Contents

1 Introduction

1.1 Background

Applications are frequently required to communicate with external services e.g. Facebook, Youtube, etc. Unfortunately, every external service specifies their own API and communication protocols. Therefore it is difficult to write applications that use many external services because a developer should write the communication layer for every external service.

The problem can be defined as following: There are a lot applications written for different operating systems and they require frequent communication with many external services. What architecture is the most appropriate for communication between applications and external services?

One of the solutions, is a simple direct communication between client and services. The client will send requests to every service and wait for responses. This solution has several drawbacks: a developer should maintain many communication layers written on different languages. When the external API changes, the developer should rewrite every communication layer. This increase the cost of development and the development time. Another problem is security, some services might have a private API that should not be used directly.

The different solution will be to introduce the middle layer between the client and external services. The client will communicate with middle layer using predefined protocol, middle layer will gather data from external services and send it to the client. This approach has several benefits: every application will be written with standardised communication API, defined by the middle layer; if external API changes only middle layer should be rewritten; the response time can be decreased by caching the data in the middle layer.

Another solution can be the usage of CDN (Content Delivery Networks). The application will send the requests directly to the external services, but the request will be intercepted and handled by the CDN Edge Servers that are located between Application and External Services. This solution requires dynamic usrage of HTTP Cache-Control headers. The advantage of using CDN is that the developer should not maintian the middleware service, the results will be cached somewhere on the way between Application and External services, therefore the response to the final user will be faster.

On the other hand, the drawbacks are: it is difficult to configure CDN to process dynamic content, the developer needs to write the communication layer for every external service.

The project tries to break the connection between applications and external services by providing the middle layer. The applications will communicate with the middleware using predefined format that is the same for every platform. The middleware will manage the requests, translate them to the format appropriate for the external services and forward them to the appropriate servers. It will also manage the responses from servers and send them back to the applications.

1.2 Project Objectives

The project in the context of networking, caching and data aggregation. The goal of the project is to investigate different solutions for implementing a caching layer used in a backend middleware, design and develop prototypes for each solution, run set of tests and select the most optimal solution. The selected solution is then further developed and integrated into an existing middleware provided by the company, and an analysis is then made in a real life scenario.

1.3 Thesis Outline

During the project the company solution will be examined and several solutions will be introduced.

The Theory section will give information about modern web caching technologies and how they can be used.

The Architecture overview will present the company solution and will identify the drawbacks.

2 Theory

2.1 Content Delivery Networks

Nowadays the Internet is available almost from any place in the world. It allows people from different countries to communicate with each other and exchange information. The content providers and applications should serve the user's requests from all around the world with the equally high speed in order to provide good user experience. For solving this task, companies should deploy their servers all around the world in multiple data centers. For world-class companies like Google and Netflix this task is solvable[9], but for middle sized companies there should be another solution because it is very expensive and sometimes complicated to deploy servers in multiple data centers. Fortunately, the content delivery networks solve this problem[6].

The Content Delivery Network(CDN) is a distributed system consisted of many servers deployed all around the world in different data centers. They act as a local content holders. The routes to the content providers and applications are configured to lay through the CDN's servers. When the client wants to access the server through the Internet the request will be processed by the local CDN server and if it will find the data locally, it will deliver the response to the client without making the request to the content provider that can be located in different country. The CDNs can store both static and dynamic content. One of the first global solutions was presented by Akamai company[6]. The graphical architecture is depicted on figure 1.

The CDN consists of several parts: routing, load balancing and web caching[7]. The web caching part will be reviewed because it is relevant to the project.

2.2 Web Caching

Web caching is the technique that allows to store temporary content that is requested from the Internet e.g. HTML pages, JSON, XML or CSS files. It alleviates the server's work by reducing the amount of requests to it and reduces bandwidth usage[4]. A web cache system serves as a communication point between client and sevrer. The client's requests and server's responses are routed through it. The web cache stores responses from the server and returns them without hitting the underlying server. It also can manipulate

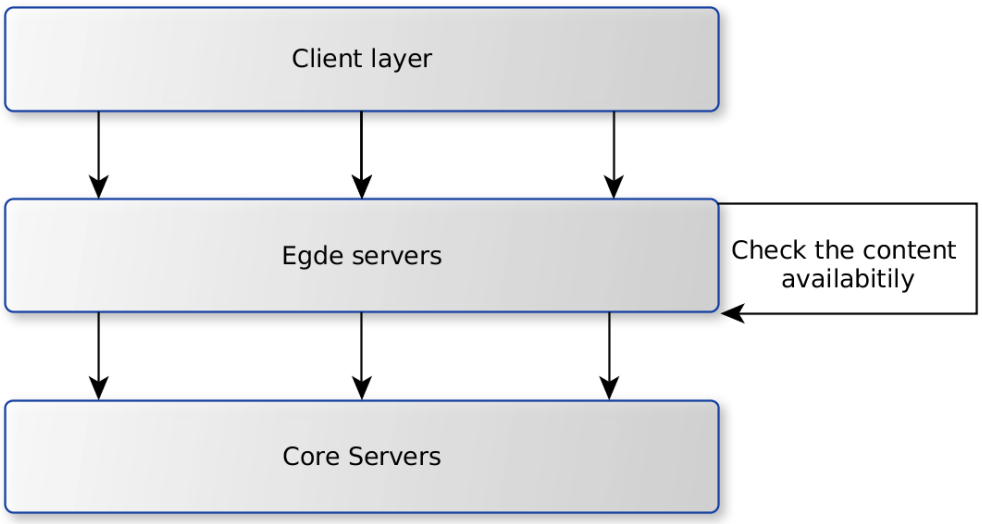

Figure 1: Content Delivery Network Overview

the request/response headers.

The benefits of web caching:

- Reduces the server workload. Using web caching the requests will go through the cache and will touch server only if the data does not present in caches or the data is stale. That will reduce the amount of requests made to server.
- Web caching improves user experience. The data will be delivered faster to the end users.
- Reduces bandwidth

2.2.1 Content types

The content stored by web caches can be static or dynamic.

The static content is a set of resources that stay the same no matter what was the user input. Static objects are identified by the unique path and can be cached for a long time by the web caches.

The dynamic content is generated at run-time, based on the user input[3]. The dynamic requests almost always processed by severs. They are the main consumers of the web server's resources. The dynamic requests are time dependant meaning that in different time the same request can produce different output, as a result they cannot be cached for a long time by the web caches. Moreover, for security reasons, usually dynamic objects that contain client's personal data and preferences are configured in order not to be cached by the external web caches and CDNs.

However almost all dynamic content is static for a short period of time. It changes only when the internal resouces e.g. databases are altered. This gives a possibility to configure web cache for storing dynamic objects.

2.2.2 Http Cache Control Headers

Web cache is controlled by the http cache control headers[8]. The control mechanisms can be specified on both request and response sides. There are several http headers that are relevant for the project:

- Cache-Control
- Vary header
- Etag, If-None-Match
- Last-Modified, If-Modified-Since
- Expires(for http 1.0)
- Widely used extension headers, like X-Cache.

There are two caching techniques: Time based caching and Data based caching.

The time based caching is represented by the next http headers: *Last-Modified* and *If-Modified-Since*, *Cache-Control* with *s-maxage* parameter and *Expires* header. Client sends the request with specified *If-Modified-Since* header, where he indicates as a parameter the time when the content was modified. The server will check the type of the content which was requested and send the corresponding response. If the static content was requested, the server will check if the resource was changed since the date specified by the client in *If-Modified-Since* header, and will send the new

content with the code 200 and updated *Last-Modified* header if it was modified, otherwise it will reply with the code 304(resouce not modified) without content and with old *Last-Modified* header. The server usually does not set the Last-Modified header to the dynamic resources because it is hard to know where exactly they were modified. During the response the server can specify the *Cache-Control* header with s-maxage parameter. In this case, if the content is public, it can be cached by the web cache servers for s-maxage time specified in seconds. If the next request will occur in the next *s-maxage* seconds, the content will be served from the web cache. This technique is used for caching both dynamic and static content. The difference between static and dynamic content is that the s-maxage parameter fot the dynamic content is set dynamically, and it is static for the static content.(change the last sentence)

The data based caching is represented by the *Etag* and *If-None-Match* headers. It is supported since in HTTP/1.1. On the first request the server will compute the hash of the content and send it to the client in the Etag header. The client will remember the hash value in the *If-None-Match* header. When the next request occurs, the server will recompute the hash of the content and will compare it with the value specified in the If-None-Math parameter. If the values are the same, the server will reply with the 304 code, without content and with the old Etag header, otherwise it will change the Etag header to the new value and send the normal response.

