REST engineering on the server- and client-side

BACHELOR THESIS

BENJAMIN GRÖHBIEL

June 2011

Thesis supervisors:

Prof. Dr. Jacques PASQUIER–ROCHA & Andreas RUPPEN
Software Engineering Group

UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

Software Engineering Group
Department of Informatics
University of Fribourg Switzerland

Abstract

Almost all popular web services offer Application Program Interfaces (API) which make use of the Representational State Transfer (REST) architectural style. This bachelor thesis is about REST engineering on the server and on the client side. It is composed of a theoretical and a practical part.

The theory part discusses the definition and key principles of REST. The practical part is about the development of a Java REST server and a client application consuming that REST service. Technical aspects, programming concepts and application design of both the server and the client side are being shared.

At the end of this thesis you will find a conclusion, which summarises interesting lessons learnt and focuses on possible future improvements for such applications.

Preamble

Foreword

This is a bachelor thesis, written by Benjamin Gröhbiel as a student of University of Fribourg, Switzerland.

Acknowledgements

I want to thank my supervisor Andreas Ruppen for being a competent and dedicated mentor. I also want to thank Prof. J. Pasquier for giving me a lot of freedom and responsibility.

Notations and Convention

- At the very end of this thesis you can find the appendixes and an overview of the references.
- Abbreviations are written out in the appendix B.
- I use **bold** font, whenever I want to point something out in the text or the code.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Context

In the last couple of years companies like Google, Twitter, Amazon and Facebook have begun to change many aspects of our daily life. We use them with all kinds of devices: computers, mobile phones and tablets. They are also embedded in many other products we use. For example instant messengers, which fetch the latest news from Facebook. There are several reasons why these companies were able to become as popular as they are today. One of them is certainly the fact that they all have publicly available API's which helped them to spread their services over the different devices and the web. In order to serve millions of API requests per minute these companies had to come up with an efficient architecture. This is where REST comes in. The majority of public API's are using the REST architectural style. They are fashionable and an alternative to Simple Object Access Protocol (SOAP) web services.

Also in the context of mashups, REST APIs become increasingly interesting. Thanks to them, developers can easily integrate data of existing services into their web service. A mashup is an application, which creates a new service by combining existing services.

1.2 Objectives

The overall objective is to get a grasp of the theoretical and practical side of REST services from the server's and the client's perspective.

Moreover, the objectives of this bachelor thesis are:

- to understand the principles of REST (Chapter 2).
- to develop a REST API in Java (Chapter 3).
- to develop a client consuming the REST API (Chapter 4).

2 REST Basics

2.1 What is REST?

The term REST was introduced by Roy Fielding in his dissertation in the year 2000 [1]. In his dissertation, which earnt him the doctor's degree in philosophy, he talks about web architectures in general and defines REST.

2.1.1 Definitions

Roy Fielding's approach of defining REST as an architectural style [1 S. 86] found wide adoption by other authors. Fielding's introduction and description of REST is more abstract and philosophical compared to other definitions. This is because REST is not bound to the web technology. It is an abstract term [2 S. 80], which can be applied in many fields.

Leonard Richardson and Sam Ruby introduce the term resource-oriented in order to define REST in the context of web services. They define concrete design principles, which describe a resource-oriented web service. Generally speaking a resource-oriented architecture serves the purpose of making resources accessible. In the context of REST: resources are data entities. Some architecture might fulfill the REST criteria better than another but there is no such thing as one true REST architecture [2 S. 79].

Burke [3 S. 4] sums up the REST architectural principles precisely. According to his definition, REST is an architecture built up with the following properties:

- Addressability
- A uniform, constrained interface
- Representation-oriented
- Statelessness
- Hypermedia As The Engine Of Application State (HATEOAS)

Pautasso, Zimmermann and Leymann define REST with the help of just four design principles, leaving out the representation-oriented criteria [4].

Wilde [5] also reduces the definition of REST to mainly two architectural principles: addressability and statelessness:

REST focuses on avoiding application state, making sure that important concepts of an application scenario are represented as URI-identified resources, and that all interactions with a client through a server contain all state information that is necessary, so that the server does not have to maintain session state with clients.

This emphasises the space for interpretation, which the term REST contains. As already mentioned, one REST architecture does not exist, hence the design criteria themselves depend on the point of view of the author.

2.2 Key Principles of REST

According to the authors mentioned before, I will point out the core principles of REST in this section.

2.2.1 Statelessness

Statelessness means that the REST server can fulfill the client request in complete isolation [2 S. 86]. As a result the client sends all the information the server needs with the request. The server does not need to maintain a session. This is one of the reasons why the architectural style of REST can build highly scalable solutions [1]. The server can process information without storing sessions, which need a lot of memory and calculations.

2.2.2 Addressability

This principle says that every resource should be made available through a unique address [3 S. 6]. In the real world this address is described by the use of Unique Resource Identifiers (URI). This principle is also crucial for the web. Resources, for example an image, should be available through an URI. On the web this is not always the case, due to Asynchronous Javascript And XML (AJAX) applications [2 S. 86], which can load content dynamically without changing the URI. To get a resource of a REST service, one can request a URI. In the real world, APIs try to keep the URI design as logical as possible. Therefore the URIs are often hierarchically designed. Hierarchies increases the readabilty and make it easier for developers to address the resources. Also, it is possible to guess resources, which make it easier for clients to discover resources deductively.

2.2.3 Uniform Interface

A uniform interface only offers a set of operations [3 S. 7]. But it still offers enough operations to retrieve, change or create data. The simplicity of the uniform interface is important to REST, because it keeps the interaction between client and server as simple as possible. In practice HTTP is used as the uniform interface of REST services. But REST does not dictate a particular protocol. Technically, REST can make use of any uniformed protocol. But because the HTTP offers all the necessary operations, is well known and widely spread, it became the de-facto standard for REST services. All the interaction between clients and resources are based on a few basic HTTP methods.

GET

The GET method of HTTP describes a request for information about a resource [2 S. 218]. To a GET request, the server will respond with a set of headers and a representation containing the requested resource.

PUT

The client can send a PUT request to modify an existing resource or to create a new one at a given URI. Usually, the client sends a representation along with the PUT request. The server will read the representation and update, respectively create a resource if the resource does not yet exist at the requested URI. Compared to POST, PUT is a much more limited operation that never does anything more than PUT one page at a specified URL.

POST

Like a PUT request, a POST request can create a resource. The difference is, it is not bound to the specified URI. Normally, POST is used when the client sends data to the server and the server then tells the client where it put the data [6]. But the server can do anything with the POST request. As mentioned, it can store it under the given URI (like PUT) but it can also send back a HTTP header or do nothing at all. This makes the POST more flexible than the PUT.

Richardson and Ruby come to the point at which they conclude:

The difference between PUT and POST is this: the client uses PUT when it's in charge of deciding which URI the new resource should have. The client uses POST when the server is in charge of deciding which URI the new resource should have. [2 S. 99]

DELETE

The DELETE request is an assertion that a resource should be removed. The client does not need to send any representation.

2.2.4 Representation Oriented

Representations describe in what format the data is being exchanged between server and client [3 S. 10]. Common are XML, JSON and HTML. But because there is a strict difference between the resource and its representation, the resource can be converted to any representation. This is one of the reasons why REST architectures are said to be loosely coupled [7]. The representation can also handle the language of the content. Practically speaking, the representation oriented architectural style allows servers to give the client exactly the format it demands. It is easy to extend the REST API with a new representation.

2.2.5 HATEOAS - Links and Connectedness

Depending on the application state, the response contains the data and links to other related resources. This means that the responses are hypermedia. They output not just the data, but also links to other resources [2 S. 94]. Hypermedia as the Engine of Application State (HATEOAS) says that, REST can use links to connect a resource to other related resources. This is similar to the web, where we use hyperlinks to link between web sites.

This REST principle has got several advantages. First, it gives the server a lot of flexibility, as it can change where and how state transitions happen on the fly [3 S. 13]. Also, this approach allows the client to browse other resources, which the server suggests to the client. Moreover, this enables the client to automatically discover the resources of the service.

2.3 Recipe: Designing a REST API

Following steps have been useful to me in order to design the REST API.

1. Define resources
2. Design URI schemas
3. Define allowed actions on URIs
4. Define representations
5. Define parameters

2.3.1 Define Resources

First, all the resources of the services need to be defined. This means that, one has got to think of which data entities will be provided through the API. Maybe some information is business-critical and is not meant to be available through the REST API. Also, it is important to think about the relation between the resources. This is important because one has got to decide what information belongs to a particular resource or whether some data is better represented by an own resource. Also, the understanding of the relations between resources is important for designing coherent URI schemas.

2.3.2 Design URI Schemas

Each resource is linked to one URI. The URI schema is important for the developers, who will be using the REST API. An easily understandable URI schema, which is self-explanatory and easy to grasp for developers, makes it easier to make use of the REST API. Often hierarchies are used to create easily understandable URI design.

2.3.3 Define Allowed Actions on URIs

Each URI supports at least one action, but can also have more than one action attached to it. With the help of actions, one can retrieve, mutate or remove resources. Since the de-facto interface of REST is HTTP, there are following actions: GET, PUT, POST, DELETE. Also, one can retrieve metadata about the HTTP headers or the HTTP status. The corresponding HTTP operations are HEAD and OPTIONS.

2.3.4 Define Representations

Representations need to be defined for the input and output of the REST service. In theory REST services can read and provide several representations. The standard representations for web services for both input and output are XML and JSON. But representations depend on the nature of the resources. A REST API for image recognition for example might make use of images instead of XML.

2.3.5 Define Parameters

In general parameters make the REST API more dynamic by allowing more specific client requests. The clients are able to make requests, which fit their purposes. For example, instead of loading all the resources at once, the client can only load the first 25 resources. Therefore parameters are important for filtering resources. They often save traffic or unnecessary computation time. Parameters also help the server to be stateless. Authentication information is also transferred with the help of parameters.

