Applying REST principles on local client-side APIs

ROBERT OLSSON

Master's Thesis at CSC
Supervisor: Olov Engwall
Examiner: Olle Bälter

Abstract

In this thesis, the application of REST principles on local client-side APIs is explored. REST is a popular software architectural style designed with simplicity, scalability and generality in mind. The benefits and drawbacks of using REST over conventional styles of designing local APIs have been investigated, with a specific focus on libspotify. Libspotify is a C API that allows third-party developers to use Spotify's music streaming service. A local REST API was implemented on top of the libspotify C API. Software metrics were applied to the APIs, showing that the implementation was less decoupled than the equivalent C API. The metric results and relevant previous studies were used to analyze REST in local APIs. The main benefit was found to be the decoupling, leading to better evolvability and maintainability of an API. The main drawback is the additional work needed to model a REST API, and to make it more user friendly. The conclusion is that libspotify can benefit from REST, but that it might require more work than designing a conventional local API.

Sammanfattning

Applicering av REST-princper på lokala API:er på klientsidan

I det här examensarbetet utforskas användningen av REST-principer på lokala API:er. REST är en populär arkitekturstil för att utveckla mjukvara, designad för att vara enkel, skalbar och generell. För- och nackdelar med att använda REST istället för konventionella stilar att utveckla API:er har utforskats, med fokus på libspotify. Libspotify är ett C-API som låter tredjepartsutvecklare använda Spotifys musikströmningstjänst. Ett lokalt REST-API implementerades ovanpå libspotifys C-API. Metoder för att mäta API:ernas egenskaper användes, som visade att det existerande API:t hade ett större beroende mellan mjukvarukomponenter än REST-API:t. Mätresultaten och tidigare relevanta studier användes för att analysera REST i lokala API:er. Den huvudsakliga fördelen visade sig vara det minskade beroendet mellan mjukvarukomponenter, vilket leder till att API:t blir lättare att utveckla och underhålla. Den största nackdelen är det extra arbete som krävs för att dels modellera ett REST-API, och dels göra det mer användarvänligt. Slutsatsen är att libspotify kan dra nytta utav REST, men att det kan innebära mer arbete än att designa ett konventionellt lokalt API.

Contents

Chapter 1

Introduction

This chapter introduces the thesis by presenting the background and the problem statement.

1.1 Introduction

In software development, it is common to separate a system into different smaller components. Components are usually separated in such a way that each component handles a distinct concern. A typical piece of client-side software could consist of components related to the graphical user interface, the logic, network communication, etc. Separating a system into components can give many benefits, such as reusability of parts of the software and better scalability. An application programming interface (API) is a set of standards for how such software components communicate with each other [17]. APIs in object-oriented programming languages like C++ and Java commonly consist of a set of exposed classes and class methods along with their definitions. APIs in procedural languages like C, similarly consist of functions and function definitions. In such languages communication between different components is thus done by function calls.

REST stands for REpresentational State Transfer and is a software architecture style that has become very common in web services. It is targeted at distributed information systems and has been designed with simplicity, scalability and generality in mind. REST APIs with their resource oriented nature differ in many ways from how local APIs are commonly designed [18].

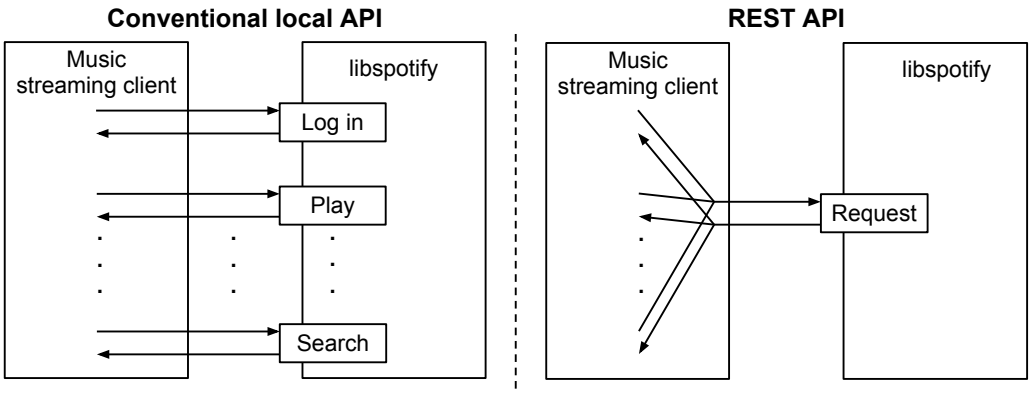

Figure 1.1. An illustration of the uniform way of performing API requests in REST, compared to a common local API.

One difference is the way that requests are made, which is illustrated in fig. 1.1. In common local APIs, API requests are usually made by performing hard-coded function calls, specific to the component communicated with. A component that wants to initiate communication simply calls a specific function in another component, which in turn can respond by returning a value. In REST, requests are instead constructed in the form of self-descriptive messages, that are sent through a common uniform interface [18].

1.2 Problem statement

This thesis will look into whether the benefits of REST that has made it popular among web based APIs can also be applied to local APIs on the client side. The goal is to investigate what parts of REST could be beneficial to apply to local APIs. To do this, a local REST API will be implemented on top of an already existing common local API. The benefits and drawbacks of the REST API over the original one will then be investigated. The thesis will also present the implemented API to give an idea of how a local REST API could look in practice.

1.3 Scope limitations

Local APIs come in several different shapes and forms, and there are many aspects of an API to consider. To limit and further define the scope of this project, the investigation will be based on the libspotify API. Libspotify is a local C API that enables third-party developers to write applications that use the Spotify music streaming service [9]. The implementation presented will be a REST API around libspotify. The hope is still that the results of the work will be relevant in a more general context than libspotify only.

1.4 Terminology and abbreviations

API
: Application Programming Interface, a set of standards for how different software components communicate with each other.

HTTP
: HyperText Transfer Protocol, an application protocol for distributed hypermedia systems used by the World Wide Web.

JSON
: JavaScript Object Notation, a lightweight text format for the serialization of structured data.

REST
: REpresentational State Transfer, a software architecture style targeted at distributed information systems.

RESTful
: An API is said to be RESTful if it conforms with the principles of REST.

URI
: Uniform Resource Identifier, a formatted string that identifies a resource.