The web cache can be implemented on:

- Client side, by using browser caches
- Proxy servers, by introducing the middle caching server between client and server.
- Server side, by implementing cache programmatically

Currently, the middleware server supports the server side caching with Redis in memory data store. We will introduce the proxy server caching and will see if it will give the benefit to the architecture.(move to abstract)

2.2.3 An overview of proxy server types

A proxy server is a server that is deployed between client and server. It serves as a middle point in communication between clients and servers. The

proxy server redirects requests to servers and responses from servres. It can improve the performance of the servers by storing the copies of frequently used resources. When a client makes a request to the server through the proxy server, the proxy server serves as a web cache, it will try to find data locally and will return the resource back to the client on success.

There are two main types of proxy servers: forward and reversed proxy[2]. A forward proxy is a one of the most common types of proxy servers. The client is aware about the proxy server and can configure requests through it. A reversed proxy is deployed by server administrators in the internal network. The client contacts the desired server, but the request is routed through the reversed proxy server. In this case, the client may not know about the underlying proxy. The reversed proxy server was selected for the project because it perfectly suits the architecture, the client should not know about the existence of the middle point. (and the web caching is mostly done by using reversed proxy servers).

2.3 Http Session Management

HTTP protocol is stateless by its nature, meaning that there is no possibility to distinguish one request from another. Http requests usually open new connection to the desired server every time. Nowadays, server can specify *Keep-Alive* header in the response in order to give browser a hint that this connection can be used again for the new request. Unfortunately, without transferring user-specific information it is impossible to distinguish users.

The session management is implemented through header fields *Set-Cookie* and *Cookie* headers. When server wants to distinguish one user from another it sets the unique identifies for the user. This identifier is transferred in the header field Set-Cookie. The browser will parse this field and remember the unique identifier. For every new request, browser will send this identifier it header field Cookie. Of course server can send additional information in Set-Cookie header, that is unique for user. Unfortunately, it is very dangerous to transfer private information(e.g. credit card number) this way. The server can store dictionary of user ids and corresponding private information in memory and retrieve this information every time when the browser specifies Cookie header.

3 Current company's solution

3.1 Server Architecture

The company architecture contains the following components: Client Application, Middleware server, Middleware Cache, Metadata Server and Content Servers. The brief architecture overview is presented on figure 2.

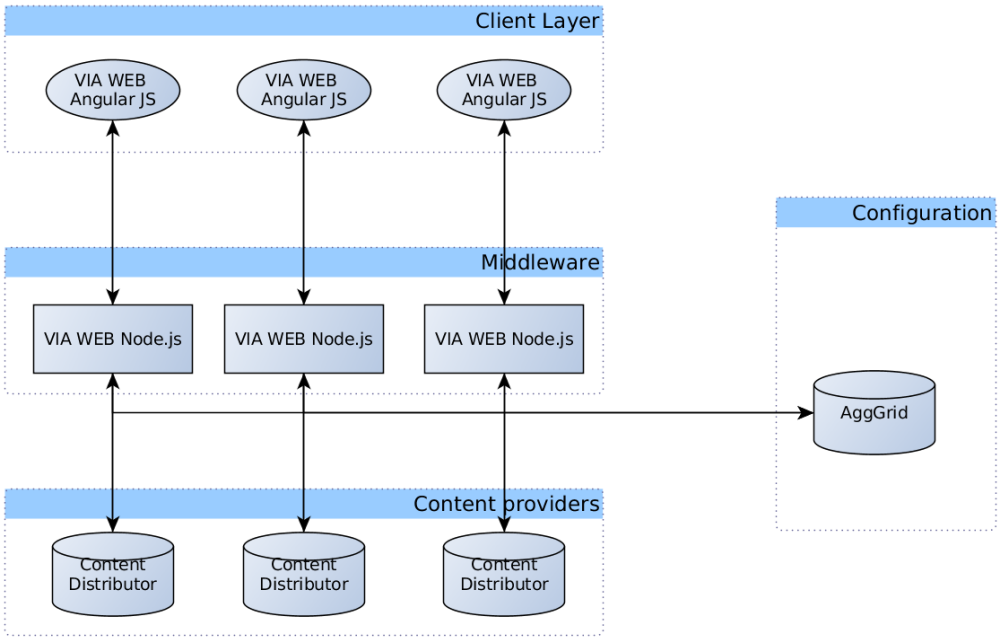

Figure 2: Global architecture overveiw

The client is a web application developed using Javascript, HTML, CSS framework and Model View Controller pattern. It communicates with middleware server through REST services based on HTTP protocol. The client has several components: Controllers, Managers, Services.

The contollers validate the user data, invoke corresponding managers and render data to the view objects represented by HTML pages. The managers are implemented using Facade pattern[**?**]. They hold and manage services, construct View Model Objects from service responses and send them back to controllers.

The services represent the communication layer between the client application and the middleware server. They communicate via REST services

based on HTTP protocol. This brings flexibility to the architecture and makes components loosely coupled.[describe why it is good?].

The middleware server serves as a transparent layer between client and inner servers. The main goal of this server is to translate requests from clients to the format understandable by the metadata server or content servers.

Content servers are applications provided by customers. They are the main sources of information that needs to be presented by the client application. They can be represented by the movie entities or music distributors.

The metadata server is the company developed application that stores and processes auxiliary and configurational information. The requests can be:

- Data gathering - required fo analytical purposes
- User action logs
- User settings - specific user settings, for example watch history for movie entities.
- Client and midldeware configuration

The client sends requests to the middleware server. The middleware server handles client requests and sends back appropriate responses. The response can be one of two types: Configurational or Data demand. If response is configurational the middleware server redirects it to the Metadata server, otherwise it redirects the request to the Content server. The middleware supports local cache and caches every response that is not assosiated with user session. The message exchanging between architecture components is depicted on figure 3.

3.2 Middleware server architecture

The middleware server is written on server side Javascript language, using asynchronous server Node.js[link?]. The server is developed using Model View Controller (MVC) pattern and communicates with other components through REST services. As a result, the server components are loosely coupled with each other, that gives great flexibility in changing and replacing components and simplifies testing.

