

POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria



Exploring Strategies for Context-Aware API Composition in Mobile Apps

Supervisor: Prof. Letizia Tanca
Co-supervisors: Prof. Maristella Matera
Prof. Florian Daniel
Prof. Vittorio Zaccaria

M.Sc. Thesis by:
Bachar Senno, 873448

Academic Year 2017-2018

*To my family, professors, friends, and everyone who helped me get
where I am today.*

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contribution	4
2	State of the Art	5
2.1	On Context and Context Awareness	5
2.2	On Data Integration and Mashups	6
3	Context-Aware Data Mashups: Concepts, Goals and Requirements	9
3.1	Concepts	9
3.1.1	Introducing CAMUS	9
3.1.2	Response format	12
3.2	Goals and Requirements	13
4	Context-Aware Data Mashups: Approach	17
4.1	Design Decisions	17
4.1.1	Service Querying with PerLa	17
4.2	Architecture	21
4.2.1	Topic Wrappers	23
4.2.2	Service Descriptors	24
4.2.3	Service Registry	25
4.2.4	Storage and Caching	27
5	Implementation and Evaluation	29
5.1	Implementation	29
5.1.1	API Wrappers	29
5.1.2	Storage and Caching	35
5.2	Evaluation	44
5.2.1	Adding New Services	44
5.2.2	Caching and Storage Evaluation	44

6 Conclusion and Future Work	49
6.1 Conclusion	49
6.2 Future Work	50
References	52
A Code	55

List of Figures

2.1 TrendsMap Interface	8
3.1 CAMUS Flow	10
4.1 High-Level Architecture View	22
4.2 Topic Wrapper	24
4.3 Service Descriptor Skeleton	25
4.4 Caching Schema	27
5.1 Caching Details	37
5.2 Sample of Query Metadata	42
5.3 Geographical Query Distribution	45

List of Tables

5.1 Performance Evaluation Results	47
--	----

Chapter 1

Introduction

1.1 Motivation

Over the past decade, we have been witnessing smartphones taking the world by storm. Since the introduction of smartphones, we have grown used to relying in so many aspects of our lives. We make so many of our decisions based on what the various apps on our smartphones tell us: some apps tell us about good places to eat, others help us book hotels when going abroad and plan our itinerary, and with the improvements in online security, now we can even opt for online banking.

Naturally, as the number of users of such applications exploded, more and more creators wanted to provide for these users. Therefore, we, as users, have now a plethora of applications to choose from, no matter what the topic is. At first glance, this might seem like a good outcome: more applications mean more options and more customizations. However, that is not really the case: firstly, it is easy for a user to get confused as to which application to use. These applications provide somewhat similar content for a certain topic, and there aren't really too many distinctive features that would allow a user to pick one application over the other. Secondly, these applications sometimes try to provide a tailor-made experience, using some machine learning algorithms, coupled with the previous searches and actions performed by a user. It is worthy to note though that due to the lack of specific contextual data pertaining to the user's conditions, the results risk being rather poor. Due to these reasons, it becomes obvious that an increase in the amount of information available doesn't necessarily cause an increase in knowledge.

In this project, we try to tackle both of these problems. On one hand, the system we are designing aims to provide a truly *context-aware* platform that takes into consideration as many details as possible concerning the user's needs. On the other hand, we try to incorporate as many data sources as possible, seamlessly integrating their outputs, while keeping in mind that adding additional sources should be simplified as much as possible.

While these problems might seem trivial at first glance, digging deeper into each of them and trying to implement working solutions reveal the hidden intricacies that make this field such fascinating yet challenging one. We will try now to provide some understanding of the notions of context-awareness and data integrations, as well as where they fit in our project.

One of the most common ways of understanding a term is breaking it down into its elementary components, defining each component separately, and then examine how these components complete one another in the scope of the main term. This is straightforward when looking at the latter part of "context awareness": being aware of something means to possess knowledge or perception of it. Therefore, context awareness means to be able to know and understand that the "context" is. However, the big question is what is "context"? What defines it? What are the factors that make up a "context", particularly within the domain of mobile applications and their users? The Oxford Dictionaries define the word "context" as "the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood". Our problems however lies in narrowing down what "circumstances" are.

In the field of computer science and ubiquitous computing, many attempts have been made to narrow down the elements that make up a user's "context". The older definitions, such as the one proposed by Schilit and Heimer[14] included a set of three parameters: the location of use, the nearby elements and objects, and the changes what these objects go through over time. More recent definitions include users themselves in the formula, which means that the context is made up of psychological and social-based outcomes of human activities[7][16].

Context awareness means being aware of the different circumstances that define a user's current situation, thus allowing the system to provide results that take such circumstances into consideration. We should also look at a user's context as a dynamic and changing element, as opposed to relatively static measures such as age, gender, or nationality, which define the profile of a user. Some factors include time, location, identity, and current activity in the scope of a user's context. Other factors can be added such as a user's role (the context of a father travelling with his family is different from that of a teacher accompanying a group of students on a field trip). Being able to pinpoint accurately what to include when studying a user's context will go a long way in helping us provide a customized and fitting user experience.

Lenzerini[10] defines data integration as "the problem of combining data residing at different sources, and providing the user with a unified view of these data". For instance, if we visit the Android Play Store or the Apple Store and we write a very simple term such as food or restaurant, seemingly endless lists of apps will pop up, and many of these apps will have decent ratings and download counts. If we take a number of these apps and perform the same search on all of them, we will also find long lists of results, with many intersections amongst the various apps. On many occasions, there is not really a clear set of criteria that would allow us to choose one of the results from a list, let alone choosing one of the apps over the others.

A good scenario for the user would be to have a single app that somehow integrates all of the outputs from the most significant apps, and shows him one comprehensive list of results. An even better scenario would also take the aforementioned context into consideration, and apply it to the list of the results. This would mean that all of the results provided to the user are customized to their specific case; in other words, all of the results represent a pool of valid options for the user, and it would come down to their personal preferences when making a choice.

In order to have a system that is able to perform these operations, a lot of background work needs to take place. The first component of this system is a backend server able to query multiple services, retrieve results, and integrate them together. It is worthy to note that the responses from different services can possibly have very different structures. Therefore, we need to let the backend system know what kind

of response to expect from each service, as well as how to extract the information that we deem relevant.

The second component is a tool that would allow a system administrator to input the context-specific details of the user. This will be our way of translating the particular needs of a user into a language understood by our system. We need to expose a list of options that is extensive enough to choose from, as the circumstances of users differ greatly, and there are many details that need to be included.

Finally yet importantly, we need a middleware that is able to take the context representation of the user and apply it to the response received from the backend server. This middleware will have to perform the mapping between the context-related information on one hand, and the collected search results on the other hand. Once the mapping is done, the results are filtered accordingly and presented to the end-user. The innovative part is designing the different components in a generic way, as to allow adapting it to as many different topics and scenarios as possible. Implementing this system will go a long way in providing a truly customized experience to users.

In what follows, we make use of the example of tourists needing to perform various searches, or asking for recommendations for different activities (where to eat, which events to attend etc...), and they perform these operations through the mobile application installed on their devices by the travel agent. This example includes enough diversity for our purposes on one hand, and on the other hand, there are enough publicly available web services providing relevant information obtainable through relatively simple API calls.

1.2 Contribution

The main contributions we are aiming to achieve by the end of this project are the following:

- A working implementation of the generic web service querying and integration operations.
- An abstract design, as well as the corresponding implementation of resource storage and query caching.

Chapter 2

State of the Art

2.1 On Context and Context Awareness

Many have tried to narrow down the definition of context, and identify the various elements that make it up. Brown et al.[3] include location, adjacency of other objects, critical states, companions, and time as elements that make up the context. Rodden[13] defines context from a different perspective: instead of looking at the circumstances surrounding the user, he looks at the user as the main point of interest. Robles and Kim[12] define context-awareness as “the concept of leveraging information about the end user to improve the quality of the interaction”. Dey et al.[1] provided in 2001 one of the most comprehensive definitions of context; the context is made up of any and all information that take part in characterizing an entity, whether this entity is a person, object, or place. As long as this entity influences the user’s situation, it should be included in the elements that make up the context. Naturally, this includes both the user themselves, and the application they are using.

Context aware systems are widely used in a variety of areas, and their popularity is increasing even more with the improvements in artificial intelligence and machine learning. Internet of things also relies greatly on context awareness to provide a personalized and tailor-made user experience to each individual user. Another area where context awareness is greatly beneficial is recommender systems, as these technologies are the main driving power behind today’s main advertising and online shopping services. Incorporating context-awareness with a multitude of real life applications is also growing more popular in the academic field, as is the case with the ADaPT (Automatic Data Personalization based

on Contextual Preferences) framework[11], which was developed in the Politecnico di Milano in this regard.

2.2 On Data Integration and Mashups

Integrating data incoming from heterogenous sources has been a topic of interest for many years now. The work proposed in [2] focuses on multi-domain queries, i.e. queries that make use of two or more heterogeneous resources available on the web. The authors try to create a formal abstraction of the query, and then map the query to multiple “query plans”. A query plan represents a possible execution schedule for these queries, either sequentially or in parallel, depending on which combination has the best performance.

Francese et al.[8] proposed an interesting approach that allows users to dynamically design their own applications on their smartphones, in a process that integrates the functionalities of their devices with Internet services. Although their approach grants great freedom to users to decide and design how to access these resources, it does not focus on the context of the users, and does not personalize the user experience based on the conditions of each individual user.

Another service dubbed “MyService” was created by Lee and Joo[9]; their platform allows experienced designers to specify rules that define the context. These rules determine which services are selected, and dynamic code generation takes care of invoking these services at runtime. However, this system only considers location as the main factor that defines the context. Our definition of context is much broader, and include many factors, one of which is location. Moreover, the selection of the services that the system should query is done by the system itself in our case, and the users do not need to decide which service to query. Service selection is done by identifying the needed resources, and querying the services that would best fulfill these needs automatically.

Casillo et al.[4] developed a context-aware mobile application that aims to help tourists in Salerno plan their trip and locate places of interest. In their approach, the authors used a context dimension tree to specify various parameters of a user’s context, and their definition of context is very close to ours. The results of their work was very promising; one

hundred users installed and used the application, and afterwards filled a survey to describe their experience. In most cases, user satisfaction rate was above 75%, and they generally agreed that the application's interpretation of their context is accurate. This is encouraging for us because it implies that using a context dimension tree to describe a user's context is a valid approach. An additional feature in our system is the dynamic inclusion of web services: the services that can be queried are not fixed, and new services can be always added to provide more -and more accurate- results.

