to the new PriorityRouter (adds a URL to the queue) and the PrioritySelector (gets a URL from the queue).

View Sketches:

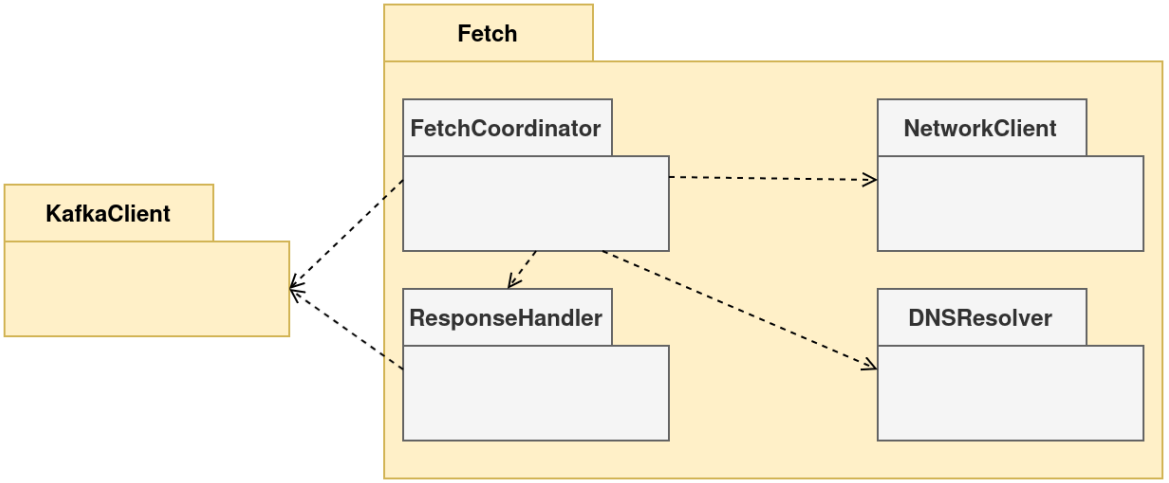
Top Level Module Decomposition View



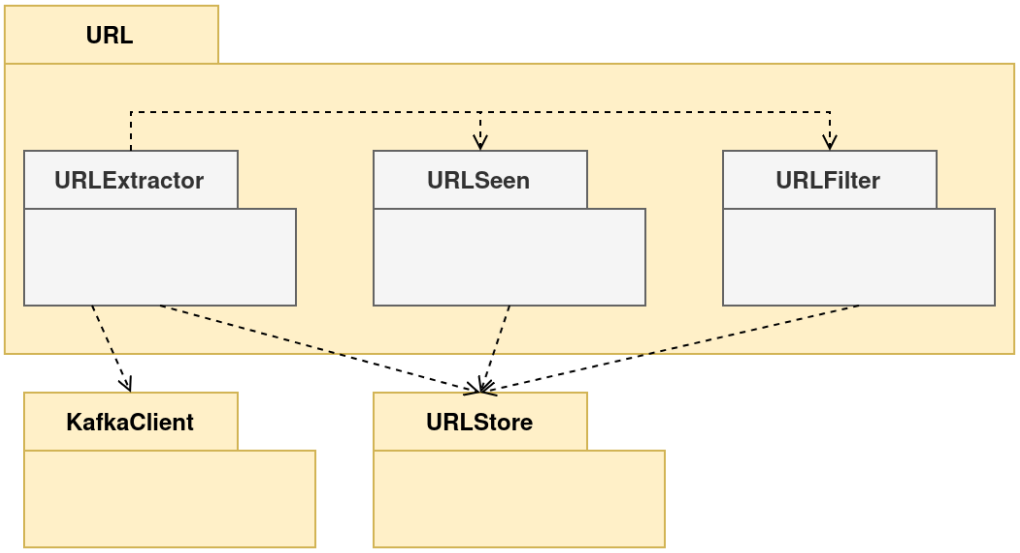
Top Level Module Uses View



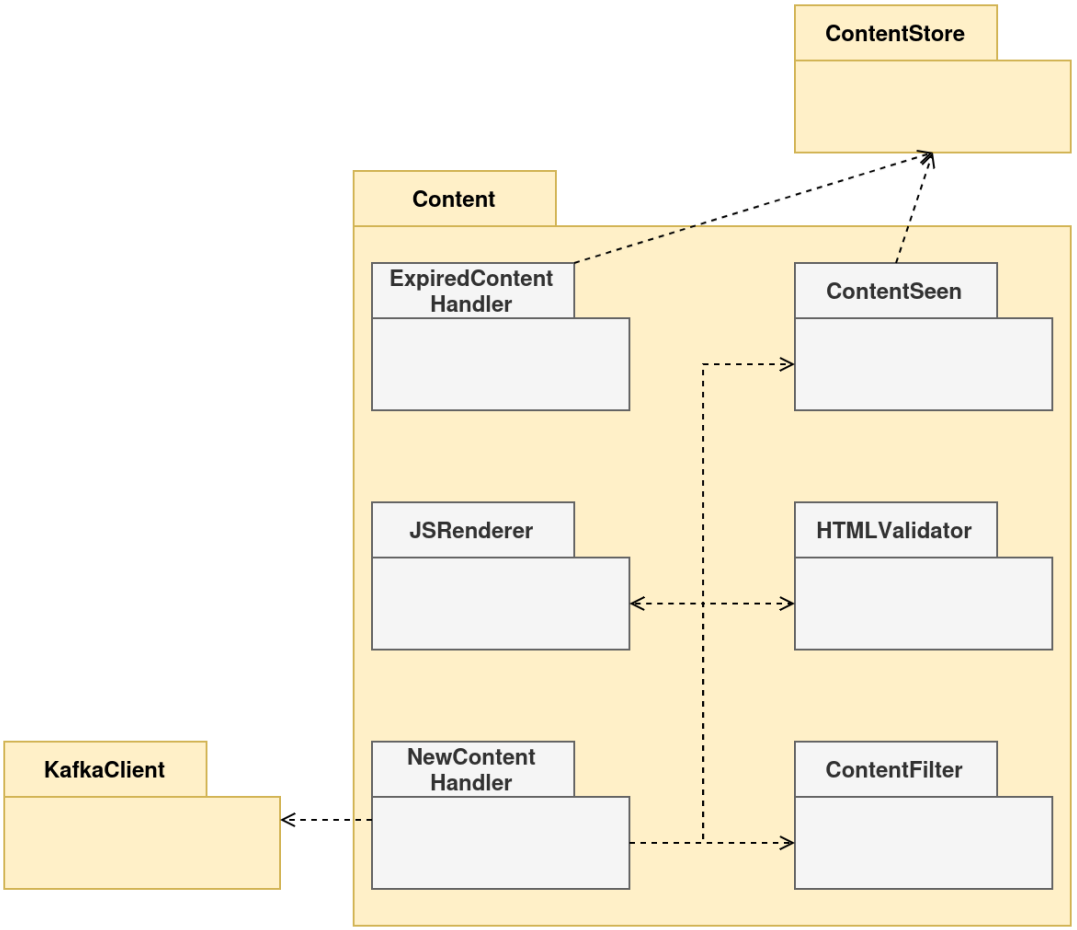
Fetch Module Uses View



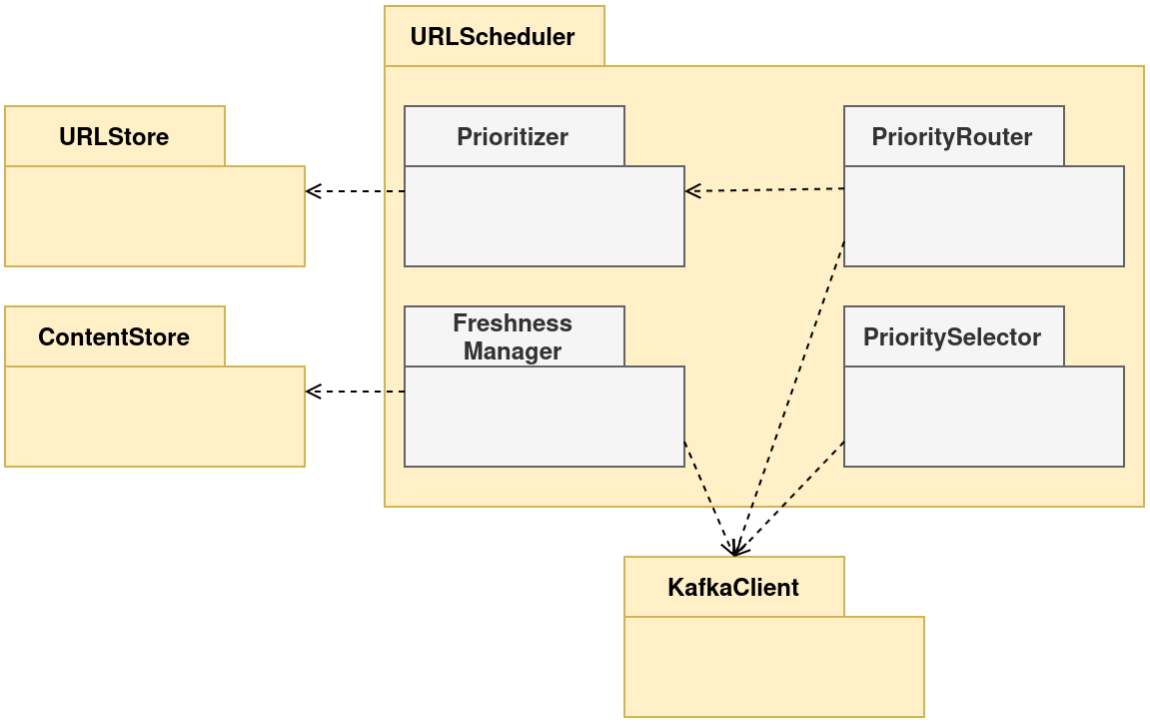
URL Module Uses View



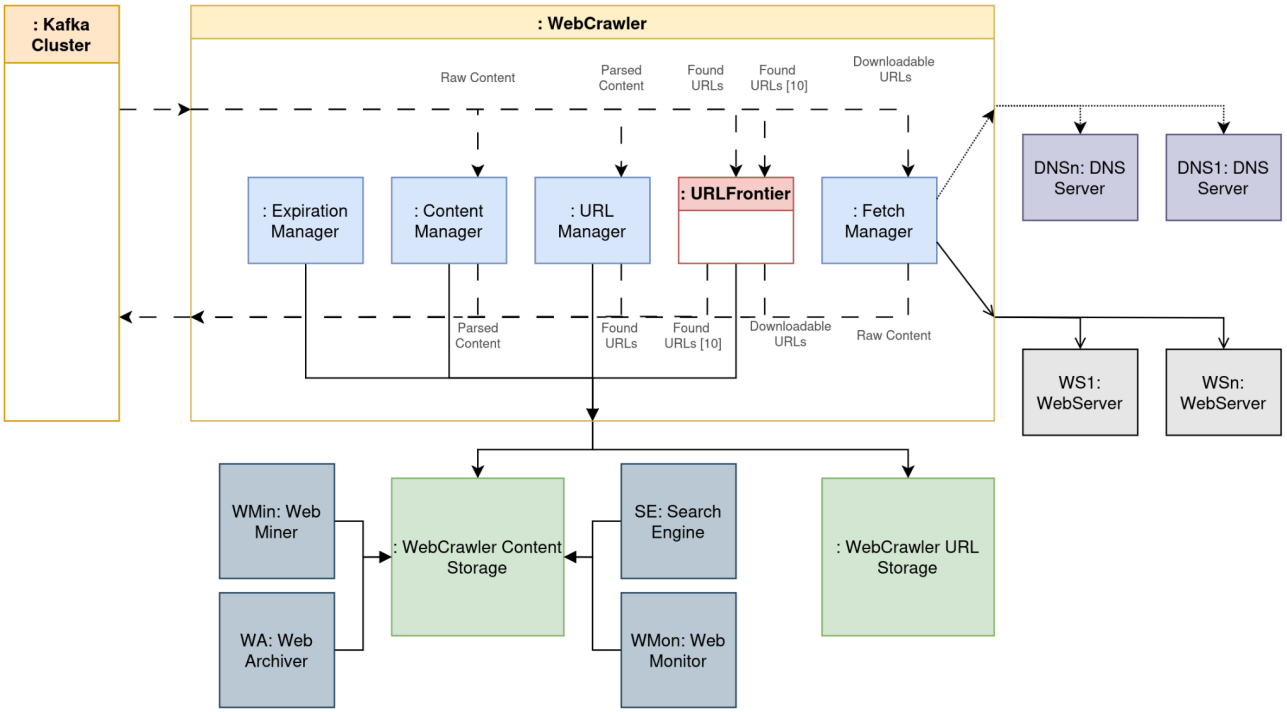
Content Module Uses View

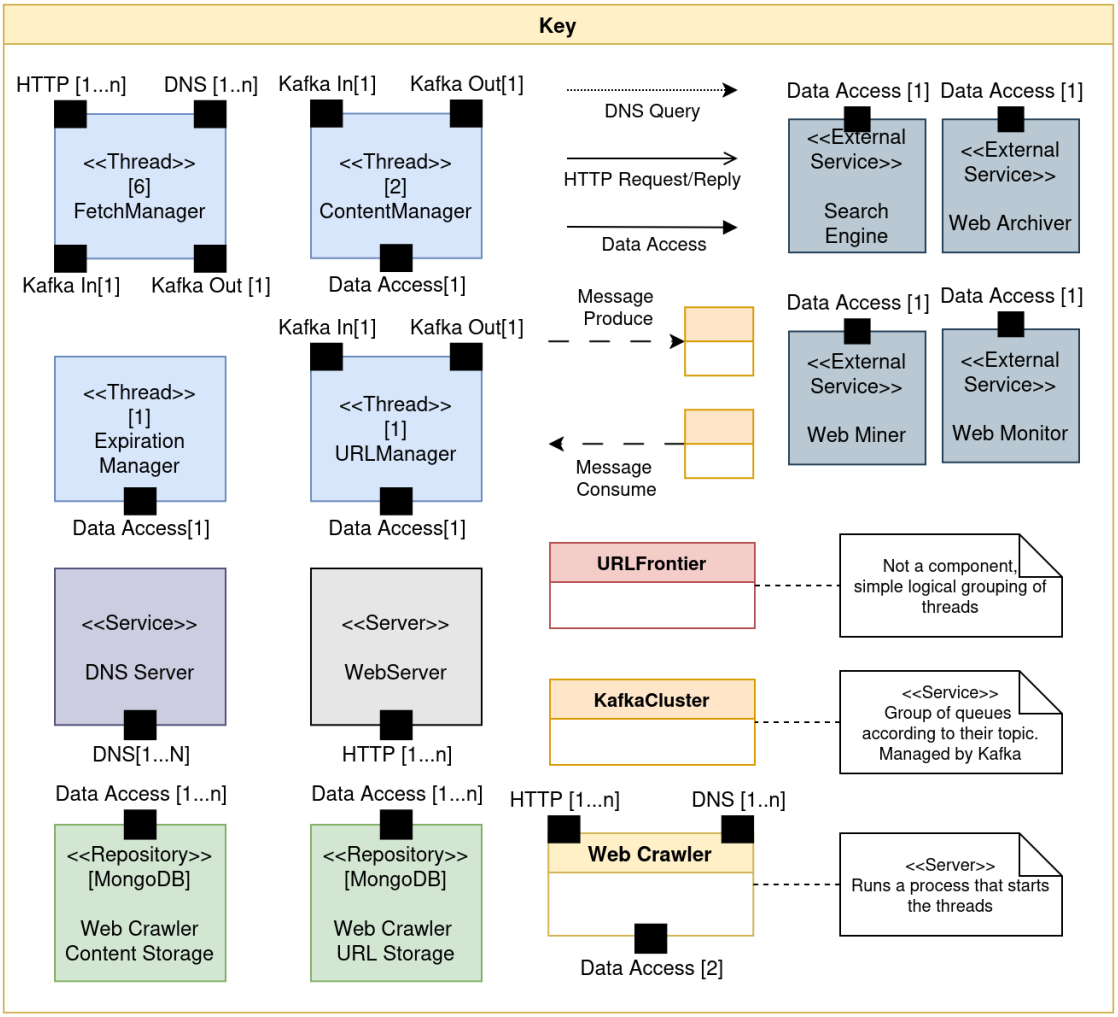


URLScheduler Module Uses View

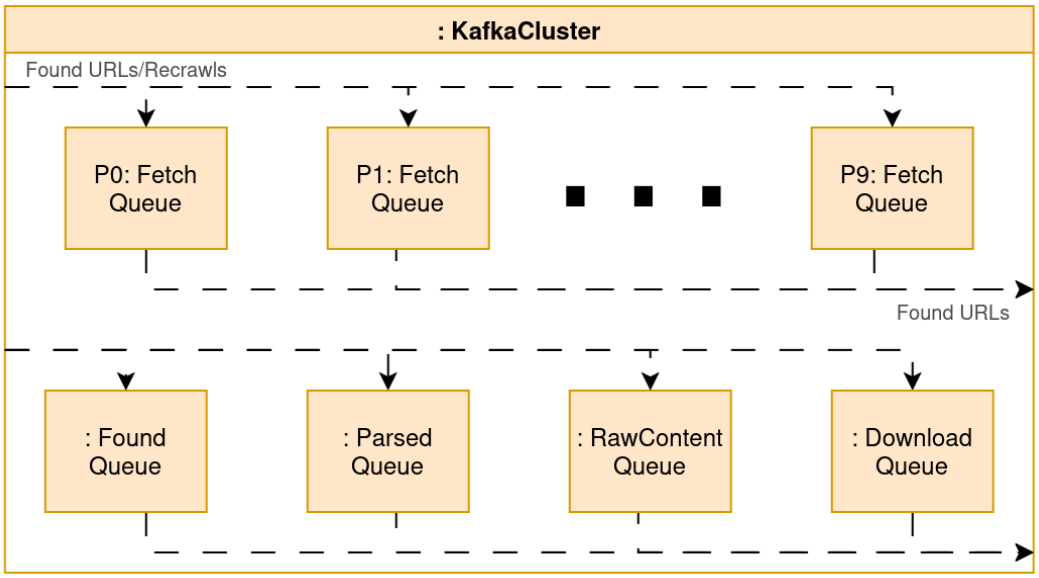


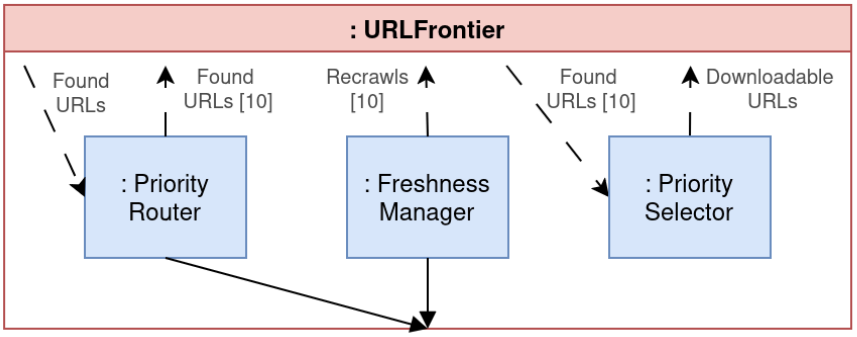
Top Level C&C View

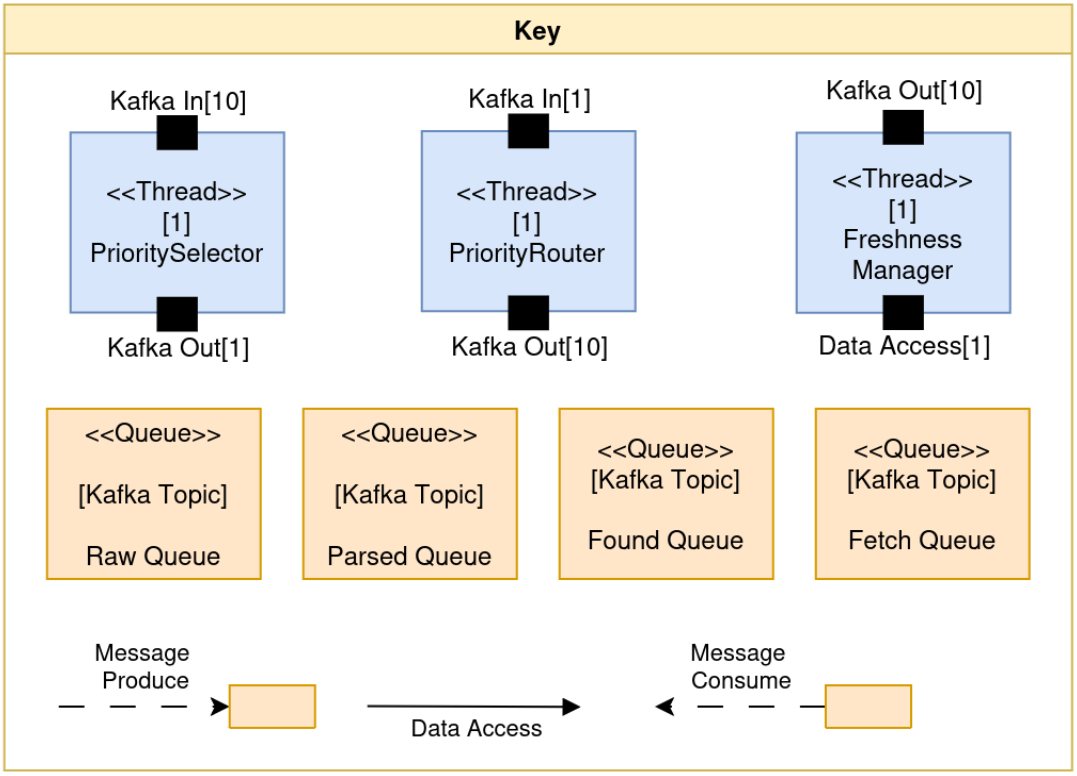




Detailed KafkaCluster and URLFrontier C&C Views







Iteration 2 - DNS Cache

The introduction of a DNS caching mechanism is essential to address performance bottlenecks related to the resolution of domain names into IP addresses. In the current design, the DNSResolver module processes DNS lookups sequentially, leading to potential delays and reduced throughput. Additionally, since many lookup requests often resolve to the same server during web crawling, introducing a caching mechanism can significantly reduce the latency of repeated lookups and improve overall efficiency.

Architectural Drivers:

P1, P2, PB2.

Elements of the system to refine:

The **DNSResolver** sub-module; The **Web Crawler** component; The **Web Crawler** module;

Design concepts:

We utilize the *Maintain Multiple Copies of Data* tactic to reduce the time spent in remote communication with DNS servers.

Architectural Elements, Responsibilities and Interfaces:

The **DNSCache** repository component stores resolved DNS records with a corresponding Time-To-Live (TTL) value and provides efficient lookups of these cached values to minimize latency. Stale entries should expire automatically. The component exposes an interface that allows for:

- Lookups: retrieve the IP of a given hostname if a valid entry exists in the cache.
- Updates: adds or updates a hostname-to-IP mapping in the cache, with the provided TTL value.

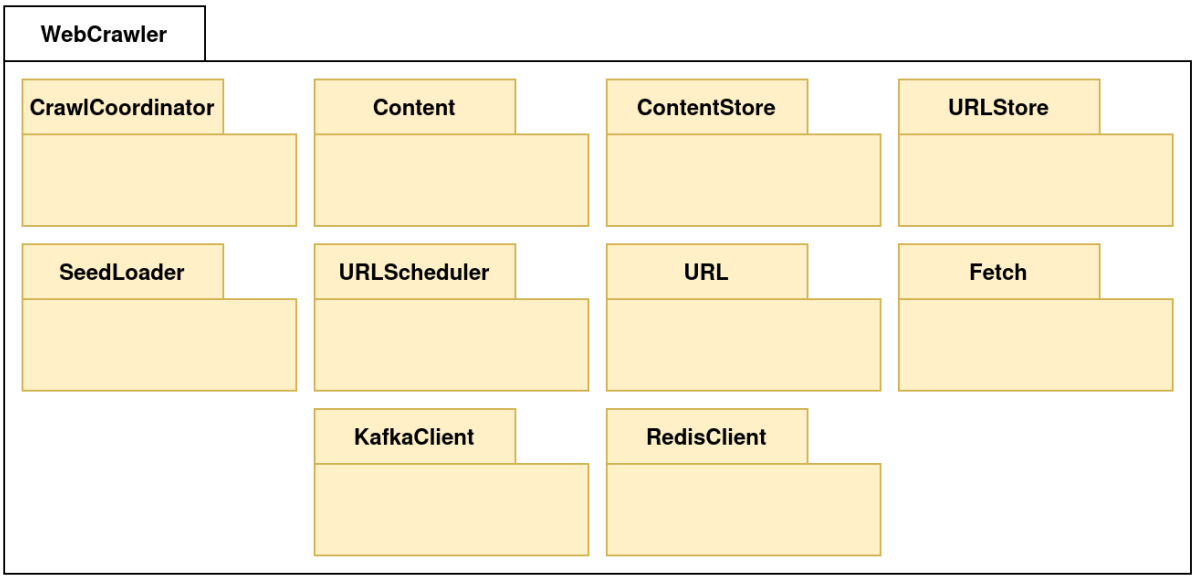
A good technology choice for this type of cache is an in-memory **key-value store**, such as Redis. We need a module for communicating with Redis - **RedisClient**. For now, this module is only used by the DNSResolver.

The **DNSResolver** sub-module will follow a uniform access principle in its use of the **DNSCache**. **DNSResolver** will remain accessible to its actors in the same way, ensuring that interactions are consistent regardless of whether the cache is used for a particular request. Requests that can be resolved using the cache will benefit from a faster response time. The **DNSResolver** will be modified in its implementation to query the **DNSCache** for a resolved hostname before initiating a DNS resolution on the network. If the result is found in

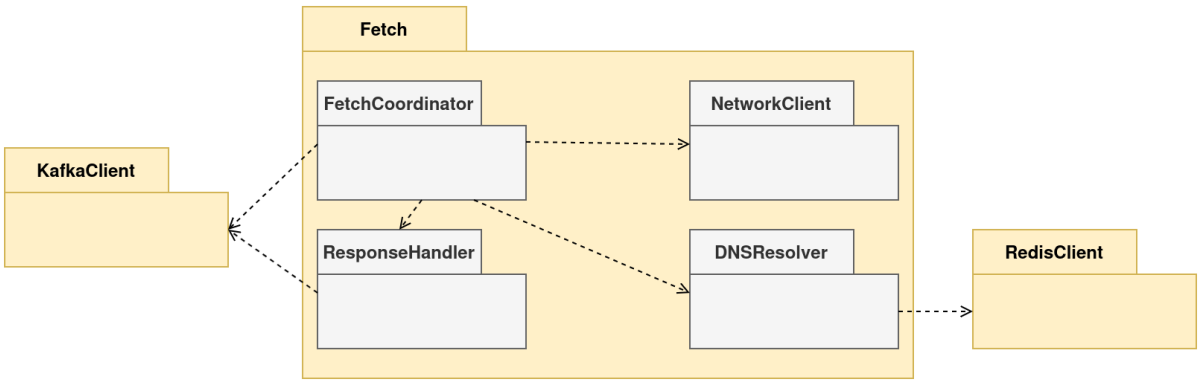
the cache, it returns it. If the cache query does not return a result, it queries the DNS servers, returns the result and updates the cache with the new entry.
Each DNSResolver instance will have its own cache, reducing dependency on shared or centralized caching solutions and minimizing network overhead. This localized deployment enhances performance by allowing the cache to respond quickly to queries, even with multiple instances of the DNSResolver.