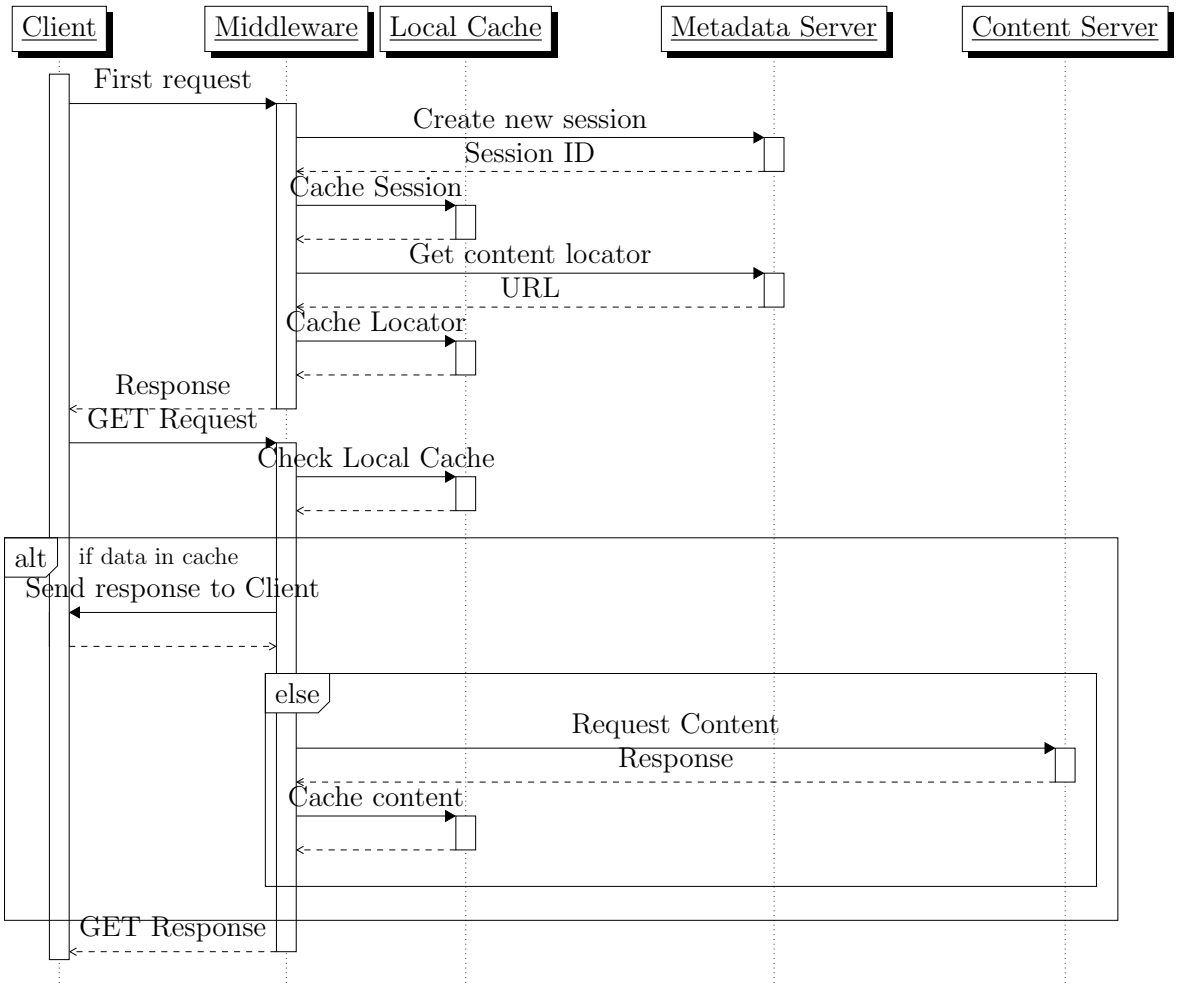

Figure 3: Sequence Diagram of message exchanging

The middleware contains the following components: Controllers, Managers, Services, Configuration and Data Model Object builders. The interraction between components is presented on figure 4.

The controllers accept requests from clients. They validate user data and invoke corresponding managers.

The managers implement facade pattern[link]. They aggregate multiple services and redirect requests to them. They gather the data from services and build immutable Data Model Objects(DMOs). These DMOs are sent back to the client as responses. The workflow of managers is depicted on figure 5.

Services communicate with metadata and content servers. They aggregate the cache layer and react according to the following rules: They check if the data is located in the local cache. If service observes cache hit, it will check the object TTL[specify description] and send corresponding object

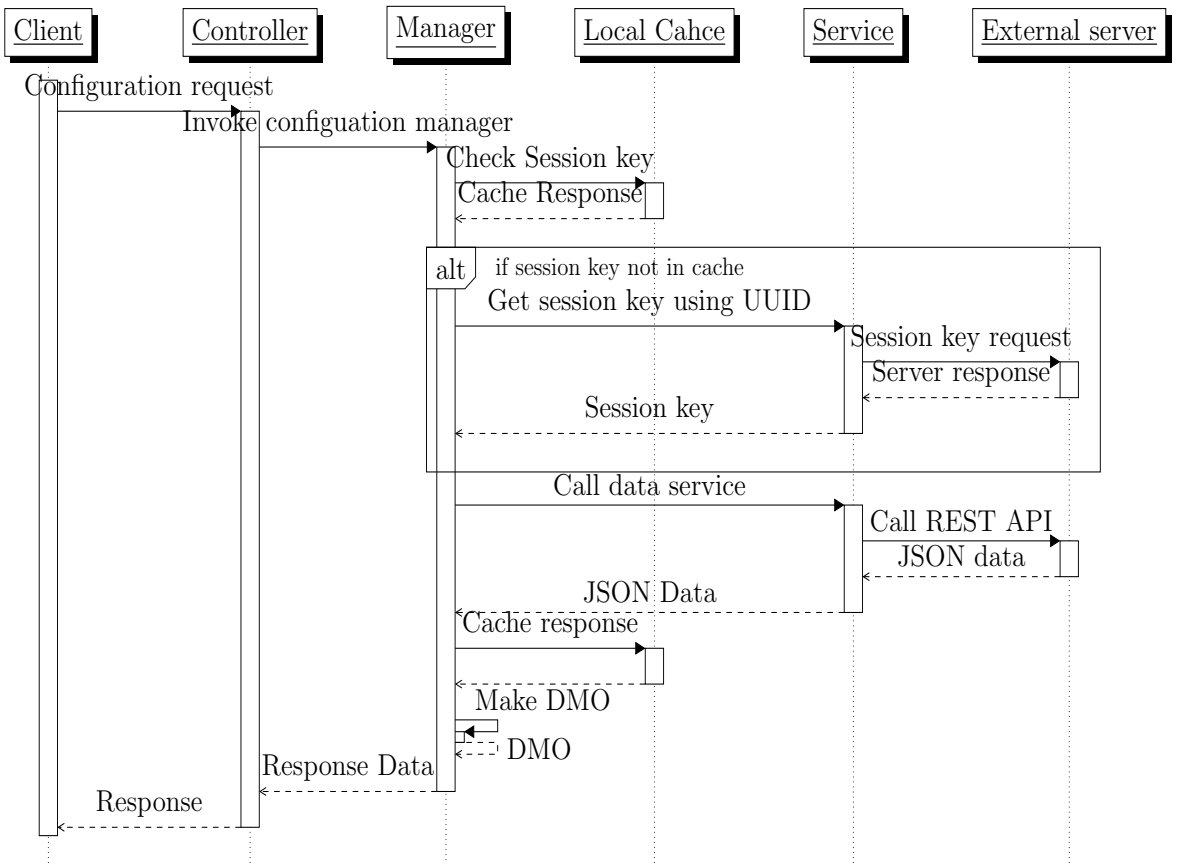

Figure 4: Sequence Diagram of Middleware server request process

back to the manager. On the other hand, if the cache miss occures, it will send the GET request through the REST protocol to the metadata server or content server, store the response locally for predefined period of time and send it back to the manager. The workflow of services is presented on figure 6.