2.4 Current Situation

According to ProgrammableWeb.com [8] out of 3086[1] APIs, 73% are implemented with REST architectural style.

The percentages in Figure **1** need to be handled with care. On the one hand they just represent the percentages of APIs listed on ProgrammableWeb.com. Therefore only the APIs which are indexed by ProgrammableWeb.com and which are also publicly available are listed. On the other hand it could well be that some of these REST APIs are GET-only APIs, which are of course implemented according to REST principles but are not real APIs. They may lack data operations such as updating or deleting resources. Also, I assume that many of the private APIs, which are used in internal business environments, are implemented in a SOAP style.

Nevertheless, the figure below emphasises the importance of REST in the context of public web services.

[1] http://www.programmableweb.com/apis, accessed on 29.03.2011.

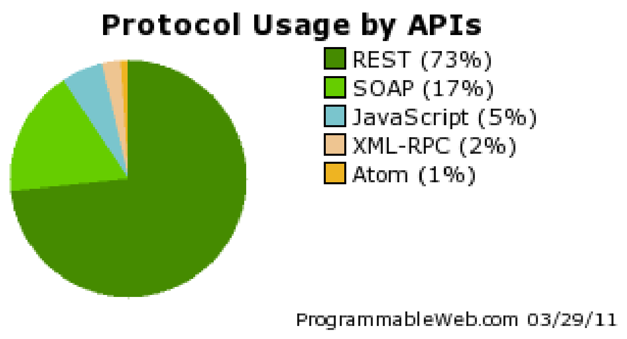


Figure 1: 73% of all API's registered on ProgrammableWeb are based on REST.

3 Server Implementation

3.1 Scenario

In the context of this bachelor thesis I developped a REST API. It is called FoodDistributor and it is all about food and locations. It comes from the general idea of mapping foods on a map. A possible use-case could look like the one depicted in **Figure 2**. Client Steven feels thirsty and wants to find all drinks in a certain area. He uses the FoodDistributor API to find all drinks in his area, including information about the prices. The REST API serves him with information about where and for how much he can buy a drink. He can use the REST API to directly buy the drink with the help of a PUT request.

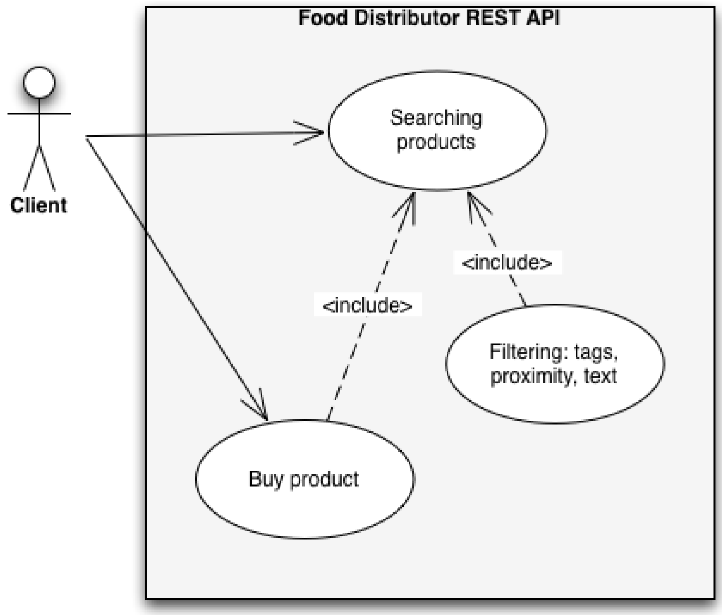


Figure 2: Use case of the REST service

3.1.1 Terminology

Foods

As you can see in Figure **3**, there are foods, which contain general information about the nature of the food. For example: an apple has got a title “Apple”, a description “Regional fruit with taste” and some tags, e.g. “fruit”, “sweet”.

Products

Products are mapped to one particular food, extending it with information about the price, vendor, quantity and location. An example would be a specific Apple, being sold by Migros for 1 CHF at 46.79827, 7.1534.

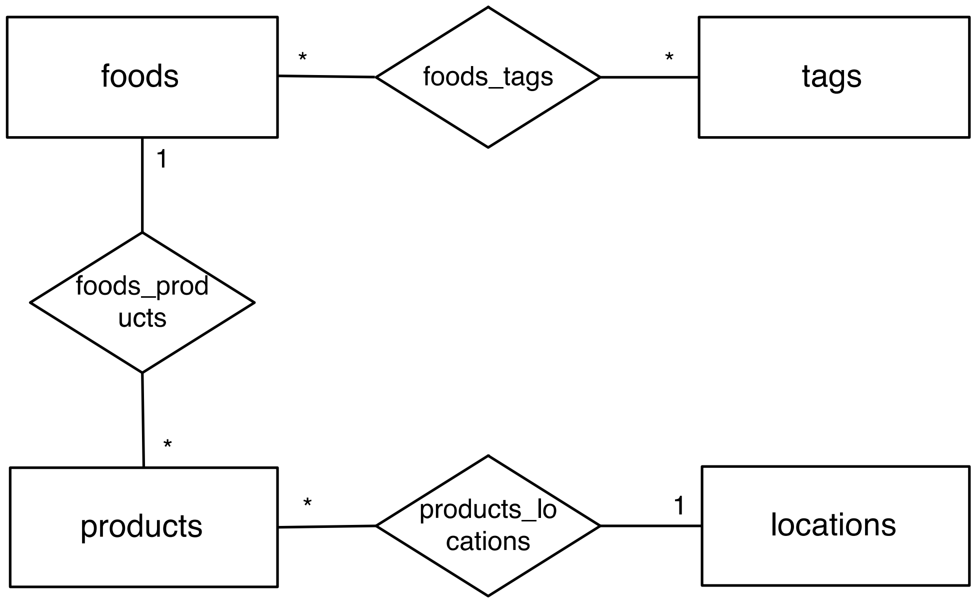


Figure 3: ER-model of the FoodDistributor

Tags and Locations

A food can have several tags and vice versa. A product is bound to one location, but at one location there can be several products (see Figure **3**).

Features

The features below are offered by the FoodDistributor:

- Filter criteria exist in order to find foods and products. Filters for foods and products are text search and tags. Products can additionally be filtered by the foods.
- Listing foods and products, matching the filters.
- Browse foods by tags.
- Listing products in a specified rectangular area, matching the filters.
- Provide information about a specific food or product.
- Buy a product.

3.2 Choosing implementation tools

There are dozens of programming or scripting languages, which are capable of implementing a REST API. If a programming or scripting language can support the key principles of REST mentioned in the chapter before, it is technically possible to implement the API with that particular language. To name a few: Java, Ruby [9], ASP.NET, PHP.

There are also many frameworks that simplify the creation of REST APIs. For Java there are for example Restlet[2], RestEasy[3] and Jersey[4].

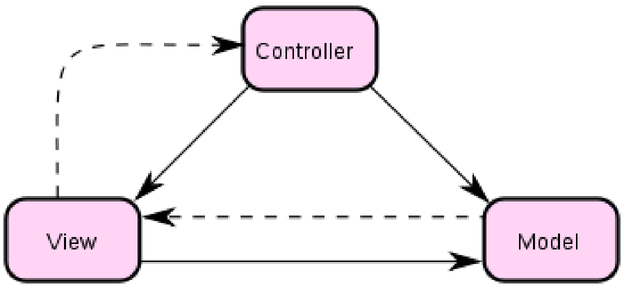
I chose Java and the Jersey framework to implement the FoodDistributor API, mainly because of the good documentation. Also, Jersey is the JAX-RS reference implementation for building REST services.

Which framework suits best is a question of preference and functionality. If you lay emphasis on GZIP compression, Restlet seems to be the right choice since it gives you a lot of control options about GZIP compression at runtime. The current release of Jersey (1.6) on the other hand does not seem to support this feature. Although since Jersey 1.0, it is possible for a client to request GZIP compression using output filters. But the server side does not have the abilty to decide whether to respond with a GZIP compressed response or not. Using GZIP compression is only possible by telling the server to compress all the responses. There exist more differences of that sort between the frameworks.

Furthermore MySQL is used to store the data and Hibernate is used as persistence framework.

3.3 Application Design

The FoodDistributor was implemented with the Model-View-Controller pattern in mind. This design pattern seemed to be handy for REST services. MVC is a software architecture, which lays emphasis on distinguish between business logic, data management and presentation. In order to achieve this there exist controllers, models and views as depicted in Figure **4**. Controllers coordinate views and models. Models take care of the data management. Views display the data, which has been loaded by the model.



[5]

Figure 4: MVC overview

Figure 5 depicts a simplified class diagram of my REST service. When a resource is being requested, a function is fired. This function is provided in one of the controllers in the Controller package. This approach of using an URL as an entry point with a function behind itis also found in Ruby on Rails and other MVC frameworks.

[2] http://www.restlet.org/, accessed on 30.03.2011.
[3] http://www.jboss.org/resteasy, accessed on 30.03.2011.
[4] http://jersey.java.net/, accessed on 30.03.2011.
[5] http://en.wikipedia.org/wiki/File:ModelViewControllerDiagram2.svg, accessed on 12.05.2011.

The models handle the database connection and the data operations. They store and retrieve data from the database and pass it back to the controller. In the FoodDistributor application, all the data operations are bundled in the model class in the data package.

Since REST services normally provide several representations, the concept of views in MVC comes in handy. It makes sense to have several views for one controller. After retrieving the data from the model, it opens the corresponding view – depending on the representation. The views of the REST service are bundled in the Output package.

Helpers are functional code snippets that can be used cross-application by controllers, models or views.