View Sketches:

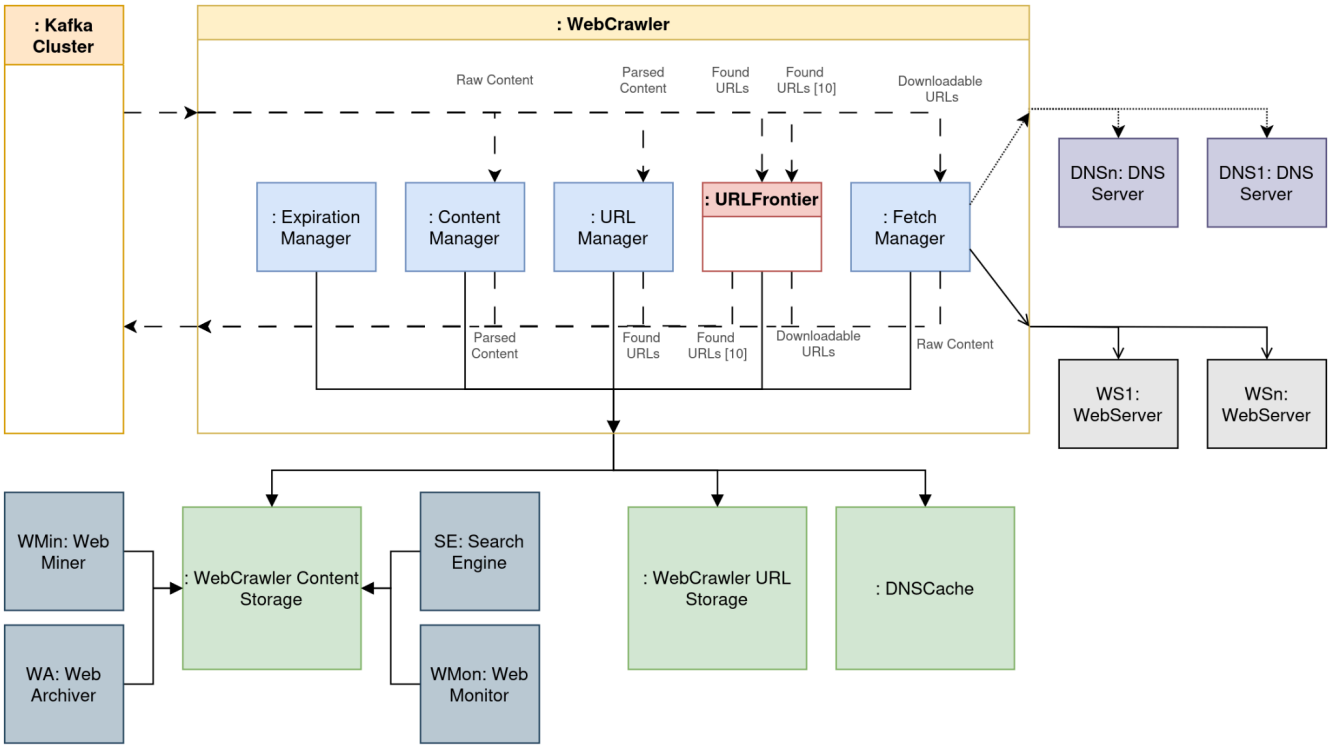
Web Crawler Module Decomposition View:

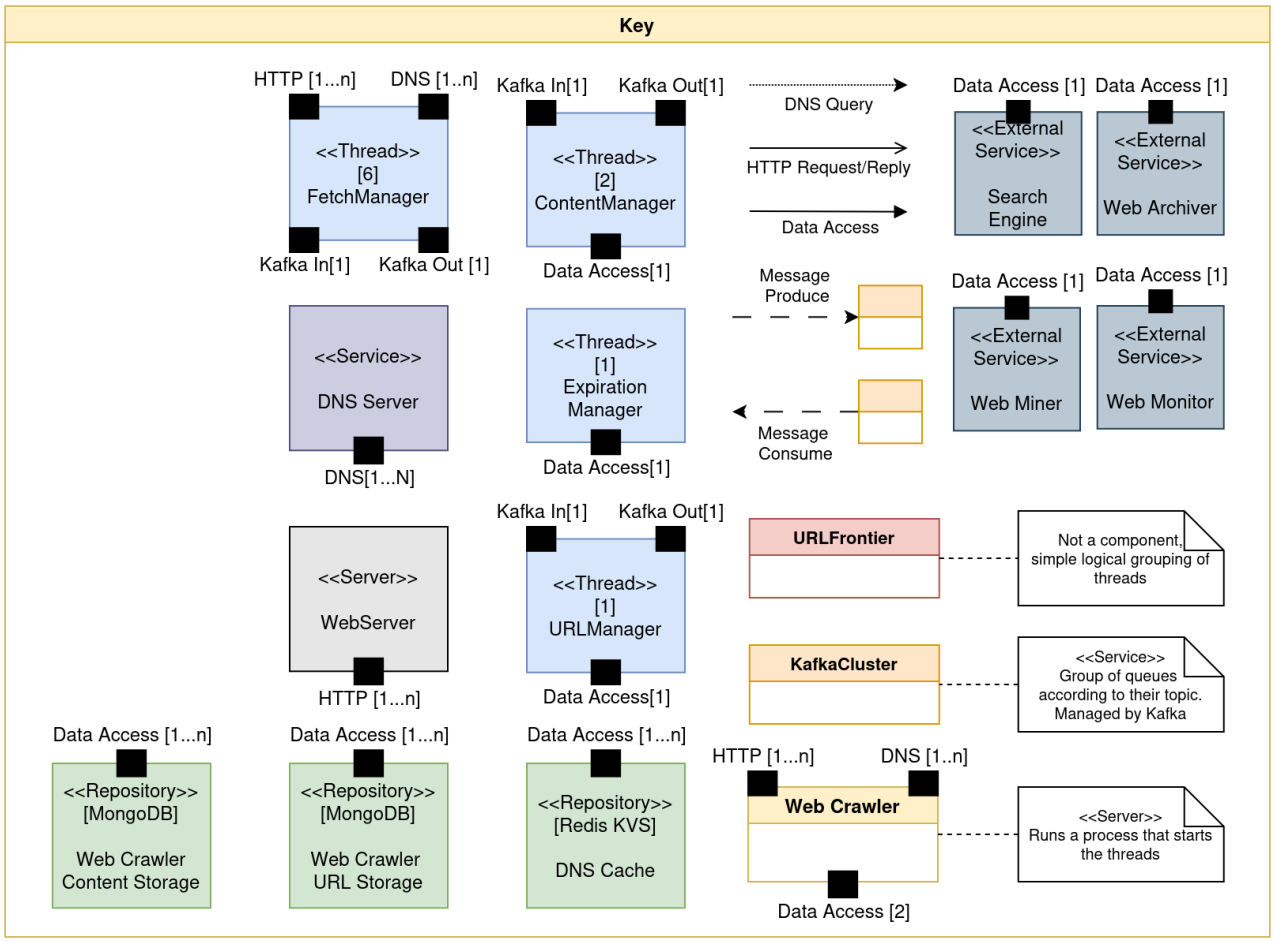


Fetch Module Uses View:



Top Level C&C View:





Iteration 3 - URLFrontier and Queue Storage

In this iteration, we wish to solve the bottleneck associated with the (now named) URL Frontier component's way of obtaining URLs to download.

Architectural Drivers:

P1, P2, PB3.

Elements of the system to refine:

All the system components interacting with the Kafka cluster;

Design concepts:

We utilize the *Maintain Multiple Copies of Data* performance tactic to reduce the latency in reading data from the disk.

Architectural Elements, Responsibilities and Interfaces:

In the previous iterations, we designed a complex queue system to implement asynchronous communication between our system's main components. This queue system utilizes Kafka, popular for its fault-tolerance and other attributes. Kafka uses the disk to store the queued data. Although this enables us to store a very large number of records, it becomes quite slow to read from. We could think about designing a buffer system, i.e, a watcher thread per queue that reads from it and stores the data in memory. Then, the crawler system would query these new components to obtain what was in the queue, but with faster read speeds. However, Kafka already enables this by default. It allows the workers it runs, called *brokers*, to return batches of data to topic consumers. Therefore, we don't require any modifications to the design to solve this problem, only a few configuration options need to be set in the Kafka broker.

Iteration 4 - Performance for Content Store Accesses

In this iteration, we wish to address the quality attribute scenarios related to the performance of reading from the ContentStore. The external systems using the crawler's data require performant and consistent data access, and they prefer that accesses to popular content have reduced latency.

Architectural Drivers:

P3, P4.

Elements of the system to refine:

The ContentStore Component; The Web Crawler Component;

Design concepts:

We apply the *Maintain Multiple Copies of Data* performance tactic to reduce the latency of reading data from the disk. Additionally, the Split Module tactic is used to allocate the new responsibility (selecting where to read the content from) into a new module.

Architectural Elements, Responsibilities and Interfaces:

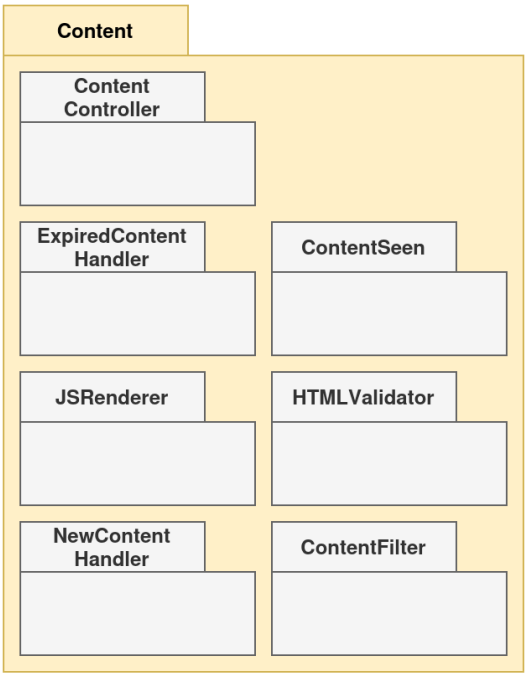
In order to reduce the latency of accessing the crawler's stored content, we instantiate the **WebCrawlerContentCache**. This is a repository component, which can be, again, a Redis Key-Value Store instance. This cache will store popular content, evicting the Least Frequently Used records.

Requiring the external systems to obtain the crawler's data via a direct data access made sense in previous iterations, but with multiple copies of data it's unrealistic to expect such systems to have to figure out where the content they need might be. Therefore, we instantiate the ContentController module, which handles all the requests for content from the external systems.