Mashups are in a way a special application of data integration: they are a kind of hybrid application which tries to use data obtained from multiple sources, merging their outputs, and displaying the result using one unique interface. These mashups are usually created under Software as a Service (SaaS) paradigm, and made available for users to use over the Internet. Most of these services are centered around a specific topic; they retrieve information related to this topic from many publicly available sources, and perform the integration, showing the user a comprehensive output that combines the data received from the queried sources. TrendsMap¹ mashes up Google Maps and Twitter Trending Topics to display the most trending topics on Twitter on the world map. Web mashups have started declining in popularity since the rise of the mobile era however, since mobile applications were able to provide similar services using a much simpler and accessible medium.

¹<https://trendsmap.com/>

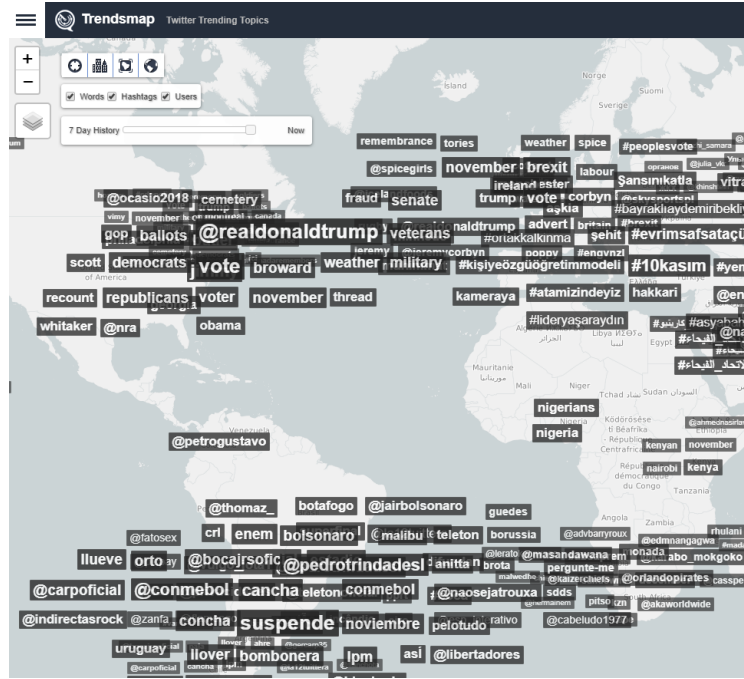


Figure 2.1: TrendsMap Interface

The main difference between our system and the available web mashups however is that we are developing a framework that does not revolve around a specific topic. Instead, the data integration procedures and the context-awareness applications are generalized and made adaptable to numerous situations.

To wrap up, there are many projects that have tried to approach data integration and mashups from different angles. Many of the presented solutions are well structured and do their intended work efficiently. However, they are still missing one or more of the key elements we are focusing on in our system, whether it is a clear and comprehensive definition of “context”, an efficient representation of this context (both in terms of abstract design and practical implementation), or a valid generalization of the web service integration operations.

Chapter 3

Context-Aware Data Mashups: Concepts, Goals and Requirements

3.1 Concepts

3.1.1 Introducing CAMUS

Context-Aware Data MashUpS (CAMUS)[5] is a project that aims to dynamically generate mobile applications, based on a framework capable of performing seamless data integration of resources obtained from multiple heterogeneous sources, and then apply context-based filtering to this data. The result of this process is an answer that is context-aware; in other words, it takes into account the particular details of the user requesting this information. Moreover, this answer integrates the different responses received from the various sources, whether they are online web services, or locally stored data.

The CAMUS project is split into multiple components, each having its own purpose and functionality. Moreover, there are multiple human agents involved in operating the system, each of them with their own roles and responsibilities which we will shortly explain. The main components on the CAMUS system are the end-user interface, the context dimension tree, the resource schema, and finally the backend server. On the other hand, the human agents who are involved in this system are the end user, the domain expert, and the system administrator.

Figure 3.1 shows the basic operational flow of the CAMUS system.

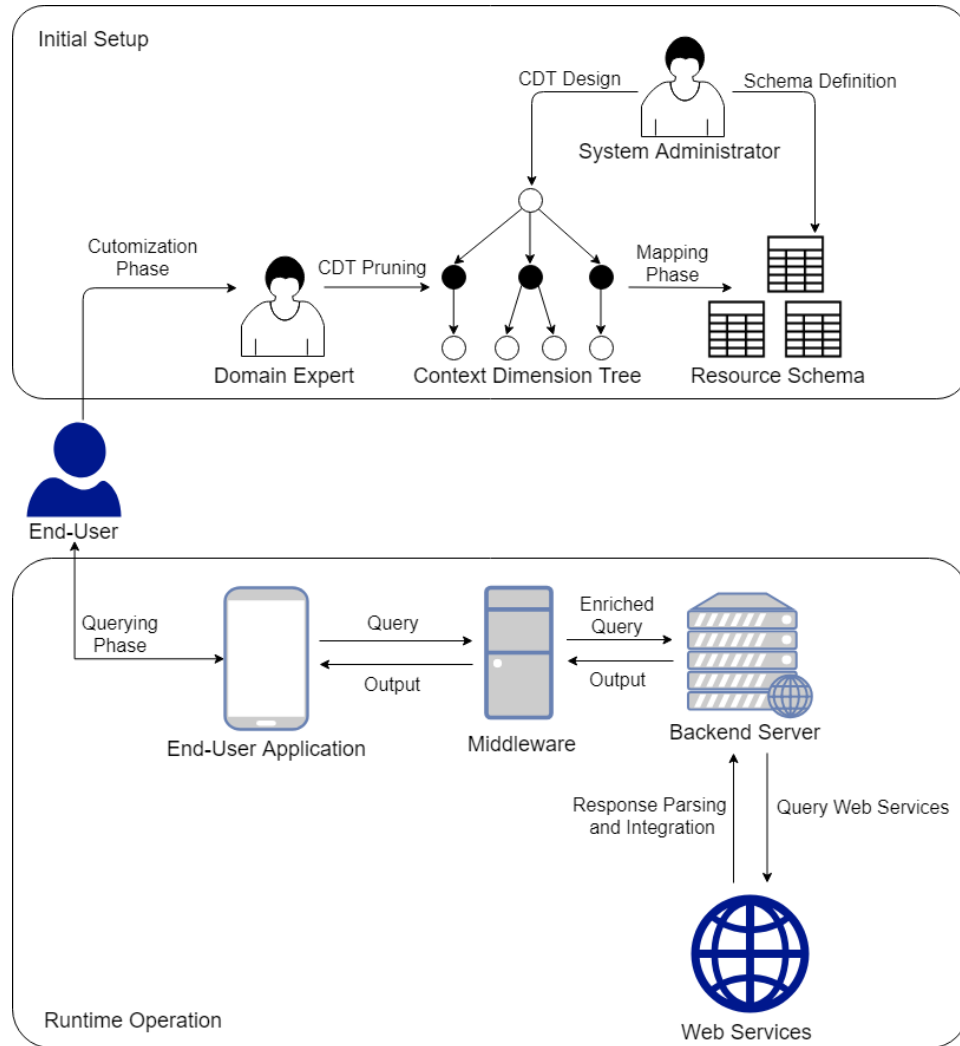


Figure 3.1: CAMUS Flow

There are two main stages in the operational flow: the initial setup, and the runtime operation. The initial setup starts with the end-user providing their context parameters to the domain expert. The domain expert perform pruning operations on the CDT so that it reflects the user's context. This CDT is initially constructed by the system administrator, and the domain expert can also provide input to make sure that it is able to accurately represent the user's context. The system administrator also constructs the resource schema, which is a representation of the various

relevant interest topics.

Once the mapping between the CDT and the resource schema is done, the system is ready to be put into use, and the end-user can install the application on their device, which moves us to the runtime stage. This is the stage that is most visible to the end-user; the user starts sending a query using the mobile application, and this query is forwarded to the middleware. At this point, the query is still context-agnostic. The middleware takes care of enriching the query with the context details which were previously supplied (using the CDT), and the enriched query is then sent to the backend server. The backend server queries the various web services, integrating their responses in a format that matches the format specified in the resource schema. This final response is forwarded to the middleware, which in turn forwards it to the mobile application so that it can be displayed to the end-user.

In what follows, we elaborate on each individual component, and we make use of the previously mentioned travel agent example to better illustrate one of the possible use cases.

The End-User Application

The end-user interface represents the mobile application installed on the mobile devices of the users. This mobile application is used as a two-way portal that both displays relevant information that users ask for, as well as takes input from the users to further improve the user experience. During the initial setup phase, the end-user (the tourist in our example) provides as many details about their needs and preferences. These details will be used to describe the “context” of the user, along with other parameters that can be collected real-time while they use the application. The travel agent takes the information provided by the tourist, and using a web interface, inputs this information into the system.

The Context-Dimension Tree (CDT)

The web interface used by the travel agent is generated based on the context dimension tree. The system administrator initially constructs the tree, and the web interface is the mean through which the domain expert can input the values provided by the end-user. This tree is a rep-

resentation of the various parameters that are part of a user’s context for a given topic. In our example, a user’s favorite cuisines, their allergies, and their ambience preference (indoors vs. outdoors) are some of factors that constitute the CDT for the “Restaurants” topic. The context, now represented by the CDT, is incorporated into the queries, transforming them from a context-agnostic state to a context-aware one.

The Resource Schema

The resource schema, also designed by the system administrator, describes the various topics supported by CAMUS for a certain domain; in other words, it includes the attributes associated with a topic. For example, within the tourism domain, the resource schema specifies that the “Restaurant” structure is part of the schema, and it includes attributes such as name, address, phone number, cuisines, rating, and so on. A mapping function takes care of matching these attributes with the CDT, which results in context-aware queries.

The Backend Server

The final component of the CAMUS framework is the backend server. This component’s task is to perform the actual querying of the various web services when required by the middleware. The middleware receives queries from the end-user application or from the domain expert, and performs the mapping between the CDT and the resource schema if needed. Then, the queries are forwarded to the backend server. This server then identifies which web services to query, sends the requests, and parses the responses. This server is also responsible of performing the data integration operations, transforming the heterogeneous output of the various web services into a homogeneous list that is made accessible to the middleware. In what follows, this component will be the main point of focus, and the aim will be to design and provide a working implementation for it.