URL
: Uniform Resource Locator, a type of URI used to identify resources in HTTP.

Chapter 2

Background

This chapter presents the background theory needed to understand the presented implementation and analysis. APIs and REST are explained, along with different software metrics used to measure properties of software. Relevant previous work is also presented.

2.1 API

Modern software applications are typically built on several APIs. An API could be seen as a problem abstraction. It specifies how software components should interact with other software components that provide a solution to that problem [13]. Libspotify is such an API, which implements a solution for applications wanting to stream music from Spotify's music streaming service. Libspotify in turn uses several other APIs to help provide its functionality, e.g. libogg and libvorbis to decode compressed audio streams [9]. The purpose of an API is thus to provide a simple logical interface to a component and its functionality, while hiding irrelevant implementation details. In local contexts, these components are often distributed as software libraries, intended to be usable by many different applications [13]. Figure 2.1 illustrates, in the form of a graph, how a music streaming client using libspotify could depend on various APIs. The nodes in the graph represent the different software components, and the edges connecting the components represent communication channels. Each edge thus also corresponds to an API.

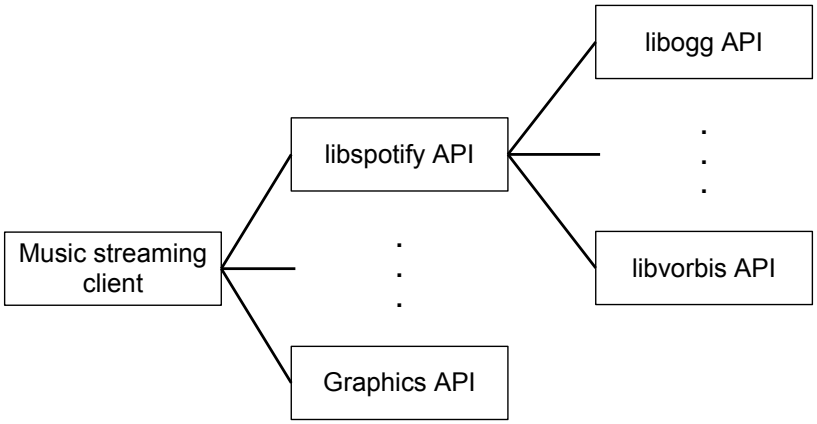

Figure 2.1. Possible API dependencies for a music streaming client using libspotify.

2.2 REST

REST is a software architecture style targeted at distributed information systems. In other words, it is targeted at systems where all components are not on the same physical machine, and have to communicate through a network. REST is defined by Roy Fielding in his dissertation [5], which is used as the main theoretical REST reference. It goes through the constraints of REST and the properties they induce.

2.2.1 Principles

Resources and representations are two key aspects of REST. Information is modeled as resources, which can basically be any concept that can be named. A resource could be a document, a person, a list of resources or even non-static concepts like "The most played songs in Sweden today". Representations are used to transfer resource states between different components, e.g. from a server to a client. A representation is just a sequence of bytes, e.g. in a specific document or image format, along with metadata describing the bytes. The representation types used are negotiated upon by the client and server [5].

2.2.2 Constraints

REST is formally described as a set of constraints applied to elements within the architecture. Because REST is an architectural style, the constraints are high-level and do not specify implementation-specific details such as specific protocols or file formats [5].

2.2.2.1 Client-server

The functionality of a server and a client should be separated. This improves portability of the client as it does not have to be concerned of server-specific concerns as data storage. Consequently the server does not have to care about the user interface, simplifying the server and improving scalability. Another benefit of this separation of concerns is that components can evolve independently [5].

2.2.2.2 Stateless

The nature of the communication between clients and servers must be stateless. No client state should be stored on the server, instead clients should include all information needed to process a request. This constraint increases server scalability and simplicity as servers do not have to store any state between requests. One disadvantage with this constraint is that it can decrease communication performance between components as clients might have to send more repetitive data. It also puts more responsibility on clients to ensure consistent application behavior [5].

2.2.2.3 Cache

Data within a server response must specify, implicitly or explicitly, whether it is cacheable or not. If a response is cacheable, a client is allowed to reuse that data for later requests. This can improve efficiency, as some interactions between the client and the server can potentially be eliminated. One disadvantage with caching is that reliability might decrease as stale cached data can differ from data on the server [5].

2.2.2.4 Uniform interface

The interface between components should be uniform to simplify overall system architecture. REST defines four interface constraints for this. Uniformity can potentially decrease efficiency as information has to be transferred in a standardized form instead of a more optimized application-specific form [5].

Identification of resources Each distinct resource has an identifier to identify it during interaction between components [18]. This could e.g. be a URI. This principle is also used on the web and is implemented in HTTP where all resources like web pages, images and other documents are identified using distinct URLs.

Manipulation of resources through representations Clients update and add resources on the server by sending representations [18]. When a client wants to e.g. create, change or delete a resource on a server, it creates and sends a representation indicating the desired state of the resource.

Self-descriptive messages Messages between components not only contain data, but also metadata describing how to process the contents of the message. Typical metadata could be the URI of the resource that the representation is representing, or the type of the representation, e.g. what file format it is in. Messages also contain control data defining the purpose of a message. A client could specify what action it wants to perform on a resource (whether it wants to create a new resource, update or delete an existing resource etc.) [18].

Hypermedia As The Engine Of Application State (HATEOAS) Except for a known initial resource identifier, the client only changes states by choosing between state transitions provided by the server in received representations. This is how the user interface of a web browser usually works; a user navigates to a known bookmark and then navigates between web pages and performs actions by clicking on links or submitting forms provided by the web server [18].

2.2.2.5 Layered system

Components should only see other components within the layer that they are interacting with. This constraint limits the complexity of the system. Together with the uniform interface constraint, it also enables the creation of components between server and clients without changing the interface between them. Such intermediate components can process and transform data passing through them, and can e.g. act as proxies, gateways or firewalls. The main disadvantage of using layered systems are increased overhead and latency as data is processed in several layers [5].

2.2.2.6 Code-on-demand

A client can receive and execute code from a server, enabling extension of the client. This can simplify clients and improve system extensibility, as the number of features that have to be implemented on the client is reduced. A web browser can e.g. download and execute JavaScript from a web server. Code-on-demand is the only optional constraint of REST [5].