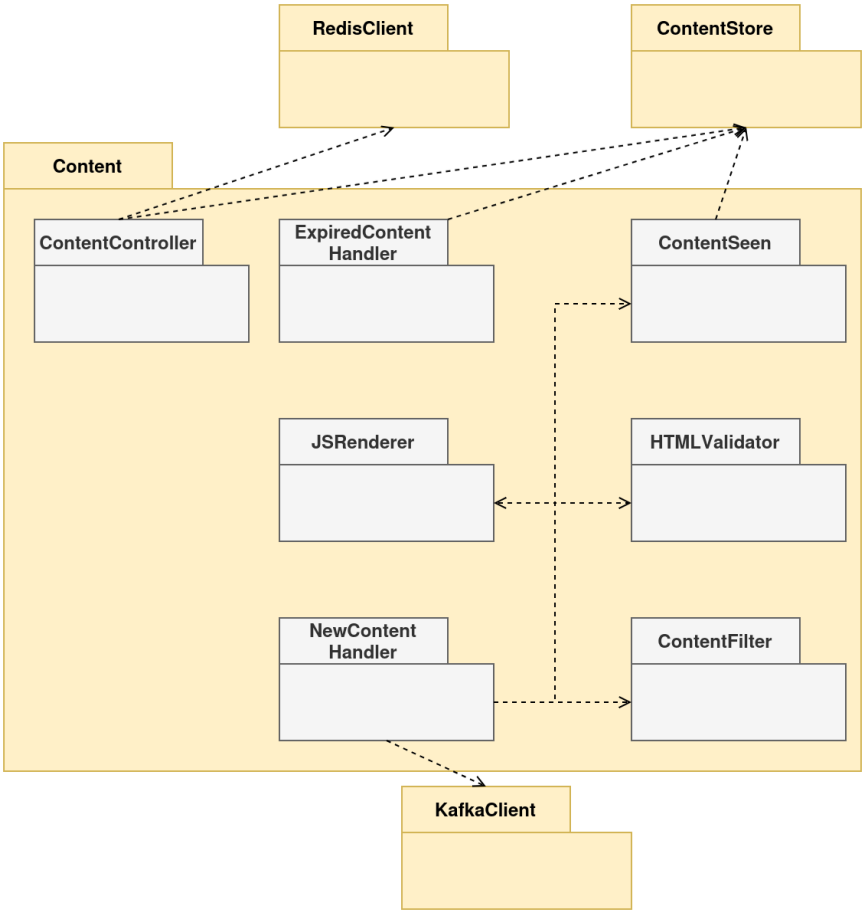
It interacts with the ContentStore, to obtain content, and it also interacts with the new cache, via the previously instantiated RedisClient. When it receives a request, it first checks the cache. If the required content is there, it returns it immediately, and registers the "use" in the cache. If it's not there, it goes to the NoSQL repository and reads from there. If the content is found, the cache is updated and the client application receives it. This module can run as a thread. If the number of requests for content is large, it can be replicated as many times as needed behind a load balancer, since it is completely stateless.

View Sketches:

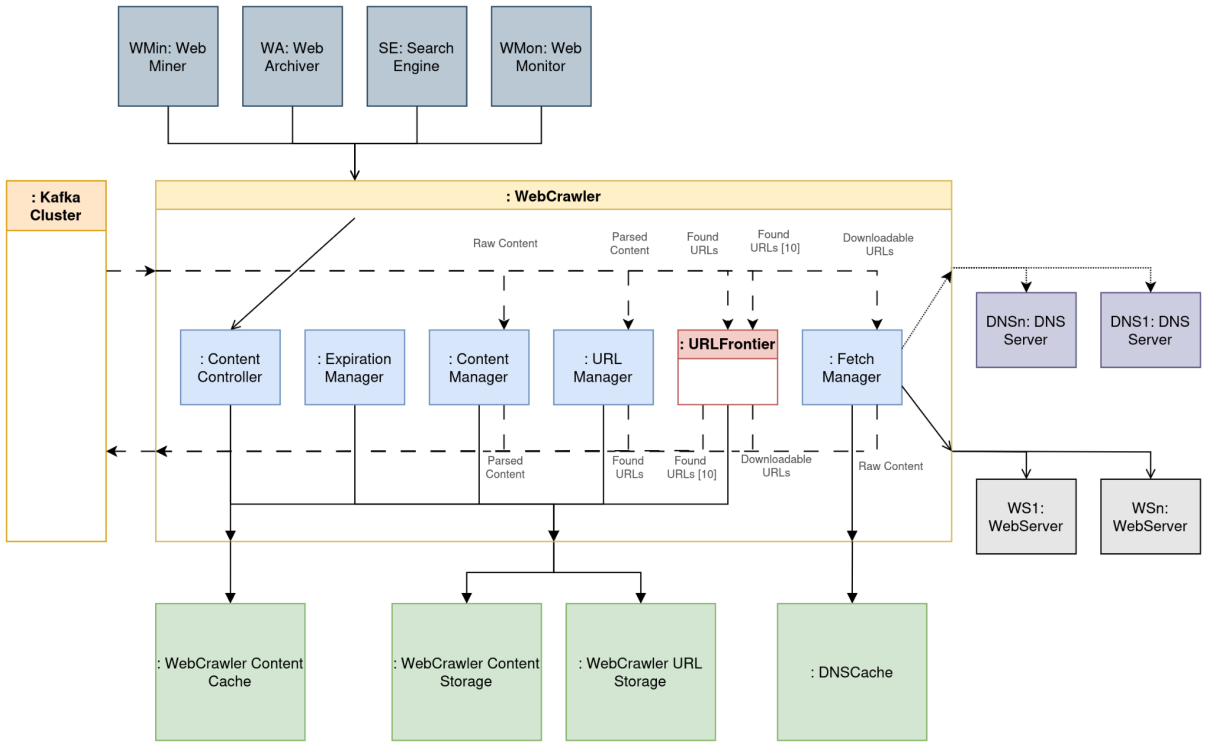
Content Module Decomposition View



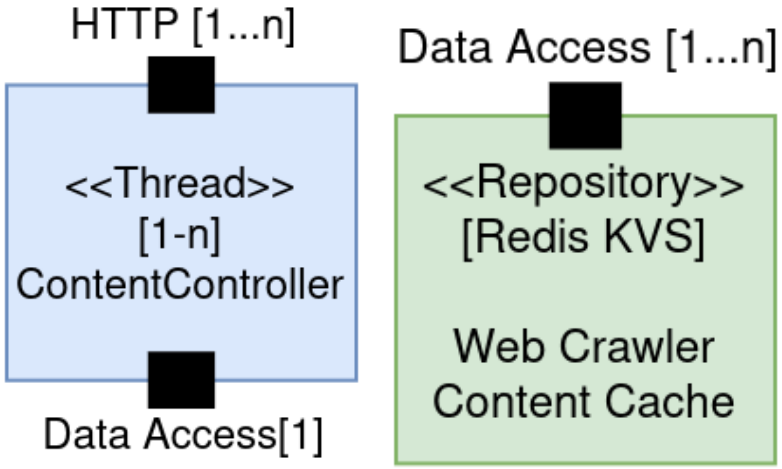
Content Module Uses View



Top Level C&C View



(to avoid repeating the entire key, only the new component types are shown next)



Iteration 5 - Normal Load Performance

In this iteration, we wish for the system to reach its required normal load performance measures. In the previous iterations the crawler's throughput has increased drastically, but it still isn't enough for the massive scale required. In Iteration 1, we mentioned the need for more machines, due to the high number of threads running in our system. The current single machine configuration is another performance bottleneck (**PB5**), which we will address in this round.

Architectural Drivers:

P1, PB5

Elements of the system to refine:

The system's components;

Design concepts:

We are utilizing the *Maintain Multiple Copies of Computations* performance tactic to increase the number of available workers, each running multiple copies of the aforementioned threads.

Architectural Elements, Responsibilities and Interfaces:

In iteration 1, we mentioned that our Web Crawler component could crawl around 100 pages per second with a multi-threaded setup. It is easy to wrongly assume that by multiplying the number of threads by 4, we could achieve our required throughput. However, depending on the machine our software is running in, adding more threads might not be beneficial due to CPU limits (real parallelization is not the same as threads), RAM limits, network bandwidth limitations, etc…

Therefore, the logical solution is to maintain a reasonable number of threads in one server and then replicate the server as many times as needed until the needed performance measures are observed.

We modify our current configuration to support this. First, we address the Web Crawler component itself. If before we were getting 100 QPS, we should get around 400 QPS by creating 4 copies of this component.

The Kafka Cluster can remain unaltered, though it should be hosted on a separate machine, so that all the new instances of the crawler can connect to it.

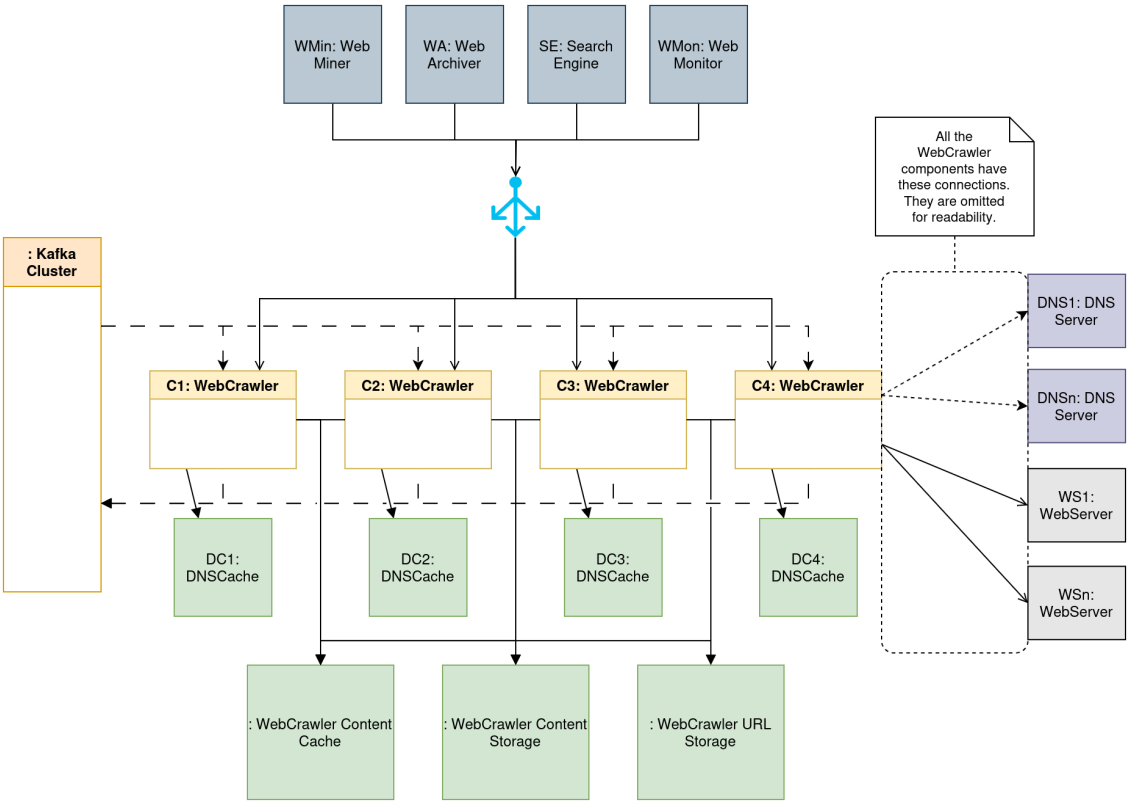
The ContentStorage, ContentCache and URLStorage repositories remain "centralized", that is, all the crawler nodes share the same data. Both of the software options used for these components (MongoDB and Redis) are already highly scalable, allowing us to grow them as needed, to support the increased amount of content to store.

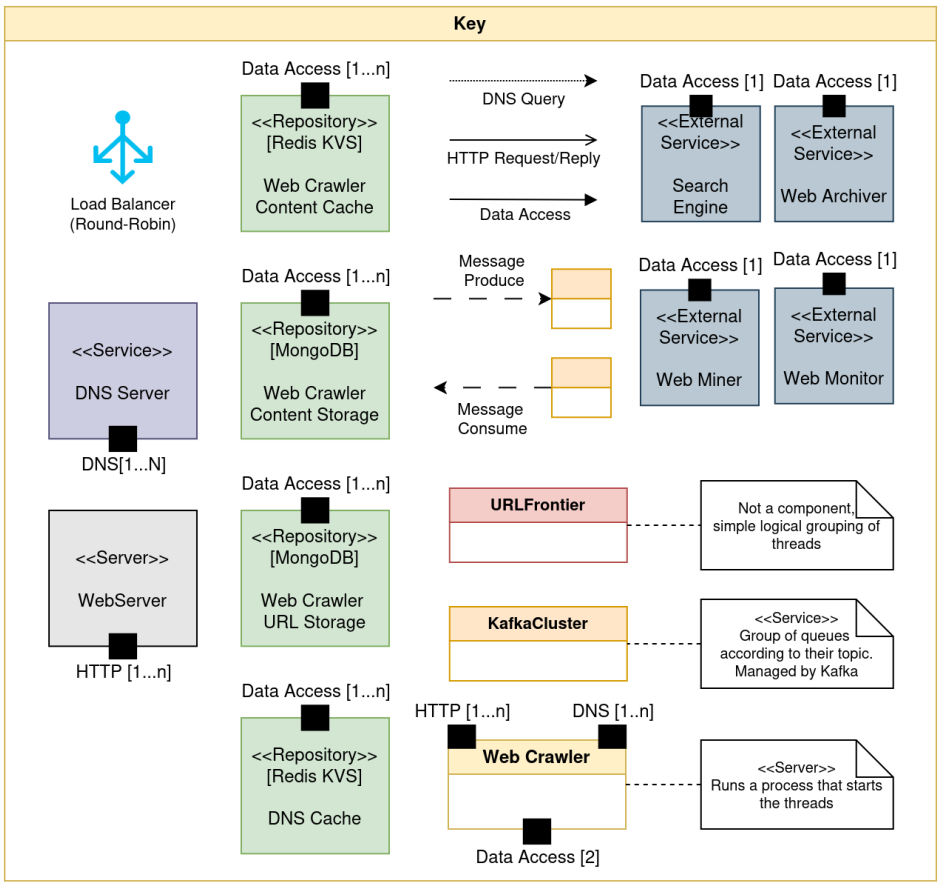
The DNSCache is handled differently - a new DNSCache is created per instance of Web Crawler. It should be on the same machine as the crawler, to avoid the network communication overhead.

Finally, we handle the issue of stored content accesses. We can't expect external systems to choose from one of our nodes in an efficient way. We solve this problem by adding a load balancer component. It will receive the requests from the Search Engine and other external client applications, and route them to an available Web Crawler Content Controller thread.

View Sketches:

Top Level C&C View:





Iteration 6 - Peak Load Performance

With the previous iteration completed, our system now behaves as expected, regarding performance, under normal system load. However, it still does not match the expected 800 QPS during peak system load. We must address this performance scenario to have a fast, large scale web crawler.

Architectural Drivers:

P2, PB4

Elements of the system to refine:

The system's components.

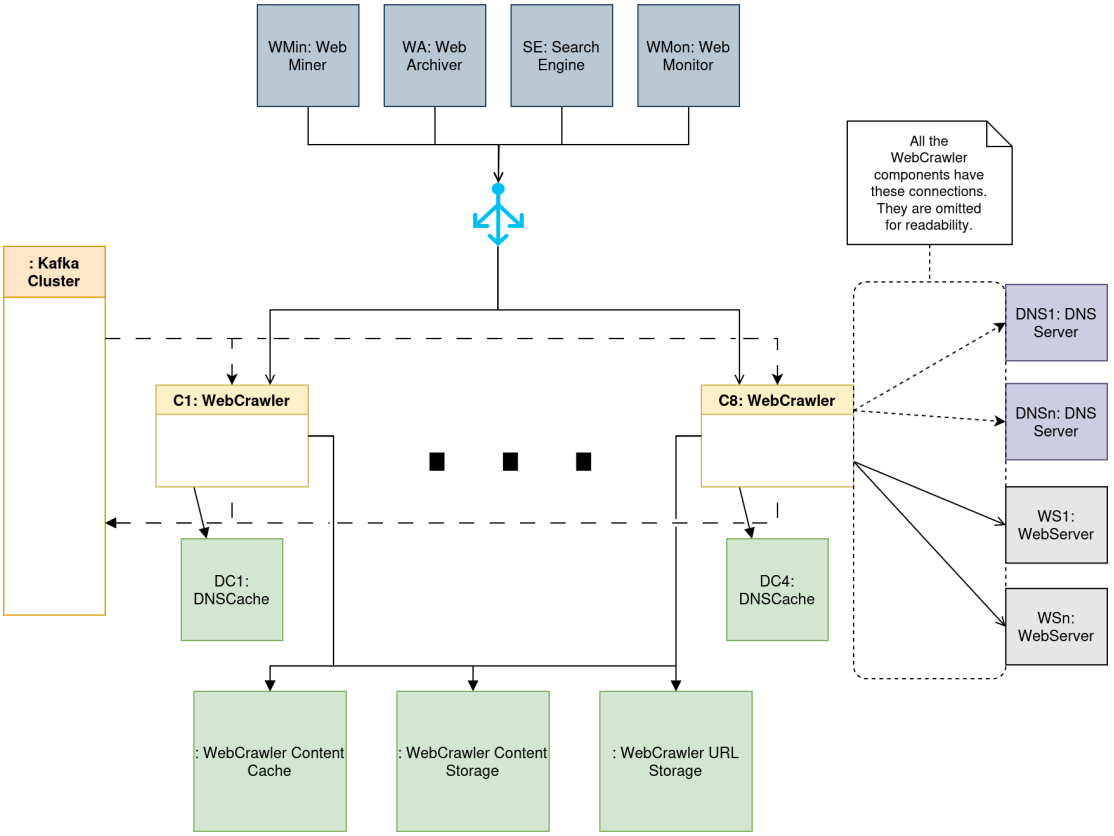
Design concepts:

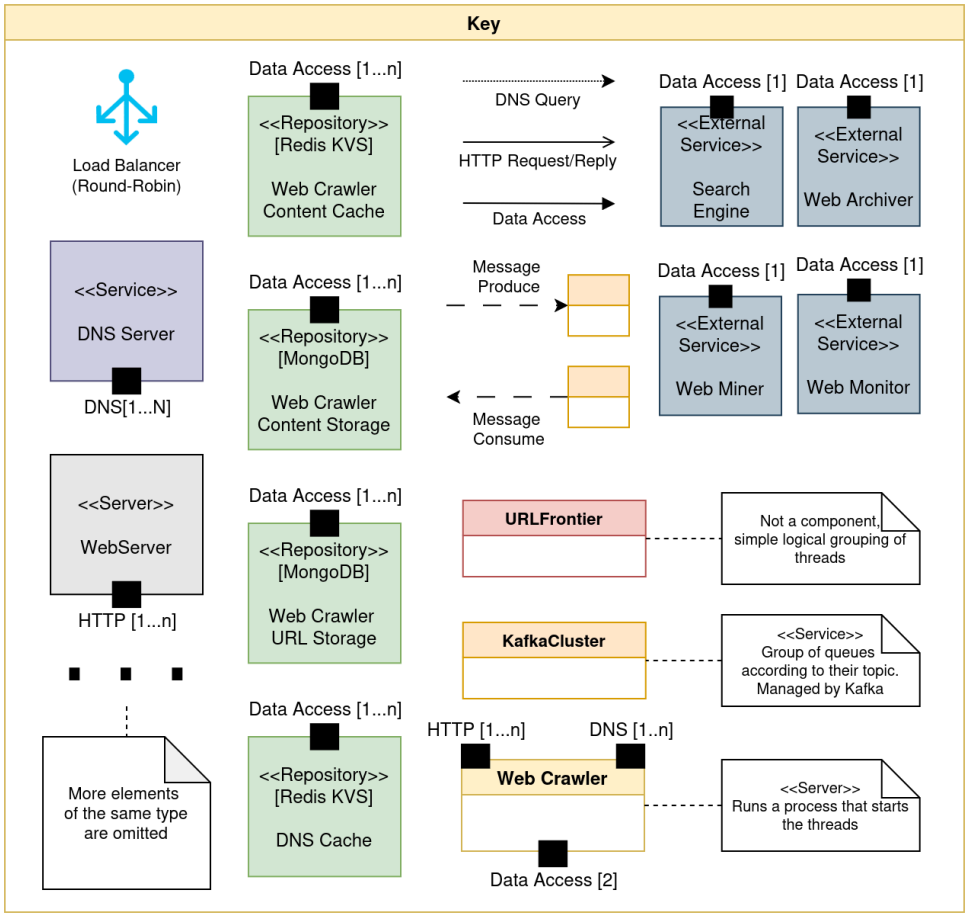
Again, we wish to increase the throughput of our system. Therefore, we apply the *Maintain Multiple Copies of Computation* performance tactic. We also use the *Reduce computational overhead*, by addressing the components geographic proximity.