3.1.2 Response format

In the classical sense, two or more data sets are considered homogeneous when they are made up of elements that belong to the same type. In other words, all of the elements that make up the data sets share the same structure. The simplest example of a homogeneous data set is an array, since generally, all of the elements of an array belong to the same

type. Therefore, two integer arrays are homogenous. Conversely, heterogeneous data sets are data sets that contain differently structured elements such as lists and unions, among others. This is the low-level definition of homogeneity and heterogeneity.

In data integration, two datasets are considered homogenous if they are structured according to the same data model. Naturally, different online services have their own structures for the objects that they return when they receive requests. This is even true for different services that operate in the same domain. For instance, two APIs specializing in food and restaurants reply differently, even if we call a simple search API that returns a list of restaurants. Even though conceptually both APIs are returning a list of restaurants, each API returns different attributes, and therefore the Restaurant objects are not identical. It does not stop at that: even the general structure of the reply, the meta information returned along with the restaurants list, and the names of the fields are different. Therefore, our objective here is to establish pair-wise homogeneity between different responses, in accordance with the structures defined in the resource schema.

3.2 Goals and Requirements

Integration: Our primary goal in this thesis is to find a way to extract the useful information from the aforementioned heterogeneous replies, therefore ending up with a homogenous structure (in this example, a collection of restaurants), and this collection will have in it all of the restaurants that were included in the various received replies.

Context Awareness: Context awareness should be included in all of the operational phases of our system. Firstly, selecting which web services to query is made context-aware by choosing services that allow filtering, or include information that match with the parameters that make up our context. These parameters are determined on a case-by-case basis, and therefore, so should be the selected services.

API Suitability: Going to a lower level, the data returned by the services can be enriched with the user's context by making sure that the query parameters of the requests that are being sent to the web services represent as accurately as possible the user's context. In order to do so,

each individual API that we are querying should be closely examined, and we should make sure that we are on one hand optimizing the values attached to the query, and that we are taking advantage of all of the supported filtering parameters offered by the web service on the other hand.

Flexibility: The process should also be flexible, in the sense that, thanks to adaptive abstractions, we should have the ability to add more services, and the algorithm should be able to include the results from the new service without making changes to the core of the code. Instead, we would just have to include a description that specifies details about the new service (what parameters it expects, and the structure of its reply).

Velocity: Another primary concern is to reduce the waiting time on the side of the end user: sending numerous requests to multiple web services, receiving the replies, and parsing these replies can take a lot of time. Therefore, we should aim to reduce the number of sent requests if possible. Mobile application users however do not expect long wait times when performing seemingly simple functions such as searching for a place to eat, and therefore we should be able to provide them with results as soon as possible.

Request Filtering: We have to include the context awareness paradigm in our workflow: the results that the user sees on their device should be filtered according to their own context. In some cases, the filtering can be done before sending the request to the web services, by making use of the request parameters that are sent along with the request.

Response Filtering: Further filtering should be done after the services send their replies: we are bound to encounter services that are not as flexible as we need them to be in terms of allowing filtering within the request parameters. In that case, we should examine all of the fields returned by these services, and make sure that context-based filtering is performed also on these responses when needed. Finally, we should also allow the end-user to provide manually additional filters, to account for changes in the context, or simply to give the user more freedom. This would create a truly dynamic, context-aware output that matches the user's needs as accurately as possible.

User-friendly Interfaces: In order to allow users of CAMUS to easily integrate new services, we should provide an intuitive frontend which can be used to describe new services, and associate them with their respective topics. The same applies to filling a user's information in the CDT, especially that this task should be doable by an individual who does not possess any programming or technical knowledge. Moreover, the system should take into account that the information that was provided as input to the CDT might change over time, and therefore, the implementation should be dynamic enough to allow changes to the CDT, and update the output shown to the users accordingly. To do so, the application installed on the end-user's device must allow them to override the previously-defined context parameters so that they reflect better the current context.

Chapter 4

Context-Aware Data Mashups: Approach

In this chapter, we will be discussing and justifying our design decisions, and we will provide a description of the high-level architecture of our system. As previously stated, our work will focus on the backend component, which takes care of service querying and data integration.

4.1 Design Decisions

4.1.1 Service Querying with PerLa

PerLa¹[15] is a middleware technology developed in Politecnico di Milano. The main aim of this middleware at the time was to query pervasive systems that exist in a heterogeneous environment, and provide a homogenous abstraction of the different devices that make up this system. In CAMUS, part of the system is going to have query different web services, and provide an abstract homogenous interface that contains the results of these queries. Therefore, using PerLa as a tool to provide such abstraction was a reasonable option.

The way PerLa works is as follows: the first step is to generate (or manually write) an XML descriptor for each device in the pervasive system. Since different sensors often have different structures for their replies, the descriptor's job is to allow PerLa to know what to expect. The descriptor specifies what the structure of the reply is, as well as some

¹<http://perla.dei.polimi.it/middleware.php>

other factors, such as the protocol used for communication (i.e. http, https etc...), the request method (GET, POST etc...), and the destination URL of the request. The descriptor also includes details on which response elements are interesting to us, so that we can retrieve and query for them. Finally, we can include at the end of the XML file any post-processing we want to do, in terms of editing the content of the response, using the PerLa scripting language.

In theory, we can adapt PerLa and use it to query REST APIs instead of pervasive systems. In this case, the web services are similar to the sensors that we were querying in the case of pervasive systems. Moreover, the descriptors would describe these web services (request parameters, response format etc...), in a similar way to how they used to describe the sensors.

Unfortunately, PerLa requires performing some complex steps to set up, before we could use it for REST APIs. As we prepared PerLa for the needed tasks, we started figuring out the missing components and performing the necessary adjustments. In particular, PerLa did not natively support including HTTP headers when sending requests. Therefore, we had to locate the area in the code where the requests were being built and sent, and then we added the required code to support adding headers to requests.

```
getRequest = new HttpGet(request.getUri());
String head = request.getHeaders();
if (head != null && !head.equals("")) {
    String[] headers = head.split(";");
    for (int i = 0; i < headers.length; i++) {
        String[] header = headers[i].split(":");
        getRequest.addHeader(header[0], header[1]);
    }
}
```

This code allows adding headers to the XML descriptor of the web service as follows: in the requests object, we add an attribute “headers”, and assign to it the header as key:value. With this code, we can also add multiple headers, and we separate the different key-value pairs using semicolons. We also edited other various areas in the code to make this code work, as building the HTTP request goes through multiple phases

in different Java classes.

Preliminary results with PerLa seemed promising: we were able to adapt this stable, solid framework to our needs. We performed these tests manually by writing XML descriptors for simple services (mostly weather services) because the responses were simple enough, and mapping them to XML was not too challenging. Naturally, we also want our code to work with more complex responses, as this is the case for most modern APIs. Modern APIs return responses that include multiple objects of various structured, nested arrays, and such structures are not simple when it comes to mapping them to XML descriptors for PerLa.

For this reason, the next step we decided to work on is finding a way to automatically generate an XML descriptor for each web service, based on the JSON responses returned by the service. The descriptor generator should be able to parse the response, figure out the schema of this response regardless of the actual data context returned, and output an XML descriptor that we would directly plug into PerLa. In this way, this descriptor should work with any response returned by that specific API endpoint, for any values of the query parameters we include when querying the service.

After doing some research, we found that there are many libraries written in various languages that can generate a JSON Schema-like structure for any JSON document. This was interesting because mapping a JSON schema to an XML schema is much simpler than mapping a JSON response to an XML schema. Thus, by using such libraries, we would already be half-way on the path of generating the XML schema that PerLa requires. We ended up using a Jskemator, a public Python library available on Github. This library did its job perfectly well, and the next step would be to add the code needed to map JSON schema to XML schema. We wrote the corresponding code in Python as well, and as we tried it on different web services, we started figuring out some particularities in the way PerLa parses the XML descriptors. Specifically, when generating the response schema, we should define the different objects in a bottom-up approach. This means that in the case of nested objects, for example, we should define the innermost object first, and then progressively go outwards until we reach the top level. At first sight, this does not seem very complicated, and all we have to do is keep track of the current nesting level we have reached, and then re-order all the struc-

tures we have generated based on the nesting order. After testing with many services, we realized that it is not easy to solve this issue, for the reasons mentioned in the following point.

The second point we noticed is that PerLa in its current implementation does not support the presence of multiple objects sharing the same ID in the same JSON response. Therefore, even if the original response contains such objects, and in case they do not share the same structure, our code must somehow make the IDs in the corresponding XML schema distinct. However, this proved to be more difficult when querying some of the more complex services we mentioned. Taking Foursquare as an example, the response contains several nested objects that share the same IDs as the parent objects (sometimes several levels going up), and this created the problem of circular dependency, as we ended up with the inner objects being dependent on the outer ones.

The third issue we ran into was when we realized that, since the URL we are querying is hard-coded in the descriptor, we needed to create a different XML file not only for each service we query, but also for each query to the same service any time we change the value of a query parameter. This is a problem because, in a real life scenario, we will be querying the services multiple times, with different parameters, and having to create a separate descriptor for each query renders using PerLa unfeasible. Another important drawback of PerLa is that, after querying different services, we will need to perform joins in order to integrate the different responses into one logical group, and such operations are not natively supported by PerLa. Therefore, we would anyway need to use an external DBMS or module to accomplish this task.

Finally yet importantly, PerLa supports querying multiple services simultaneously. However, in case the services share objects that have the same name, PerLa does not support choosing which service to query for this object. This is yet another thing that we will have to add to the core code of PerLa. After doing these tests and realizing the shortcomings of PerLa, we opted not to use it, since the work we will need to do in order to add the needed features will be very time-consuming, and this outweighs the advantages of using a this well-established framework. In the end, we decided to perform simply the queries to the services manually using any programming language of our choice, therefore bypassing the need to upgrade PerLa to have it satisfy our requirements.

4.2 Architecture

After making sure that in its current state, PerLa does not satisfy our needs, we decided to move forward without including it as a component in the CAMUS system. Therefore, we now had complete freedom in choosing the language in which to implement the different components we need. We opted to use PHP to send the requests and parse the responses. Next, we had to find numerous free APIs belonging to a specific category, and these APIs must fit within the Travel Agent scenario. In the end, we selected three APIs that allow searching for events in a specific location (PredictHQ, Ticketmaster, Eventful, Eventbrite), and three APIs that lists restaurants also given a location (Zomato, Yelp, Foursquare). This makes sense in terms of travelling activities because tourists are usually interested in both finding places to eat and events to attend. This also was interesting because it opens the possibility of performing joins between these two categories, e.g. searching for sushi restaurants that are located around a certain event.