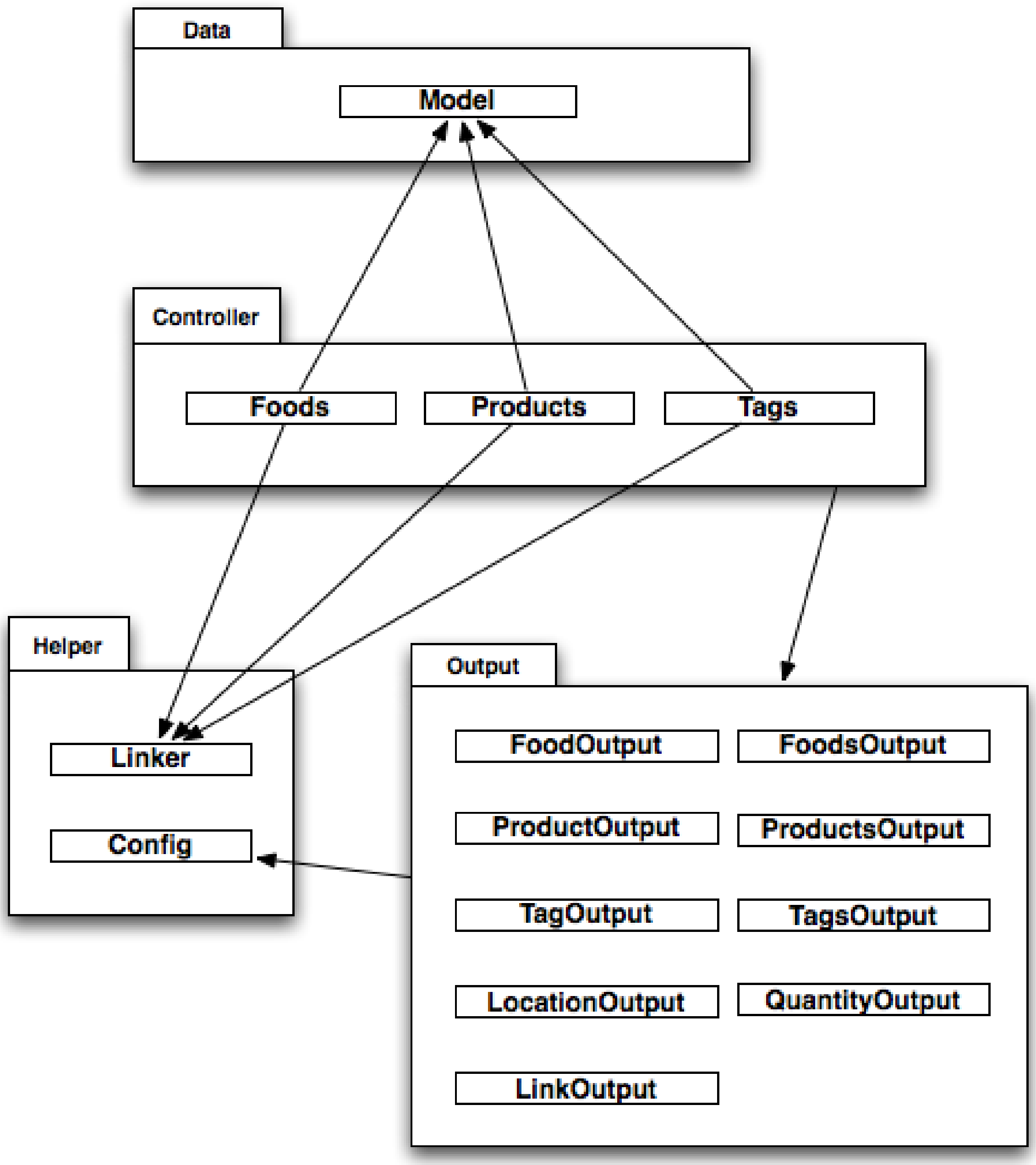


Figure 5: Simplified class diagram of the FoodDistributor API

3.4 REST Principles Implemented with Jersey

Jersey is the reference implementation [10] of the JAX-RS specification [11]. JAX-RS is a specification designed for building web services embracing the REST design principles.

3.4.1 Statelessness

Statelessness cannot be achieved by the Jersey framework itself. It is a question of own implementation. Technically, it is possible to create a stateful application with the Jersey framework. Statelessness is maintained by not using sessions.

3.4.2 Addressability

The Jersey framework provides addressability through paths and parameters.

Path

When a client requests a resource as depicted in **Listing 1**, Jersey calls the corresponding function. Thanks to code annotations, Jersey knows which function needs to be called. If no function is defined for a certain URI schema, Jersey replies a 404 HTTP error message to the client. The @Path annotation can be static or dynamic. In Listing **2** the “foods” resource is static, while {id} indicates a dynamic variable. This variable can be read with the @PathParam annotation.

```
GET /foods/2 HTTP/1.1
Host: www.example.com
```

Listing 1: GET Request

```
@GET
@Path("/foods/{id}")
public Response getAllFoodsHTML(@PathParam(“id”) int id) {
    ...
    return response;
}
```

Listing 2: Mapping a URI to a function with Jersey

Paths are important for the addressability because they are the foundation of the URI schema. Therefore they are implemented thouroughly, offering a lot of flexibility because they can consist of static and dynamic parts at the same time.

Parameters

Parameters are part of the JAX-RS specification and exist amongst others to make the REST server stateless. Parameters are information, which can be passed along besides the message body.

There exist different kinds of parameters. Here, I will only discuss the parameters I have used in my REST API. If you are looking for a brief summary of all the parameters that exist, I can recommend Chapter 5 *JAX-RS Injection* of Java with JAX-RS by Bill Burke [3].

QueryParam

Query Parameters can be passed along with the URI. For example, to retrieve only foods, which are linked to a certain tag and consist of a certain title (see **Listing 3**).

```
GET /foods?tag=2&search=snickers HTTP/1.1
Host: www.example.com
```

Listing 3: Request with query parameters

```
public Response getAllFoods(
   @QueryParam("tag") @DefaultValue("") String tag,
   @QueryParam("search") @DefaultValue("") String search,
…
```

Listing 4: Use of @QueryParam

@QueryParam annotation is used to read the query parameters (see Listing **4**). The @DefaultValue annotation allows the developer to specify a default value for a certain query parameter, in case it is not specified. This also works with all parameters and is a very handy feature. In Java, unlike Python, this concept of allocating a default value for a variable is not used. Thanks to the Jersey library one can use this concept in the context of REST web services.

There is no limit for query Parameters in regard to the bytes length of the parameter. But it is recommended that the GET request is not over 255 bytes [12]. Otherwise problems with different clients or servers can occur.

HeaderParam

Header parameters are different than the query or path parameters. Header parameters reside in the header of the HTTP request and are not part of the URL. The FoodDistributor application uses HeaderParams to store geolocation information. A GET request to /products can contain the HeaderParam "Bounds" as depicted in **Listing 5**.

```
GET /products HTTP/1.1
Bounds: 46.7874,7.1437;46.7990;7.1668
Host: www.example.com
```

Listing 5: Request with HeaderParam

Reading the header parameters works the same way as reading query or path parameters (see **Listing 6**).

```
Public Response getProducts(
   @HeaderParam("Bounds") @DefaultValue("") String bounds,
…
```

Listing 6: Reading the HeaderParam

3.4.3 Uniform Interface

The Jersey framework uses HTTP to fulfill the uniform interface principle. The following HTTP operations can be used directly as annotations: @GET, @POST, @PUT and @DELETE.

Listing 7 shows a function, which is being triggered when Jersey registers a GET request to the /foods resource. Instead of @GET, one could also make use of the other HTTP operations mentioned above.

```
@GET
@Path("/foods")
public Response getAllFoodsHTML() {
    ...
    return response;
}
```

Listing 7: GET annotation

3.4.4 Representation-oriented

Because representations concern the input and the output of the REST service, there are two annotations: @Produces and @Consumes.

Listing 8 shows the usage of the @Produces annotation. When triggering a function, which is annotated like shown, Jersey automatically sets the Content-Type to application/json.

```
@Produces(MediaType.APPLICATION_JSON)
```

Listing 8: @Produces annotation

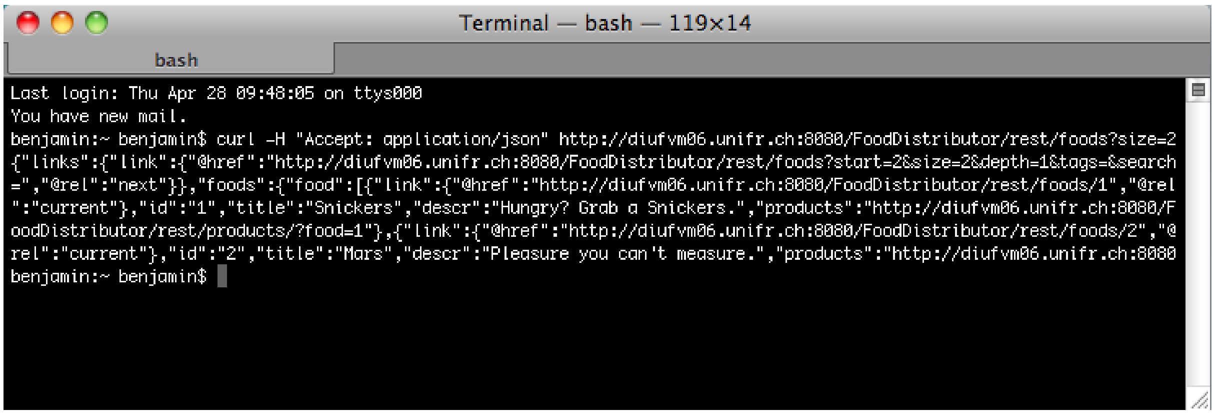
The same works for the input of the REST service. The @Consumes annotation, shown in **Listing 9**, tells the corresponding function, that the message body is formatted in the representation specified. In this case, the content-type of the message body would be JSON.

```
@Consumes(MediaType.APPLICATION_JSON)
```