Architectural Elements, Responsibilities and Interfaces:

Similarly to the previous iteration, to increase the throughput of the crawler, we need to add more server nodes. In this case, we will make the number of available workers to be 10. This should give us, even considering the delays added from inter-process communication (with the kafka cluster, or with the repositories), the required 800 QPS. These workers should be deployed in different locations across the globe, so that latency does not increase with distance to the target web server.

View Sketches:





Round 3 - Politeness and `robots.txt`

With the system now being capable of impressive throughput numbers, we need to be careful not to overwhelm the external web servers we are crawling. It is one thing for a server to handle a few requests per hour from a client. Hundreds of requests per second will overwhelm the server, possibly causing it damage or denial of service, or making it treat the crawler as an attacker performing a DDOS. Therefore, we need to be careful. By the end of this round, our system will successfully integrate the external web servers it's getting pages from, avoiding the violation of rate limits. Additionally, the system will be able to respect the `robots.txt` protocol specified by each host it is crawling.

Iteration 1

Architectural Drivers:

I1

Elements of the system to refine:

The Kafka Cluster, the URLScheduler module, the URLFrontier component.

Design concepts:

The *Split Module* tactic is applied to separate new responsibilities into their modules.

Architectural Elements, Responsibilities and Interfaces:

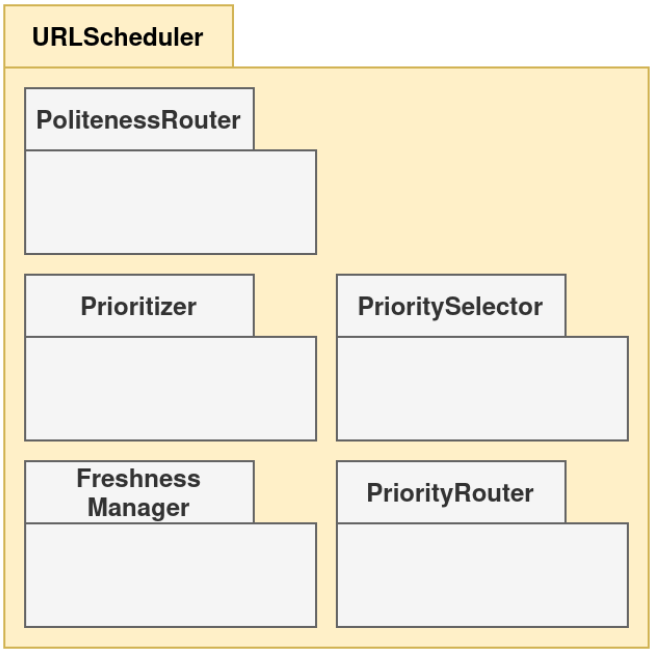
To enforce politeness, we want to download at most 1 page at a time from a particular host. We implement the politeness constraint by maintaining a mapping from website hostnames to Kafka topics.
Each FetchManager thread will now be associated with a single topic, which contains URLs from a single host. FetchManager threads can be reassigned (or else we would have a limit on the number of possible hosts to crawl), but the important thing is that at most one of these threads is mapped to a Topic.
We introduce the **PolitenessRouter module** (which runs in a thread component with the same name) as a sub-module of URLScheduler. Additionally, we introduce a **MappingTable** component - another in-memory KVS (Redis) that maps running hostnames to Topics. The PolitenessRouter uses this MappingTable to send URLs to the correct Topic.
Finally, we need to route these queues to the correct threads. The MappingTable can be used for this. The mapping in this table is of the type <hostname, topic, threadId>. Each FetchManager thread, on startup, generates a unique ID. Then, it assigns itself to an empty

topic and starts consuming from there. If it does not get URLs for a brief period of time, it reassigns itself. After a while of processing the same host, it also reassigns itself. This allows us to effectively cover all hosts efficiently, but keep request rates low for each host.

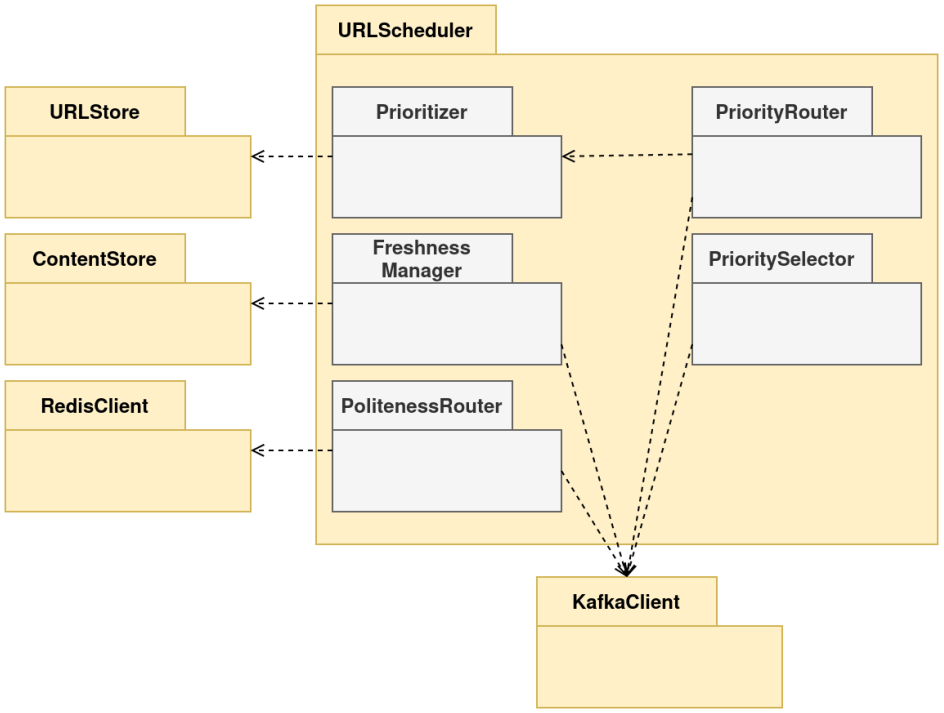
Now, we need to handle the `robots.txt` protocol. When a host is discovered, it might enforce limits on what pages our crawler is allowed to download. These limits are described in the `robots.txt` file, typically at the root of the host (e.g. google.com/robots.txt).
If each thread has to first get the `robots.txt`, and then crawl the page it meant to crawl (if allowed), we effectively double the latency of downloading each page. This is very unideal. Therefore, we introduce another cache type, this time for `robots.txt` files only. This component, called **RobotsCache,** is another Redis instance. It evicts based on the least frequently accessed files. Similarly to the **DNSCache**, there should be an instance of this cache per instance of the Web Crawler component, and both should be deployed with high proximity (ideally, on the same network). While we end up having some redundancy (same file in two or more caches), we prevent the delay that would be added by querying the cache over a remote network.

View Sketches:

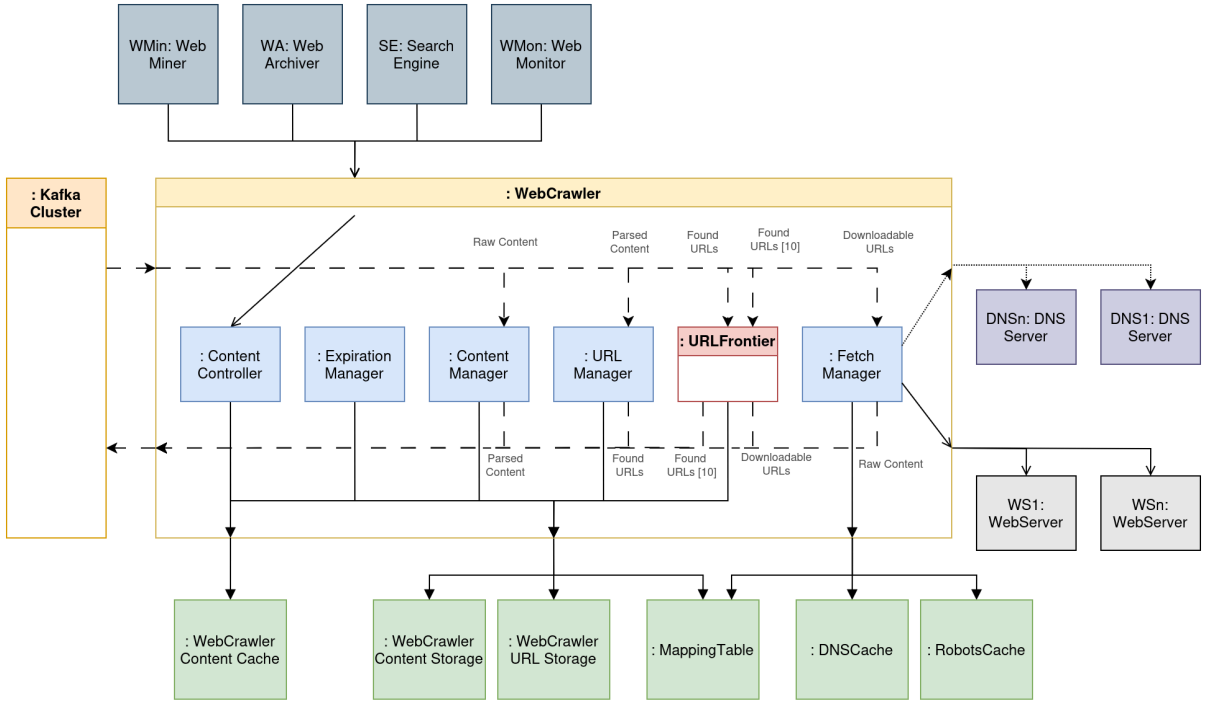
URLScheduler Module Decomposition View

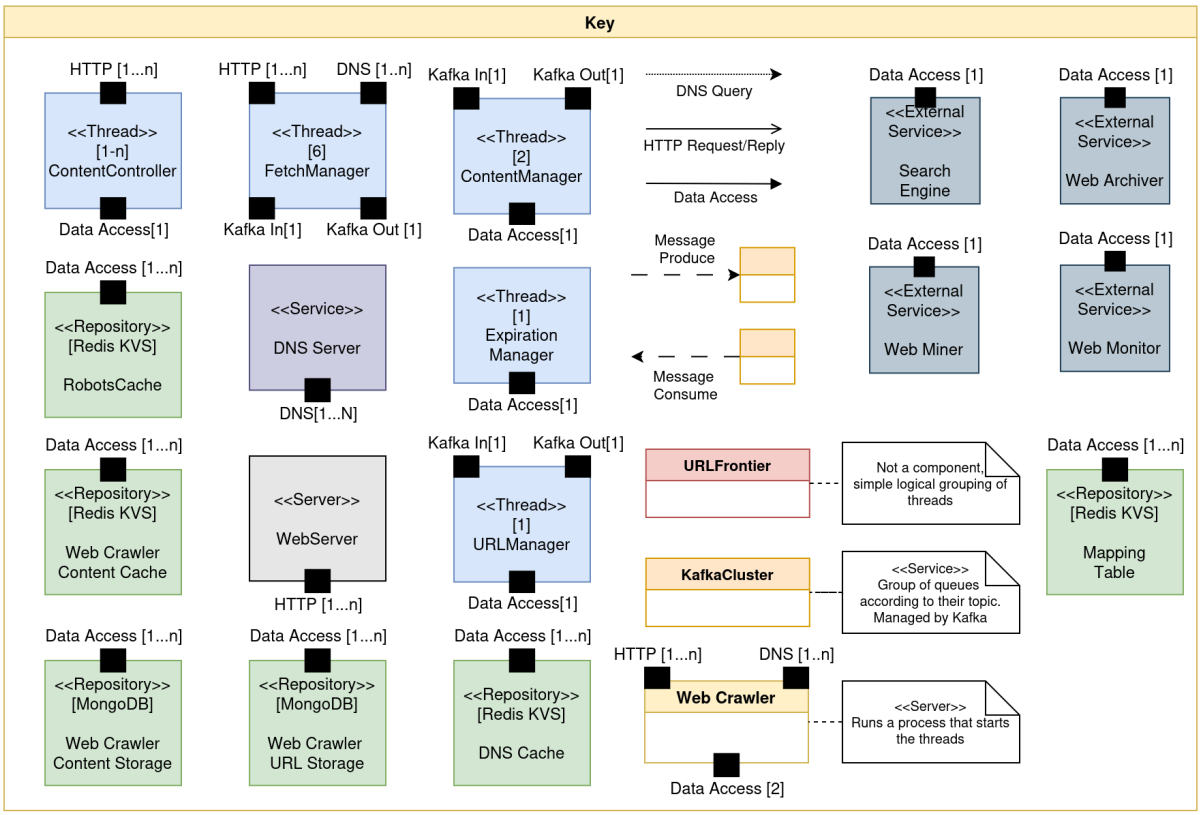


URLScheduler Module Uses View

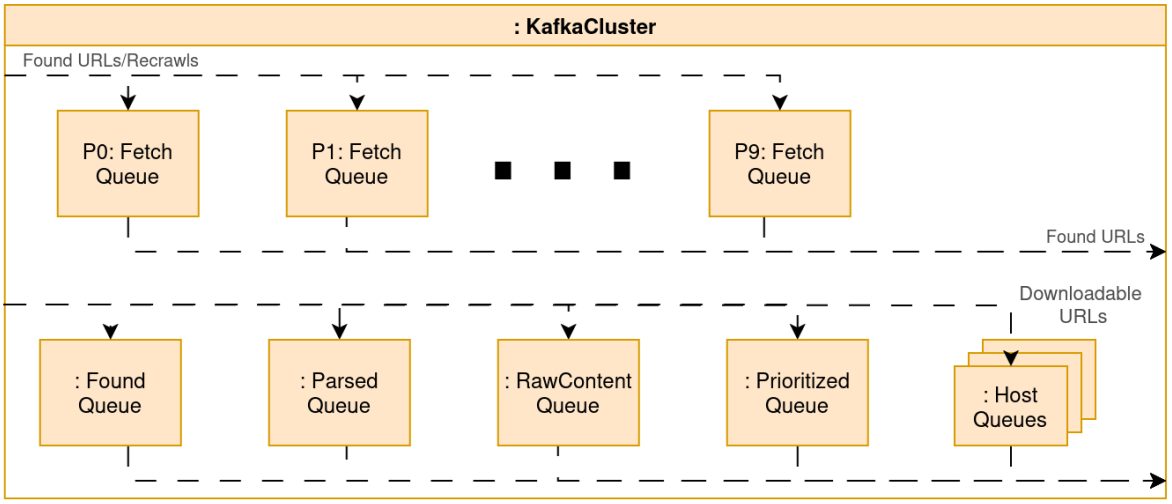


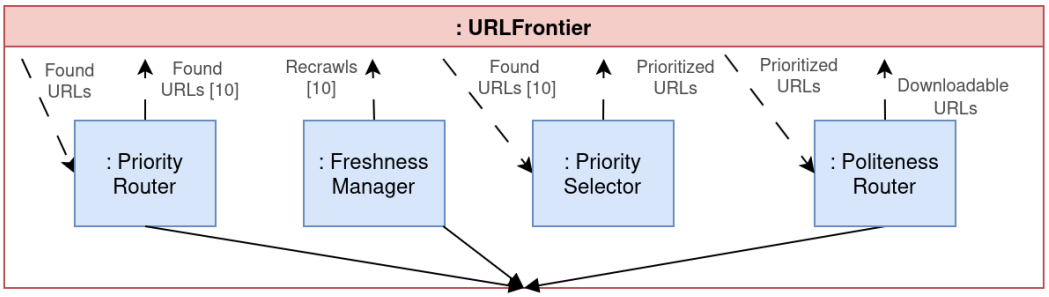
Web Crawler C&C View

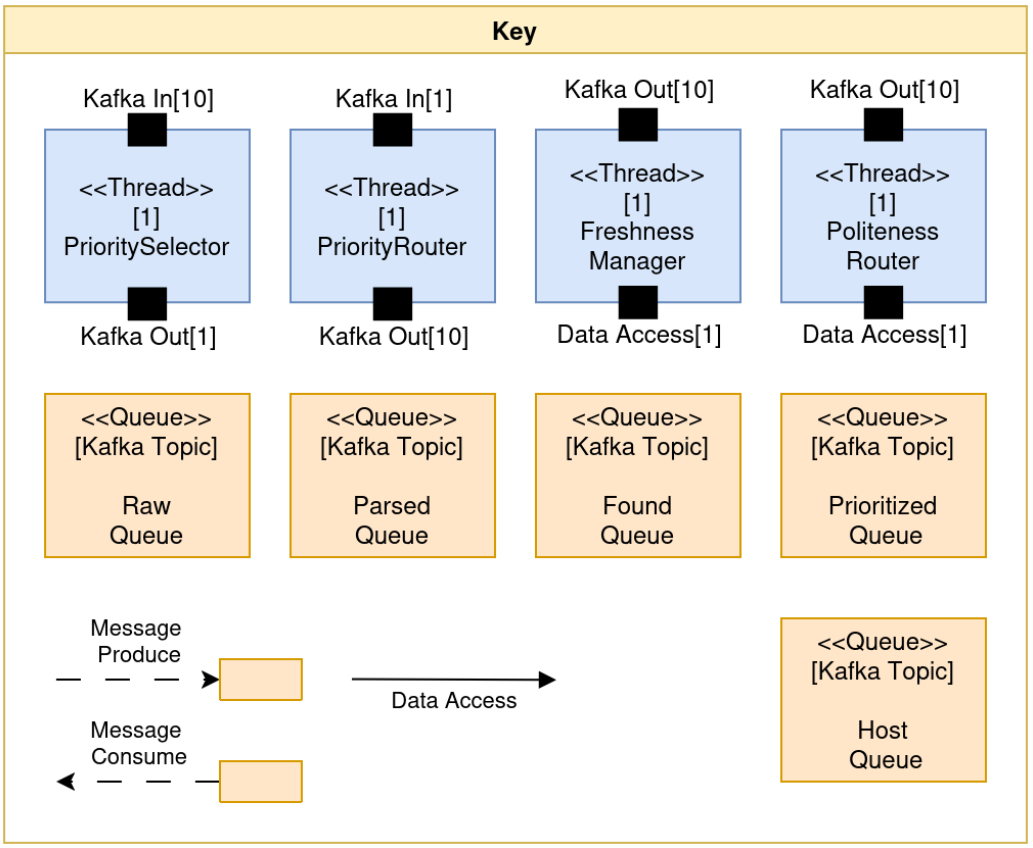




URLScheduler and KafkaCluster C&C View







Round 4 - Scalability

By the end of this round, the system should automatically increase the resources available when the load increases. We introduce auto-scaling for some components. Scaling the storage will also be possible with little impact on throughput and data integrity.