After choosing these seven sample APIs, we wrote the code needed to query each of them, and proceeded to study the differences in the way each of them expects a request and provides a reply. Naturally, different APIs require differently formed requests, and respond in variously structured formats. For instance, some APIs require header authentication, while others perform authentication using URL parameters.

Figure 4.1 shows the high-level architecture of the backend component, and we will elaborate on the role of each part in the following parts.

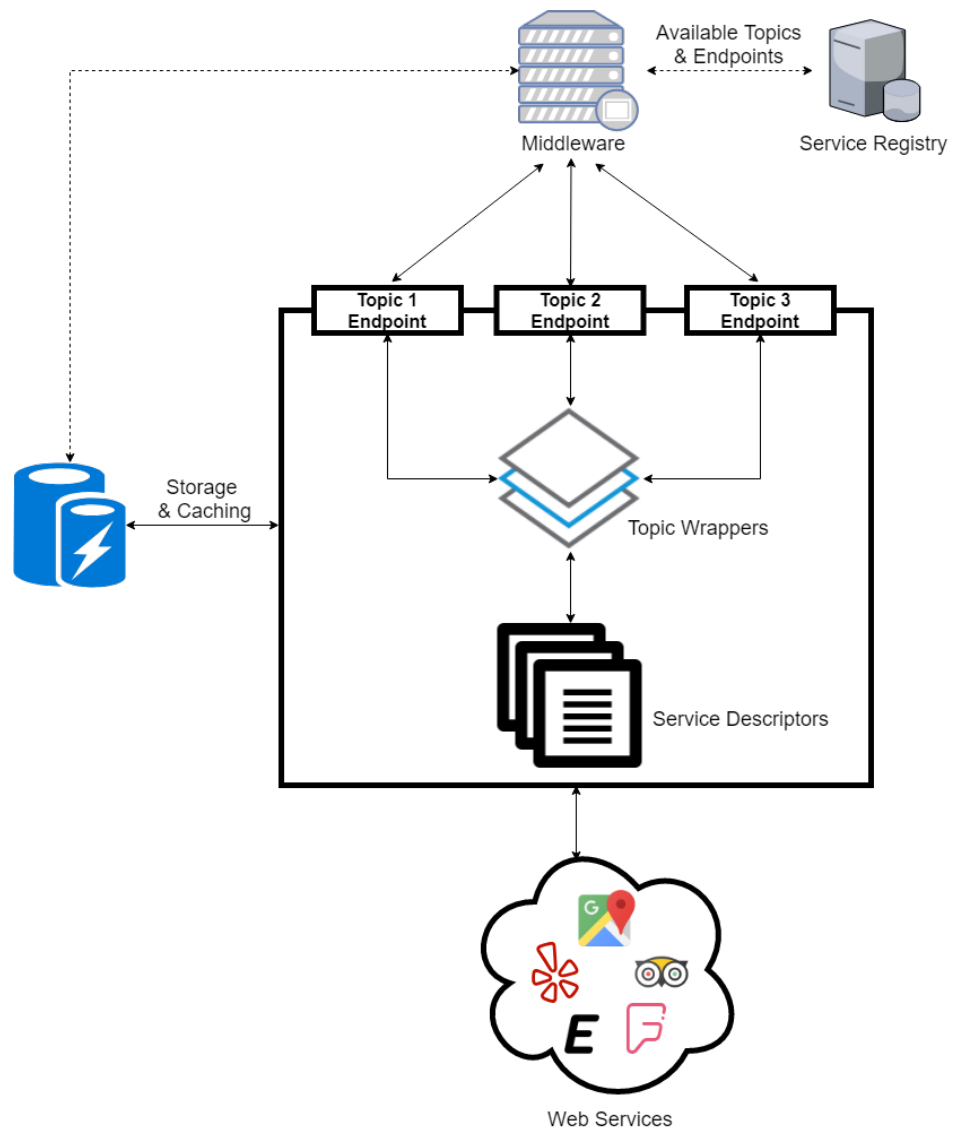


Figure 4.1: High-Level Architecture View

4.2.1 Topic Wrappers

The next step was to write code that is generic enough, allowing it to be easily adapted to any additional APIs we decide to add later on. Therefore, we decided to implement the concept of wrappers to achieve this task. In the book “Principles of Data Integration”, Doan et al.[6] state that a wrapper is the medium that communicates with data sources by sending them requests and then parsing the corresponding responses, usually converting these responses to a unified homogeneous format. A wrapper’s complexity varies greatly, depending on the data sources that it interfaces with. For instance, a wrapper would be relatively simple to construct if its purpose is to integrate two or more databases, since the mapping between the structures in these databases is static. Therefore, all that is required is to define the elements of interest, and how to extract them from the respective databases (i.e. where to find them, table name, column name and so on).

This process becomes more complicated when dealing with web services: the replies returned by web services usually differ greatly in structure, and even within the scope of one service, the reply structure might also depend on the request parameters that were initially sent. Therefore, a wrapper for web services should be able to dynamically accommodate for such variations, and still find the relevant information within the received replies.

Therefore, we needed to create a generic wrapper that is able to receive a request from the middleware, and using the service descriptors, adapt this generic request so that it can be sent to each of the relevant web services, thereby transforming the resource schema-based requests into service-specific requests. Then, the wrapper extracts the relevant information from the replies received from the web services, and integrates the resulting outputs into one structure that is based on the resource schema. Finally, this end result is finally sent back to the middleware. This process is illustrated in figure 4.2.

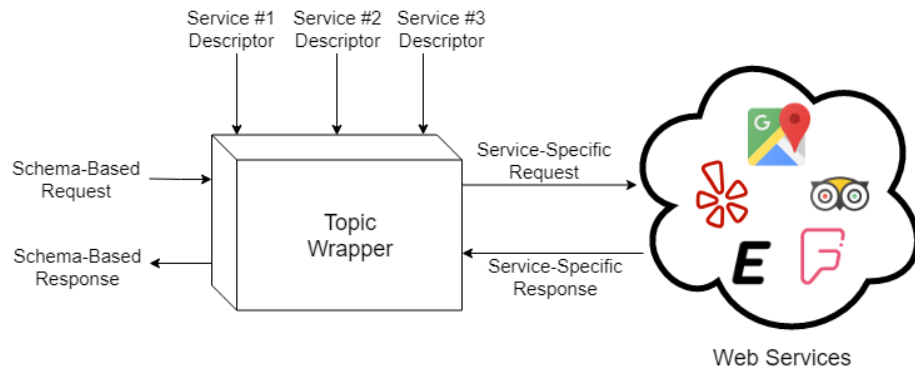


Figure 4.2: Topic Wrapper

In order to implement this functionality, we defined the structure of the objects we are querying for (i.e. restaurants and events), specified the fields that interest us, and defined the corresponding classes along with their constructors. The aim here is to extract the information we need from various APIs, and then use the constructors of these classes in order to create a homogenous array of objects that contains this information.

4.2.2 Service Descriptors

Sticking with the idea that we want our code to be generic and devoid of any service-specific details, we decided to include such details in external JSON files (descriptors). When needed, these files will be parsed in order to identify how to query said services, and what parameters to include in the requests.

The following figure shows a skeleton of a JSON service descriptor. The role of each object included in a JSON descriptor will be explained later on, but the point behind these descriptors is to supply the wrapper with knowledge on how to query the services, what parameters they expect, and how to parse their responses.

```

{
  "PreRequest": {
    .....
  },
  "MainRequest": {
    .....
  },
  "PostRequest": {
    .....
  },
  "ResponseParams": {
    .....
  },
  "CustomObject": {
    .....
  }
}

```

Figure 4.3: Service Descriptor Skeleton

4.2.3 Service Registry

Since the middleware is the component that creates and sends the initial request, we have to provide a way for the middleware to know what kind of resources are available, what are the required request parameters for each available API, as well as the structure of the response that the backend will generate, so that the middleware can parse these responses. To that, the ListServices API was created, and the response of this API is a list of all of the available services, along with the relevant details. The following is a sample response, containing only two services (restaurants and events). Whenever new services are added, a corresponding description should be added to this API's response.

```
{
  "restaurants": {
    "endpoint": "GETRestaurants.php",
    "params": {
      "city": "string",
      "query": "string",
      "lon": "float",
      "lat": "float"
    },
    "attributes": {
      "name": "string",
      "address": "string",
      "url": "string",
      "latitude": "float",
      "longitude": "float",
      "geohash": "string",
      "cuisines": "string",
      "thumb": "string",
      "phone": "string",
      "rating": "float"
    }
  },
  "events": {
    ...
  },
  "news": {
    ...
  },
  "weather": {
    ...
  }
}
```

4.2.4 Storage and Caching

After we developed a working system that can send requests to multiple services and integrate their responses, a testing phase took place in order to study the efficiency of this system and find ways to improve it. Right away, we noticed that querying multiple services and doing all of pre- and post-processing is a costly operation, especially in terms of running time. For instance, querying for music events that are taking place in Boston had an average running time of 30 seconds, during which we queried four different related APIs and performed all the necessary operations to integrate their responses. From a regular user's perspective, 30 seconds is a long wait for such a simple query, especially in the case where they want to perform multiple searches with different keywords.