Listing 9: @Consumes annotation

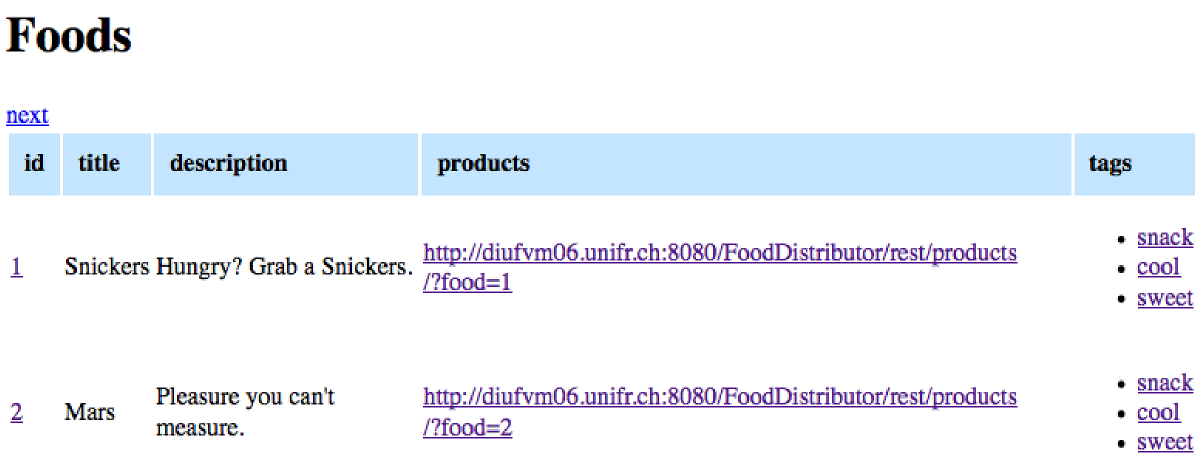
The JAXB library [13] exists in order to offer XML and JSON representation. Thanks to this library, it is easily possible to convert Java Entities into XML or JSON. Other representations, like HTML, are not supported by JAXB and must be implemented independently.

Following screenshots show, that the FoodDistributor REST API supports several representations. **Figure 6** shows the /foods resource in JSON representation while **Figure 7** shows the HTML representation of the same resources.



```
Last login: Thu Apr 28 09:48:05 on ttys000
You have new mail.
benjamin:~ benjamin$ curl -H "Accept: application/json" http://diufvm06.unifr.ch:8080/FoodDistributor/rest/foods?size=2
{"links":{"link":{"@href":"http://diufvm06.unifr.ch:8080/FoodDistributor/rest/foods?start=2&size=2&depth=1&tags=&search
=","@rel":"next"}},"foods":{"food":[{"link":{"@href":"http://diufvm06.unifr.ch:8080/FoodDistributor/rest/foods/1","@rel
":"current"},"id":"1","title":"Snickers","descr":"Hungry? Grab a Snickers.","products":"http://diufvm06.unifr.ch:8080/F
oodDistributor/rest/products/?food=1"},{"link":{"@href":"http://diufvm06.unifr.ch:8080/FoodDistributor/rest/foods/2","@
rel":"current"},"id":"2","title":"Mars","descr":"Pleasure you can't measure.","products":"http://diufvm06.unifr.ch:8080
benjamin:~ benjamin$ 
```

Figure 6: Data in JSON representation



Foods

next

id	title	description	products	tags
1	Snickers	Hungry? Grab a Snickers.	http://diufvm06.unifr.ch:8080/FoodDistributor/rest/products/?food=1	• snack • cool • sweet
2	Mars	Pleasure you can't measure.	http://diufvm06.unifr.ch:8080/FoodDistributor/rest/products/?food=2	• snack • cool • sweet

Figure 7: Data in HTML representation

Problem with other representations, not supported by JAXB

At the moment a library, which offers the conversion from Java entities to HTML markup, does not exist. Therefore one has go to genereate the HTML output by oneself. This is possible for example by using the Viewable class[6] of the Jersey framework. But this solution quickly becomes cumbersome due to duplication. Each representation has got to be maintained individually.

[6] http://jersey.java.net/nonav/apidocs/1.5/jersey/com/sun/jersey/api/view/Viewable.html, accessed on 16.04.11.

Therefore, when offering many representations the application can become hard to maintain.

Binary Representation

For testing purposes, the FoodDistributor also supports binary representations for some resources. After testing the binary representation the following three major drawbacks came up:

1. The client needs the exact same Plain Old Java Objects (POJO) – which reveals a lot about the server, hence this is a security issue.
2. The data size, which was transmitted by the server, was bigger than it was with XML.
3. It is developer unfriendly, due to the binary nature of the responses.

These drawbacks resemble the drawbacks of Remote Methods Invocation (RMI). RMI does not seem to be as widely spread as web services supporting XML and JSON. This is a sign, that binary representation is not as practical as for example XML or JSON representations.

3.4.5 HATEOAS

Because HATEOAS strongly depends on the output, it is not supported by Jersey. I implemented HATEOAS features as a part of the views and also as a helper, providing pagination.

Implementing the HATEOAS functionality for the FoodDistributor REST API was rather difficult. The problem is that all the different represenations implement HATEOAS differently. Despite the pagination helper, XML and JSON have got another implementation of HATEOAS than HTML. This leads to inconsistency, which causes code repetition. This again makes it harder to keep the representations up-to-date and to maintain the code in general. Of course it is possible to encapsulate the logic for pagination, but because the output looks different in every representation, this does not avoid code repetition entirely.

As mentioned HATEOAS is a REST principle and therefore very important. It makes it possible for machines, but also human clients to explore the REST service automatically, respectively easily.

4 Client Implementation

4.1 Scenario

The idea of the client is to offer a browser based and attractive user interface for the FoodDistributor API. The application, called FoodDistributorClient, allows the user to find products near its location with the browser. For example if user Steven is looking for a drink, he can open up his browser and find all drinks in his proximity. Google Maps is used to map the products, whereas the FoodDistributor API provides the application with foods and products. So, the client application will use map related data from Google Maps and food related data from the FoodDistributor API. This will turn the application into a mashup.

4.2 Choosing Implementation Tools

A REST client needs a programming or scripting language, which can fire HTTP operations and proceed XML or JSON. Therefore one can write REST clients with scripting languages like JavaScript or PHP. But because REST services often make extensive use of XML or JSON, decent support for serialising or deserializing XML or JSON is important. That is why it makes sense to use a programming or scripting language with native support for XML and JSON.

With that in mind the following two frameworks seem to be worth examining:

- Ruby on Rails: Well known web framework, based on Ruby.
- Google Webtool Kit (GWT): Based on JavaScript with an arbitrary server backend.

	Ruby on Rails (short Rails)	GWT
Google Maps API Support	It is possible to make use of all of the features of Google Maps by using plain JavaScript. There are some Ruby plugins, which have implemented some of the Google Maps API. [7][8]	There is a Google Maps API plugin for GWT, maintained by Google. All Google Maps features have been implemented and are steadily updated. [9]
REST consuming Capabilities	Native support for serialisation and deserialization of JSON, XML, YAML. Supports the HTTP operations too. Ruby is famous for its REST consuming capabilities. [10]	There is very limited REST support on the client side due to the Same Origin Policy (SOP) browsers. There are plugins, based on JavaScript, that all of them suffer from the SOP problem. The backend, which can be Java for example, enables full REST capabilities.
Geolocation support	The HTML5 Geolocation API can be made use of by using plain	There is a plugin, which supports the HTML5 Geolocation API. [12]

[7] http://ym4r.rubyforge.org/, accessed on 31.03.2010.
[8] http://ym4r.rubyforge.org/tutorial_ym4r_georuby.html, accessed on 31.03.2010.
[9] http://code.google.com/p/gwt-google-apis/wiki/MapsGettingStarted, accessed on 31.03.2010.
[10] http://www.quarkruby.com/2008/3/11/consume-non-rails-style-rest-apis, accessed 31.03.2011.

	JavaScript. There is also a plugin for IP location. [11]	There also exist libraries for IP location[13].
Conclusion	Ruby on Rails has got native REST support, which is well known under the keyword ActiveRecord. Also Rails is well known for rapid prototyping, which also might come in handy. But the lack of a Google Maps API for Ruby on Rails is a major drawback for a mash-up, which heavily involves Google Maps.	On the client-side GWT is based on JavaScript, which does not allow sophisticated REST support. Thanks to the backend, which can be Java for example, GWT can offer full REST capabilities. The Google Maps API for GWT is fully implemented and up-to-date.

Table 1: Comparison between Ruby on Rails and GWT

I decided to use GWT, which in retrospect turned out to be the right choice, because of the sophisticated Google Maps API support.

4.3 Consuming REST with GWT

As already mentioned, consuming REST with GWT can be very hard on the client side, due to the Same Origin Policy (SOP), which depends on the browser. The SOP is a security concept, which amongst others should prevent cross-site scripting.

HTTP calls are supported by the browser's XMLHttpRequest function. The XMLHttpRequest function though depends on the SOP of the browser. Almost all browsers do not allow requests which reach for resources outside the current site.[14] Firefox 4 introduced a new security concept called Content Security Policy (CSP), which would enable cross-site HTTP calls by defining secure sites [15]. At the moment, CSP is only supported in Firefox 4. When CSP is not found, the browser will fall back to SOP. But as soon as other browsers adopt this security policy, it will be possible to write REST clients just with JavaScript for a wide audience.