Iteration 1 - Auto-scaling the crawler

In this iteration, we wish for the crawler to self-modify, in order to increase available resources when needed. This is different from the previous rounds, where we addressed performance from a “static” point of view. In this iteration, we want to adjust the available resources for the crawler based on the required throughput at runtime.

Architectural Drivers:

M2

Elements of the System to refine:

The whole system as a component;

Design Concepts:

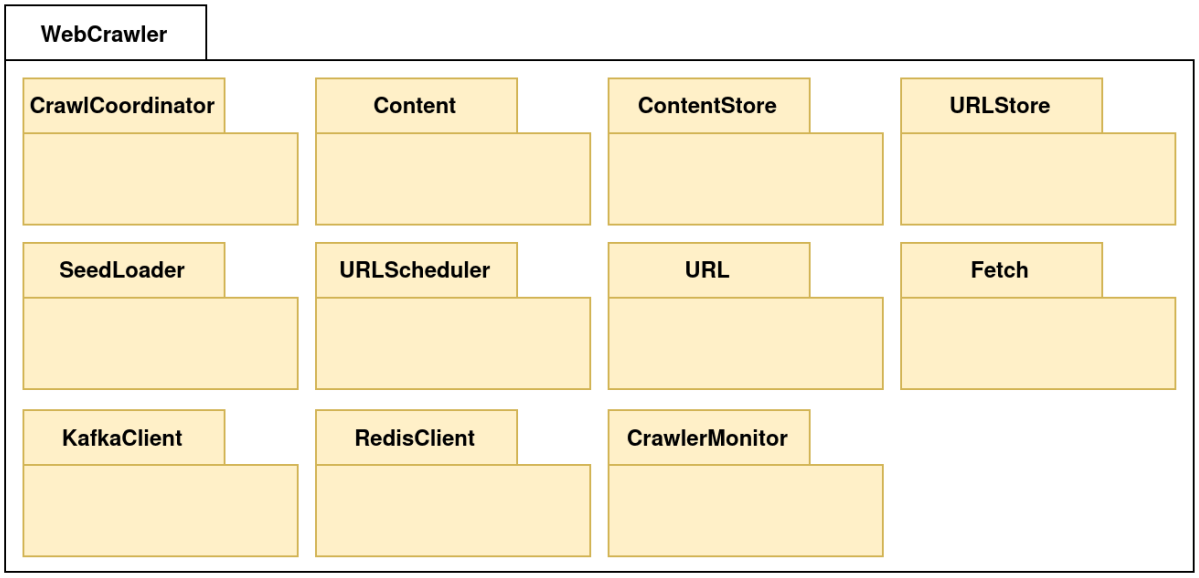
The *Maintain Multiple Copies of Computations* performance tactic is applied to grow the number of available computational resources of the system.
The *Split Module* tactic is used to separate the new responsibilities into their own modules.

Architectural Elements, Responsibilities and Interfaces:

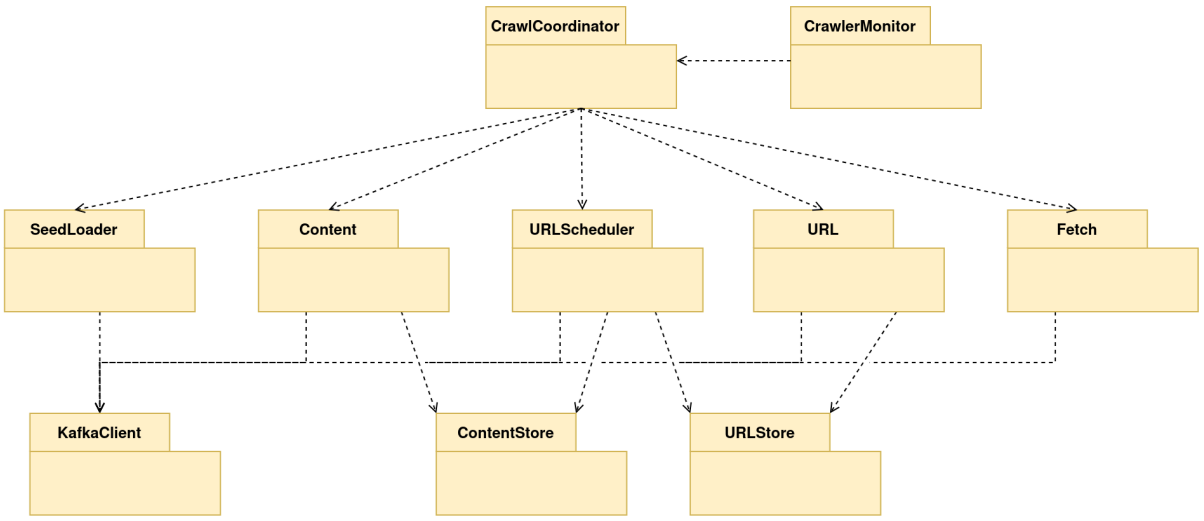
To effectively auto-scale our system, we introduce the CrawlerMonitor component, which will determine, mostly based on repository usage (spike in URLs added = more demand for crawling), the current required throughput. If it sees a big increase in demand, it will start new instances of the WebCrawler component.
We instantiate a top-level module, with the same name, that contains the code which runs in this component. We divide it based on key responsibilities - **Watch**, for gathering data from the repositories and **Control**, for starting/stopping nodes on demand.

View Sketches:

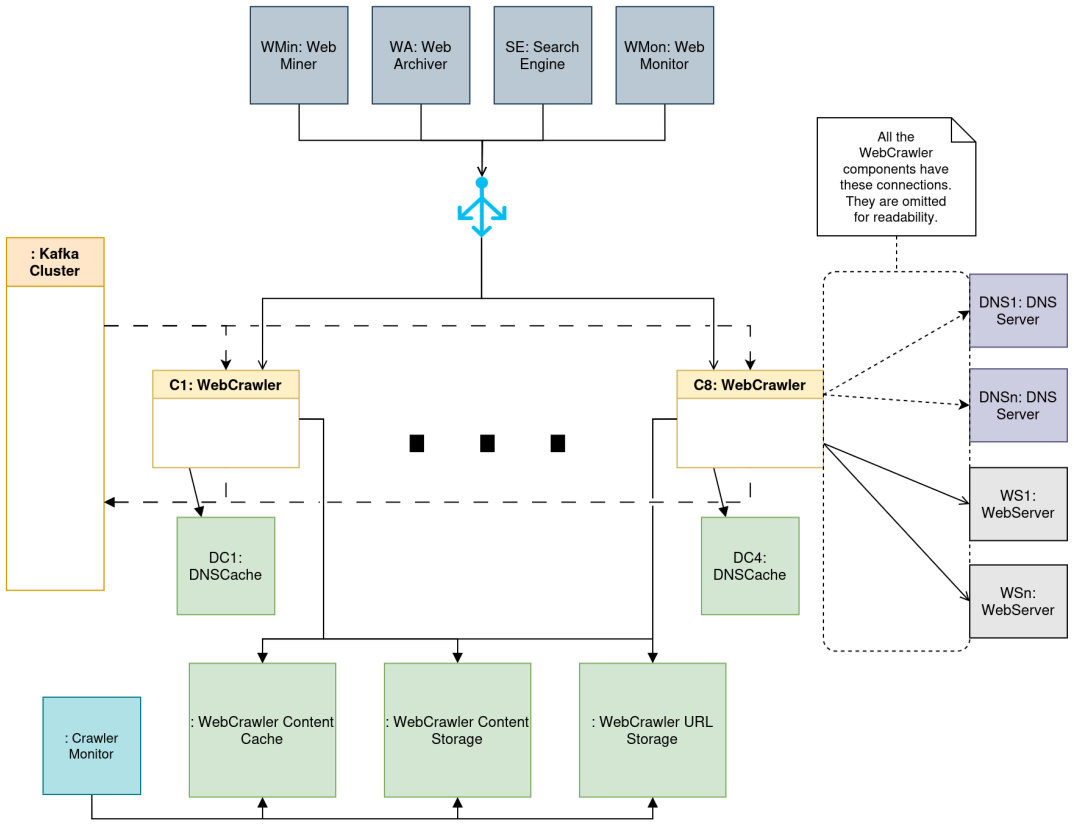
Top Level Decomposition View

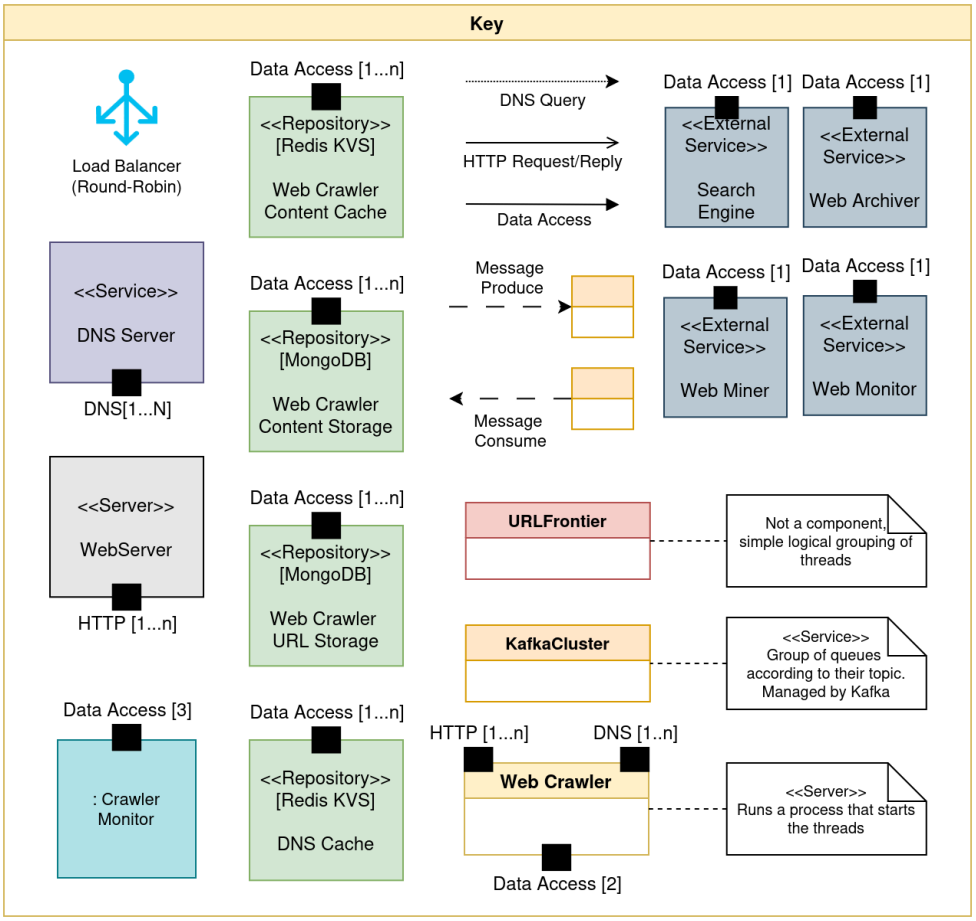


Top Level Uses View



Top Level C&C View





Iteration 2 - Scaling the Content Storage

By the end of this round, the system should respond correctly to an increase in the demand for content storage capacity.

Architectural Drivers:

M3

Design Concepts:

We utilize the *Increase Resources* performance tactic to help us solve the storage capacity problem.

Elements of the System to refine:

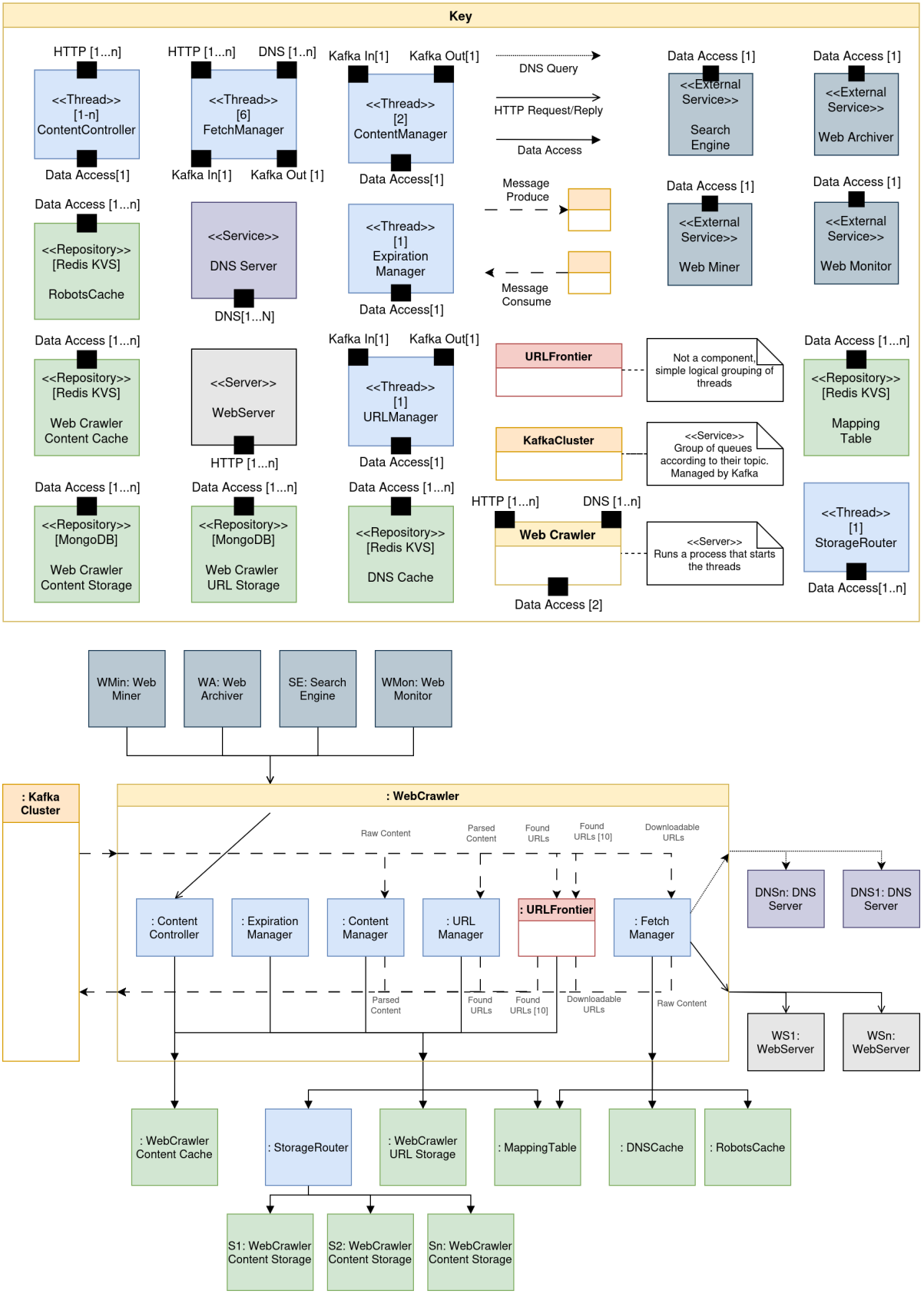
The WebCrawler Content Storage repository component.

Architectural Elements, Responsibilities and Interfaces:

When an increase in demand for storage capacity occurs, the capacity of the storage node needs to increase. Eventually, it will reach a point where it is unfeasible to only have one single storage node. We require database horizontal scaling via sharding. The sharding logic should apply consistent hashing techniques, so that adding/removing new nodes does not break the previous configuration.
We introduce a new component to help us manage this - the **StorageRouter thread**. It receives all the requests for storing content and, via consistent hashing, selects the appropriate repository (shard) for this piece of data. With this component in place, all we need to do is add as many NoSQL shards (nodes) as we need, until the capacity is in place.

View Sketches:

Top Level C&C View



Round 5 - Robustness

In this round, we wish to address the external problems (not attacks yet) that might cause our crawler to be unavailable.

Iteration 1 - Unresponsive web servers

In this first iteration, we address a common problem with network related tasks - unresponsive or disabled web servers. We will discuss how they might affect the crawler, and how the system might react to such a fault in order to maintain optimal performance and availability.

Architectural Drivers:

A2

Design Concepts:

We utilize the *Retry* availability tactic to help us solve the unresponsive web server problem.

Elements of the System to refine:

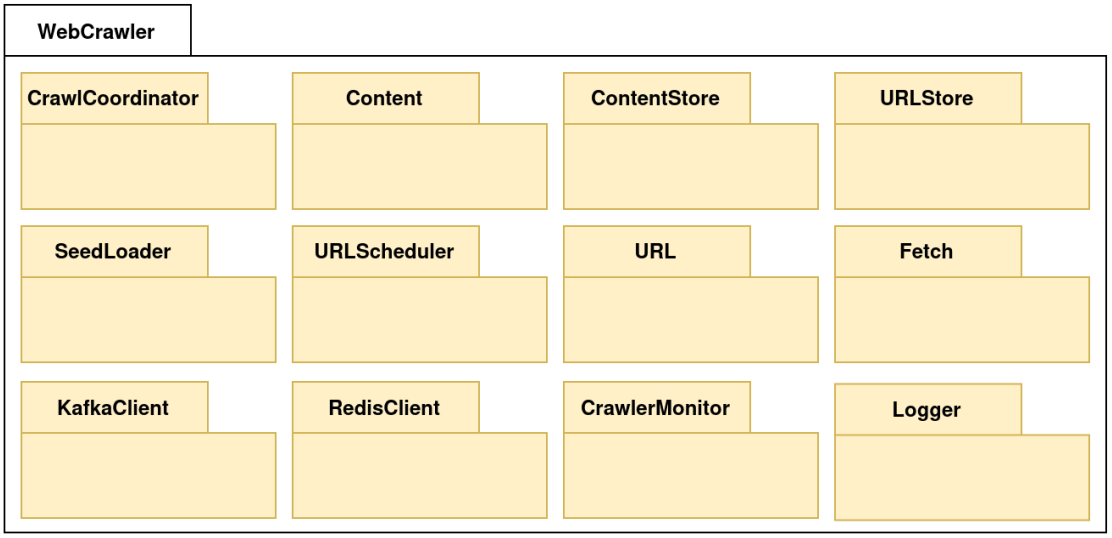
The Fetch module. The Web Crawler Module.