To mediate this drawback, we decided to add caching functionalities, thus reducing the amount of requests we send out to the various web services. The caching will be done as shown in the following figure:

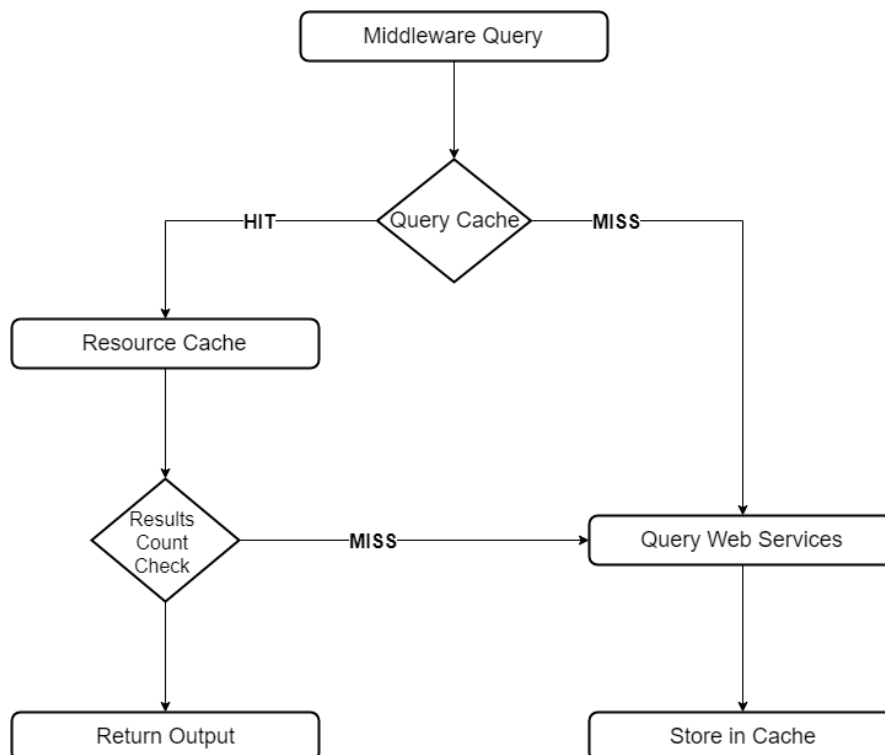


Figure 4.4: Caching Schema

The caching is done in two phases: at query time and at response time. The first phase, query caching, consists of saving the information related to every query that is received from the middleware. Query caching allows us to check whether a query similar to one that we are about to send now has already been issued within a certain (recent) timeframe. In case the cache check results in a miss, that means that no similar query has been issued recently, and therefore we must query the web services to retrieve the requested information. However, it is worthy to note that as the system scales and the number of users increases, the number of instances where the query cache will result in a miss should decrease noticeably.

In case the query cache check results in a hit, we can safely assume that a similar query has been sent. This is where the second phase of caching comes into play: it consists of saving the output of responses we receive when querying the web services. The information we store in the cache matches the structure of the resource schema, and this means that no further processing is needed whenever we query for this data later on.

The outcome of this process is a reduction in the number of requests we issue to the web services, which reduces the running time a lot since this is the actual bottleneck in our operation flow. Nonetheless, we still have to make sure that the cardinality of the response resulting from querying the second level of cache is above the required threshold. If that is not the case, we have to fall back again to querying the web services in order to update the resource cache with new data and refresh what already exists as well.

Chapter 5

Implementation and Evaluation

5.1 Implementation

5.1.1 API Wrappers

Building upon the generic wrapper concept, we decided to include the API-specific details of each web service in a JSON file associated with it. This means that all of the details concerning this service, such as the request URL, the authentication method and key, the expected URL parameters, and the format and fields of the response would be included in this JSON file. In a way, these JSON files are analogous with the XML descriptors that PerLa made use of to define the details of different web services. This proved to be a good approach because it allowed the actual code to be devoid of any API-specific code. This is an example of the details associated with the Yelp web service:

```
{
  "MainRequest": {
    "key": "Authorization: Bearer *Authentication Key*",
    "url": "https://api.yelp.com/v3/businesses/search?",
    "query": "term=",
    "city": "&location=",
    "lon": "&longitude=",
    "lat": "&latitude="
  },
  "ResponseParams": {
    "arrayLocation": "businesses",
```

```
    "name": "name",
    "address": "location.address1",
    "url": "url",
    "latitude": "coordinates.latitude",
    "longitude": "coordinates.longitude",
    "cuisines": "categories.0.title",
    "thumb": "image_url",
    "phone": "display_phone",
    "rating": "rating"
  }
}
```

Mandatory Objects

The JSON files include two mandatory objects, and three optional ones.

MainRequest: The first mandatory object is the *MainRequest* object. This object contains details needed to know how to send the main request to the web service, such as the authentication key, the URL, and the query parameters. In case the authentication is done through a URL parameter, it is not included as a separate attribute; instead, it is directly added to the corresponding URL. The presence of a “key” attribute means that the authentication details are included in the header of the request (as opposed to URL authentication) and the value of this attribute is the header that must be attached before sending the request.

ResponseParams: The second mandatory object is *ResponseParams*, and its job is to define the structure of the response, as well as where to find the values we want to extract. In normal cases, the “arrayLocation” attribute points to the location of the object array so that we can extract; this object array in the case of Eventbrite for example is the actual array containing the individual events returned by the web service. If the object array is nested within other objects, the path to the array is encoded as `objectName1:::objectName2(...)`, and the code handles traversing the response based on the provided `arrayLocation` attribute until it reaches the desired location in the response.

In order to facilitate parsing the responses, particularly when it comes to multi-level nesting, we opted to use a flattening function, which flat-

tens a multi-level array into a single level one. Moreover, the flattening function is implemented in a way that prevents loss of information, and that is guaranteed since the keys of the objects existing at deeper levels are simply concatenated to those of the parents objects, separated by a token. This means that even though we are flattening the array, we are still preserving the information concerning the keys and values of nested objects, as well as their nesting level.

Although the responses from various APIs share some similarity in the general structure, there are some that are quite different, and these differences have to be encoded and included in the corresponding JSON descriptors. For instance, Zomato's response is peculiar because the object array does not contain simple restaurant objects; instead, each restaurant is encapsulated within an additional JSON object. Therefore, we need to loop over the array and extract the restaurants from these objects. This is specified in the JSON descriptor as follows:

```
{
  "PreRequest": {
    "key": "user-key: *Authentication Key*",
    "url":
      "https://developers.zomato.com/api/v2.1/cities?q=",
    "result": "location_suggestions.0.id"
  },
  "MainRequest": {
    "key": "user-key: *Authentication Key*",
    "url": "https://developers.zomato.com/api/v2.1/search?",
    "query": "q=",
    "city": "&entity_type=city&entity_id=",
    "lon": "&lon=",
    "lat": "&lat="
  },
  "ResponseParams": {
    "loop": true,
    "preloop": "restaurants",
    "postloop": "restaurant",
    "name": "name",
    "address": "location.address",
    "url": "url",
    "latitude": "location.latitude",
```

```
    "longitude": "location.longitude",
    "cuisines": "cuisines",
    "thumb": "featured_image",
    "phone": "phone_numbers",
    "rating": "user_rating.aggregate_rating"
  }
}
```

The “loop” attribute specifies that a loop is required to extract the restaurant instances from the response, the “preloop” attribute corresponds to the name of the outer structure that we need to loop over, and the “post-loop” attribute represents the name of the inner object that contains the actual restaurant we want to extract.

Optional Objects

PreRequest and ***PostRequest***: Next comes the optional part of our JSON descriptors: *PreRequest* and *PostRequest*. The first object is responsible for any preliminary work that needs to be done before sending the requests to the web services, while the latter serves to perform all of the required post-processing on the received replies. Both of these objects are similar to some extent to the *MainRequest* object, but their purpose differs. These details are needed because some services are not as intuitive as others; in other terms, sometimes, more work is required in order to get the desired response from an API, as opposed to simply sending a single request.

For example, the Zomato search API is designed in such a way that it does not accept city names as search parameters. Instead, the search request should include a CityID parameter if the aim is to do a city-based search. In order to perform the mapping between city names and city IDs, a request should be sent to the City API before sending the main request. The reply of this preliminary request contains the city ID that corresponds to the supplied city name. This ID will be embedded in the query that we will send later on using the *MainRequest* object.

Similarly, the Eventbrite API requires additional work that must be done after receiving the response to the main request: Eventbrite returns a list of events, and each event object contains a venue ID linking to the venue where the event will take place. Since we are interested in pro-

viding the user with details about the venues, we are required to send an additional request, per event, to the Venues API. This request will have the Venue ID as parameter, and the response will contain details on the venue, such as the venue's name, address, and URL. This process is defined in the *PostRequest* object as shown below:

```
{
  "MainRequest": {
    "key": "Authorization: Bearer *Authentication Key*",
    "url": "https://www.eventbriteapi.com/v3/events/search?",
    "query": "q=",
    "city": "&location.address=",
    "lat": "&location.within=5km&location.latitude=",
    "lon": "&location.longitude="
  },
  "PostRequest": {
    "key": "Authorization: Bearer 26MMNKPXGH5HEISEKXPY",
    "scope": "item",
    "url": "https://www.eventbriteapi.com/v3/venues/",
    "param": "venue_id"
  },
  "ResponseParams": {
    "arrayLocation": "events",
    "title": "name.text",
    "venueName": "name",
    "venueAddress": "address.localized_address_display",
    "venueURL": "resource_uri",
    "eventURL": "url",
    "allDay": null,
    "startTime": "start.local",
    "endTime": "end.local",
    "longitude": "longitude",
    "latitude": "latitude"
  },
  "CustomObject": {
    "constructor": "$eventObject = new
      Event(isset($v['title'], $event[$v['title']]) ?
        $event[$v['title']] : '', isset($v['venueName'],
        $postObject[$v['venueName']]) ?
        $postObject[$v['venueName']] : '',
```

```

        isset($v['venueAddress'],
        $postObject[$v['venueAddress']]) ?
        $postObject[$v['venueAddress']] : '',
        isset($v['venueURL'], $postObject[$v['venueURL']]) ?
        $postObject[$v['venueURL']] : '',
        isset($v['eventURL'], $event[$v['eventURL']]) ?
        $event[$v['eventURL']] : '', isset($v['allDay'],
        $event[$v['allDay']]) ? $event[$v['allDay']] : '',
        isset($v['startTime'], $event[$v['startTime']]) ?
        $event[$v['startTime']] : '', isset($v['endTime'],
        $event[$v['endTime']]) ? $event[$v['endTime']] : '',
        isset($v['latitude'], $postObject[$v['latitude']]) ?
        $postObject[$v['latitude']] : '',
        isset($v['longitude'], $postObject[$v['longitude']])
        ? $postObject[$v['longitude']] : '');"
    }
}

```

This object is almost identical to the `PreRequest` object, with only one additional parameter “scope”. This parameter specifies whether the post-processing should happen on the response as a whole, or on an item-by-item basis. In the case of Eventbrite, since we need to get the venue details for each individual event, the scope is set to “item”.

CustomObject: The fifth and final object that can be included in the JSON descriptors is the *CustomObject* object. This object serves to provide a way to define a custom constructor when needed. Since we have information coming from two different sources (the Event object and the Venue object), we had to define a custom constructor for the Event object, as opposed to the generic one defined in the main code file. The custom constructor call defined in the *CustomObject* object is parsed using the PHP Eval method, which interprets a string as actual PHP code and executes it. This allowed us to circumvent the need to include two different constructors in the code (a generic one, and a second one specifically for Eventbrite). This object can be abstracted as an optional object that can include any service-specific code that we might need to execute in some special cases, therefore preserving the generality of the main wrapper code.