Because of the SOP restrictions it is necessary to have a backend, which does not depend on any browser. In GWT the backend normally uses Java or Python. To call the REST API one

[12] http://code.google.com/p/gwt-mobile-webkit/wiki/GeolocationApi, accessed 31.03.2011.
[11] http://geokit.rubyforge.org/, accessed 31.03.2011.
[13] http://sourceforge.net/projects/javainetlocator/, accessed on 12.05.2011.
[14] http://www.w3.org/TR/XMLHttpRequest/, accessed 31.03.2011.
[15] http://www.heise.de/security/artikel/Abhilfe-gegen-Cross-Site-Scripting-und-Clickjacking-1214277.html, accessed 13. 05.2011.

can make use of any REST framework written in Java or Python. Technically, it is also possible to use Java's or Python's native HTTP support to use the REST API.

Another needed feature when consuming a REST service, is the deserialisation of JSON or XML. This can be done either on the backend or on the frontend. There exist JavaScript based solutions working in the frontend. Similarly, there exist Java, respectively Pyhton solutions, which take care of the deserialisation on the backend. For more about this, see 4.4.1 Deserialisation.

4.4 Application Design

For large scale application development, Google suggests the Model-View-Presenter pattern [14]. Because such an approach involves more complexity and effort than it is necessary for small applications like this, I chose to create my own application design.

FoodDistributorClient is the entry point of the application and manages both a DataManager and a BaseLayout instance (see **Figure 8**). The DataManager class manages the data. The BaseLayout manages the layout. DataManager and BaseLayout can communicate with each other via the FoodDistributorClient class.

The BaseLayout class makes use of so called widgets, which are UI elements. Outsourcing them in their own classes improves the maintainability and readability. The DataManager makes use of the FoodsResource and the ProductsResource classes, which turn the server response into data objects. All the mentioned classes are part of the client package and run in the JavaScript environment of the browser.

FoodDistributor and FoodDistributorImpl are instanciated by the GWT framework itself. When calling functions of FoodDistributorAsync, GWT forwards the request to the backend calling the FoodDistributor class and finally the FoodDistributorImpl class.

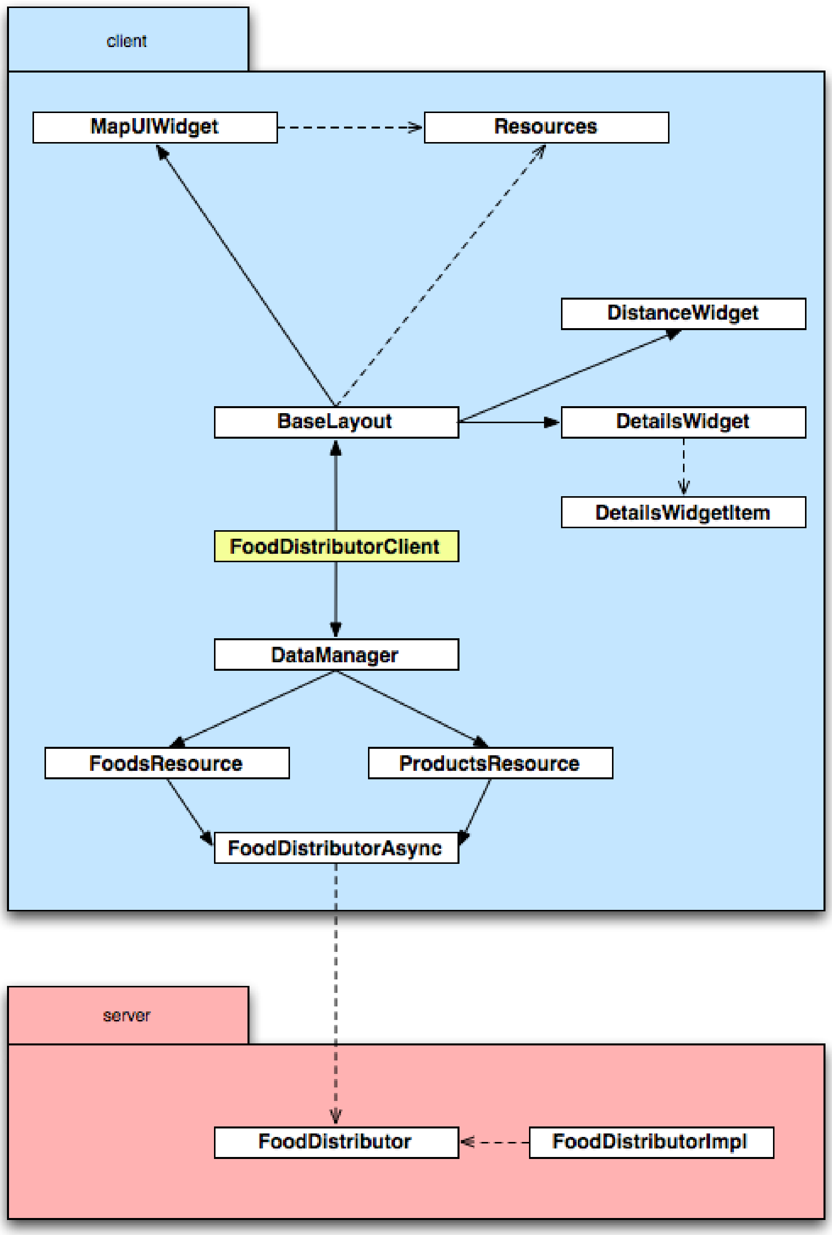


Figure 8: Simplified class diagram of the GWT application

The sequence diagram depicted in Figure **9** shows the chain of all the classes necessary to retrieve data. In this example a button in the BaseLayout triggers the request for foods. The button click listener hands the request over to the DataManager class, which forwards the request to the corresponding helper class, namely the FoodsResource class. Until this point, everything was taken care of by the frontend. The FoodsResource class passes the request to the backend. The FoodDistributorImpl class on the backend sends the GET request to the REST API and retrieves an answer. It passes the response back to the frontend asynchronously. At the end, the response will be returned to the calling function in the BaseLayout class.

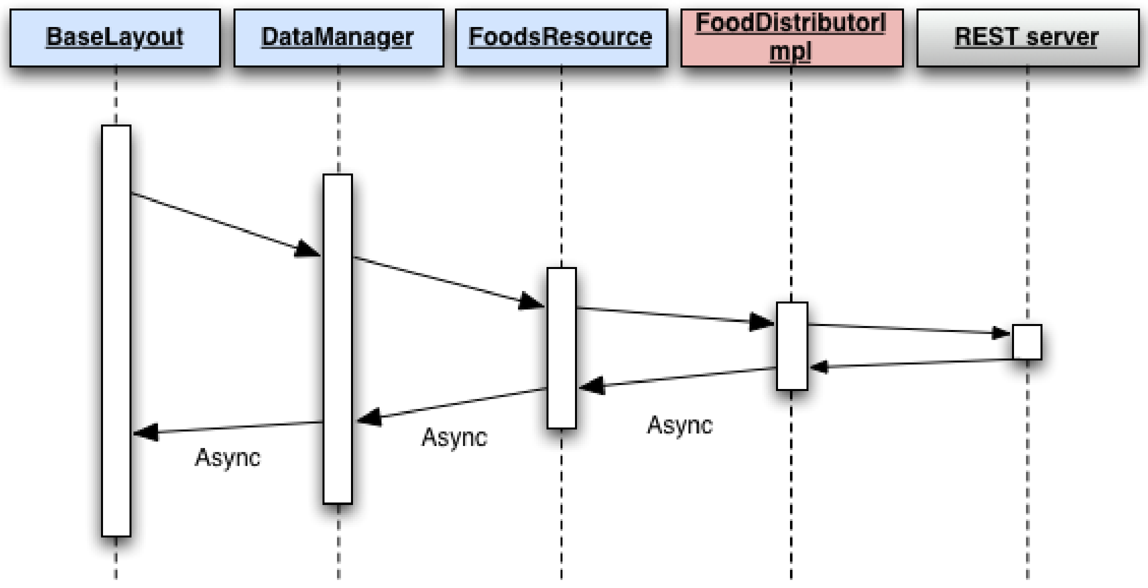


Figure 9: Sequence diagram

4.4.1 Deserialisation

The deserialisation of the JSON response can take place:

1. On the backend, using a deserialisation library.
2. On the frontend, using a GWT plugin.

In my application I decided to source out as much as possible to the frontend. This is also the paradigm of GWT, which is designed to create scalable applications. By doing so, I can save computation time on the server and shift it to the client. In applications with thousands of users this matters. A down-side of this approach is that, because the client's browser has to execute more JavaScript, it might produce an unsmooth user experience in the browser.

On the other hand, if the POJO's are created on the server part, they need to be transmitted to the client side too. This also involves the browser's JavaScript engine in order to decode the RPC call in a readable response.

For future studies, this might be an interesting field of research.

4.4.2 Using POJOs

Thanks to GWT, POJO's can be used to store data like in other object-oriented solutions. Almost all java.util collections are supported by GWT, which makes using POJO's more convenient. Being able to store data in such an object-oriented way is astonishing, given that GWT is only based on JavaScript. The FoodDistributorClient uses POJOs to store foods, products and locations. They are being stored in HashSets, which seem to be the ideal collection because of two reasons: First, HashSets do not allow double entries, which means that the POJO's in the HashSet are distinct. Secondly, an entry of the id and the value exists. The id makes the POJO easily accessible. This is an advantage compared to lists, which would need to be iterated in order to find a certain POJO.

4.4.3 Package Design

GWT dictates the separation between server, client and shared functions. Hence, the code that is executed by the frontend goes in the client package, code that is executed by the backend goes in the server package and code that can be executed by the frontend or the backend go into the shared package.

This is useful in order to remind oneself, that there are differences between frontend and backend. Code in the client package is compiled into JavaScript. Code in the server package is executed as Java.

4.4.4 Plugins

Plugins extend the native capabilities of GWT. In my application I use three plugins:

1. GWTQuery: adding jQuery functionality to GWT. This is especially useful for altering the Document Object Model (DOM).
2. Google Maps Api: this plugin gives full access to the Google Maps Api. Java Code is automatically translated to JavaScript.
3. GWT HTML5 Geolocation: this plugin adds HTML5 Geolocation support to the application.