Architectural Elements, Responsibilities and Interfaces:

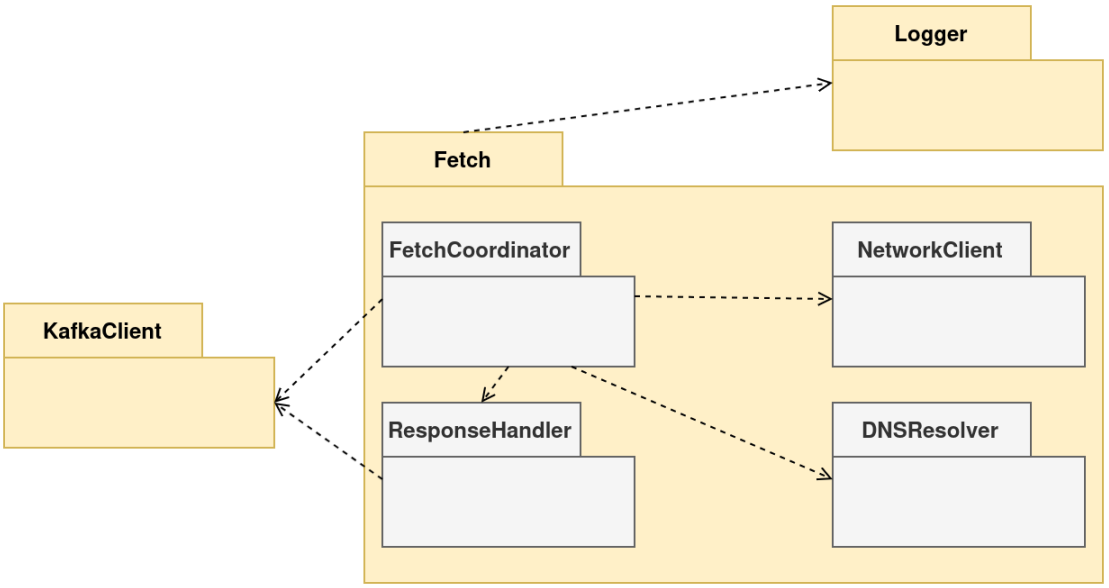
When a FetchManager thread attempts to download a page, the target host might be down (temporarily or permanently). It might also have just errored on this specific request, and it would be wrong for the crawler to assume it was totally offline. Therefore, we modify the behaviour of the Fetch module, such that, when it does not get a response, it will retry up to 5 times, with increased delay between each retry. If, at the end of the attempts, it does not get any reply, it should log the fault (using the new **Logger** top level module), and temporarily stop targeting this host. This last behaviour can be implemented by making the thread assign itself to a different host queue (see Round 3).

View Sketches:

Top Level Decomposition View



Fetch Module Uses View



Iteration 2 - Malformed URLs

In this first iteration, we address the event of finding an invalid or malformed URL. We will discuss how it might affect the crawler, and how the system might react to such a fault in order to maintain optimal performance and availability.

Architectural Drivers:

A3

Design Concepts:

We utilize the *Ignore Faulty Behaviour* availability tactic to help us solve the malformed found URLs problem.

Elements of the System to refine:

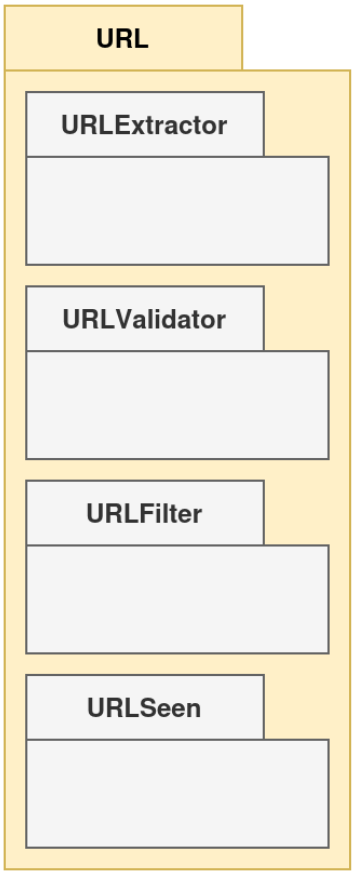
The Fetch module. The Web Crawler Module.

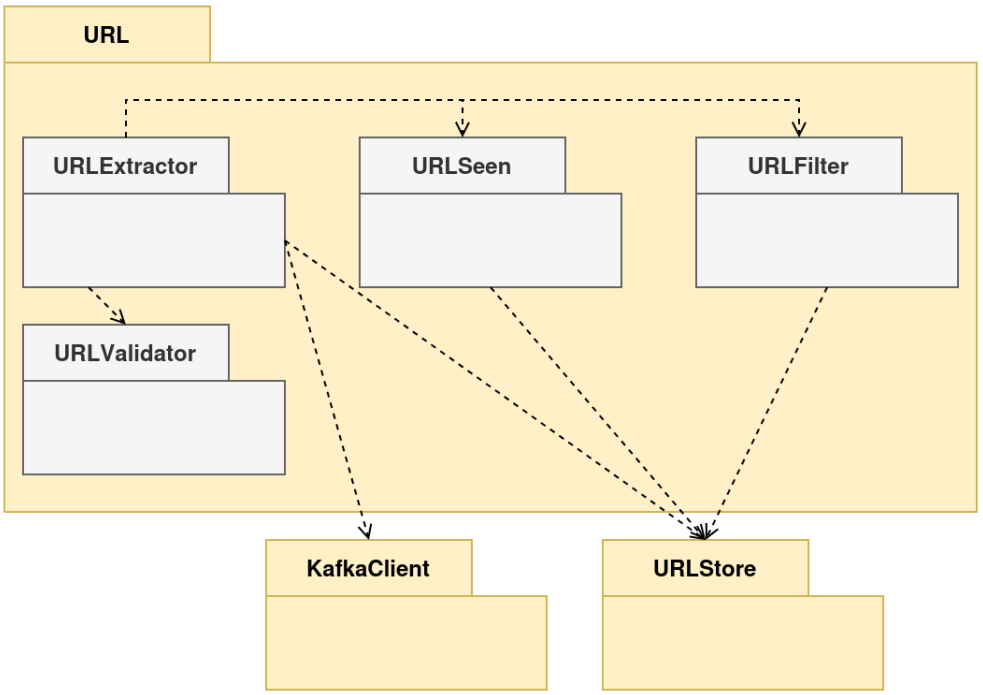
Architectural Elements, Responsibilities and Interfaces:

When the LinkExtractor module finds a malformed URL in a page, it should be able to detect it before storing it, as not doing it might cause problems later on (attempting to download invalid URLs, crashes during other operations involving the URL, etc…). Therefore, the only thing we need to instantiate in this round is a new **URLValidator** module, that can be called to discover if a given piece of text is a valid and well formed URL.

View Sketches:

URL Module Decomposition and Uses Views





Round 6 - Security

In this round, we wish to address the attacks that might attempt to access our crawler's data, reduce the system's availability or do any other harm to the system or its stakeholders.

Iteration 1 - Spider traps

A spider trap is a web page that causes a crawler to enter an infinite loop. It is typically an infinitely deep directory structure, such as:
www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/…

Architectural Drivers:

S1

Design Concepts:

We utilize the *Detect Service Denial* and *Validate Input* security tactics by filtering each URL found, ignoring the ones matching suspicious patterns such as many levels of recursion.

Elements of the System to refine:

The URLFilter module.

Architectural Elements, Responsibilities and Interfaces:

Spider traps can be partly avoided by setting a maximal length for URLs. There is no perfect solution to avoid them. Ideally, combining the setting of a maximum length for URLs with strong monitoring and performance analysis might result in avoidance of these types of attacks. For the focus of our system, we modify the URLFilter module to also discard URLs with many levels of depth.

Iteration 2 - Malicious Content

Besides attempts to reduce availability, pages can contain malicious JavaScript that attempts to hijack system resources or access data stored on the victim machine. Despite most of these attacks targeting regular internet users, our crawler must still protect itself.

Architectural Drivers:

S2

Design Concepts:

We utilize the *Validate Input* security tactic by validating a page's JavaScript before attempting to execute it (render the page via SSR).

Elements of the System to refine:

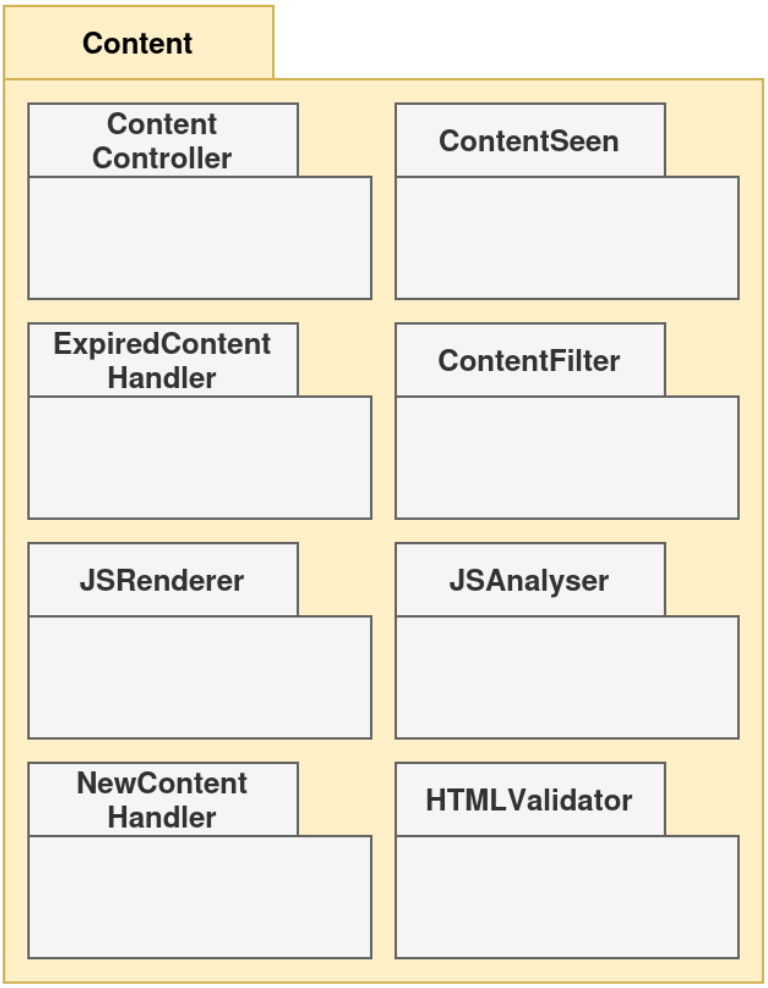
The Content module;

Architectural Elements, Responsibilities and Interfaces:

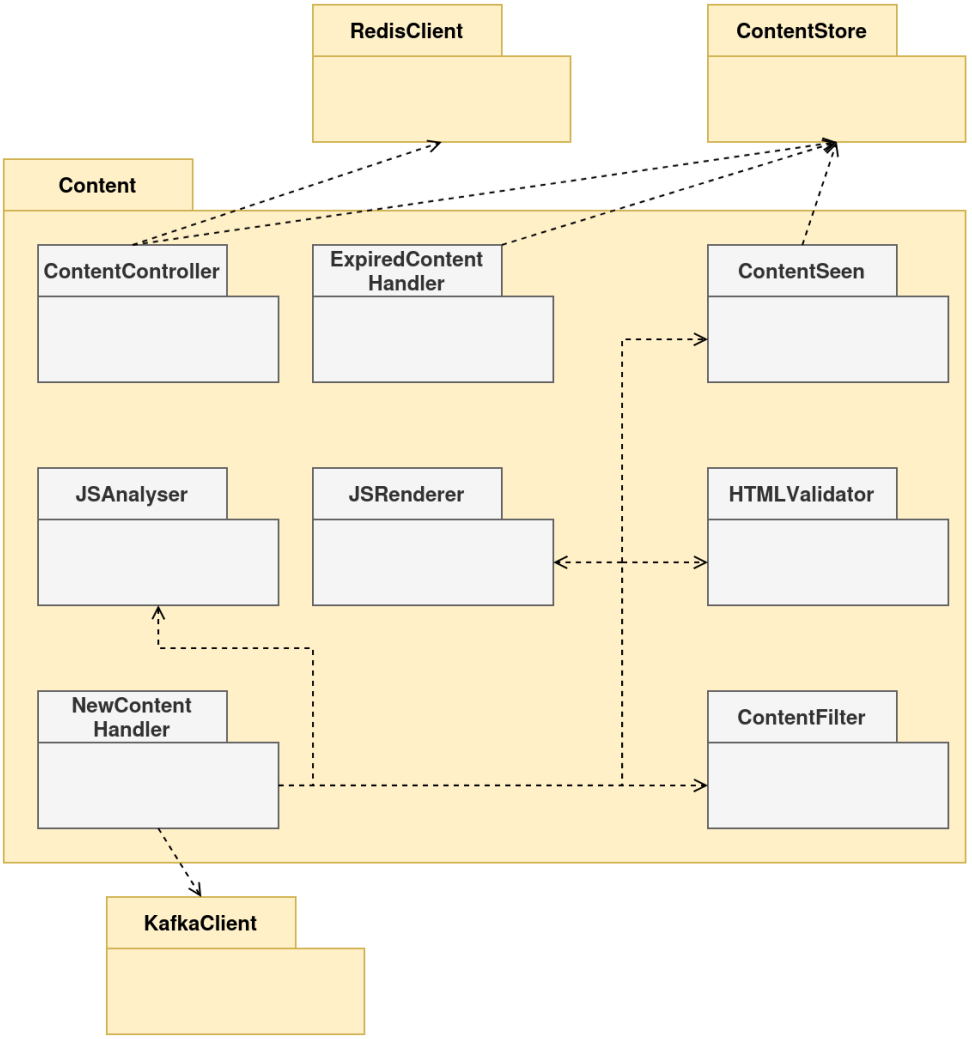
We introduce a new module - the **JSAnalyser**, that analyses javascript before it is rendered by the JSRenderer. The scripts that match patterns for viruses or other malicious content are not rendered, and the crawler resorts to the plain HTML/CSS on these pages to extract links.

View Sketches:

Content Module Decomposition View:



Content Module Uses View



Round 7 - Extensibility

In this round, we wish to address the possible changes that the system's stakeholders might require. The goal is for these changes to be as cheap (both in time and money) as possible, and so we design an architecture that is highly modifiable.

Iteration 1 - Modifying the Content Types

In this iteration, we will address the possible change in content types to be stored/parsed. We will use PDF files as an example, but any other content such as images and videos will follow a similar process.

Architectural Drivers:

M1

Design Concepts:

We utilize the *Split Module, Encapsulate* and *Abstract Common Services* modifiability tactics by refactoring the modules of our system to better adjust to new or altered requirements.

Elements of the System to refine:

The Content module; The URL module;

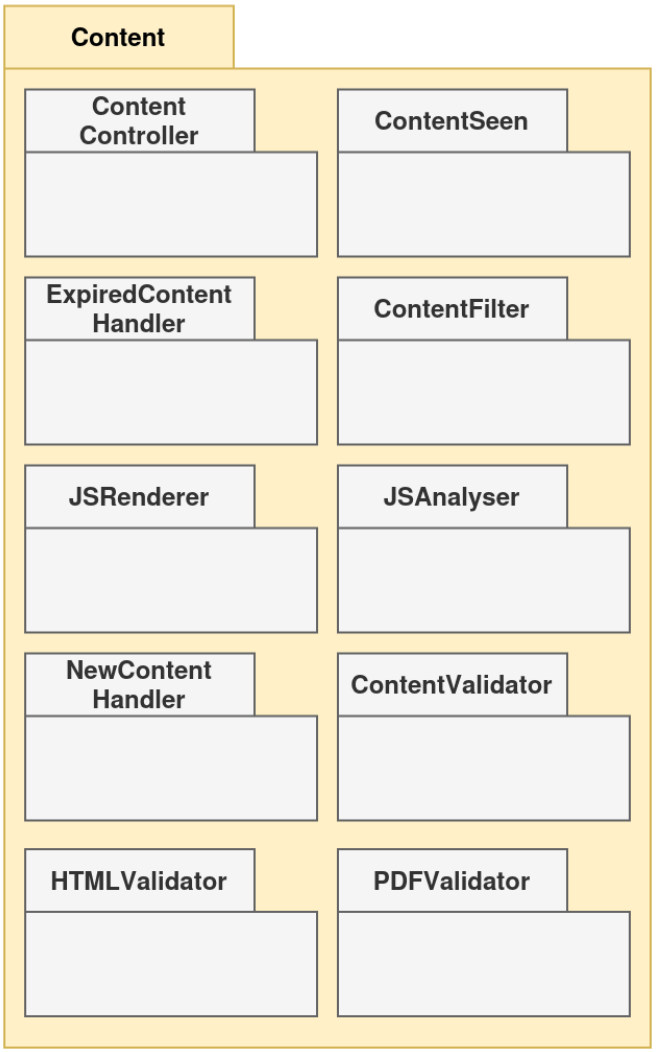
Architectural Elements, Responsibilities and Interfaces:

In order for the crawler to support PDF files, we must first define what “support” actually means. Downloading a HTML file isn’t different from downloading a PDF. However, the process of extracting links might suffer some modifications. The process of validating the content might need changing as well. For the purposes of this system, we will assume that we wish to be able to treat PDF pages almost as if they were HTML - we want to download them, validate them, extract links from them and store them in our ContentStorage.

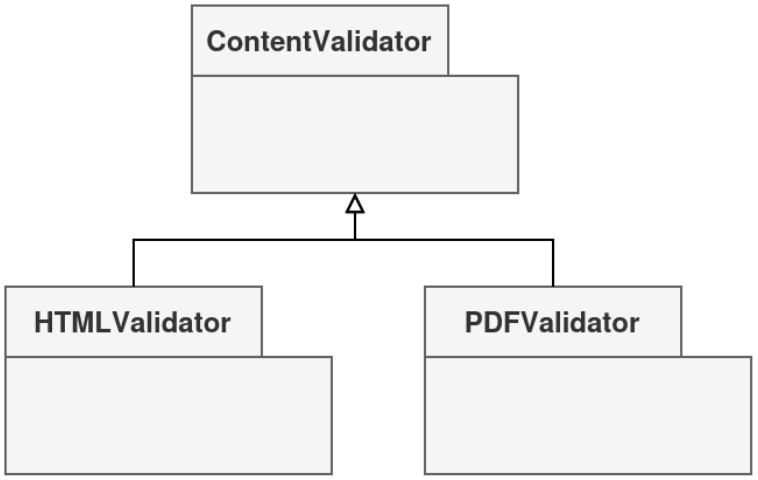
Within our Content module, the HTMLValidator sub-module now “is a” new **ContentValidator** module. We add the **PDFValidator** as another specialization of ContentValidator to handle the validation of PDF files.
Within the URL module, we can do the same for the LinkExtractor, i.e, we now specialize it into two sub-modules - **HTMLinkExtractor** and **PDFLinkExtractor**.