5.1.2 Storage and Caching

Caching Mechanisms

A cache, in its simplest form, is a hardware or software component that can store data, allowing future requests to be served faster. Since the data is already saved, the system wouldn't have to repeat the same process multiple times (an example of such processes would be performing complex calculations). Instead, it would simply return the result which is already stored in the cache.

In software, various methods exist when it comes to caching:

RDBMS: One of the most intuitive caching techniques simply consists of saving the query history and the received responses in a relational database. Relational database management systems (RDBMS) offer functionalities that go beyond simply storing data: they allow performing more complex operations such as filtering the data we're selecting, joining different tables in a database for a more comprehensive data selection, as well as some more advanced features such as creating triggers, views, and setting up live replication and recovery. Moreover, such databases are highly scalable, and can store virtually unlimited amounts of data since they rely on disk-based storage.

In-Memory Cache: Another valid option would be to use in-memory caching, a technology offered by tools such as Memcached, or the newer and improved counterpart Redis. These caching technologies follow a different paradigm, as they do not offer relational functionalities; instead, both Memcached and Redis are essentially key-value stores that rely on storing information in the RAM.

This is the most fundamental difference between the two alternatives: unlike disk-based storage, memory-based storage does not offer as much storage space as the former, since RAM capacity is much more limited than disk capacity. On the other hand, disks cannot offer the same read/write speed offered by RAM, and that is why Redis is blazingly fast: the I/O operations are non-blocking, and Redis can perform over 100,000 read/writes per second¹.

¹<https://static.simonwillison.net/static/2010/redis-tutorial/>

It is obvious that we have a tradeoff here between capacity and speed: a RDBMS makes more sense in terms of capacity, and Redis is the more logical option if speed is our primary concern and if we are dealing with somewhat limited amounts of data. In our case, we mentioned earlier that we have two different structures that we want to cache: we want to cache the queries, namely the category a query corresponds to, the location a query is issued from, the specific text linked to the query, and finally a timestamp representing when this query was issued. Using an in-memory cache for the queries turns out to be a good approach: the speed-space compromise works in our favor, the memory size is more than enough to store details about the queries, and we do not need any of the additional functionalities offered by relational databases.

This does not apply to storing the responses received from web services, since scalability in this case is a much more prevalent issue: one simple query can result in a response containing tens, or even hundreds of elements. As the number of queries increases, the amounts of data we have to save will also increase, and a Redis approach will not be able to keep up with the space requirements.

Moreover, we want to preserve the structure of the different objects when saving in the database, in a way that matches the resource schema. This cannot be done using a key-value database unless we find a way to encode the primary key of each object as the key of the key-value pair, and encode everything else, from the object-specific details to the various timestamps, as the value of the key-value pair. Performing such operations will add significant overhead, and this will become even more problematic when we need to decode this data. Finally, using a key-value cache to store objects will cause us to lose the join functionalities, which can be very beneficial when performing some complex operations. Therefore, a RDBMS seems to be the better option in this case.

The following diagram shows the caching schema, including details on which kind of cache is used for each component. It is identical to the diagram shown in figure 4.4, and follows the same flows, but it also includes the exact technologies we will use for each caching phase.

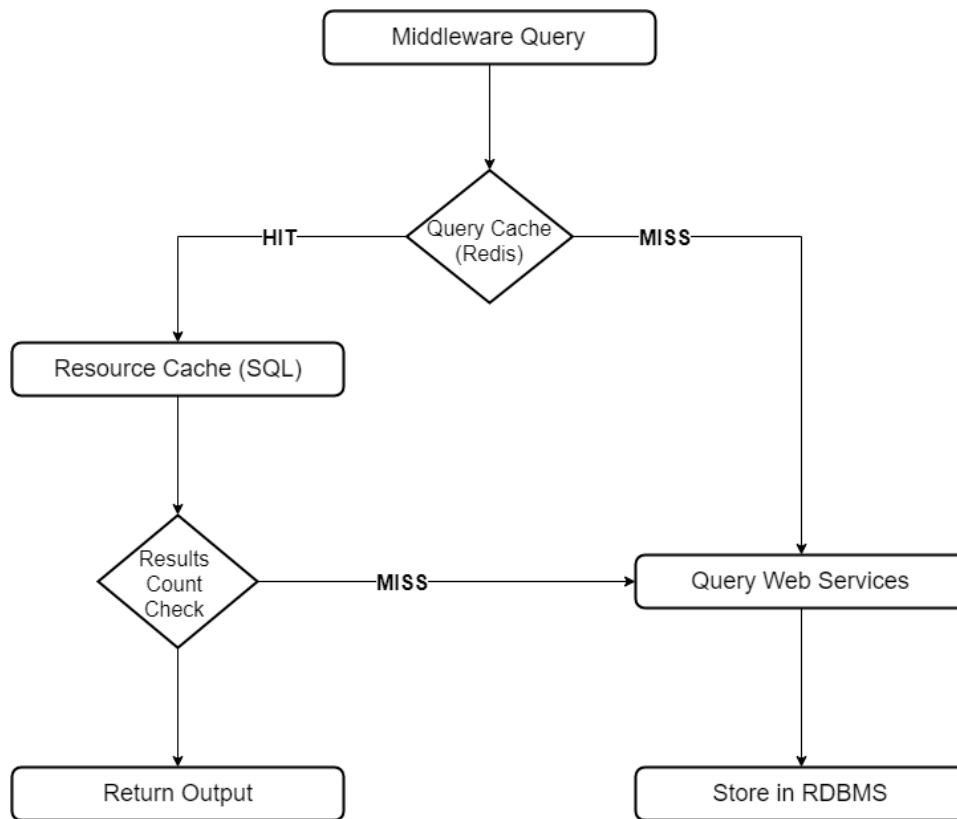


Figure 5.1: Caching Details

Query Caching

As we mentioned earlier, a query is identified by the following characteristics:

- Category of the requested option (corresponding to an object type in the resource schema).
- Location of a query, originally received from the middleware as (latitude, longitude) pair.
- Query text, as in the user-provided content of the query.
- Timestamp of when the query was initially sent.

The first three characteristics allow us to identify whether two queries are, in a sense, “identical”. Two queries are “identical” if they belong to the same category, have been issued from the same location (within a certain radius, and we take 2.4 kilometers as an example), and have

the same text (or a close synonym). Thus, we have to find a way to determine whether two location points are within a certain distance from one another on the fly, and whether two terms can be considered close synonyms.

For the first part of the key, e.g. the category, it is easy to distinguish whether two queries share the same category since this information is already provided by the middleware, and we simply have to include it as part of the key in the key-value pair.

In what follows, we try to explain the adopted strategies for establishing location equivalence and query text equivalence.

Location Equivalence

Geohash² is a public domain geocoding system which encodes a geographic location -initially represented as a (latitude, longitude) pair- into a short string of letters and digits. The longer the encoded string is, the more accurately a location is encoded. The following table represents the radius of an area represented by a Geohash string, with respect to the length of this string:

#	km
1	± 2500
2	± 630
3	± 78
4	± 20
5	± 2.4
6	± 0.61
7	± 0.076
8	± 0.019
9	± 0.0024
10	± 0.00060
11	± 0.000074

This means that if two Geohash strings start with the same character, then the two corresponding locations are at most 2500 kilometers away from one another. Applying the same idea to our case, and taking two query locations as an example, if the corresponding Geohash strings have the same 5-character prefix, then the distance between the two

²<http://geohash.org/>

locations of queries is less than or equal to 2.4 KM. In other words, a 5-character prefix corresponds to a circle of diameter 4.8 kilometers, and all queries that share the same 5-character prefix belong in the same circle. This makes our task much easier, since we now have a reliable way to distinguish whether two queries match in terms of location, by performing simple calculations.

Query Text Equivalence

So far, we have successfully encoded two of the three key characteristics as part of the Redis key, and the key is structured as follows:

Category:::Geohash

Example:

Restaurants:::drt2z

The next step would be to add the query text to the key. However, if we base our key checks on literal comparisons of various query texts, we would not be establishing a real equivalency relation. For instance, if a user performs a search for “music” events in an area, and another user shortly after that searches for “band”, there will be many common elements in the results of these two operations. Similarly, if a user searches for “Asian” restaurants, and another searches for “Chinese” restaurants in the same location, many restaurants will exist in both result lists. Therefore, we should establish some semantic equivalency measure between these terms. We want to be able to link two queries by comparing the text attached to them, from a semantic perspective, and decide whether they can be considered equivalent. There are multiple approaches that allow us to compare semantic resemblance between terms, and we will discuss in what follows.

- Word2vec

Word2vec³ is a tool developed by a team of researchers at Google in 2013. This tool uses shallow, two-layer neural networks as models, and these models are trained over large data sets in an attempt to find semantic and linguistic relations between words. Word2vec produces a vector space that represents the data set, each words being a vector in this space. The distance between two vectors is proportional to the semantic similarity between the two words associated with these vectors.

The code below calculates and prints the following information:

- Similarity between the words “first” and “second”.
- Similarity between the words “sushi” and “japanese”.
- The ten most similar words to “sushi”.

The training data set that we used is made up of the top English language pages on Wikipedia, and its size is around 1 Gigabyte.

```
import gensim, logging
import gensim.downloader as api

info = api.info()
# download the model and return as object ready for use
model = api.load("glove-wiki-gigaword-300")

print model.similarity("first", "second")
print model.similarity("sushi", "japanese")
print model.similarity("sushi", "music")
print model.wv.most_similar(positive="sushi")
```

The output of this program is as follows:

- Similarity between “first” and “second”: 0.8507271
- Similarity between “sushi” and “japanese”: 0.2515804
- Similarity between “sushi” and “music”: 0.07649763

³<https://code.google.com/archive/p/word2vec/>

– Ten most similar words to “sushi”:

- * sashimi
- * restaurant
- * restaurants
- * chefs
- * diners
- * seafood
- * chef
- * steak
- * sandwich
- * tempura

Looking at the output, we notice that there is a variable degree of accuracy in terms of semantic similarity; the similarity coefficient between “first” and “second” is 85%, while it is 25% for “sushi” and “japanese”, and 7% for “sushi” and “music”. At first glance, the output might make sense. However, for our purposes, Word2vec might not always offer the kind of semantic analysis that we are going for. For instance, Word2vec gives a high similarity coefficient between “sushi” and “burger”, since both of them are types of food. However, in our context, we want a very low similarity coefficient between these two terms. More investigation resulted in many more examples of cases where Word2vec alone actually leads us the wrong way. Therefore, even though it can provide valuable information in some contexts, we cannot solely rely on it, and had to look for other approaches.