2.2.3 REST in practice

Although REST does not specify any specific protocols, HTTP is the most commonly used application protocol in conjunction with REST. This has led to a common misconception that any HTTP-based interface is a REST API, which is not always the case. A typical pitfall when designing a REST API is to break the HATEOAS constraint by hard-coding URIs (other than the URI for the API entry point) on the client side. By blindly assuming that certain resources exist with

static URIs, a server loses control of the client's available state transitions and the URI namespace [6].

When using HTTP, resources are identified by URLs. Actions are performed on these resources in a uniform way by using a set of predefined verbs like *GET*, *POST*, *PUT* and *DELETE*. HTTP messages also contain headers that allow descriptions of the representations that they contain. HTTP headers can contain information about media type, cacheability, client state in the form of cookies, etc. [7].

2.3 Software metrics

Software metrics are used to measure different properties of software in a quantifiable way. Metrics are important in all sciences, as they can provide objective and reproducible measurements. Continuous effort in computer science is thus put into bringing such approaches to software development [19].

2.3.1 Coupling

Coupling is a measure of dependence between components in a software system. Coupling has been correlated to maintainability, traceability and robustness in software. High levels of coupling have been associated with greater design effort, greater rework and lower productivity. There exists metrics to quantify the amount of coupling between software components, which could assist when comparing the coupling of different APIs [1].

Fenton and Melton have defined some different types of coupling between two software modules. The types are ordered from the highest and most undesirable levels of coupling, down to type 0, which is no coupling at all. They suggest the coupling metric shown in eq. (2.1), to measure the coupling between two modules x and y. i is the greatest coupling type between the modules, and n is the number of interconnections between the modules. A higher value of $M(x, y)$ indicates a higher coupling between x and y [4].

$$M(x, y) = i + \frac{n}{n + 1} \tag{2.1}$$

2.3.1.1 Content coupling (level 5)

Content coupling is the strongest type of coupling according to Fenton and Melton's metric. Two modules x and y are content coupled if x refers to the inside of y, i.e. it branches into or changes data in y [4]. Listing 2.1 shows an example of two modules represented as two C++ classes. x communicates with the class y by directly modifying the variable A that is internal to y, making the two modules content coupled.

```
class x {                          class y {
  void B() {                         static int A;
    y.A = 5;
  }                                  ...
};                                 };
```

Listing 2.1. An example of content coupling between two classes in C++.

2.3.1.2 Common coupling (level 4)

Common coupling is when two modules x and y share global data [4]. Listing 2.2 is an example of common coupling between two classes. The class y writes to the global variable A, and x reads the same variable.

```
int A;

class x {                          class y {
  void B() {                         void C() {
    int D = A;                         A = 5;
    ...                              }
  }                                };
};
```

Listing 2.2. An example of common coupling between two classes in C++.

2.3.1.3 Control coupling (level 3)

Control coupling means that a module x communicates with another module y by passing parameters, with the intention of controlling the behavior of y [4]. Listing 2.3 shows an example of control coupling. The class x communicates with the class y by calling the function B. The value of the function parameter determines the behavior of y, as the parameter is evaluated in an *if* statement.

```
class x {                          class y {
  void A() {                         static void B(bool a) {
    y.B(false);                        if (a == true) {
  }                                      ...
}                                      } else {
                                         ...
                                       }
                                   }
                                 }
```

Listing 2.3. An example of control coupling between two classes in C++.

2.3.1.4 Stamp coupling (level 2)

If a module x communicates with another module y by passing parameters in the form of data structures, they are stamp coupled [4]. Listing 2.4 is an example of stamp coupling between two classes. The class x calls a function in the class y with a structure as parameter.

```
class x {                          class y {
  Person p;                          static void B(Person p) {
                                       float age = p.age;
  void A() {                           ...
    y.B(p);                          }
  }                                  }
}
```

Listing 2.4. An example of stamp coupling between two classes in C++.

2.3.1.5 Data coupling (level 1)

Data coupling means that a module x communicates with another module y by passing parameters that consist of single data elements or a homogeneous set of data that do not include any control elements [4]. Listing 2.5 shows an example of this, where the class x communicates with y by passing a *float* parameter. The behavior of y is not controlled by the parameter, as it is only used in a calculation.

```
class x {                          class y {
  void A() {                         static void B(float a) {
    float value = y.B(9);              return a * a;
    ...                              }
  }                                  }
}
```

Listing 2.5. An example of data coupling between two classes in C++.

2.3.1.6 No coupling (level 0)

Level 0 is the lowest level of coupling between two modules. It indicates that the two modules do not communicate at all and are totally independent [4].

2.3.2 Performance

While coupling is related to the overall system design, performance is specific to runtime. Performance is an indication of a system's responsiveness, and is measured in the terms of either throughput or latency. Throughput is a measure of how many events that occur in a given amount of time. Latency is a measure of the time it

takes to respond to any event. The performance of an application can affect its scalability [10].

2.4 Related implementations

Some related implementations were found in the form of frameworks and protocols that implement REST or similar principles.

2.4.1 Resource-Oriented Computing

Resource-oriented computing (ROC) is a model for designing software systems derived from research started at Hewlett-Packard Laboratories and later continued by 1060 Research [8]. It is inspired by the characteristics of Unix systems and the World Wide Web. ROC revolves around the concept of abstract resources referenced by identifiers and has much in common with REST. It is meant to complement object-oriented programming rather than to replace it [2]. Different tiers of an architecture are interconnected using logical channels instead of using physical memory references to each other.

NetKernel is a ROC implementation in the form of a ROC platform [8]. Because of the similarities between ROC and REST, NetKernel will be of interest for the implementation.

2.4.2 Restlet

Restlet is a REST framework for Java that provides a set of classes and interfaces to serve as a foundation for RESTful applications [15]. Although it is mainly targeted at web applications, it abstracts away the protocol that connects the components from the actual API. Apart from HTTP, Restlet provides several communication protocols, including local protocols such as Restlet Internal Access Protocol (RIAP). RIAP uses internals calls to communicate between components, enabling an entire RESTful system with components to run and communicate locally within one physical machine.