The client can make two types of requests: configurational and data demand requests. The configurational request has the following purposes:

- Provides configuration parameters for both middleware server and client application
- Writes user activy
- Checks the health of metadata server
- Get Content Server url
- User settings and preferences

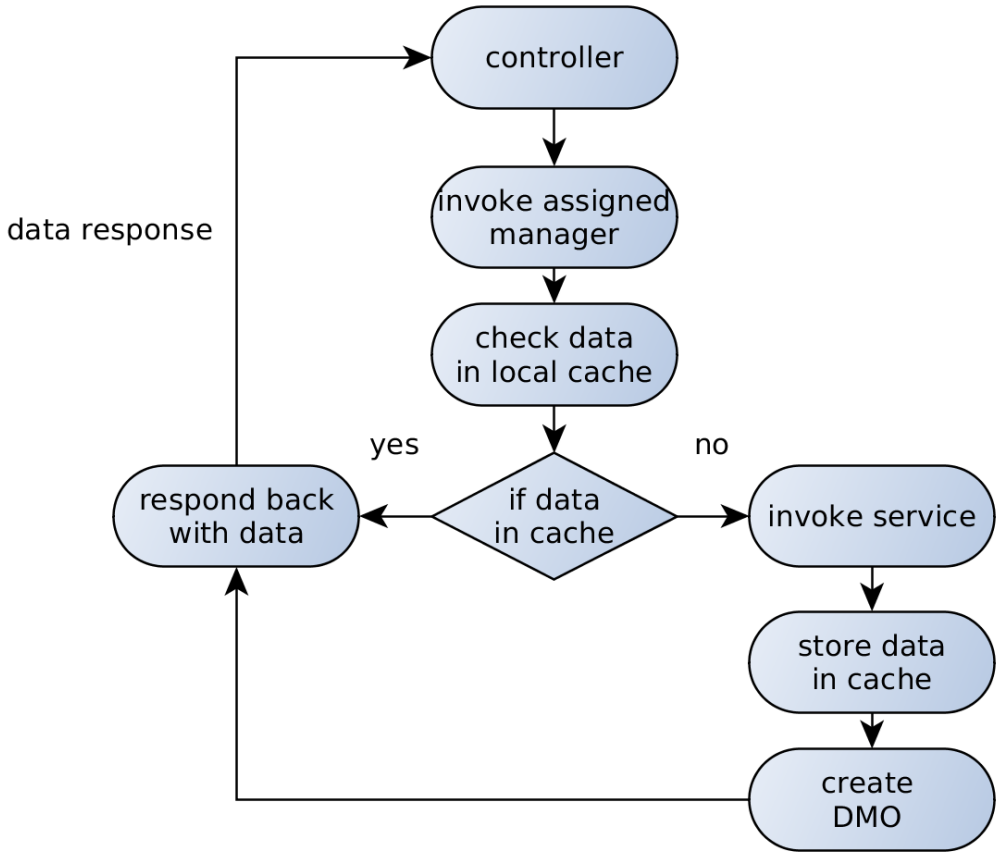

Figure 5: Manager workflow

- Analytics

The data demand request is a request to the content servers. Content servers are customer servers. It means that the response can have any structure and doesn't have predefined structure.

The controll flow can be described as following:

The controller recieves the request, validates user data and redirects it to the assigned manager. The manager invokes corresponding service. When the client makes the data demand request, the manager invokes content services that redirect request to the assigned content server. The response then is stored in the cache, translated into Data Model object(DMO) and transferred back to the client in JSON format. The builders are in charge for translating JSON data into Persistent Data Model objects. For every DMO there is an assigned manager and service. As an example, if the content server responses with video object, there will be VideoManager and VideoService components.

The client layer and middleware layer are loosely coupled and communicate with each other through REST services. It gives application great

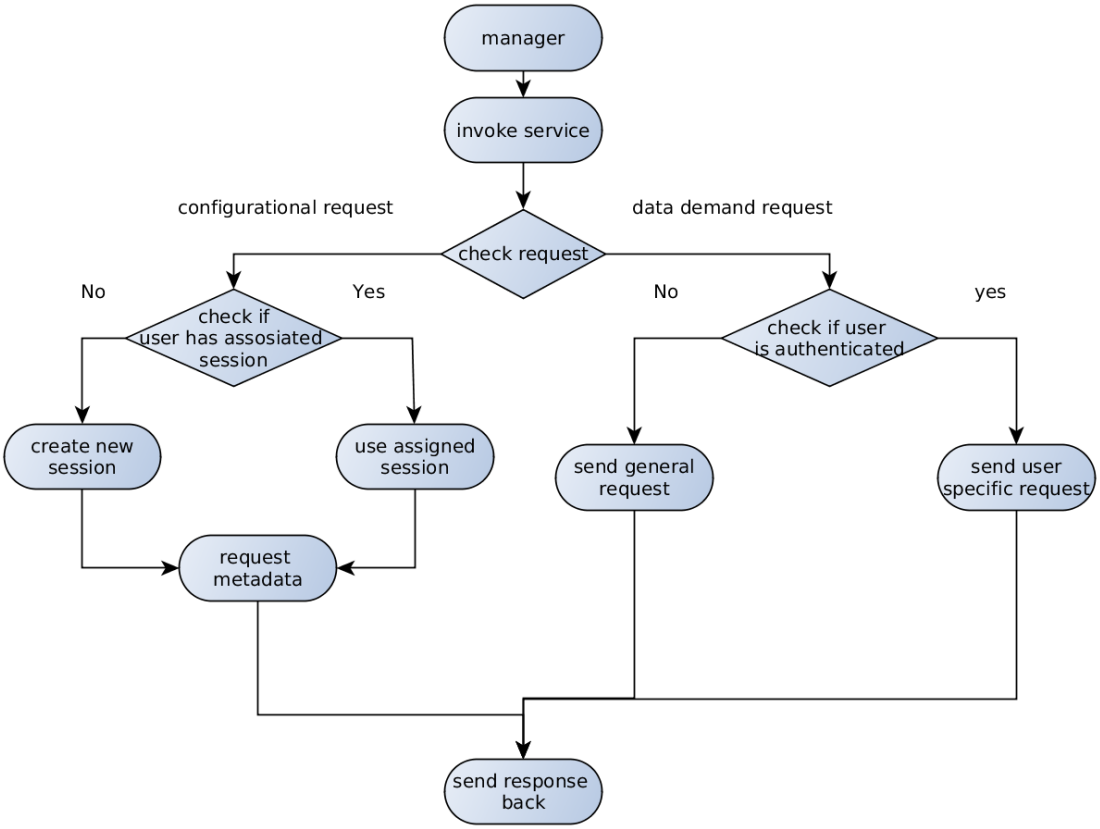

Figure 6: Service workflow

flexibility.

3.3 Session management

In order to process communication between middleware server and inner servers, the security layer was implemented. The security is implemented through the dynamic session management.

The session consists of two parts:

- The client session
- The middleware session

The client session is represented by the unique session identifier and the browres id. These parameters are generated by the middleware server and transferred to the client through the Set-Cookie header. The browser remembers the data and sends it back with every request in Cookie header.

The middleware session is generated when the the client makes the initial request. It contains the metadata session key and user session, if the user is

authenticated. The metadata session is obtain by making a request to the metadata server. The middleware server sends the aplication key parameter, which is specified in the configuration file and browser id. The metadata server checks the validation of the application key and generates new session for the middleware server. The middleware server assignes this session to the client and stores it locally in memory. Every time when the client will communicate with the middleware server it will send the client's session data. The server will find the metadata session associated with the client and retrieve the medata session. Using this session the middleware can make configurational requests to the metadata session. The sequence diagram of session management is presented on figure 7.

The purpose of sessions is to distinct users and provide corresponding analytics to the customers.