4.4.5 Geolocation

GWT HTML5 Geolocation plugin can load the location of the current browser, if the user accepts to share its location. After embedding the plugin in the application, one can make use of it as depicted in **Listing 10**.

```
1     private void getLocation() {
2
3         if(Geolocation.isSupported()) {
4               Geolocation geo = Geolocation.getGeolocation();
5               geo.getCurrentPosition(new PositionCallback() {
6                   public void onFailure(PositionError error) {
7                       center = LatLng.newInstance(46.80170, 7.14669);
8                       …
9                   }
10                  public void onSuccess(Position position) {
11                      Coordinates coords = position.getCoords();
12                      …
13                  }
14              });
15
16        } else {
17              center = LatLng.newInstance(46.80170, 7.14669);
18              …
19        }
20    }
```

Listing 10: Geolocation snippet

On line 3 it is being checked whether the HTML5 geolocation feature is supported by the current browser. If the HTML5 geolocation feature is enabled, the PositionCallback on line 5 takes over. Either everything works (onSuccess) and the plugin is able to get the location or something goes wrong (onFailure). Within the function onSuccess one can access the location data, which the browser provided through position.getCoords() as shown on line 11.

The GWT plugin transforms this Java function into JavaScript, which on the other hand accesses the browser's HTML5 geolocation API and returns it.

In case the HTML5 geolocation API is not supported by the browser, a static location is used instead. Instead of using static coordinates, I could have used IP-tracking to get a more precise estimate of the location.

4.5 User Interface Design

In this section I want to go into the most important concepts of my user interface. Most parts of the user interface were created with the GWT Designer[16] (see **Figure 10**).

[16] http://code.google.com/webtoolkit/tools/gwtdesigner/index.html, accessed 11. April 2011.

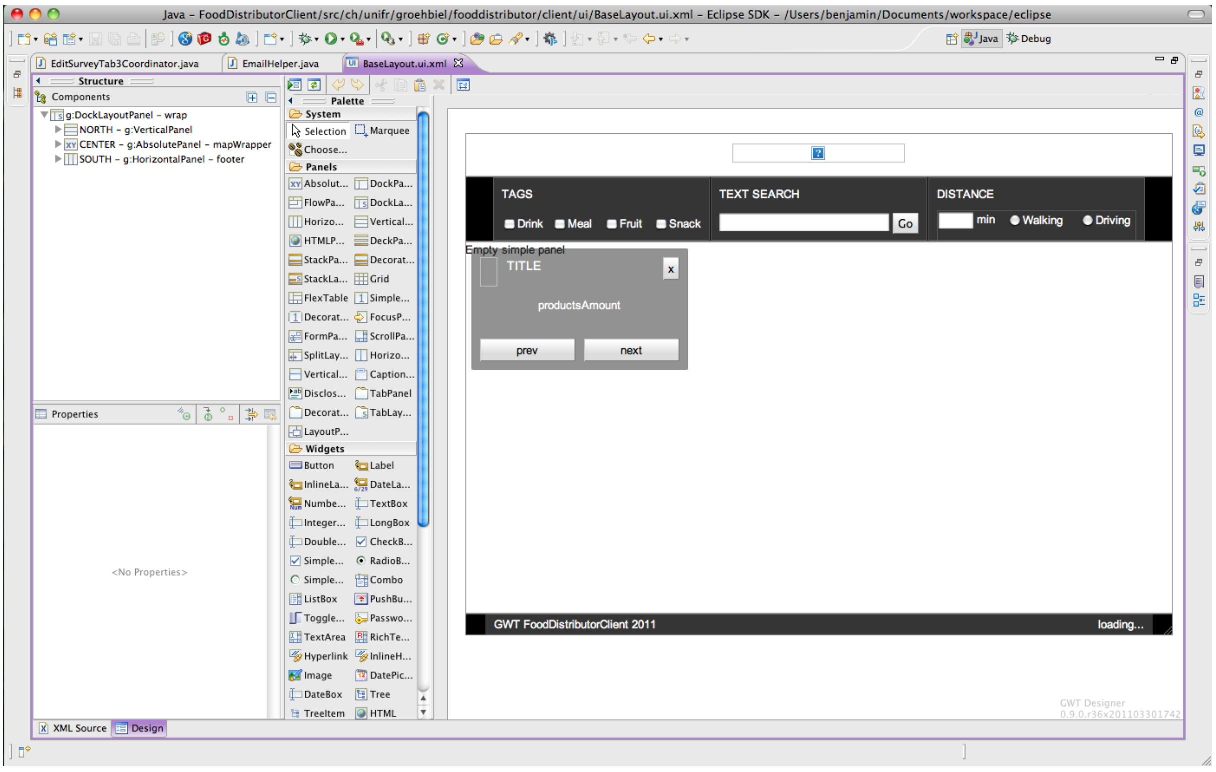

Java – FoodDistributorClient/src/ch/unifr/groehbiel/fooddistributor/client/ui/BaseLayout.ui.xml – Eclipse SDK – /Users/benjamin/Documents/workspace/eclipse
EditSurveyTab3Coordinator.java
EmailHelper.java
BaseLayout.ui.xml
Structure
Components
g:DockLayoutPanel – wrap
NORTH – g:VerticalPanel
CENTER – g:AbsolutePanel – mapWrapper
SOUTH – g:HorizontalPanel – footer
Properties
<No Properties>
Palette
System
Panels
Widgets
TAGS
Drink
Meal
Fruit
Snack
TEXT SEARCH
Go
DISTANCE
min
Walking
Driving
Empty simple panel
TITLE
productsAmount
prev
next
GWT FoodDistributorClient 2011
loading...
XML Source
Design

Figure 10: GWT Designer

4.5.1 Construction

In order to create a layout, which automatically adjusts height and width to the current browser size, the following construction is necessary.

The red boxes indicate the DockLayoutPanel [15] with three areas:

- **North**: containing the logo and the filters.
- **Center**: containing the map.
- **South**: containing the disclaimer and the zip information.

As a definition the areas north and south must be defined by fixed heights, whereas the center automatically takes the space left. As long as the browser height is higher than the heights of north and south together, there will be no scrollbars as a result.

North itself is a VerticalLayout [15], which lists the logo and the filters vertically. The filters themselves (yellow boxes) are listed horizontally with a HorizontalLayout [15].

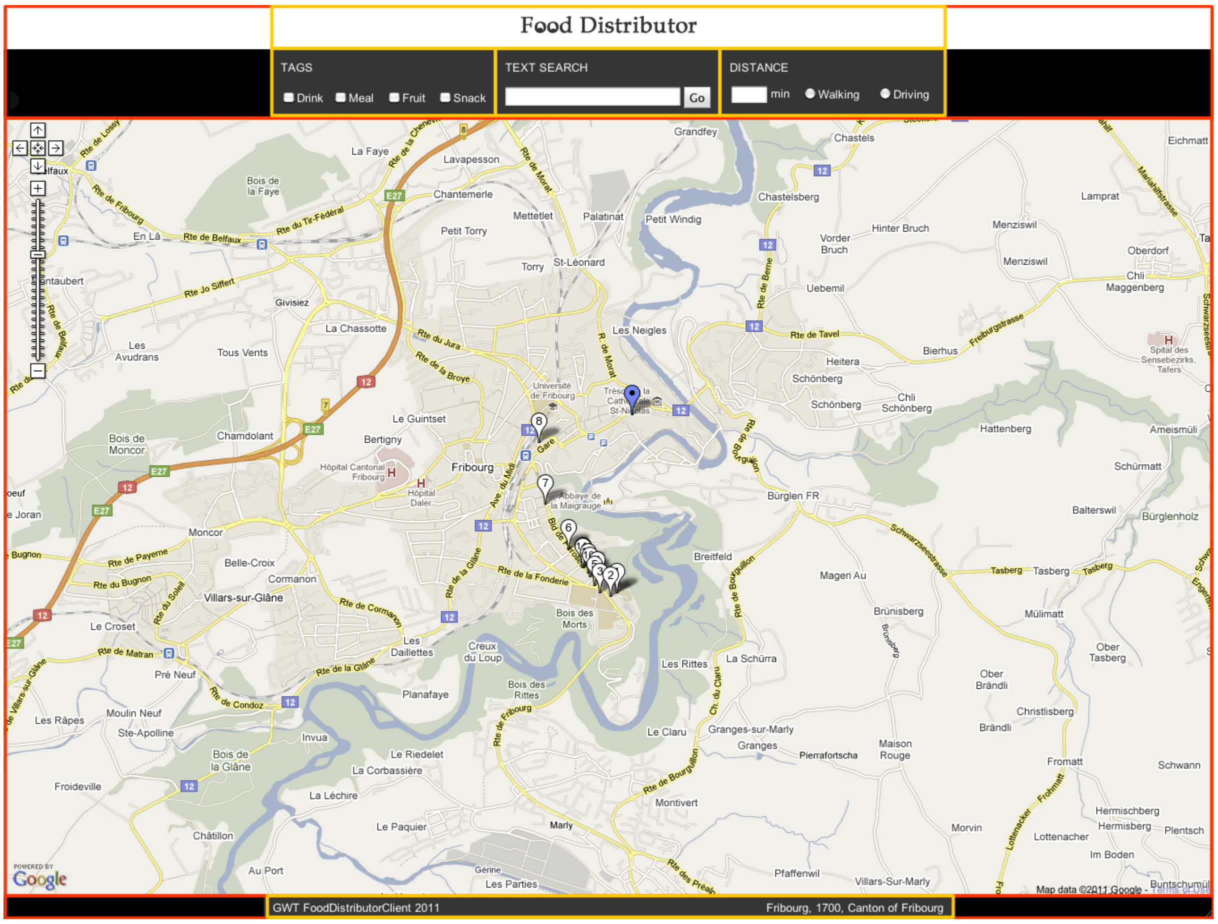


Figure 11: Construction with GWT panels

4.5.2 Status Quo

Once loading the application a loading animation shows up. Once the loading is done, the map centers at the current position of the user (see Geolocation 4.4.5). It invokes a blue marker, indicating the position of the user, and white markers, indicating available products. As soon as the user is located, the application consults the server to load the products in the current area. This means, all the products, which are located within the rectangle of the map, are loaded. The result of this is depicted in **Figure 12**.