2.4.3 HTTP

Hypertext Transfer Protocol (HTTP) is an application protocol for distributed hypermedia systems and is used by the World Wide Web. REST and HTTP have some shared history, as Roy Fielding, the introducer of REST, also is one of the authors for the HTTP specification [7]. HTTP implements many of the properties needed for REST, so when used correctly HTTP can be used to design RESTful APIs. Although HTTP is web based, many of its principles should also be applicable to local contexts.

Chapter 3

Methodology

This chapter presents the methodology used to accomplish the goal of the project.

The first stage of the project consisted of research on the theory behind REST, on related implementations and on similar studies. The second stage was to implement a local REST API around libspotify. The third and last stage was to analyze the benefits and drawbacks of REST in local contexts.

The work of the different stages overlapped and was carried out iteratively. As the implementation and analysis progressed, new sources were looked up as needed. Because local REST APIs is quite an unexplored area, it was hard to anticipate what sources were needed beforehand. This also allowed for implementation improvements, and to find more relevant sources.

3.1 Implementation

The implementation that was done was based on the related implementations found. There were two main goals with the implementation. The first goal was to present how a local API could be implemented. The second goal was to aid the analysis. The implementation provided a tangible local API that was used for comparison purposes. The experiences gained from creating the local implementation was compared to what is presented in the REST study by Pautasso, Zimmermann and Leymann, where they compare REST to other web service standards from an architectural perspective [12].

3.2 Analysis

The comparison and analysis is mainly based on the similar studies found. These studies cover different aspects of REST, including architectural aspects [12], coupling [3], and various areas mentioned by Fielding in his dissertation [5]. Arguments presented in these studies that are applicable to the problem statement will be

considered, and the focus will lie on areas that are mentioned by several sources. Because REST is mainly an architecture designed for distributed systems, most of the existing studies do not focus on local REST interfaces. To help verify that the found arguments also hold in a local context, experiences from the implementation will also be considered and compared to the studies.

3.3 Metrics

Software metrics were applied where possible to objectively compare the libspotify C API and the implemented local libspotify REST API. Not all aspects of software are easily quantifiable, but metrics for the areas covered in the analysis that were found were used.

3.3.1 Coupling

The uniform interface and the decoupling is one of the benefits of REST mentioned both by Fielding, and other studies found [11, 12, 16]. To quantify the amount of coupling in the local REST API and C API, coupling was one of the software metrics chosen. The coupling metric suggested by Fenton and Melton was applied to the APIs, as that metric was found to be widely cited. Another considered coupling metric was the well-cited metric suggested by Dhama [3]. Dhama's coupling metric considers more variables than Fenton and Melton's, like the number of different parameter and variable types connecting different software modules. It was deemed too complex and time consuming for this purpose, as it would require analysis of the libspotify C API implementation.

As only part of libspotify was implemented in the REST API, the existing libspotify C API is much more extensive than the REST API. Only the parts common to both APIs were thus considered in the metric to get a more fair result.

3.3.2 Performance

Potential performance drawbacks introduced by some REST constraints is mentioned by Fielding [5]. To measure how significant the overhead introduced by the local REST API is, it was measured and compared to the C API. Because the latency introduced by the REST layer is so small that it might be hard to measure with reasonable precision, the throughput was measured. To isolate the REST layer, no libspotify calls were included in this metric. Including libspotify calls could add uncertainties in the form of network delays, cached responses etc. Instead, a dummy API was set up using the REST implementation, with requests of varying processing times, together with an equivalent C API. Ideally, the throughput for the API requests should be the inverse of the processing time of the requests. In practice, processing overhead in the API layer will affect the throughput for both APIs.

Throughput of requests with processing times between 0.01 ms and 10 ms were chosen. The processing time was simulated using a sleep function. The lower bound, 0.01 ms, was chosen because of the limited precision of the sleep function. Testing processing times over 10 ms was not deemed to be necessary, as by that point the processing time would dominate the throughput, and the API overhead would be relatively insignificant. Each request was called as many times as possible in a time frame of about 10 seconds. The call count was divided by the time taken, giving an average value for how many requests that could be called each second. Because background processes running on the test computer could affect the results, the measurement was repeated five times for each API, and the values were averaged.

Chapter 4

Implementation

This chapter presents the implementation of the libspotify REST API. The different parts of the API are explained, and an example of how the API can be used is shown.

4.1 The existing API

The existing libspotify C API is split up into modules. Different modules implement different parts of the API, and they each consist of a set of functions and data structures [9]. The modules that have been used during the REST API implementation are presented in table 4.1.

Module	Used functions	Used structures/types
Session	sp_session_create sp_session_login sp_session_logout sp_session_player_load sp_session_player_play sp_session_process_events sp_session_user sp_session_userdata	sp_session sp_audioformat sp_session_callbacks sp_session_config
Track	sp_track_add_ref sp_track_is_loaded	sp_track
Link	sp_link_as_track sp_link_create_from_string sp_link_release	sp_link
Error	sp_error_message	sp_error

Table 4.1. The different functions and structures of libspotify used in the REST API implementation.

The session module contains the *sp_session* structure that represents a session, and functions like *sp_create_session*, *sp_session_login* and *sp_session_player_play*

to manage a session. The track module contains the *sp_track* structure which represents a track, that can e.g. be played using functions from the session module. The link module contains the *sp_link* structure which represents a Spotify URI, which can be a reference to a track, album, artist, etc. The error module contains the *sp_error* type, which represents an error. The different functions throughout the libspotify API return an *sp_error*, to indicate whether an error occurred, and what the error was. Other modules exist for managing other parts of libspotify, like playlists, albums, toplists, etc. [9].

4.2 Overview

The implementation is written in C++. Internally the API utilizes object-oriented programming, like any other C++-based API. Externally, the interface that is exposed is a general REST interface consisting of two C++ methods through which API requests are performed. The mindset during the implementation of the API has been to reuse existing standards and technology when possible, to focus the time on areas where established standards do not exist. HTTP has been the main inspiration for the API. The representation structure and the uniform way of performing actions on resources using a set of predefined verbs are based on HTTP.

4.3 Representations

A representation is stored in a general representation class that contains a header and a data part. Components communicate with each other by passing around such objects. The header is a dictionary that maps field names to values. The idea is to make the messages self-descriptive to conform with REST. Some standard header fields have been predefined. The *URI* header field specifies the resource that is being represented or manipulated by the representation. The *Status* field specifies the result of a client request, i.e. whether the request was successfully completed or if an error occurred.