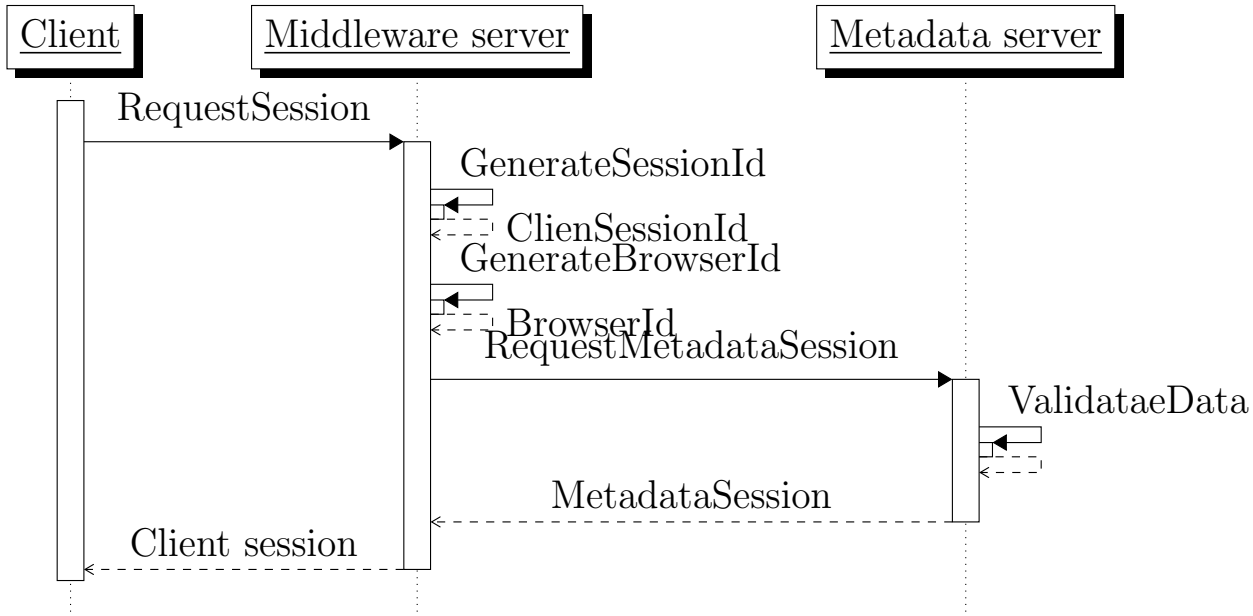

Figure 7: Sequence Diagram of Middleware Session generation

3.4 Model objects

The architecture uses the Data model objects, Video Model Objects and Application model objects. The purpose of the middleware server is to take the data from content servers, store it in Redis, adjust data from Metadata server if necessary and send back to the client application. The data from the content servers differs from one to another, as a result the structure for

representing content server objects should be generated dynamically. These objects are called data model objects(DMO). They represent the entities gathered from the content servers. It can be information about videos, music or any other entity. The DMO is generated by the DMO builders from JSON.

The middleware server generated DMOs and sends them to the client application. The client gatheres several DMOs and constructs View Model Object(VMO). This object is than presented to the users. As a result, the VMO is constructed on the client side. For every html page single VMO is constructed.

The Application model object(AMO) is the array of View Model Objects. The AMO represents the information and objects that can be fetched from the single content server.

3.5 Drawbacks of the current architecture

After careful examination, two categories of problems were defined: Client side drawbacks and Middleware side drawbacks.

In order to present the page, the client needs to generate a View Model Object(VMO). The VMO contains several Data Model objects(DMOs). The client makes request for every DMO, aggregates the response, generates VMO from DMOs and renders it to the html view. The drawback is that the client has to make several HTTP requests in order to generate single VMO. It would be better for client to make a request for VMO instead of DMO. This approach has several advantages: the client will make less HTTP requests, that will increase the performance by reducing the latency, it will simplify the client logic, because the client will not be requered to generate VMOs from DMOs.

Another problem with the current client implementation is that it is not generic. If the new content server will be introduced, a lot of code have to be changed on the client side in order to impelment the new logic.

The client can maintain the caching layer that will cache VMOs from the responses.

The client implements MVC pattern, which produces dublication with middleware server. This approach increases the complexity of the system, the developers should support boch client and middleware MVC applica-

tions. We can simplify client and assign two tasks to it: caching and rendering VMOs.

On the middleware side, the DMOs are not generic. The purpose of the middleware server is to serve as a transparent layer, but without dynamic DMO generation a lot of code has to be changed when the new content server is introduced.

The middleware cache can be replaced by the Content Delivery Network(CDN). The middleware caches only information that is common for every user. This work can be done by the CDN edge servers. These will decrease the middleware complexity, decrease the cost of maintaining middleware server.

4 Cache solutions

4.1 Solution

The figure 8 shows the amount of requests and time that browser needs to make to generate a single VMO and render a page. The browser needs to make more than 20 AJAX requests for a single page. The middleware was deployed on the local machine meaning that there is no latency bentween browser and middleware server. As can be seen, these are not optimistic numbers. The amount of requests is too high and computation have to be done on the both client and server sides. Several questions arises:

- Is Redis a good caching layer for this project?
- Can Redis be replaced by something else? Maybe it would be better to use the configurational cache, represented by the web caches and control it through the http cache control headers.

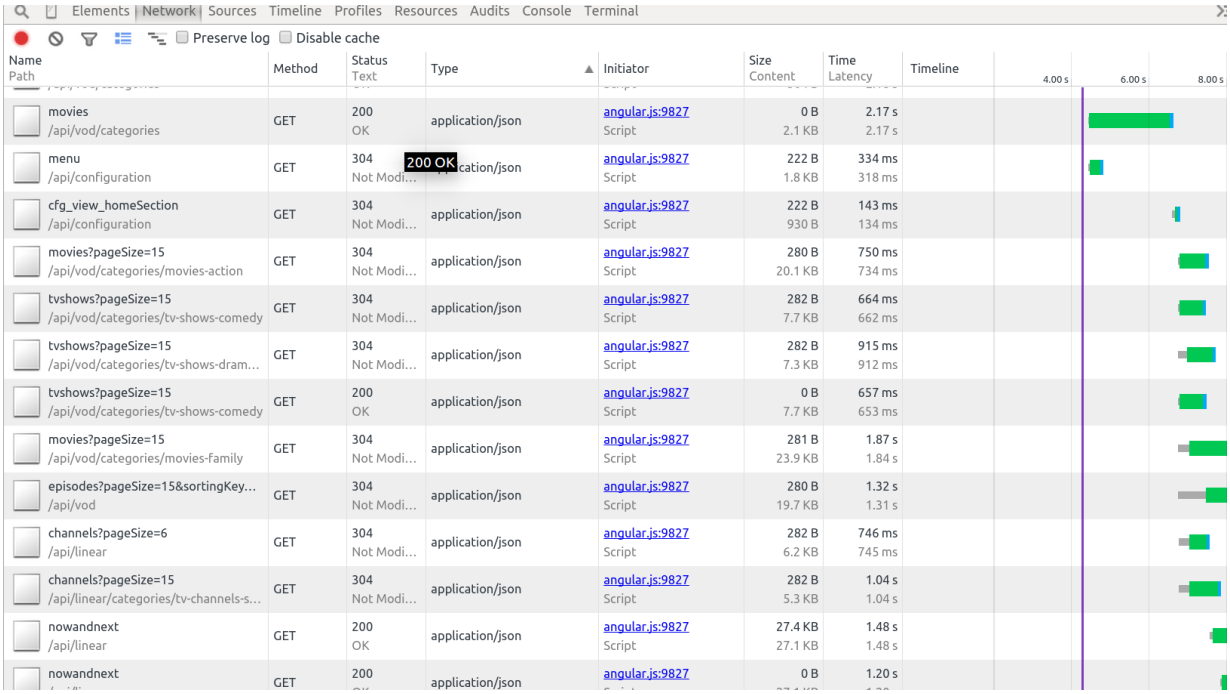

Figure 8: Example of page generation by browser

The client application will make http requests through a reversed proxy server. The proxy sever will decide if the content is stale or not and will act as a web cache. The benefit of this solution is that instead of contacting server and than caching the object, the cache is contacted first and than

the server. It theory it should give the performance and flexibility. The redis cache could be replaced by the Content Delivery Network solution in future.

4.2 Web cache selection

The web cache for the project should satisfy several parameters:

- Configurable. The client application sends requests both for DMOs and for writing user actions. The web cache should not cache the requests for writing user actions. The sessions still should be supported in order to write user actions.
- Responsive, easy maintanable. The web cache should be deployed easily and provide statistics and internal logging.
- High performance. The web cache should be transparent and must not increase the request time and latency if the cache object was not found locally.

There are a many web cache solutions available both commercial and open source. For this project only open source solutions were considered. The initial candidates were: Squid, Varnish, Apache server with proxy module, Nginx server as a cachable reversed proxy and Apache Traffic Server.