- Database Query Text Associations

As mentioned earlier, we are storing in a relational database the content of the replies we are receiving from the various web services, after parsing them and matching the needed attributes with the resource schema objects. However, we are also saving information about the queries that led us to these results. An example of this is shown in the figure below:

source	query_lat	query_long	query_city	query_text	name
Yelp	42.3601	-71.0589	Boston	sushi,asian,japanese	Crudo
Yelp	42.3601	-71.0589	Boston	sushi,asian,japanese	PABU B
Yelp	42.3601	-71.0589	Boston	sushi,asian,chinese	Q Rest
Yelp	42.3601	-71.0589	Boston	sushi,asian,korean	Koy
Foursquare	42.3601	-71.0589	Boston	japanese,korean	Apollo I
Yelp	42.3601	-71.0589	Boston	sushi,japanese	O Ya
Yelp	42.3601	-71.0589	Boston	sushi,japanese	Douzo
Yelp	42.3601	-71.0589	Boston	asian,japanese	Shojo
Yelp	42.3601	-71.0589	Boston	asian,japanese	Oisa Ra

Figure 5.2: Sample of Query Metadata

For each received object (restaurants, in this case), we save the source (web service) that sent it, as well as the location and the text of the query associated with it. For example, the first result in the table is a restaurant called “Crudo”, and there are three different queries that returned this same restaurant as part of the result set: “sushi”, “asian”, and “japanese”. The third restaurant “Koy” is associated with queries having “sushi”, “asian”, and “korean” as query text.

The interesting point here is that since some query texts return the same restaurant, then there is a chance that the associated terms have some kind of correlation among them. This correlation might be a way to define the semantic similarity we are looking for. Therefore, once the number of users scales enough and we have more data, we might be able to mine for valid semantic associations between various terms.

- Manual Ontology Definition

The third approach to determine whether two queries are equivalent is by relying on static rules that are defined and saved in the form of domain ontology. The system administrator, along with someone who is experienced in the relevant domain, can define these rules and save them. While this might seem unfeasible due to the number of variations a word can have, we only need to save the root of the words manually, and link them with other semantically related roots.

For example, in the case of restaurants, the administrator would define through rules that “asian”, “japanese”, and “sushi” are linked. Likewise, there would be another rule linking “pizza”, “pasta”, and “italian”. After doing this, we can generate programmatically some variations of these words using string manipulation and regular expressions. In other words, terms like “japan”, “japanese food”, and “sushi bar” are generated automatically once the corresponding roots have been added to our semantic dictionary.

One last point worth mentioning is the following: we are aiming to develop CAMUS as a framework that can be adopted and implemented in as many fields as possible. The tourism scenario we have been using is just an example. The system however is being developed in a generic way that allows it to be beneficial in other domains as well. However, some of these domains might be much more sensitive and critical than tourism; some examples of that would be anything related to medicine or security. In those cases, any kind of association rules would anyway need to be looked over by a specialist from the field, and this even applies to structuring the corresponding CDT and resource schema. Therefore, the idea of manually setting up some rules, even if it is just the most basic and important ones, now becomes even more appropriate.

5.2 Evaluation

The evaluation process on one hand aims to confirm that the developed components are able to perform the needed functionalities successfully without any errors. On the other hand, the software development process is iterative, and therefore, there is always room to optimize and improve the performance of the various parts that make up our framework.

In this performance evaluation process, we focus on the parts that we worked the most on, namely the back-end system, and its communication with the middleware. We study different aspects in our system, such as its ability to include new web services given valid descriptors and the time it takes to retrieve and integrate data. Moreover, we focus on judging how advantageous introducing caching and storage mechanisms has been, as well as whether the theoretical design decision we made are supported by the results we get by doing experimental tests.

5.2.1 Adding New Services

Testing the flexibility of our algorithms, and the ability to add new services yielded satisfying results: our initial configuration included two main categories (restaurants and events). We also were integrating data from four different web services when searching for events, and from three when searching for restaurants. Adding a fourth web service to the restaurants API was easy and intuitive as we aimed for it to be: by generating and including the corresponding descriptor, the response received from this web service was integrated seamlessly with our previous ones.

Adding a new category (news) proved to be simple as well: we used the same code we had for the previous categories with very minimal changes, defined the new endpoint, and provided descriptors for three web services that provide news headlines. The new endpoint returned the expected response: a list of news headlines that integrates data received from all three new web services.

5.2.2 Caching and Storage Evaluation

One of the main reasons we introduced caching and storage is to reduce the amount of time required to fetch data from web services and integrate the responses. This process proved to be very time consuming,

and therefore we proposed this approach as a solution. In order to test the efficiency of our solution, we devised the following plan:

- Generate a test dataset made up of 1000 queries, issued from Milano. The queries follow a normal distribution in terms of location, having Milano Duomo as center, to keep the scenario as realistic as possible.
- The dataset includes entries for restaurants with various search queries (“sushi”, “burger”, “pizza” etc...).
- Choose a caching policy, and then forward the queries to the backend. Log the total running time, the standard deviation, and the variance for each of the policies.

The following map visualizes the geographical distribution of the queries.

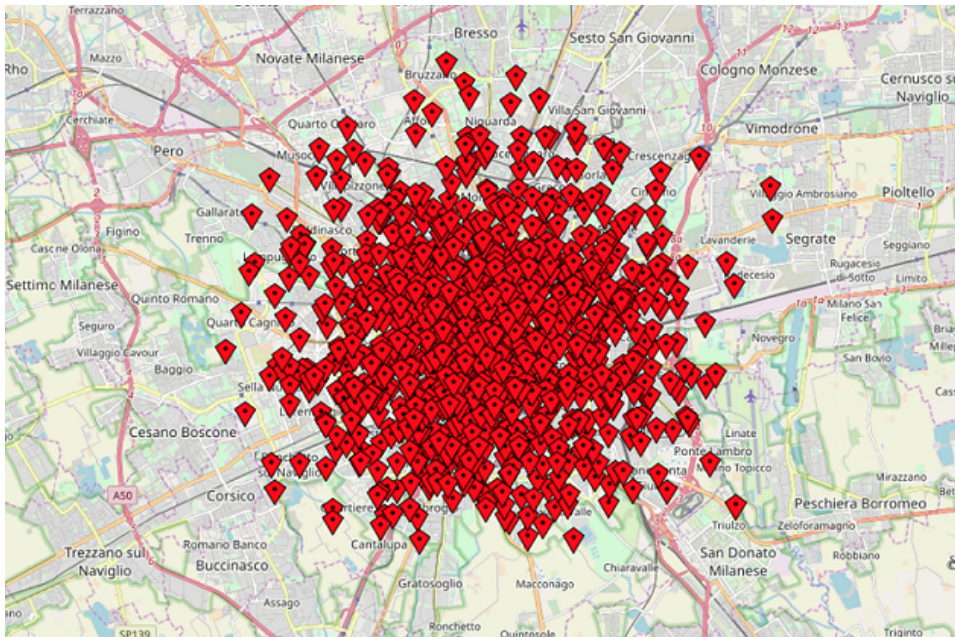


Figure 5.3: Geographical Query Distribution

Moreover, we ran the tests using four different caching policies:

- **NO CACHE:** This policy does not implement any kind of caching or storage. For every query, the middleware forwards the query to the backend through an endpoint, and the backend queries the corresponding web services and integrates their outputs.
- **DB RESOURCE CACHE:** Each time the backend integrates the responses received from web services, it stores the result in an SQL database, and it specifies the metadata that led to this result for each object (location and text of the corresponding query). For subsequent requests, the middleware tries to retrieve results from this same database, and only proceeds to forward requests to the backend in case the results retrieved from the database are deemed insufficient.
- **DB CACHE (RESOURCE + QUERIES):** Same as DB RESOURCE CACHE, but the backend also stores information about the queries (category, location, text, and time). Instead of querying the resource tables (which will scale exponentially as the number of users and queries increases), it simply checks whether an equivalent query has been issued within a certain timeframe. In theory, since the number of rows in a queries table is much smaller than that of a resource table, this process should be much faster.
- **DB CACHE (RESOURCE) + QUERIES CACHE (REDIS):** Same as the previous case, but save the queries in a REDIS cache instead of an SQL database. The key-value pairs follow the same patterns defined in section 5.1.2.

The following table summarizes the results of the tests for each caching policy.

Caching Policy \ Value	Average	Standard Deviation	Variance	Hit Ratio
NO CACHE	2.4911	2.5054	6.2772	N/A
RESOURCE CACHE (SQL)	0.3675	1.0087	1.0175	85.6%
RESOURCE CACHE (SQL) + QUERIES CACHE (SQL)	0.7833	1.430	1.3065	72.8%
RESOURCE CACHE (SQL) + QUERIES CACHE (REDIS)	0.4184	1.1545	1.333	83.6%

Table 5.1: Performance Evaluation Results

Right away, we can see the advantage gained by introducing caching. The average time per query (measured over 1000 queries) went down from 2.5 seconds to around 0.5 seconds for any caching type. Moreover, in accordance with our theoretical estimations, using REDIS has proved to be more efficient than using SQL databases as a query cache: this is where the in-memory storage and the non-blocking operations come into play.

One interesting observation is that caching the resources only in an SQL database has had better values compared to using both query and resource caching, even if minimally. However, this is probably because our resource cache started from an empty state, and therefore, towards the end of the test, it did not scale to the size that would match that of a real life scenario. The idea here is that the resource cache increases in size at a pace that is much higher than that of the query cache, since there are multiple resources associated with each query. Therefore, it becomes increasingly costly to go through the entirety of the resource cache and check whether it contains enough entries corresponding to the query and category we are looking for.

To sum up, the results we obtained after running these tests were encouraging, and they confirmed that we are on the right track with our current design decisions and overall architecture.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this work, we have proposed both a high-level design, as well as a low-level, detailed architecture of some of the components of the CAMUS system. The sheer number of details that should be examined and studied in this system, as well as the deep complexity surrounding some of them, renders finalizing all of them within the scope of this thesis nigh impossible. However, every small step gets us closer to having a fully operational framework.

We have tried to propose an architecture, focusing for the most part on the backend of CAMUS. We have studied some tools that allow us to perform data integration, and ended up developing our own solution, which offered us the biggest amount of freedom when it comes to optimization strategies. We have successfully implemented these designs, and the early tests we have performed seemed to be quite promising.