The representation class can be extended to simplify management of different representation types. Much of the data passed from and to the API is JSON-based. JSON is a common lightweight format for representing structured data in web services. A JSON representation class was implemented to make it easier to manipulate such representations. A component is not required to know of all the different representation types and subclasses of the representation class, as all representations are serializable and have a uniform way of being represented. This is usable for components that need to handle representations but not necessarily understand the data they contain, like routers or caches.

4.3.1 Methods

To provide a uniform way of performing actions on resources, the representation header field *Method* is used to specify an action in client requests. The methods defined are the verbs *CREATE*, *READ*, *UPDATE* and *DELETE*. These are similar to the verbs used in HTTP, but they have been renamed to make their purpose clearer. *CREATE* is a method used to create a new resource on the server at the given resource URI, or as a subordinate of that resource. *READ* is used to retrieve a resource, and should not have any side-effects, to ensure stateless communication. *UPDATE* is used to replace an existing resource at the given URI. *DELETE* is used to delete the resource at the given URI.

4.3.2 Types

Each representation should have the header field *Type*, specifying the type of the representation data. The format of a type follows the same standard as media types in HTTP. The type for JSON data is e.g. *application/json*. Types not registered with the Internet Assigned Number Authority (IANA) should be prefixed with *x-*, e.g. *application/x-spotify-track+json*, which is the type for a track in the implementation.

4.4 Interface

Each server component in the API implements a general connector interface. The interface provides two methods for clients to perform requests on a component. These methods are uniform across the entire API. Both methods take a representation object, the request. One of the methods is used to perform asynchronous requests, and also takes a callback function that is called when the response is available. The other method performs blocking requests and returns the response representation when it is available. These methods provide a uniform interface and enables the creation of layered systems. A client communicates with a connector and does not care whether the component associated with it is a server component, or if it happens to be an intermediate proxy or cache.

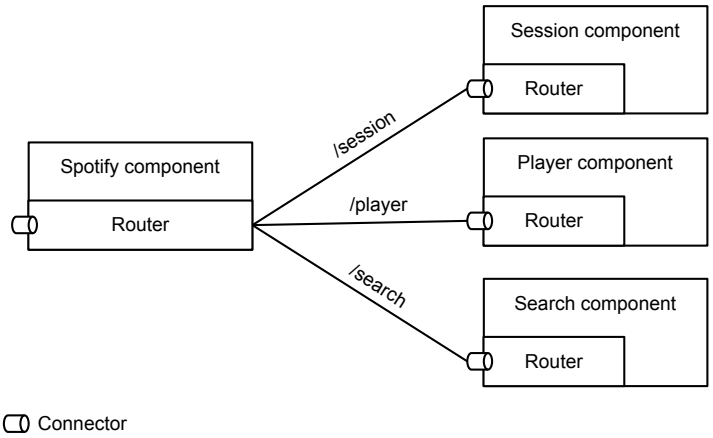

Figure 4.1. A process view of the API, showing the different components and the top level URI space.

To improve modularity, different parts of the API are separated into different internal components, e.g. a session component that manages login and a player component that manages music playback. To give a client a single interface to communicate with several server components like these, a router component was implemented. Figure 4.1 illustrates this in the form of a general process view of the API. A router holds pointers to several connectors along with their corresponding URI address spaces, but provides only one connector for clients. The router inspects the URI header field of each incoming request and routes the request to the correct connector. This is similar to how a network router works with its routing table. Routers are also used by components to manage their local address space. In that case address spaces are mapped to methods within the component rather than connectors. The router was inspired by Restlet, that also uses a similar router solution to delegate requests [14].

4.5 Model

Libspotify is a typical C API, consisting of function and structure definitions. This model was turned into a resource-oriented model. The functionality provided by the functions was mapped to resources and the predefined set of uniform methods as shown in table 4.2. Generally the different resources correspond to different modules of libspotify, e.g. session management, player and search. Functions in the C API corresponding to just accessing data are implemented using the *READ* method. Functions modifying state are implemented using *UPDATE* or *DELETE*.

URI	Method	Corresponding calls in C API
/session/user	CREATE DELETE	sp_session_login sp_session_logout
/player	READ UPDATE	 sp_session_player_*
/search/{query}	READ	sp_search_*

Table 4.2. The resource model of the API

4.6 Limitations

Because of the limited time of this project, only a very basic libspotify REST API was implemented. Libspotify is quite extensive, and functionalities like playlists, top lists, browsing and social were not implemented.

4.7 Example

This section presents a concrete example of how the final implemented REST API works, by showing C++ code excerpts from a simple demo application that uses the API, along with equivalent calls in the C API for comparison. The example only shows the interface of the implemented API, what a user of the API would work with, not any underlying implementation details.

To start off, a Spotify object has to be created (listing 4.1). The Spotify object represents a REST component, and it has implemented the connector interface. Upon creation it also creates the standard set of components (session, player and search), and sets up its internal router to point to them (see fig. 4.1). All requests to the API is done through the request method of the Spotify object. Listing 4.1 shows how the C API is set up.

```
Spotify connector;
```

Listing 4.1. Create a Spotify component in the REST API.

```
sp_session_callbacks callbacks;
callbacks.logged_in = &login_callback;
...

sp_session_config config;
config.callbacks = &callbacks;
config.cache_location = "/tmp";
...

sp_session_create(&config, &session);
```

Listing 4.2. Setting up libspotify in the C API. The *config* structure contains various libspotify options and callback functions for the API.

To start interacting with the different resources, their URIs must first be known. To get the available top-level resources, a read request is sent to the entry point of the API, which is predefined as "/". This resource contains the available resources, and it responds with a JSON representation of them. In the example in listing 4.3 the player and session user resources are extracted from the JSON response. No equivalent call exists in the C API, as all function names are already known and defined in its documentation [9].

```
Json resources = connector.request(Representation(READ, "/"));
string playerUri = resources["player"].getString();
string userUri = resources["session_user"].getString();
```

Listing 4.3. Read the top-level resources in the REST API.