The Squid server is a forward proxy server, but it can be configured as a reversed proxy server. Squid is preferably used for storing static content.

The Varnish was developed as a reversed proxy server from the beginning. It is fast, reliable and lightweigth. It uses Varnish Configuration Language(VCL) for configuration and describing the data workflow in cache. The VCL is translated to the C code and compiled to a shared object which is then dynamically linked into the server process. It is powerful tool that helps to set up Varnish as a dynamic reverse proxy server.

The Apache traffic server was developed by the Yahoo group and moved eventually to the Apache Incubator [5]. According to Yahoo inc. Apache traffic server can handle more than 400TB of internet traffic per day and works as a forward as well as reversed proxy server. It has a growing community and continous improvement. The configuration is simple and consists of changing several files. All these, makes Apache traffic server a good candidate.

Other solutions(Nginx and Apache server) was not developed to be proxy servers, but have additional modules that one can install and configure. They are not well-configured and work worst than solutions described above [5][change].

The thorough comparison and performance evaluation of Varnish and Apache Traffic Server could be found in [1].

4.3 Web cache configuration

Before performing execution and comparative study reversed proxy servers should be properly configured. They should aggregate and store requests that contain public data and skip analytical requests and requests with private user information(e.g. payments).

Proxy servers should also work with http sessions. Usually, when session is specified(the set-cookie http header included in the server response), proxy servers are transparent, meaning they are skipping these requests and not store them in memory. As was described in previous chapters, the metadata server uses http session for analytical purposes. It means that even anonymous users will have the unique session.

In order to solve the problem, the Varnish was configured to replace Cookie header with X-Cookie header. This gives the possibility for Varnish to store the requests and still have the analytical requests available. The metadata server was modified in order to treat X-Cookie header as a Cookie header.

The Varnish configuration is presented in Appendix A.

The Apache traffic server did not require a specific configuration.

4.4 Testing tools

In order to conduct necessary experiments the folowing tools were selected: Apache Benchmark tool and JMeter.

JMeter is a tool that helps to measure the performance of applications that are using HTTP protocol. Jmeter was configured in order to simulate JSON and HTML requests that are required to render single page. The figure of Jmeter configuration is presented in Appendix B **??**. JMeter also builds the plot and the table of the request that was made with the corresponding time parameters.

Apache benchmark is a good tool for evaluating a performance of the developed solution. It emulate the user requests and execute multiple HTTP requests simultaneously.

4.5 Performance Comparison of Redis and Web caches

During the project several experiments were conducted. First, the figure [add figure of node.js jmeter] shows the results of executing JMeter script for existing company solution. The meaning of the columns is described below:

- Label – The name of the script that was executed
- Samles – The amount of responses that was executed
- Average – The average response time, represented in milliseconds
- Min – the minimum time for the response
- Max – the maximum time for the response
- Std. Dev. – standard deviation from the average response time
- Error – the persentage of error responses
- Throughput – the throughput represented in responses per second
- KB/sec – the data transferred per second
- Avg. Bytes – the amount of bytes transferred per response

5 Hierarchical VMO Generator

5.1 View model objects

The company's solution was developed in order to provide flexibility and customisation to the end user. The middleware server retrieves all configuration from the metadata server. The configuration can be user specific information as well as global information e.g. the location of content distributors. The middleware translates the responses from the content distributors to Data Model Objects. They are the immutable objects that are transferred to the client's application. On the other hand, the client application operates with View Model Objects. They are used to reder HTML pages. Single HTML page can contain single View Model Object. The VMO is constructed from mutliple data model objects and some additional information(e.g. the location of the image).

One important drawback could be noticed: application dublication. The middleware server and the client application have the same pattern of execution: they both operate with data and build new data patterns from existing ones. The difference is that the middleware server is doing it with content distributor responses and the client aplication is processing middleware responses. Because of this situation, the client application should make several HTTP requests for retrieving necessary DMO objects from the middleware server. The example of the single page generation is on the picture [put picture]. As can be seen, in order to generate a single main page, the client application has to make about 10 Ajax requests. What, if there was a possibility to generate the VMO objects on the middleware side? Maybe, there would be a possibility to get rid of the dublication? What should be done for it? This section will answer these questions.

5.1.1 VMO properties

Typical DMO and VMO objects are presented in appendix C[Add appendix C]. As could be seen a VMO object is a JSON dictionary build from DMO objects. Lets define the set of properties that VMO supports:

Generic. The DMO objects are specific and unique for every content distributor. They are build according to the specific rules in the middleware server. The VMOs are generated from DMOs according to another set of

rules specified in the client application. As a result in order to move VMO generation to the server side some descriptive language should be introduced that lets developers to describe the VMO structure.

Hierarchical. DMOs are basically data from content distributors. That means that the DMOs can be depend on other DMOs. For example, the figure [add DMO dep. figure] shows that in order to build a movie list DMO, first the DMO that represents movie categories should be generated, than for each movie category, the set of movie objects should be fetched, and only on the third step, the list of movies could be build. These means that the VMOs have a hierarchical structure. The typical VMO is presented on picture [figure of VMO].

5.2 New Middleware server architecture

In the current architecture, the single instance of the middleware server is deployed for a single content distributor. This is done due to the specifics of content distributors and data that they are providing. The question arises, is there a possibility to move the individual logic that requires to processes data to the client application while keeping the middleware as generic as possible? The theoretical benefits of this approach are: there will be needed single middleware server or cluster deployed. It will simplify the deploying system: instead of supporting middleware for every content distributor, there will be just one. The new architecture is presented of figure 9. As can be seen, the configuration server(Appgrid) is now treated like a content distributor. The benefit is that we do not need specify separate logic for it.

Lets consider the architecture from another prespective. The middleware server receives queries for the VMO generation. It supporsts several content distributors(databases), it makes queries to the content distributors for retrieving data model objects(tables). The middleware server resembles a lot a relational database system. The comparison of database management system and middleware server:

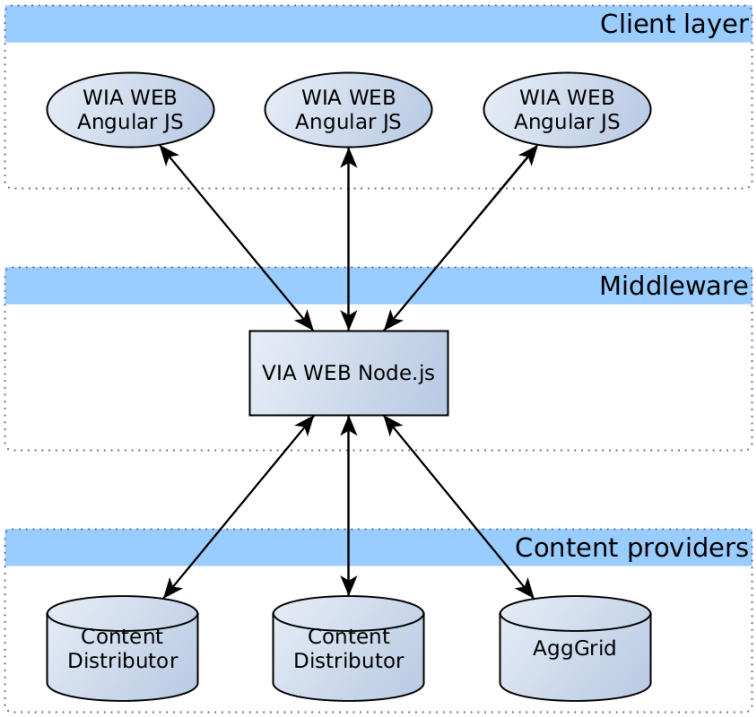

Figure 9: Manager workflow

Database	Middleware server VMO generation
serves multiple clients	should serve multiple client applications
supports creation of new databases	should support insertion of new content servers
supporst creation and alternation of tables	should support creation and alternation of new endpoints
supports dynamic queries to the desired databases and tables	should support queries to the content distributors and theis endpoints

5.3 Approach description

Let's consider relational database management systems and applications that are using them. There is usually a single instance or cluster of databases insalled and deployed on servers, depending on the amount of requests it should serve. Nobody makes new database instance for a single client, because it is neither efficient nor effective. In order to get data from the

database the queiries are executed. The SQL queries are basically the rich descriptive language that is interpreted by the database and executed for specified tables. The data is provided in two forms: array of table rows and array of view rows. Tables are the single source of information that resembles a lot DMO. Views are the mixed information from tables that looks like VMO.