Our data integration model, along with the implementation, were able to fetch data from various heterogeneous sources and integrate the different outputs. Adding new services proved to be simple as well: if we follow the guidelines when adding a new service, and provide to the parsers the information pertaining to these new services (as descriptors), then the system will seamlessly integrate the new endpoints into the relevant wrapper, without any extra coding required whatsoever.

The storage and caching strategies were able to reduce greatly the running time required to perform a full cycle between the backend and the

web APIs. This cycle includes building the requests that need to be sent considering the representation of the current context, customizing them to match what the web APIs expect to receive in these requests, retrieve and parse the responses, and finally integrate them into one homogeneous structure. These operations are quite costly in terms of running time, which is a big constraint when we are considering mobile applications. We were able however to meet this constraint by adding query caching functionalities, as well as saving the end result of the data integration operations in our own database structure.

Finally yet importantly, we were able to adhere to the domain-independence paradigm when designing the various components: the data integration model, as well as the storage mechanisms, work no matter what kind of data we are dealing with, as long as suitable descriptors are provided. We want CAMUS to be a framework that can work for as wide of a variety of domains as possible. To that, the code should not depend on the kind of resources we are dealing with. Naturally, however, in some cases, a domain specialist might be needed to provide insight on how which attributes to include in the CDT and the resource schema. The technical implementation however does not depend on such factors.

6.2 Future Work

Having presented a solid foundation for the back-end component, work can now be done to integrate this component with the other elements. Once the work on the individual components is completed, the system as a whole can be tested to see the accuracy of our context model, and our data integration strategy can be also further improved. Concerning the backend, the resource schema structure is as of now hard-coded into the back-end component. However, it would be better if the resource schema is defined in an external descriptor, using a similar logic to what we did to describe the web services. Doing this will increase the modularity of the back-end, and it will facilitate editing existing resources and adding new ones.

Moreover, the current implementation lacks the user interfaces that would expose the various functionalities that are available. In particular, an interface that exposes that resource schema and the context dimension tree to the administrator -allowing him to make changes as

necessary- would remove the need to write manually the necessary code instead. This also applies to the interface that will be used by the domain expert to project the different factors that make up the end-user' context onto the context dimension tree; a suitable interface which shows the entirety of the context dimension tree without overwhelming the domain expert, and which allows easily editing and pruning needs to be developed.

The final interface in need of work is the actual application that will be used by the end-user. This interface represents the culmination of all of the work that is taking place behind the scene. It is also worthy to note that this interface is the *only* interface seen by the end-user, and therefore, it has to be intuitive, simple, and it has to adhere to the latest UI guidelines.

Additionally, in order to increase the usability and robustness of the system, a formal methodology should be proposed, and this methodology would clearly define the steps required to have the system up and running as intended. Doing this for every agent in the system help make sure that everyone knows their role, and what kind of input is expected from them, as well as how to supply this input. Once we have all of this finalized, we can move on to testing real-life scenarios, and only then would we be able to judge the accuracy of our model, and the efficiency of our components.

Bibliography

- [1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In Hans-W. Gellersen, editor, *Handheld and Ubiquitous Computing*, pages 304–307, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [2] Daniele Braga, Stefano Ceri, Florian Daniel, and Davide Martignghi. Optimization of multi-domain queries on the web. *Proc. VLDB Endow.*, 1(1):562–573, August 2008.
- [3] Peter J. Brown. The stick-e document: a framework for creating context-aware applications. In *Proceedings of EP’96, Palo Alto*, pages 259–272. also published in it EP-odd, January 1996.
- [4] Mario Casillo, Francesco Colace, Massimo De Santo, Saverio Lemma, and Marco Lombardi. A context-aware mobile solution for assisting tourists in a smart environment. In *HICSS*, 2017.
- [5] Florian Daniel, Maristella Matera, Elisa Quintarelli, Letizia Tanca, and Vittorio Zaccaria. Context-aware access to heterogeneous resources through on-the-fly mashups. In John Krogstie and Hajo A. Reijers, editors, *Advanced Information Systems Engineering*, pages 119–134, Cham, 2018. Springer International Publishing.
- [6] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.
- [7] Paul Dourish. What we talk about when we talk about context. *Personal Ubiquitous Comput.*, 8(1):19–30, February 2004.

- [8] R. Francese, M. Risi, G. Tortora, and M. Tucci. Visual mobile computing for mobile end-users. *IEEE Transactions on Mobile Computing*, 15(4):1033–1046, April 2016.
- [9] E. Lee and Hyung-Joo Joo. Developing lightweight context-aware service mashup applications. In *2013 15th International Conference on Advanced Communications Technology (ICACT)*, pages 1060–1064, Jan 2013.
- [10] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02*, pages 233–246, New York, NY, USA, 2002. ACM.
- [11] A. Miele, E. Quintarelli, E. Rabosio, and L. Tanca. Adapt: Automatic data personalization based on contextual preferences. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1234–1237, March 2014.
- [12] Rosslin John Robles and Tai-Hoon Kim. : Context aware tools for smart home development. 2010.
- [13] Tom Rodden, Keith Chervest, Nigel Davies, and Alan Dix. Exploiting context in hci design for mobile systems. In *in Workshop on Human Computer Interaction with Mobile Devices*, 1998.
- [14] Bill Schilit and Marvin Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8:22–32, 1994.
- [15] F. A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, and G. Rota. Perla: A language and middleware architecture for data management and integration in pervasive information systems. *IEEE Transactions on Software Engineering*, 38(2):478–496, March 2012.
- [16] Sakari Tamminen, Antti Oulasvirta, Kalle Toiskallio, and Anu Kankainen. Understanding mobile contexts. *Personal Ubiquitous Comput.*, 8(2):135–143, May 2004.

Appendix A

Code

The following code takes a JSON schema of a response received from web services, and generates the corresponding XML descriptor. The XML descriptor can then be interpreted by PerLa, and then used to communicate with this web service. The JSON schema is generated using one of the many publicly available schema generators, such as the one hosted on <https://github.com/gonvaled/jskemator>

```
import simplejson
from pprint import pprint
from collections import defaultdict, OrderedDict
from operator import itemgetter
import sys
import getopt

Elementary = {"integer", "float", "string", ""}
Basic = {"integer", "float", "string"}
value_object = defaultdict(list)
cleaned_value_object = OrderedDict()
parent_list = dict()
depth_array = dict()
parent = "root"
maxDepth = 0

def process_options(argv):
    filename = None
    try:
        opts, args = getopt.getopt(argv[1:], "f:s", ['filename='])
    except getopt.GetoptError, err:
```

```

# print help information and exit:
    print str(err)
    sys.exit(2)
for o, a in opts:
    if o in ("-f", "--filename"):
        filename = a
return filename

def generate_value(key, value):
    if key == "formattedAddress":
        print ''
    if value["type"] == "array":
        if value["properties"] and value["properties"][0]["type"] in
            Basic:
                return '<js:value name="' + key + '" type="' +
                    value["properties"][0]["type"] + '" qualifier="list" />'
        return '<js:value name="' + key + '" type="' + key + '"
            qualifier="list" />'
    elif value["type"] == "object":
        return '<js:value name="' + key + '" type="' + key + '" />'
    else:
        return '<js:value name="' + key + '" type="' + value["type"]
            + '" />'

def isElementary(o):
    if o["type"] in Elementary:
        return True
    elif o["type"] == "object":
        return False
    elif o["type"] == "array":
        i = o["properties"]
        if not i:
            return True
        return isElementary(i[0])

def isBaseObject(o):
    for key, value in o.iteritems():
        if value == None:
            value = dict()
            value["type"] = "string"
        if isElementary(value) or key in Elementary:
            continue
        else:
            return False, key
    return True, True

```

```

def getObject(o):
    if o["type"] == "array":
        return o["properties"][0]["properties"]
    else:
        return o["properties"]

def parseObject(o, parent, depth):
    depth = depth + 1
    global maxDepth
    if depth > maxDepth:
        maxDepth = depth
    if parent in parent_list:
        parent_list[parent] = parent_list[parent] + 1
    else:
        parent_list[parent] = 1
    parent = parent + ":::" + str(parent_list[parent])
    for key, value in o.iteritems():
        if key == "groups":
            print ''
        if key == "id":
            continue
        if value == None:
            value = dict()
            value["type"] = "string"
            o[key] = value
        if isElementary(value) or key in Elementary:
            value_object[parent].append(generate_value(key, value))
            continue
        val, k = isBaseObject(getObject(value))
        if val == True:
            parseObject(getObject(value), key, depth)
            Elementary.add(key)
            value_object[parent].append(generate_value(key, value))
        else:
            temp = getObject(getObject(value)[k])
            parseObject(temp, k, depth + 1)
            Elementary.add(k)
            parseObject(getObject(value), key, depth)
            value_object[parent].append(generate_value(key, value))
    depth_array[parent] = depth

def cleanupValues():
    for key, value in value_object.iteritems():

```

```

    if ":::" in key:
        k = key.split(":::")[0]
        if k in cleaned_value_object.keys():
            for val in value:
                if val not in cleaned_value_object[k]:
                    cleaned_value_object[k].append(val)
        else:
            cleaned_value_object[k] = value

def reorderValues():
    global cleaned_value_object
    for key, value in cleaned_value_object.iteritems():
        for key2, value2 in depth_array.iteritems():
            k = key2.split(":::")[0]
            if key == k:
                if type(cleaned_value_object[key][-1]) == str:
                    cleaned_value_object[key].append(value2)
                elif cleaned_value_object[key][1] < value2:
                    cleaned_value_object[key][1] = value2
    cleaned_value_object = sorted(cleaned_value_object.items(),
                                   key=lambda x: x[1][-1], reverse=True)

def printValues(valueArray):
    for value in valueArray:
        print "\t" + value

def generateXML():
    for key, value in cleaned_value_object:
        if key == None:
            print ''
        if key == "root":
            continue
        else:
            print '<js:object id="' + key + '">'
            printValues(value[:-1])
            print '</js:object>'
    print '<js:object id="root">'
    printValues(value_object.get("root:::1")[:-1])
    print '</js:object>'

def main():
    filename = process_options(sys.argv)
    with open(filename, 'r') as f:
        data = simplejson.load(f)
    data = data["properties"]

```



```
    parseObject(data, "root", 0)
    cleanupValues()
    reorderValues()
    generateXML()

if __name__ == '__main__':
    main()
```