The next step is to log in using a Spotify account, by creating a session user resource. To do this, a representation containing a user is sent, with the method header field set to *CREATE*. The representation contains an account username and password. The example in listing 4.4 shows how a login is performed, in this case taking the username and password as the first command line arguments. This time a callback function is also sent with the request. This makes the request call asynchronous, avoiding locking up the thread in which it was called. Instead of returning the response immediately, the response is delivered to the specified callback function when it is available. An equivalent login call in the C API is shown in listing 4.5.

```
Json user(CREATE, userUri);
user["username"] = argv[1];
user["password"] = argv[2];
connector.request(user, &loginCallback);
```

Listing 4.4. Log in to the service with a Spotify account in the REST API.

```
sp_session_login(session, argv[1], argv[2], false, NULL);
```

Listing 4.5. Login in the C API.

A request callback function takes a representation as its only parameter, which represents the response to the corresponding request. The callback shown in listing 4.6 will be called when the login is complete. A typical use case after a user has logged in is to start playing music. This is achieved by updating the player resource. A representation containing the new desired values is sent. To start playing a track using a Spotify URI, the *track* attribute is set to the desired URI, and the *playing* flag is set to *true*. Pausing the playback is simply done by updating *playing* to *false*. Listing 4.7 shows how a track is played in the C API.

```
void loginCallback(Representation response) {
  if (response.getHeaderInt(HEADER_STATUS) == STATUS_OK) {
    Json player(UPDATE, playerUri);
    player["track"] = "spotify:track:6JEK0CvvjDjjMUBFoXShNZ";
    player["playing"] = true;
    connector.request(player, &playCallback);
  }
}
```

Listing 4.6. Start playing a track in the login callback if the login succeeded in the REST API.

```
void login_callback(sp_session *session) {
  const char *uri = "spotify:track:6JEK0CvvjDjjMUBFoXShNZ";
  sp_link *link = sp_link_create_from_string(uri);
  sp_track *track = sp_link_as_track(link);
  sp_track_add_ref(track);
  sp_link_release(link);

  // Called when track has loaded
  sp_session_player_load(session, track);
  sp_session_player_play(session, play);
}
```

Listing 4.7. Playing a track in the C API when login is completed.

To finally log out the user, the session user resource is simply deleted (listing 4.8). In the C API, the logout function is called, as shown in listing 4.9.

```
connector.request(Representation(DELETE, userUri));
```

Listing 4.8. Log out the user in the REST API.

```
sp_session_logout(session);
```

Listing 4.9. Log out the user in the C API.

Performing some actions in the C API require more steps than the REST API, which is a result of the implemented API being more simplistic than the existing API, rather than the properties of REST. The difference between listing 4.6 and listing 4.7 is one example of this, where the C API requires more steps to play a track, but in turn is more powerful in what it can do. The main differences between the API styles are that resources and types are represented in a more general way in the REST API, as they are all encapsulated in a representation object, and sent through the same request method.

Chapter 5

Analysis

This chapter attempts to answer the problem statement by analyzing the found studies and the performed measurements.

5.1 Modeling

Pautasso, Zimmermann and Leymann have compared REST to other web service standards like SOAP, from an architectural perspective [12]. They focus on the differences in the architectural decisions that a web service developer has to make using the different standards. They conclude that more decisions must be made with REST, like the choice of representation types. This also results in more flexibility and control. Significant development efforts and technical risks can also lie in designing the resource specifications and the addressing scheme. They also find that REST does not have as good tool support as other standards.

Some of their findings are consistent with what was found during the implementation of the local REST API. This could indicate that designing a local REST API could involve more work and be more time consuming than designing a more conventional local API. Tool support for local REST APIs was found to be even worse than that for REST web services, and very few relevant tools could be found (see section 2.4). As there are no widespread standards to follow when designing local REST APIs, the implementation was inspired by technology and standards used by REST web services. Some standards, like media types, could be used directly. Other standards were not directly applicable in a local context. HTTP for example, which is a very common application protocol for REST web services, relies on an underlying network transport protocol like TCP.

Time was also spent on designing the resource model. Depending on the underlying structure of how data is represented and manipulated in the system, this could implicate more or less work. The libspotify C API is not very REST or resource-oriented to begin with, so several steps had to be taken to make it RESTful before any code could be written on the implementation. The list of functions in the existing C API had to be conceptually transformed into a set of REST modules,

resources and types. Some parts of the REST API were conceptually harder than others to model. Some local APIs like libspotify are e.g. not just about retrieving and sending data, but also about performing actions like playing and pausing playback. These types of actions do not directly correspond to any of the predefined verbs, but still have to be mapped to resources in some way. One way of implementing a play action could be to introduce completely new verbs, like *PLAY*, *PAUSE* and *STOP*, that could act on certain resources. Another approach could be to implement a player resource, whose play state could be updated using the *UPDATE* verb. The latter approach was chosen in the implementation, as it felt more general and did not introduce a need for new specialized verbs.

5.2 Decoupling

According to Fielding, the uniform interface between components is what mainly distinguishes REST from other network-based architectural styles. The benefit of this constraint is presented to be a simpler overall system architecture, and improved visibility of interactions [5].

5.2.1 Metric

The result of the coupling metric is presented in table 5.1. Several sources mention that REST reduces coupling, when compared to other web service standards like SOAP [11, 12, 16]. This is backed up by the metric, which supports that the implemented REST API is less coupled than the C API, with a value of 3.75 compared to 3.93.

	Coupling type	**Interconnections**	**Final metric**
C API	3	13	3.93
REST API	3	3	3.75

Table 5.1. Application of Fenton and Melton's coupling metric on the APIs

The metric clearly states which API is less coupled, but it is not apparent how significant the difference of 0.18 is. Fenton and Melton do not discuss how to interpret the magnitude of this value when they present the metric, so it is hard to draw any additional conclusions with the value alone. By looking at what the metric actually takes into account, more can be understood. According to eq. (2.1), the metric takes the greatest coupling type between the two modules, and the number of interconnections between them into account. Both APIs have the same greatest coupling type, control coupling, which is described in section 2.3.1.3. Both use parameters to communicate, and the parameter controls the behavior of the APIs. The number of interconnections is what differs. The C API has one interconnection for each public function and callback function, whereas the REST API only has two methods to make requests, and one method type for callbacks. The uniform

way of performing requests in REST is ultimately what lowered the coupling in the implementation according to the metric. Because the REST API only provides a subset of libspotify's functionality, only the corresponding parts of the libspotify C API was considered in the metric.