In order to solve the problem of dublication and reduce the amount of requests, the hierarchical VMO generator(HVG) was developed. The HVG basicaclly the descriptive language, that developers are writing in a JSON format. The json is then translated into the GET HTTP or POST HTTP request and sended to the middleware server. The middleware server parses the request, builds the asyclic graph and fetches the corresponding data model objects. The data model objects then combined together and sended as a response to the client application. The client application executes specific logic on the data received from server and builds HTML pages.

5.4 Path in VMOs

This section introduces the definition of path in the View Modelw Object. The main format for data transferring is JSON. The JSON format consists of two main objects: array and dictionary. The example of dictionary:

Listing 1: The example of JSON dictionary

```
{key1:value1,key2:value2,key3:{key1:value1}}
```

The example of array:

Listing 2: The example of JSON array

```
[value1,value2,value3:{key1:[value1,value2]}]
```

Very complicated entities can be built using combinations of these two objects. Let's introduce the definition of path: The path is the route to the specific object or a set of objects in a JSON entity. The examples of path are presented below:

Listing 3: The examples of objects in different routes

```
Object: {root:{child:value}}
Path: root.child
Value: value
```

Input values:	{*id:[value1,value2,value3]*}
Endpoint:	*http://testdomain.ext/*{*id*}
Result:	[*http://testdomain.ext/value1*
	http://testdomain.ext/value2
	http://testdomain.ext/value3]
Endpoint:	*http://testdomain.ext/*{*id**}
Result:	*http://testdomain.ext/value1,value2,value3*

```
Object: {root:[{key:key1, value: value1},{key:key2,value:value2}]}
Path: root.key
Value: [value1,value2]
```

As can be seen, the path identifies a single object if it lays though a set of dictionaries. However, if the array is found on the route, all elements will be traversed and a set of objects will be extracted.

5.5 Hierarchical VMO Generator overview

In order to solve the problem of multiple requests and middleware server dublication, the descriptive hierarchical VMO generator was implemented. It consists of three parts: content provider dictionary, queries and parser.

The content provider dictionary has a tree based structure and its scheme is presented on 10. Rectangles indicate the single instance of object and parallelograms indicate the dictionary of objects. The content provider dictionary contains a dictionary of content providers. The key is a unique identifier that is used by query objects that are described below. Each content provider consitst of two parts: location and a dictionary of resources. The location is a domain with corresponding scheme. The resource has one field: endpoint. The endpoint is a path in URI[RFC to URI] with several embedded templates. The template can be one of two types: {*param_id*} and {*param_id**}. The first type indicates that only one value can replace the template. If array of values is given, it will produce the array of resolved endpoints. On the other hand, the second type produces single endpoint, even if array of values is given. The examples of templates are presented in table **??**.

The example of content provider map is given in Appendix C.

They query represents the request from the client application to the middleware server. The tree based structure is presented on figure 11. The

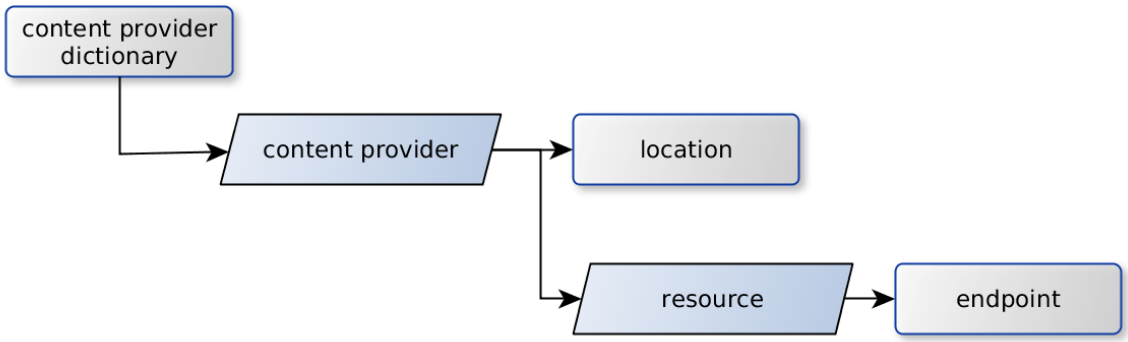

Figure 10: Content Provider dictionary structure

single query consists of several query objects. Each query object may have up to three parameters: *content_distributor*, *path_parameters*, *query_parameters*. The obligatory parameter, *content_distributor*, corresponds to the key in content provider dictionary desrcibed above. Through it the set of resources is retrieved from the content provider dictionary. The *path_parameters* and *query_parameters* are similar to each other, as a result only one of them will be described. The *path_parameters* is a dictionary of parameters, the key corresponds to the template parameter in resource object described above. Each *path_parameter* has type field. It can have one of two values: *constant* or *dependency*. If the value is *constant*, only one additional field should be specified: *values*. This field serves as a container of values that should replace the corresponding template in the endpoint.

On the other hand, if the value of *path_parameter* is *dependency*: the query object depends on another query object e.g. for getting the list of movies from the content distributor, the list of categories should be re-

quested first. That means that in order to request a query object, the parent object should be requested first and specified dependency variables for template parameters should be retrieved. In this case, additional parameters should be specified: *parent*, *parent_parameter_type*, *path* and *property*. The *parent* specifies the query object identifier of the parent object. The *path* is a JSON path to the object described above. The *property* is a key in the dictionary retrieved by using *path* parameter. The *parent_parameter_type* can be one of two values: *key* or *constraint*. If the value is *key* then the dependency variables will be extracted using path: *path+property*.

If the value of *parent_parameter_type* is *constraint* then the objects in *path* that obey constraints will be retrieved. In this case, the additional parameter should be specified in corresponding path parameter: *constraints*. The *constraints* is an array of constraints. Each constraint has two parameters: key and value. The key specifies the name of the property that the object should have and the value specifies the value of the property. The summary of parameters is presented in the table below:

Hierarchical Query Parser(HQP) is a module that translates client requests into JSON format. The available methods are:

The example of HQP usage is given in Appendix D.

5.6 HQL workflow

The developer writes the request using HQL parser methods. The request is translated into a JSON format that contains a dictionary of query objects. The JSON is sent as a POST HTTP request to the middleware server. The middleware server builds the asyclic graph (if there were dependency cycles it will generate error). The direct asyclic graph is then traversed using breath first search algorithm, on every node(query object) the data is fetched from appropriate content distributor (if the parent data has already been fetched) and is written into the output array. The output array is then sent as a response to the client POST request.