View Sketches:

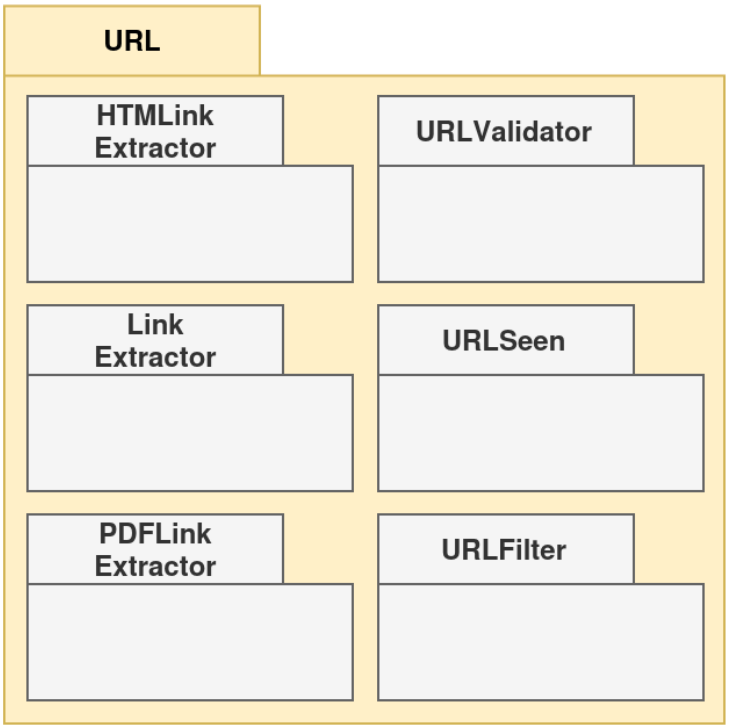
Content Module Decomposition View



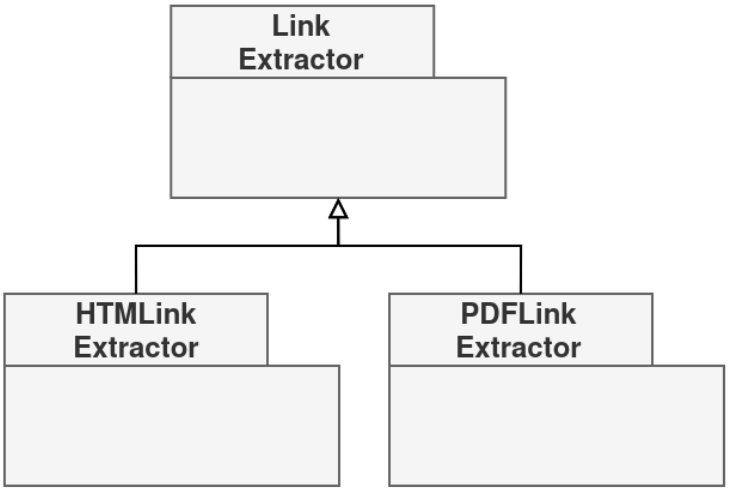
Content Module Generalization View



URL Module Decomposition View



URL Module Generalization View



Iteration 2 - Integrating a new Web Monitor Client

In this iteration, we wish for our system to be easily modifiable at Design Time to support a change in the client application, in this case, we want to support a Web Monitor, i.e, software that monitors the web for specific types of content, such as copyright infringements.

Architectural Drivers:

I2

Design Concepts:

We apply the *Split Module* modifiability tactic.

Elements of the System to refine:

The Content module; The system's components;

Architectural Elements, Responsibilities and Interfaces:

To support detection of copyright infringements, we would modify our system so that, when a page is downloaded, its content is checked against copyright protected data. If the system detects that the page is violating copyright, it stores it. Otherwise, it proceeds with crawling the web for additional pages. We can do this by simply adding a new module, called **WebMonitor**. This module would be called by the NewContentHandler and the logic would proceed from there. To store the data used for detecting infringements, we could use a new CopyrightProtected repository, a MongoDB instance.

Iteration 3 - Integrating a new Web Archiver Client

In this iteration, we wish for our system to be easily modifiable at Design Time to support a change in the client application, in this case, we want to support a Web Archiver, i.e, software that stores historical web page content for future access.

Architectural Drivers:

I3

Design Concepts:

We apply the *Increase Resources* tactic.

Elements of the System to refine:

The WebCrawlerContentStorage component; The ContentFilter module;

Architectural Elements, Responsibilities and Interfaces:

For integrating this new client, we need to store data for a much longer period than the current 5 years. Such a client might not want to store all pages, so the ContentFilter module can be modified to only accept relevant content (e.g, content from a certain host such as a library).

Round 8 - Availability

We have arrived at the last round of ADD. With the system fully functional, performant, resistant to attacks and modifiable, we now concern ourselves with the web crawler's availability.

Iteration 1

Architectural Drivers:

A1

Design Concepts:

We utilize the *Redundant Spare* availability tactic to guard the crawler against unexpected and uncontrollable failures (in this case, hardware).

Elements of the System to refine:

The whole system as a component;

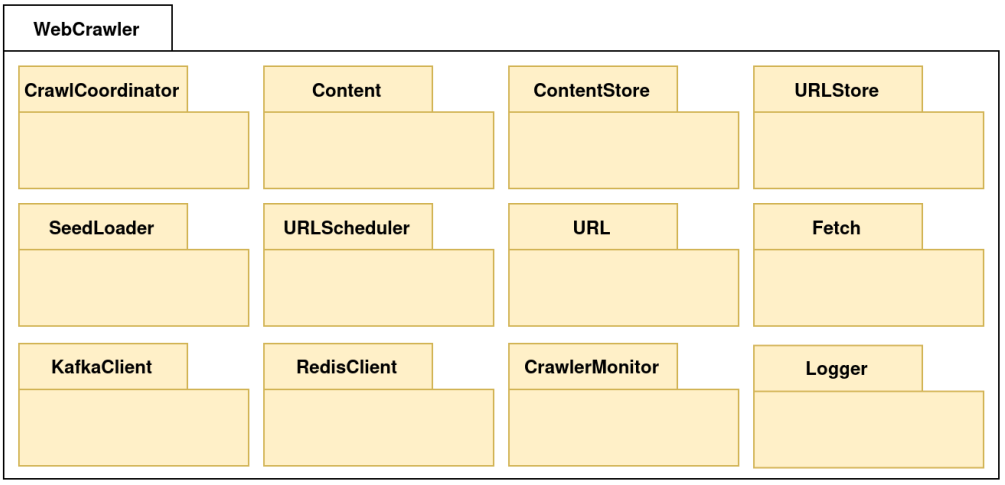
Architectural Elements, Responsibilities and Interfaces:

We wish for the system to be available 99,9% of the time, so, the combined downtime for all parts of our system should be lower than 0,1%.
Looking at the components of the crawler system, we wish to determine which don't match the required availability figures. The Kafka Cluster is running "by itself", and so are each of the queues within it. However, Kafka as a software already takes care of this, by replicating the queues in a fault-tolerant way. Within our web crawler, most of our components run as threads and are replicated more than once. This means that for the crawler to fail, all the Web Crawler components must fail, which is quite hard.
We have a few problems with the repositories. Within each crawler node, the DNSCache and RobotsCache might fail. In the event this happens, we just restart them - no need for a spare here, since these are only small performance optimizations. The caches will fill up again in a short time.
The real problem is in the MappingTable. If it fails, the URLFrontier will not know where to send URLs to, the FetchManager threads won't know what queue to read from… Therefore, this repository must be replicated, preferably using active spares (PAXOS or other consensus protocols). Our URL storage also isn't replicated, which might cause some issues. For this, a Warm Spare is a great solution. Losing the most recent URLs crawled isn't

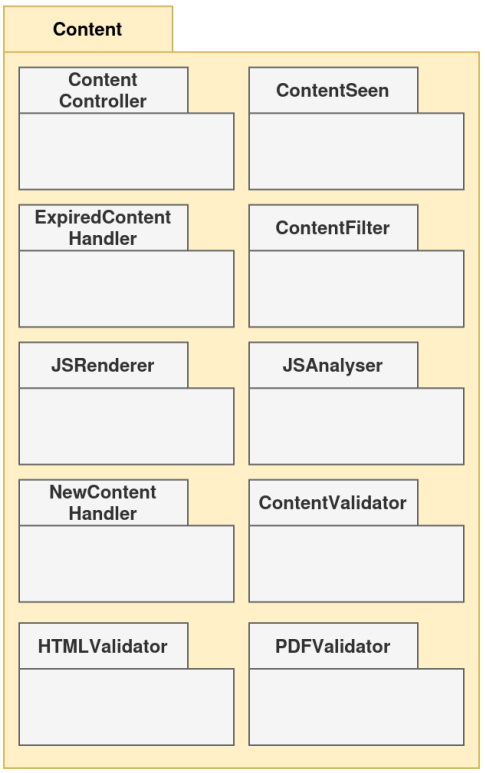
a disaster - losing valuable computational power and time replicating the storage across multiple nodes in an active manner is.
The remaining components that only have 1 instance, or 1 instance per node running, can mostly all be restarted via a cold spare, or even from no spare at all (for stateless components).

Final Views

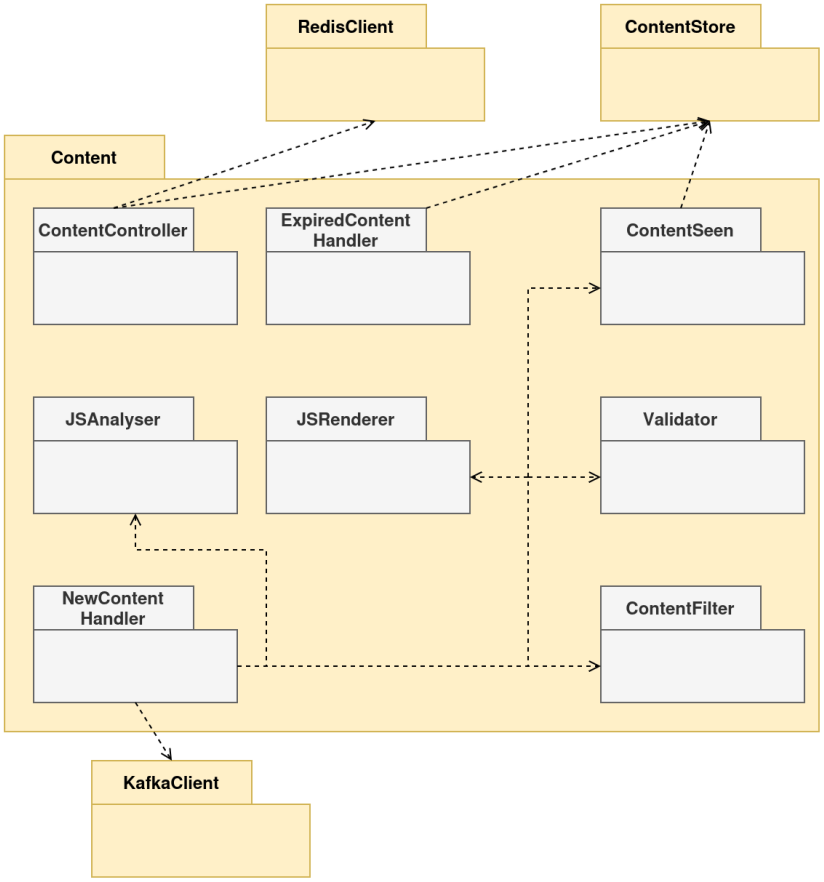
Top Level Decomposition View:



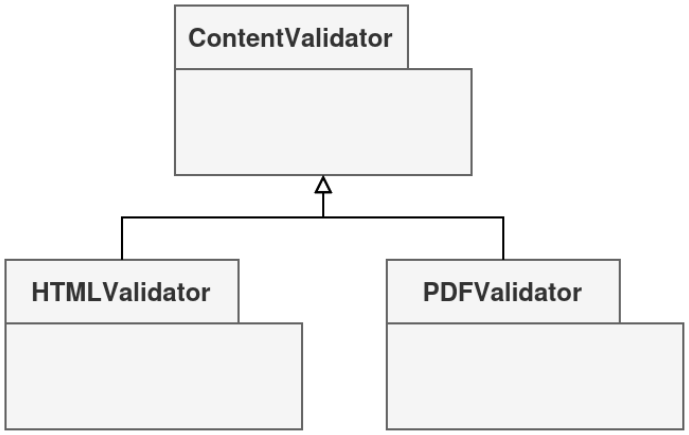
Content Module Decomposition View:



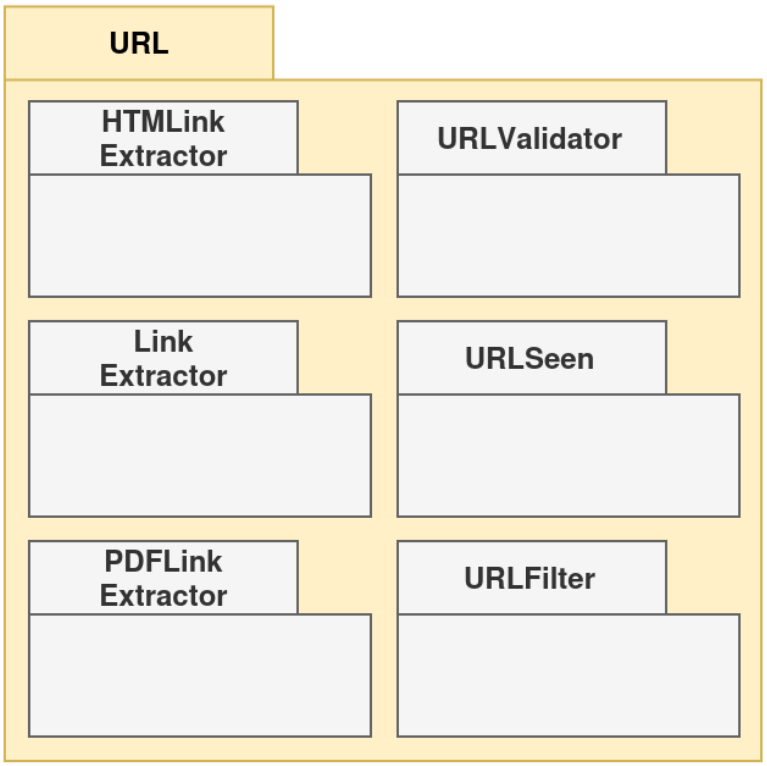
Content Module Uses View



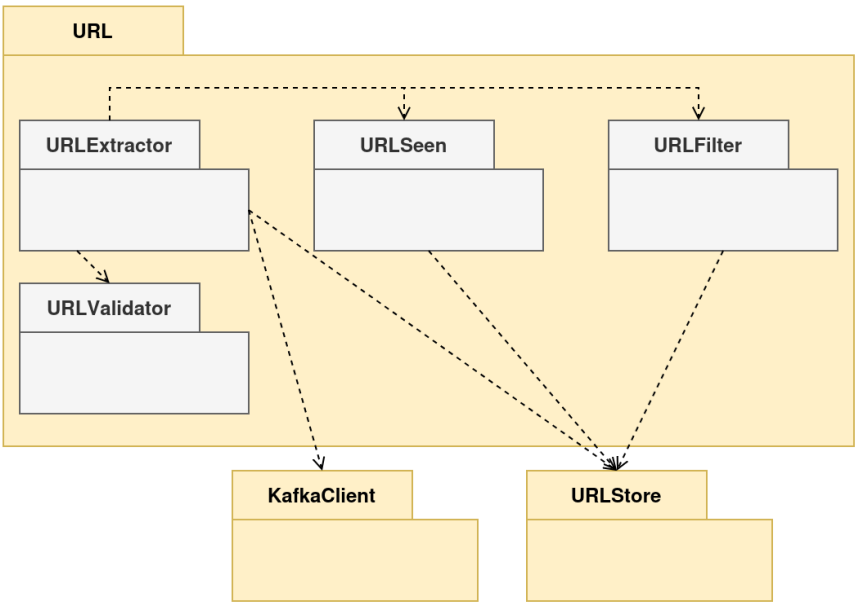
Content Module Generalization View



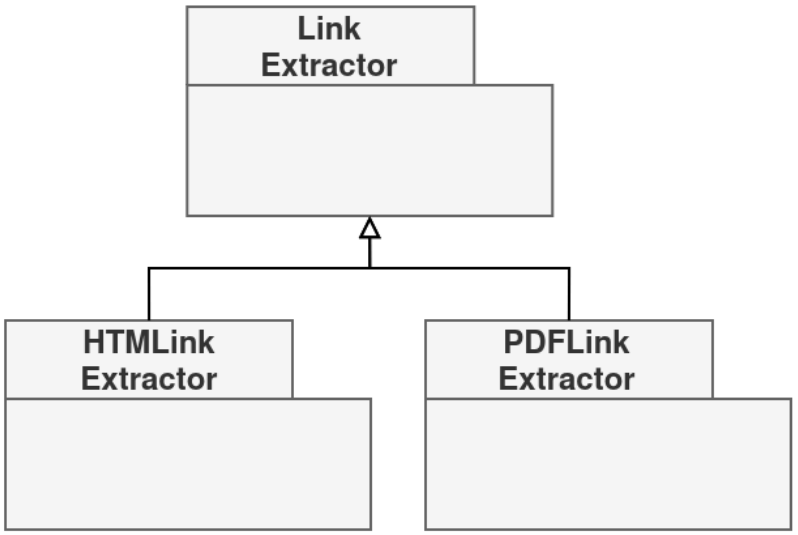
URL Module Decomposition View



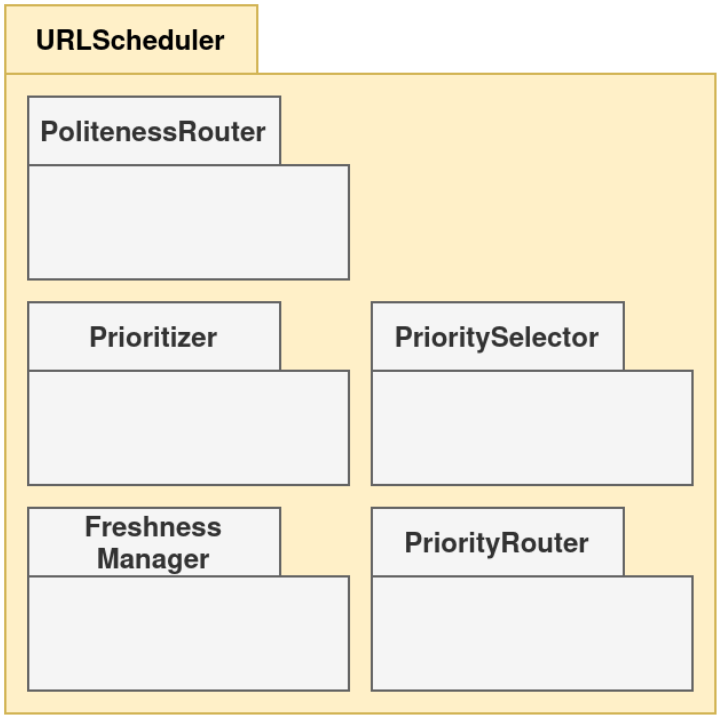
URL Module Uses View



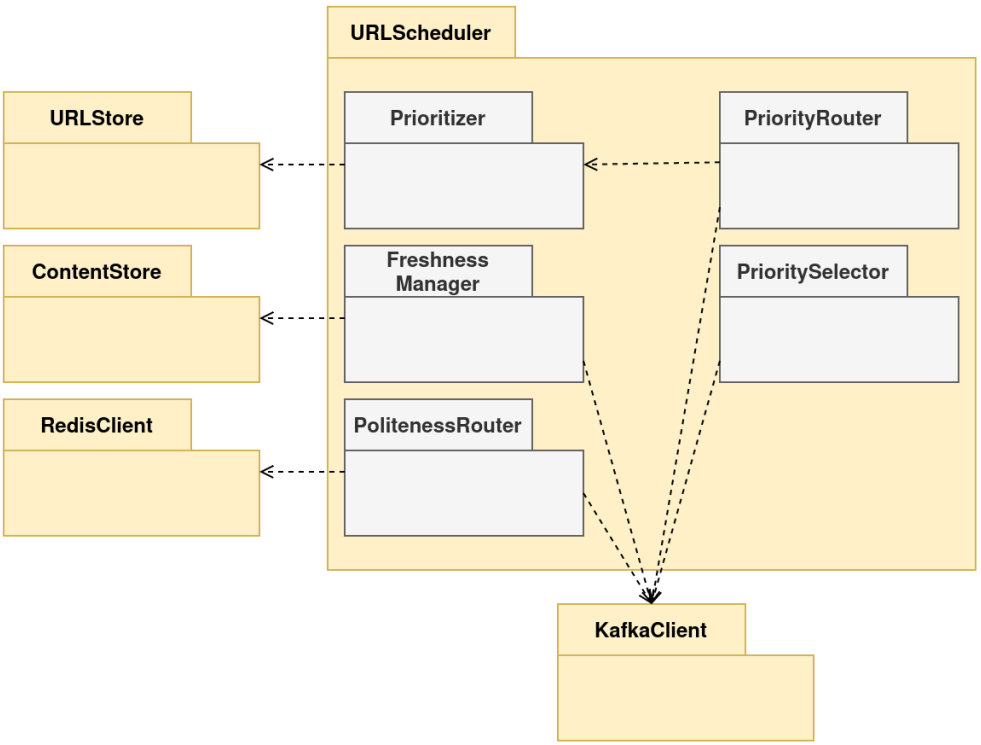
URL Module Generalization View



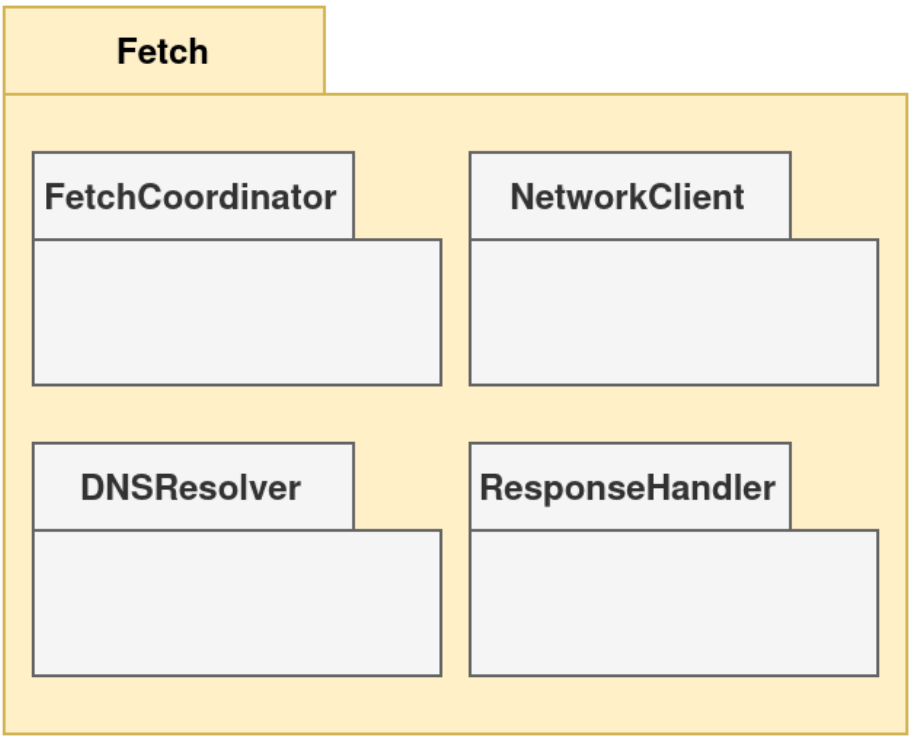
URLScheduler Module Decomposition View



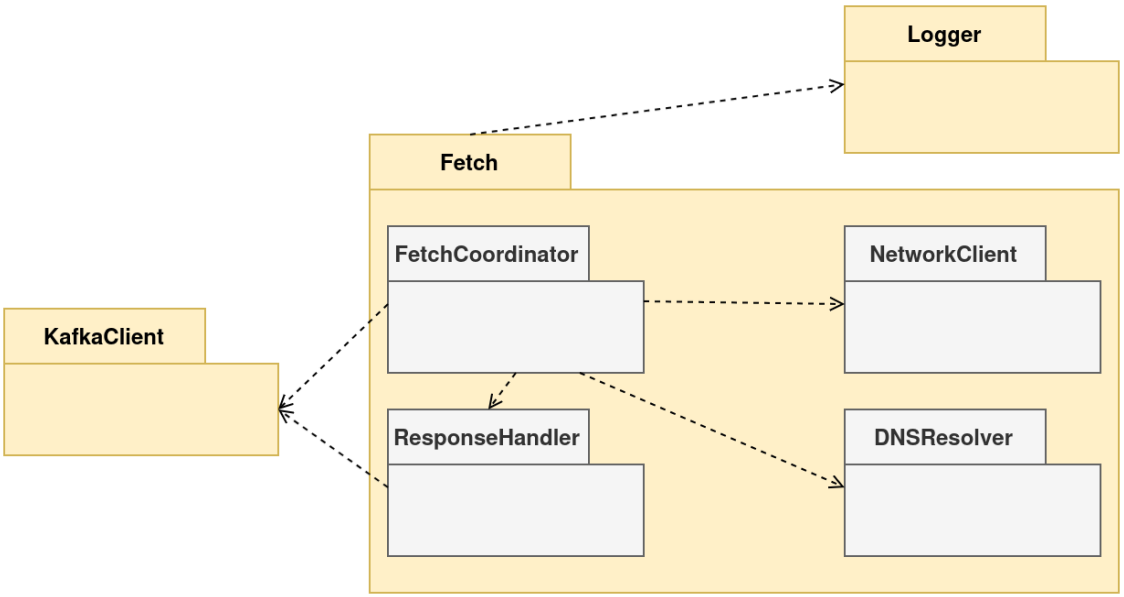
URLScheduler Module Uses View



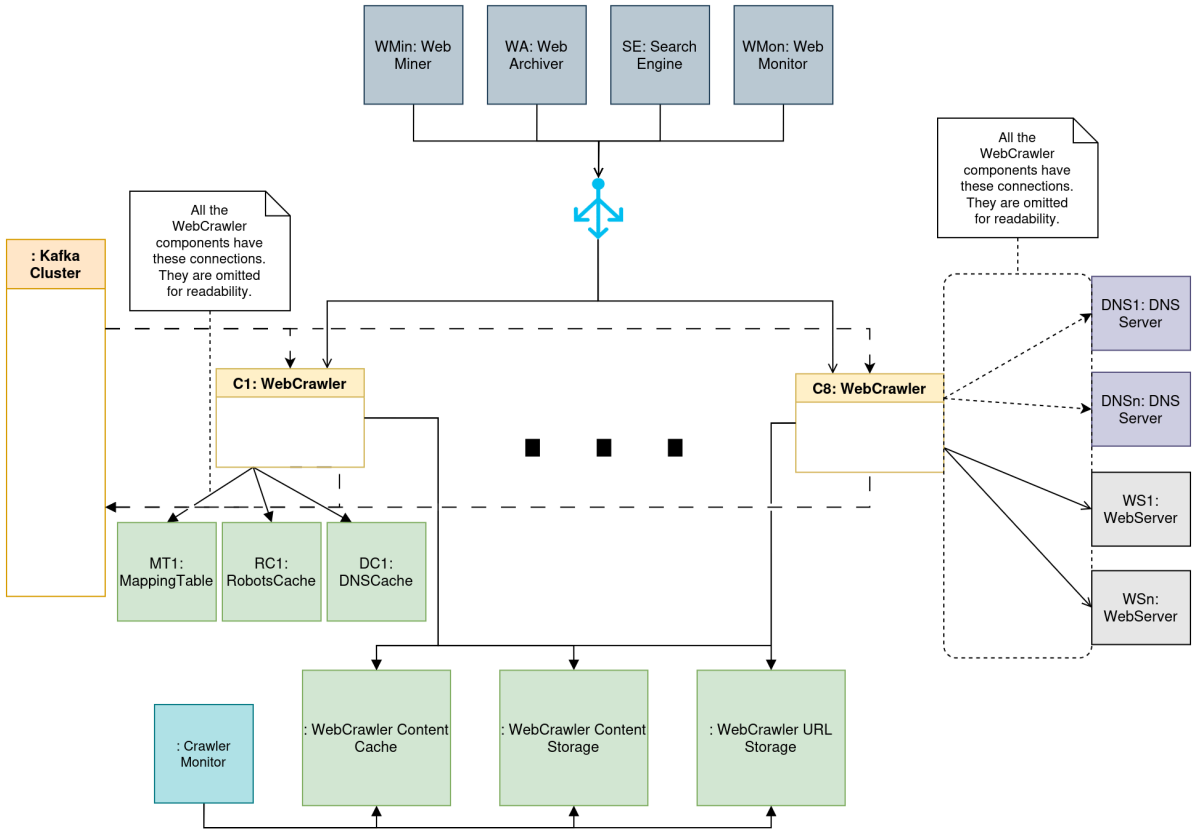
Fetch Module Decomposition View

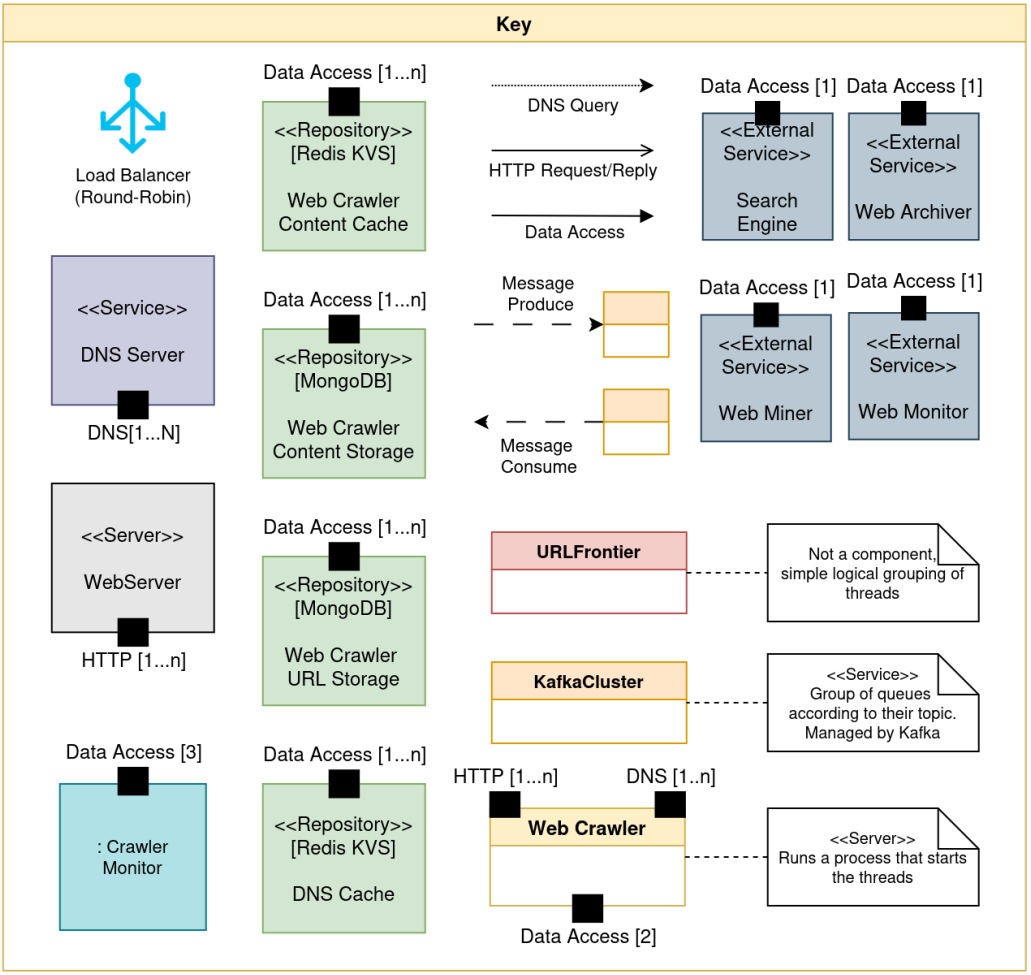


Fetch Module Uses View

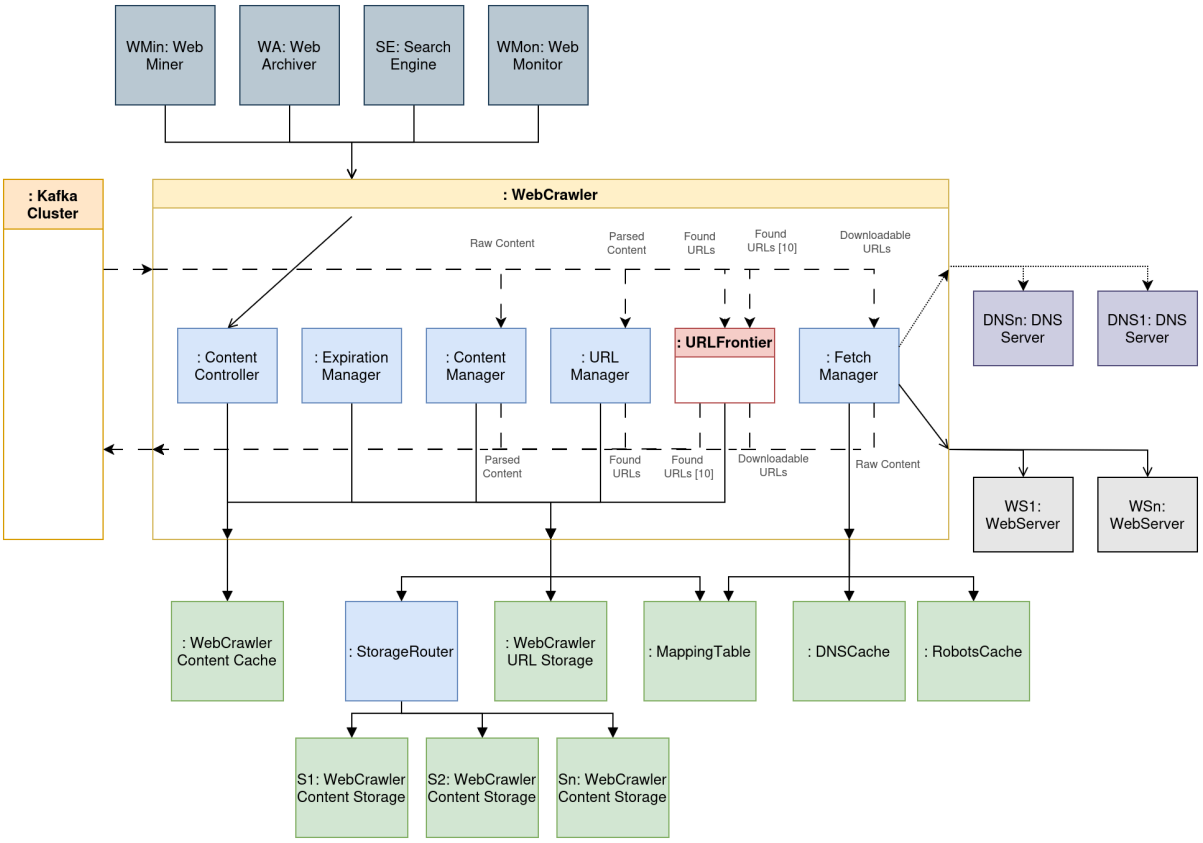


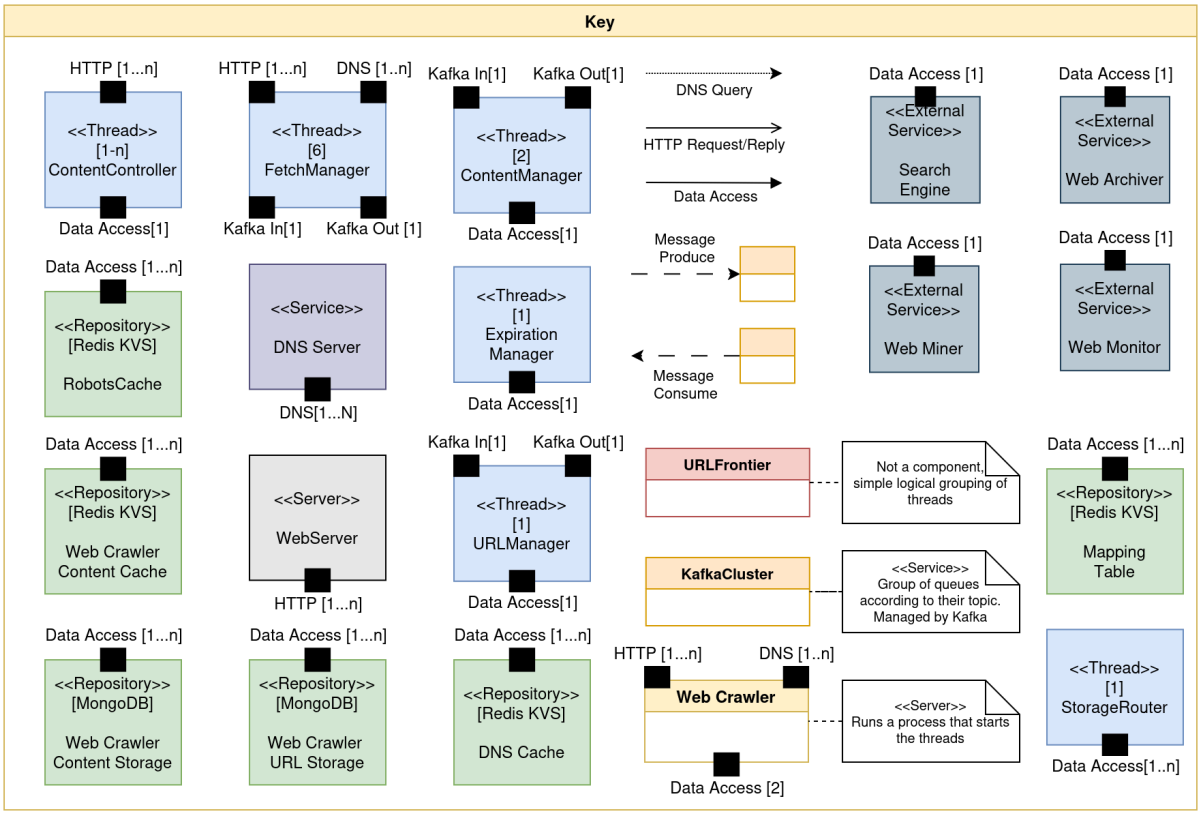
Top Level C&C View




Key
Load Balancer
(Round-Robin)
<<Service>>
DNS Server
DNS[1...N]
<<Server>>
WebServer
HTTP [1...n]
Data Access [3]
: Crawler
Monitor
Data Access [1...n]
<<Repository>>
[Redis KVS]
Web Crawler
Content Cache
Data Access [1...n]
<<Repository>>
[MongoDB]
Web Crawler
Content Storage
Data Access [1...n]
<<Repository>>
[MongoDB]
Web Crawler
URL Storage
Data Access [1...n]
<<Repository>>
[Redis KVS]
DNS Cache
DNS Query
HTTP Request/Reply
Data Access
Message
Produce
Message
Consume
URLFrontier
KafkaCluster
HTTP [1...n]
DNS [1..n]
Web Crawler
Data Access [2]
Data Access [1]
<<External
Service>>
Search
Engine
Data Access [1]
<<External
Service>>
Web Archiver
Data Access [1]
<<External
Service>>
Web Miner
Data Access [1]
<<External
Service>>
Web Monitor
Not a component,
simple logical grouping of
threads
<<Service>>
Group of queues
according to their topic.
Managed by Kafka
<<Server>>
Runs a process that starts
the threads

Web Crawler C&C View





Detailed URLFrontier and KafkaCluster View