5.2.2 Data decoupling

Vinoski presents several properties of REST that contributes to decoupling between a server and a client. The freedom for resources to represent their states using different media types, and clients to indicate their desired types, reduces data coupling. Supporting a wider variety of client applications can be easier if clients can choose media type after what is best for them. Using standardized media types provided by IANA, which are widely used on the web, can lead to further decoupling. These standardized media types will not change, removing uncertainty and the need for versioning of the types [16].

5.2.3 Evolvability

According to Fielding, decoupling in REST also encourages independent evolvability of components [5]. Vinoski gives one example of that, also related to types. It is common to use specialized constructed types, like structs, when passing data to and from a local C API like libspotify. This results in high coupling. In larger applications, different parts might have been developed by different teams, at different times, with different versions of underlying software and even in different programming languages. In such applications specialized constructed types can be difficult to use by developers not connected to the team that manages the types. In REST, media types can be versioned, and should not be language or protocol specific [16].

5.2.4 Other benefits

The small and generic interface of the implemented REST API, which consists of only two request methods and a response callback, leads to other benefits as well. Intercepting all requests for e.g. debugging or other processing purposes can be done at a single location. During the development of the Spotify REST API, a simple printing component was implemented, that was connected between the Spotify component and the client. The component simply prints the contents of the requests and responses passed through it, making it simpler to visualize the API messages and find any problems. Implementing the same thing in the C API would mean that each function had to be modified, and specialized routines might have to be implemented to print different data structures.

Local REST APIs can also simplify creation of bridges between different programming languages. Android is a platform where such bridges are used, where applications written in Java can access APIs written in C or C++. Java Native Interface (JNI) is the framework used to enable this. The more functions that have to be called through the language barrier, the more JNI code has to be written.

5.3 Performance

The properties of REST introduce both potential performance gains and performance losses. While the uniformity constraint might result in processing overhead in REST APIs, caching can improve efficiency.

5.3.1 Metric

The measured throughput of the APIs is presented in fig. 5.1. The difference in throughput between the two APIs increases as the request processing time gets shorter. For requests taking about one millisecond, the difference in throughput is 2.4 %. For 0.1 millisecond requests the difference is 30 %.

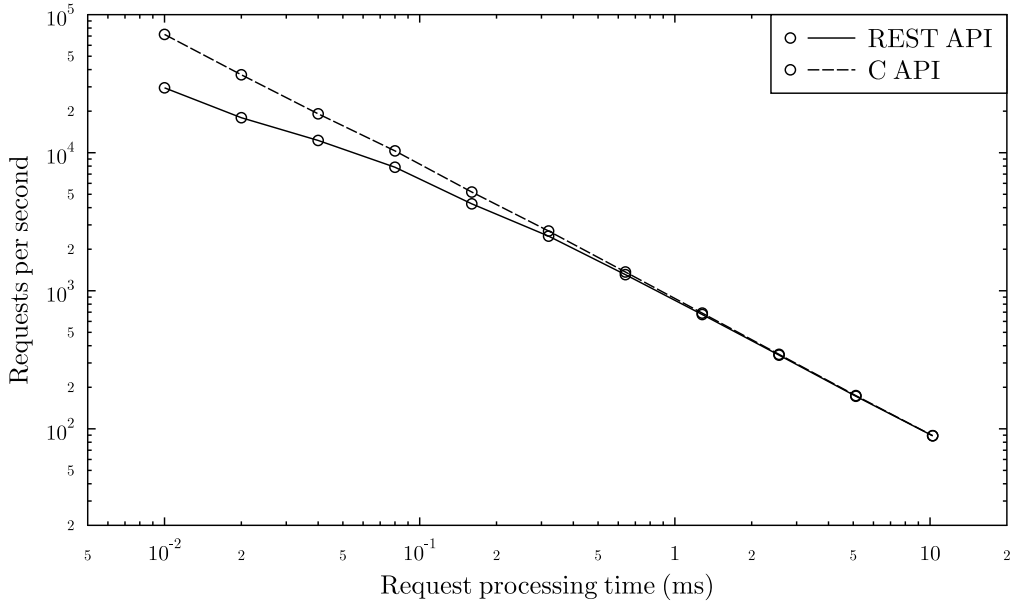

Figure 5.1. Performance throughput of the REST API layer compared to an equivalent C API. The scales of both axes are logarithmic.

The performance metric applied to the implemented dummy APIs shows that the REST API has notable performance overhead for shorter and frequent requests. For less frequent requests the difference was smaller. For 10 millisecond requests the difference is only 0.025 %, where both APIs could process around 89 requests per second. Requests to the libspotify REST API are quite sparse, as they are mainly performed when a user performs actions like logging in and starting to play a track. In normal circumstances such actions would be significantly less frequent than any of the performance measurements. The performance overhead will thus be negligible for the implemented API.

5.3.2 Uniform interface

According to the constraints of REST, data should be transferred in a uniform and standardized way. This leads to degraded efficiency according to Fielding, because data is not sent in an application-specific way [5]. Representation data sent in the API has to be transformed when sent, and then re-transformed when arriving at the receiver. Representations in the implemented API containing e.g. JSON objects have to be serialized to strings when transferred, and then deserialized back to a JSON object for the receiver to use that data. This can add a considerable overhead to each request compared to function calls in C, where data in function calls usually is transferred by copying or by pointers. This partly explains why the REST API performed worse than the C API for requests with shorter processing times. If a component's processing time to handle a certain request is short, the overhead introduced by the REST layer can dominate the total processing time.

5.3.3 Layers

Fielding mentions that processing in intermediate component layers like routers add overhead and latency to API requests in a REST system [5]. At each router in the implemented API, the URI of a request is looked up against the structure containing the different address spaces and their corresponding connectors. Requests are often routed through several routers (see fig. 4.1), which happens in runtime. In contrast, method calls to a C library are linked to the correct memory address once during compile time or when the program starts. This is also part of the reason why the measured throughput of the REST API was worse than that for the C API.