In the black ribbon[17] in the top, the user can filter the products. On the left, there are map controls, provided by the Google Maps API. In the footer, on the right, information about the zip code of the current position is displayed.

Information is only pulled from the server at the application launch and whenever the filter change. In the beginning information was pulled from the server when the window size or the zoom levels changed. But since especially zooming is a very common user action, this resulted in many server calls. This on the other hand lead to a lot of JavaScript processing,

[17] http://en.wikipedia.org/wiki/Ribbon_(computing), accessed 11. April 2011.

which created a less responsive user experience. To keep the UI as responsive as possible I abandoned events that would be fired when the window size or the zoom level changes.

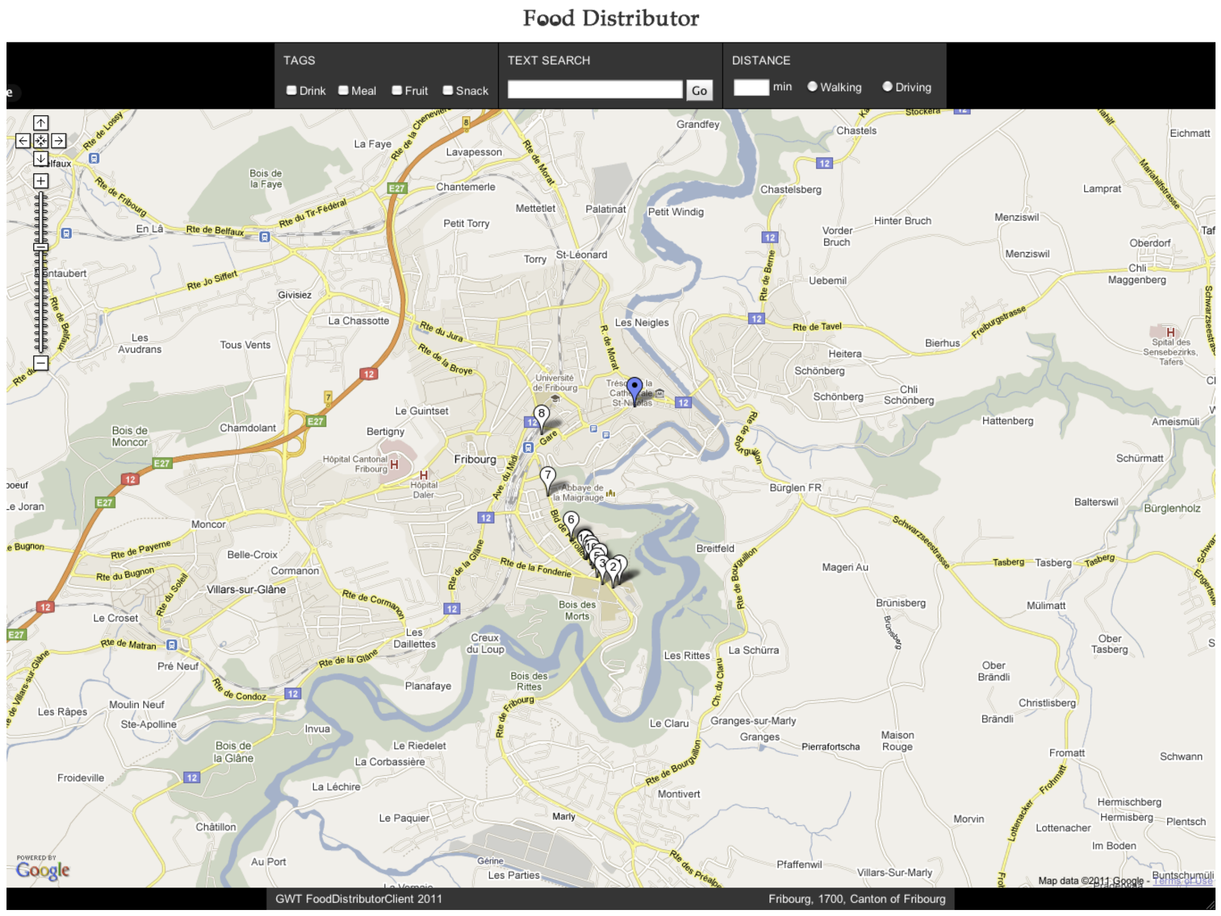

Food Distributor
TAGS
Drink
Meal
Fruit
Snack
TEXT SEARCH
Go
DISTANCE
min
Walking
Driving
GWT FoodDistributorClient 2011
Fribourg, 1700, Canton of Fribourg
Map data ©2011 Google - Terms of Use

Figure 12: FoodDistributor client status quo

4.5.3 Filters

There are two filter options which are depicted in **Figure 13** and **Figure 15**.

- Filter by tag
- Filter by search keyword

Changing these filters will fire requests to the server, pulling new information. For example, if the tags are changed to “Drink” and “Fruit”, products matching these tags are loaded and displayed on the current map. Now only products, which match these criteria, are being displayed, as depicted in **Figure 14**.

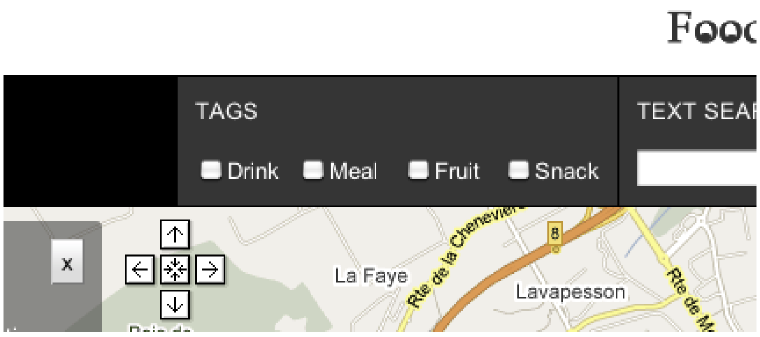


Figure 13: Filter by tags

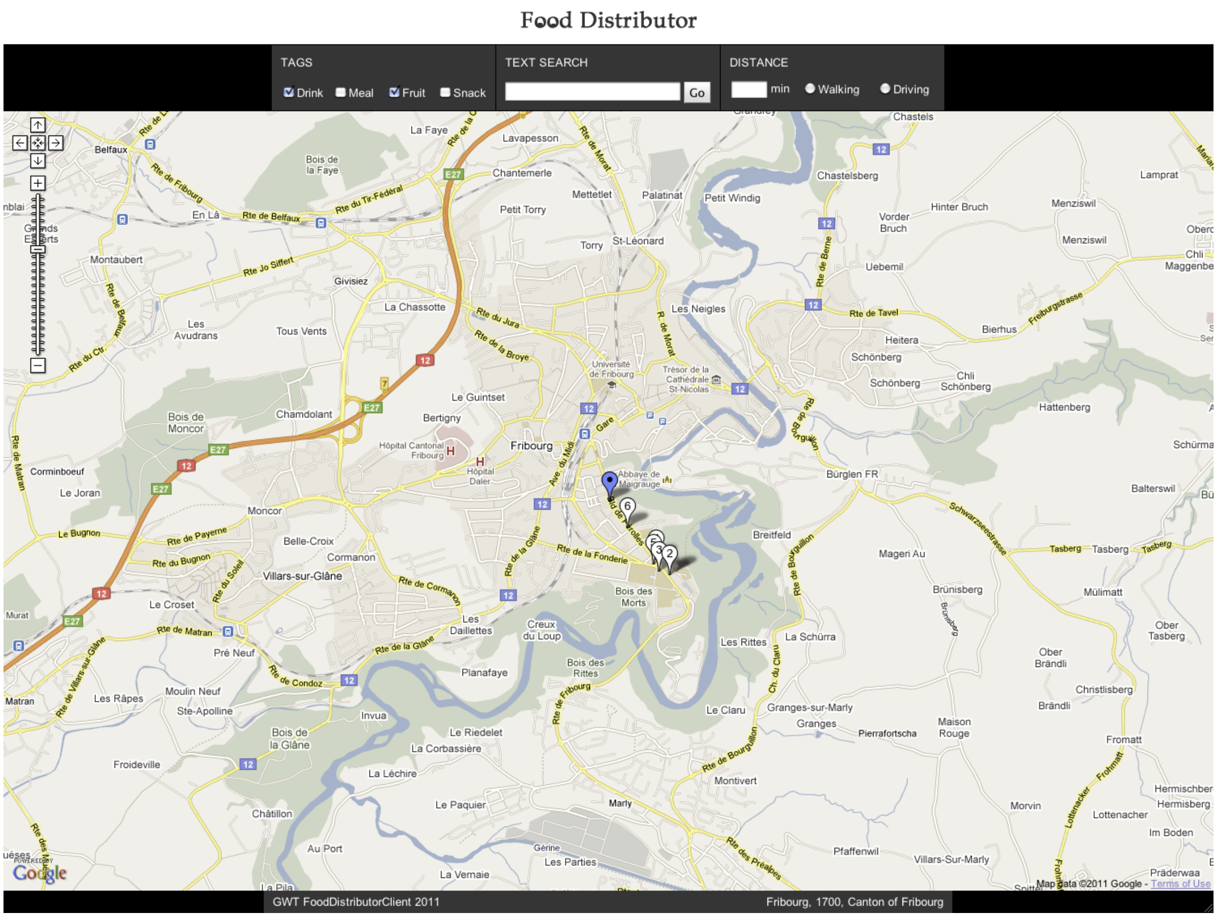

Food Distributor
TAGS
Drink
Meal
Fruit
Snack
TEXT SEARCH
Go
DISTANCE
min
Walking
Driving
Fribourg
GWT FoodDistributorClient 2011
Fribourg, 1700, Canton of Fribourg
Map data ©2011 Google - Terms of Use

Figure 14: Using the tag filter

If the tags change, the client application will fire a GET request to the REST API. In the figure above, the following request has been fired:

```
GET www.example.com/rest/products?tags=2;6 HTTP/1.1
```

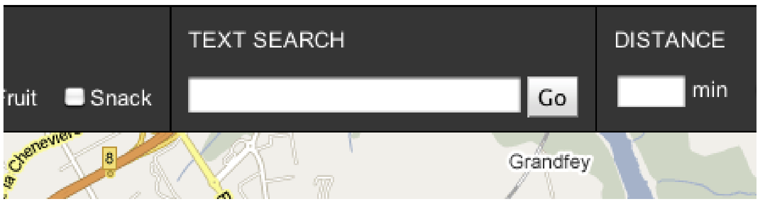


Figure 15: Filter by text search

The text form supports suggestions. When starting to type, matching foods show up, as shown in **Figure 16**. The suggestions are loaded once at the application launch. Unlike the tag checkboxes, the request is not fired when the value of the form changes. The request to the REST API is not fired until the user clicks on "Go".