Table 1: Description of Query Object's parameters

Parameter	Description
content_distributor	the location of resource dictionary in provider
path_parameters	dictionary of variables that will be retrieved in runtime and replace the corresponding template parameters in endpoint for building a request URL
type	the type of path_parameter, can be constant or dependency
parent	the parent id of query object. Note: specified only when type=dependecny
parent_parameter_type	the parent parameter type, can be key or constraint. Note: specified only when type=dependecny
path	the path for the objects in parent response. Note: specified only when type=dependecny
property	the name of property that contains dependency value. Note: specified only when type=dependecny
constraints	the array of constraint. Note: specified only when type=dependecny and path_parameter_type=constraint

Table 2: Description of HQP module

Method Name	Parameters	Description
select	object id, content distributor id	creates an empty query object with specified content distributor id
setConstantParameter	parameter id, values	appends the path parameter which has a type *constant* to the query object's path_parameters dictionary
setParentParameter	parameter id, parent query object, path , property	appends the parent parameter to the query object
setQueryParameter	key, value	appends query parameter to the query objects query_parameters dictionary
addConstraints	path parameter id, array of constraints	appends the constraint array to the query object's path patrameter
build		translates query objects into JSON format

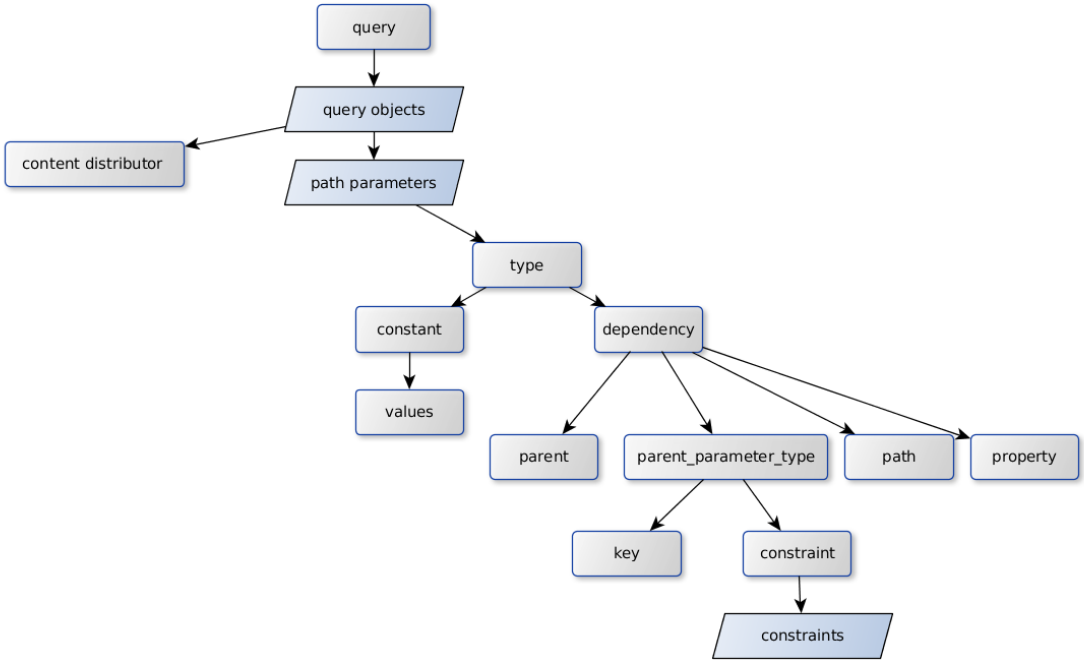

Figure 11: Query structure

5.7 Comparison

References

[1] Shahab Bakhtiyari, *Performance evaluation of the apache traffic server and varnish reverse proxies*, (2012).

[2] Katia Way Admiralty Rey Marina Barish, Greg Obraczka, *World wide web caching: Trends and techniques*.

[3] Matt Copeland, George Mcclain, *Web caching with dynamic content*.

[4] J. Rabinovich M. Gadde, S. Chase, *Web caching and content distribution: A view from the interior*, Computer Communications **24** (2001), 222–231.

[5] Yahoo inc., *Yahoo's cloud team open sources traffic server*, November 2009.

[6] Ramesh K. Sun Jennifer Nygren, Erik Sitaraman, *The akamai network: a platform for high-performance internet applications*, SIGOPS Oper. Syst. Rev. **44** (2010), 2–19.

[7] Rajkumar Pathan, Al-mukaddim Khan Buyya, *A taxonomy and survey of content delivery networks*, Grid Computing and Distributed Systems GRIDS Laboratory University of Melbourne Parkville Australia **148** (2006), 1–44.

[8] J. Reschke R. Fielding, M. Nottingham, *Rfc 7234: Hypertext transfer protocol (http/1.1): Caching*, 2014.

[9] F. Hao M. Varvello V. K. Adhikari, Y. Guo, *Unreeling netflix: Understanding and improving multi-cdn movie delivery*.

Appendices

Appendix A

Listing 4: Varnish Configuration

```
vcl 4.0;

backend default {
    .host = "127.0.0.1";
    .port = "9000";
}

sub vcl_recv {
#       set req.http.X-Cookie=req.http.
   Cookie;
#       unset req.http.Cookie;
        if(req.url ~ "^/api/vod/" || req.
           url ~ "^/api/asset/" || req.url
           ~ "^/api/health/" ||
                req.url ~ "^/api/linear/"
                   || req.url ~ "^/asset"
                   || req.url ~ "^/partials
                   /" ||
                req.url ~ "^/extensions/"
                   || req.url ~ "^/
                   bower_components/" ||
                   req.url ~ "^/scripts/"
                   || req.url ~ "^/images
                   /"){
                set req.http.X-Cookie=req.
                   http.Cookie;
                unset req.http.Cookie;
                return(hash);
        }
        if(req.url ~ "^/api/configuration")
           {
                set req.http.X-Cookie=req.
                   http.Cookie;
                unset req.http.Cookie;
                return(hash);
        }
        return(pass);
}

sub vcl_hash {
        hash_data(req.url);
        if(req.http.host){
                hash_data(req.http.host);
        }
        return(lookup);
```

```
}

sub vcl_hit {
        return(deliver);
}

sub vcl_miss {
        return(fetch);
}

sub vcl_pass {
        return(fetch);
}

sub vcl_backend_response {
    # Happens after we have read the
       response headers from the backend.
    #
    # Here you clean the response headers,
       removing silly Set-Cookie headers
    # and other mistakes your backend does.
        if(bereq.url ~ "^/api/configuration
           "){
                #unset bereq.http.Cookie;
                set beresp.ttl = 24h;
        }
    return(deliver);
}

sub vcl_deliver {
    # Happens when we have all the pieces
       we need, and are about to send the
    # response to the client.
    #
    # You can do accounting or modifying
       the final object here.
        return(deliver);
}
```

Appendix B

Appendix C

Listing 5: Content Provider Dictionary

```
{
  'accedo.ovp': {
      'url': 'https://ovp-staging.cloud.
         accedo.tv',
      'categories.movies': {
                'endpoint': '/category/{id
                   }/movie'
            },
```

```
        'categories.tvshows': {
                  'endpoint': '/category/{id
                     }/tvshow'
                },
        'episodes': {
                  'endpoint': '/episode'
                },
        'tvshow': {
                  'endpoint': '/tvshow/{id*}'
                },
        'tvseason': {
                  'endpoint': '/tvseason/{id
                     *}'
                }
    },
  'accedo.appgrid': {
        'url': 'https://appgrid-api.cloud.
           accedo.tv',
        'metadata': {
                  'endpoint': '/metadata'
                }
    }
}
```

Appendix D

Listing 6: The examples of objects in different routes

```
var parser = new HQLParser();
var j = parser
          .select('category_movies', 'accedo.ovp.categories.movies')
          .setConstantParameter('id', ['movies-action'])
          .setQueryParameter('pageSize', 2)
          .select('category_tvshows', 'accedo.ovp.categories.tvshows')
          .setConstantParameter('id', ['tv_shows_comedy'])
          .setQueryParameter('pageSize', 2)
          .select('episodes', 'accedo.ovp.episodes')
          .setQueryParameter('pageSize', 2)
          .select('tvshow', 'accedo.ovp.tvshow')
          .setParentParameter('id', 'episodes', 'entries.metadata', 'value
          .addConstraints('id', [{key: "name", value: "VOD$tvShowId"}])
          .select('tvseason', 'accedo.ovp.tvseason')
          .setParentParameter('id', 'episodes', 'entries.metadata', 'value
          .addConstraints('id', [{key: "name", value: "VOD$tvSeasonId"}])
          .build();
```