5.3.4 Caching

Caching can be used to reduce interactions like network interactions, disk interactions or computations. This can improve efficiency and user-perceived performance [5]. The uniformity, layerability and cache constraints make caching simple. Because different representation types go through a uniform communication channel and have a uniform way of being represented, a general cache can be built to manage different representation types. As libspotify already handles local caching, it was not implemented in the REST layer.

5.4 Usability

From an API user's perspective, the REST API is quite different to use compared to traditional C or C++ APIs. This is not an area that is mention in any of the studies covering web based APIs, but some observations were made during the implementation of the local REST API. Making a request in the API generally requires more code than making a call in an equivalent C/C++ API. One example is the four line login call shown in listing 4.4, where a representation is created,

the user details are set and the representation is finally sent. The equivalent call in libspotify is a single call to a login function (listing 4.5). Another observation related to types and resources was made. Because types and resources are not defined in language specific ways (i.e. well defined structures and functions in the case of C and C++), the compiler has no concept of these. This means that it can not perform any checks related to these during compile time (e.g. misspelled URIs or type attributes). This also limits the functionality of code completion, as code editors has no knowledge of REST resources or types.

5.4.1 Type checking

Because of the nature of REST, type checking in the API is performed at runtime. Types are not known at compile time, as it is up to the client and server to negotiate what types to use when performing requests. Languages where most of the type checking is performed at runtime are called dynamically typed. C and other languages like C++ and Java, are on the other hand statically typed. In those languages, type checking is performed during compile-time, as the arguments and return types of API calls are known. Dynamically and statically typed programming languages have different properties, and the benefits and drawbacks of them will not be discussed in great detail. One benefit of static typing is that it can increase software reliability by detecting type errors in compile-time before they occur in runtime. A benefit of dynamic typing is the flexibility of choosing types.

5.4.2 Wrapper classes

One downside to having a dynamically typed API in an otherwise statically typed language like C++ is that it might add inconsistency. Developers are probably used to APIs using the style and conventions of the language they work with. One way to provide more consistency is to create language-specific wrapper classes for the various media types. This way representations can still be passed through the uniform API, and also be handled in a more language-specific way, like conventional well-defined structures or objects in the case of C or C++. An example of such a wrapper class is the JSON class presented in section 4.3, which was created for this very reason. Making the REST API more coherent with the programming language which it is implemented in can thus require additional effort.

Chapter 6

Conclusions

This chapter concludes the findings of this project, and presents potential areas that could be of interest for future work.

6.1 REST in local APIs

To conclude, it is fully possible to apply REST principles on local APIs, as shown in the presented implementation. Using REST over more conventional principles when developing a local API introduces both benefits and drawbacks. In the end these have to be weighed against each other to decide if a REST API could be appropriate in a case by case basis. The main discovered strength is the decoupling, along with the potential evolvability and type handling benefits. The additional modeling effort needed, and the extra work needed to make the API language-consistent is found to be the main downside.

In some cases a local REST API might not be appropriate at all, like when API performance is crucial, or for very simple APIs where the work overhead might be unjustified. In other cases a REST API could definitely be worth considering, e.g. if the API is expected to be used by and worked on by a large number of people.

6.2 REST in libspotify

In the case of libspotify, the performance overhead was shown to be negligible. The main downsides are instead related to additional effort in designing the API and making it more user friendly for developers using the API. Applying REST principles could in the other hand provide benefits in the form of better maintainability and evolvability of the API. Due to the small interface of REST APIs, it could also simplify the creation of language bindings. This could be beneficial for an API like libspotify that is intended to work on many different platforms. Libspotify could

thus be a good candidate to develop a REST API around, if one would be ready to spend the additional time and effort.

6.3 Social and ethical aspects

Social and ethical aspects, including economically, socially, and ecologically sustainable development, have been considered during this project due to requirements from KTH. Because of the nature of the project, they have been regarded as irrelevant.

6.4 Future work

One area that has not been covered in this report is the implementation of some more advanced concepts related to REST, like type relations, lists, and modeling of more complex actions. Some types, e.g. lists that refer to other types, might require a way to specify relations between types. Non-standard actions, like login and managing playback, have been modeled very ad hoc in this REST implementation. A more general and formalized procedure to model REST resources from functions in procedural languages could be another interesting area to explore.

Bibliography

[1] Jarallah S. Alghamdi. *Measuring Software Coupling*. 2008.

[2] *Developer's Introduction to Resource Oriented Computing*. 1060 Research, Ltd, 2007.

[3] Harpal Dhama. *Quantitative Models of Cohesion and Coupling in Software*. 1995.

[4] Norman Fenton and Austin Melton. *Deriving Structurally Based Software Measures*. 1990.

[5] Roy Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.

[6] Roy Fielding. *REST APIs must be hypertext-driven*. Oct. 2008. URL: `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven` (visited on 02/18/2013).

[7] Roy Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. 1999.

[8] *Introduction to Resource-Oriented Computing*. 1060 Research, Ltd, 2007.

[9] *Libspotify*. 2013. URL: `https://developer.spotify.com/technologies/libspotify/` (visited on 03/11/2013).

[10] J.D. Meier et al. *Microsoft Application Architecture Guide, 2nd Edition*. Microsoft Press, 2009.

[11] Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. *Developing web services choreography standards: the case of REST vs. SOAP*. 2005.

[12] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. "Restful web services vs. "big"' web services: making the right architectural decision". In: *Proceedings of the 17th international conference on World Wide Web*. WWW '08. 2008.

[13] Martin Reddy. *API Design for C++*. Elsevier, 2011.

[14] *Restlet 2.1 - Tutorial*. 2013. URL: `http://restlet.org/learn/tutorial/2.1/` (visited on 04/05/2013).

[15] *Restlet FAQ*. 2013. URL: `http://restlet.org/discover/faq` (visited on 03/06/2013).

[16] Steve Vinoski. “Demystifying RESTful Data Coupling”. In: 2008.

[17] Wikipedia. *Application programming interface*. 2013. URL: `http://en.wikipedia.org/wiki/Api` (visited on 03/04/2013).

[18] Wikipedia. *Representational state transfer*. 2013. URL: `http://en.wikipedia.org/wiki/Representational_state_transfer` (visited on 05/08/2013).

[19] Wikipedia. *Software metric*. 2013. URL: `http://en.wikipedia.org/wiki/Software_metric` (visited on 05/08/2013).