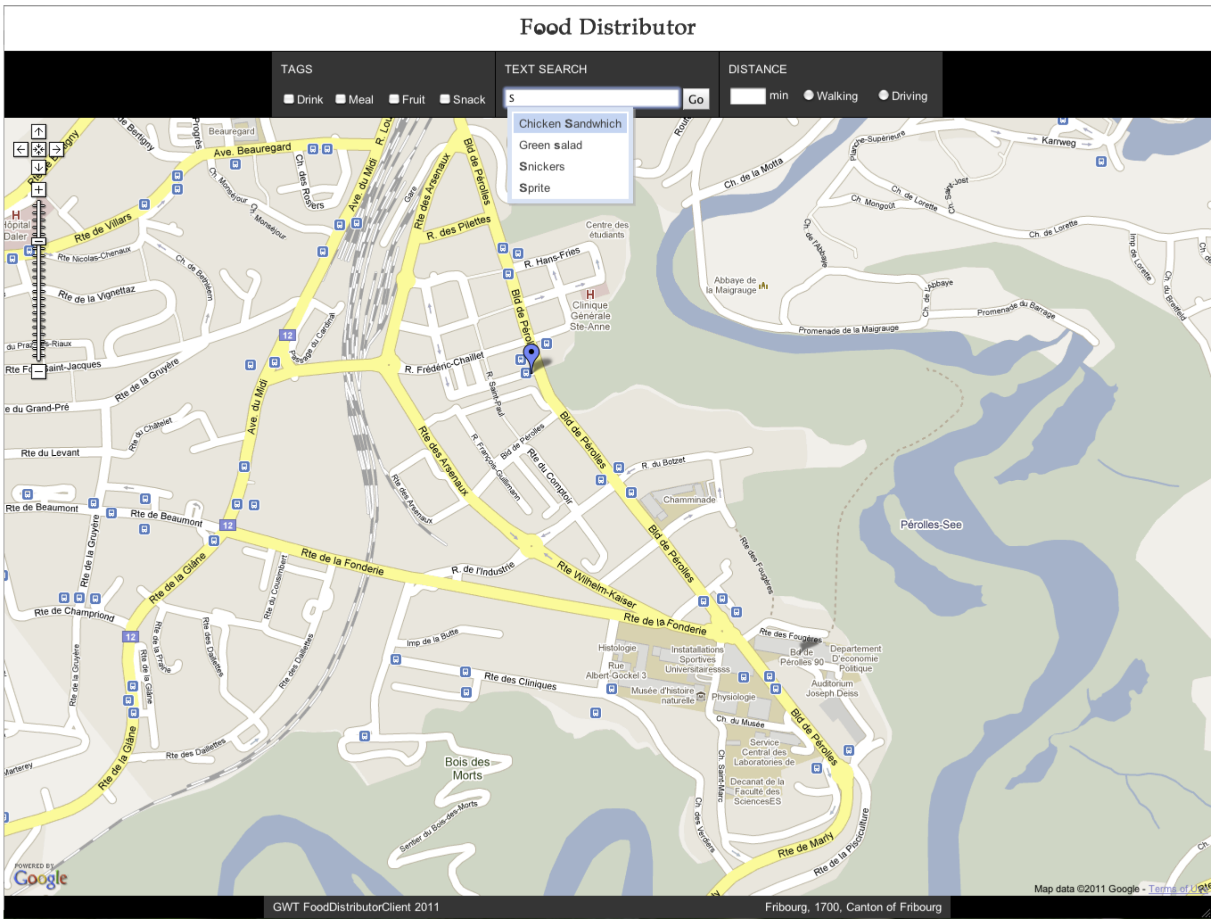


Figure 16: Using the text filter.

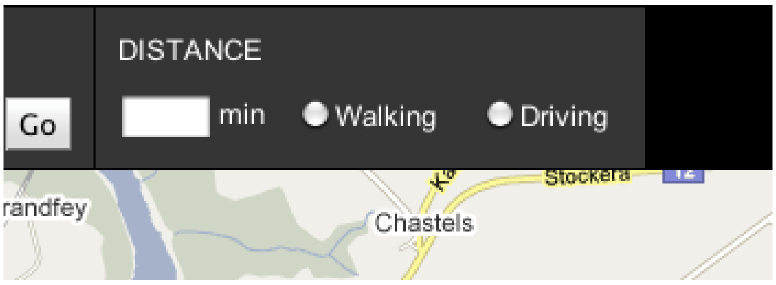


Figure 17: Filter by distance

The distance filter, depicted in **Figure 17**, is not a proper filter like the tag or text filter. The distance filter does not change the data being displayed nor does it pull any information from the REST API. It only adds an overlay (see **Figure 18**) to the map, indicating a rough radius. This helps the user to find products in its proximity. The dimensions of the overlay are calculated by an average movement speed and air-line distance.

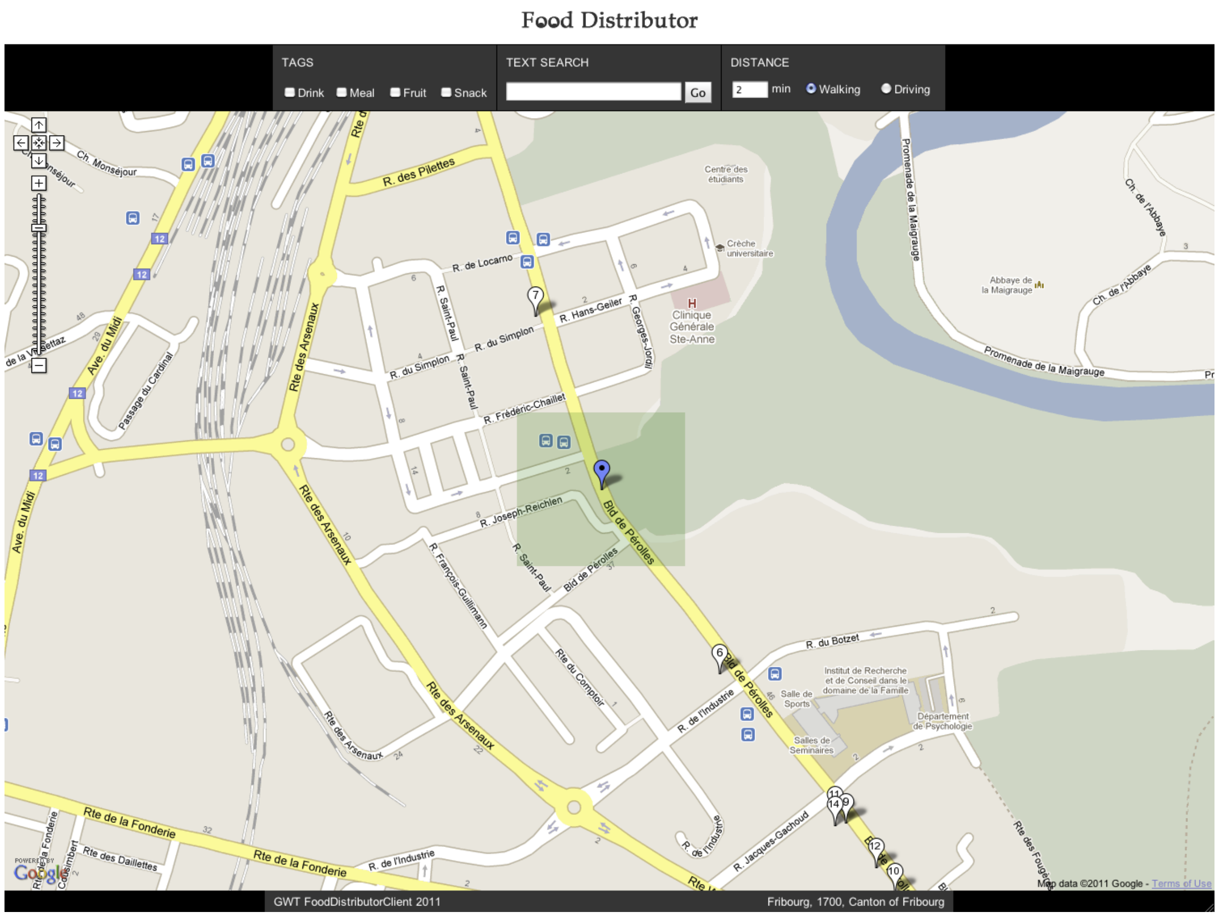


Figure 18: Distance overlay indicating a rough estimate.

4.5.4 Selecting Locations

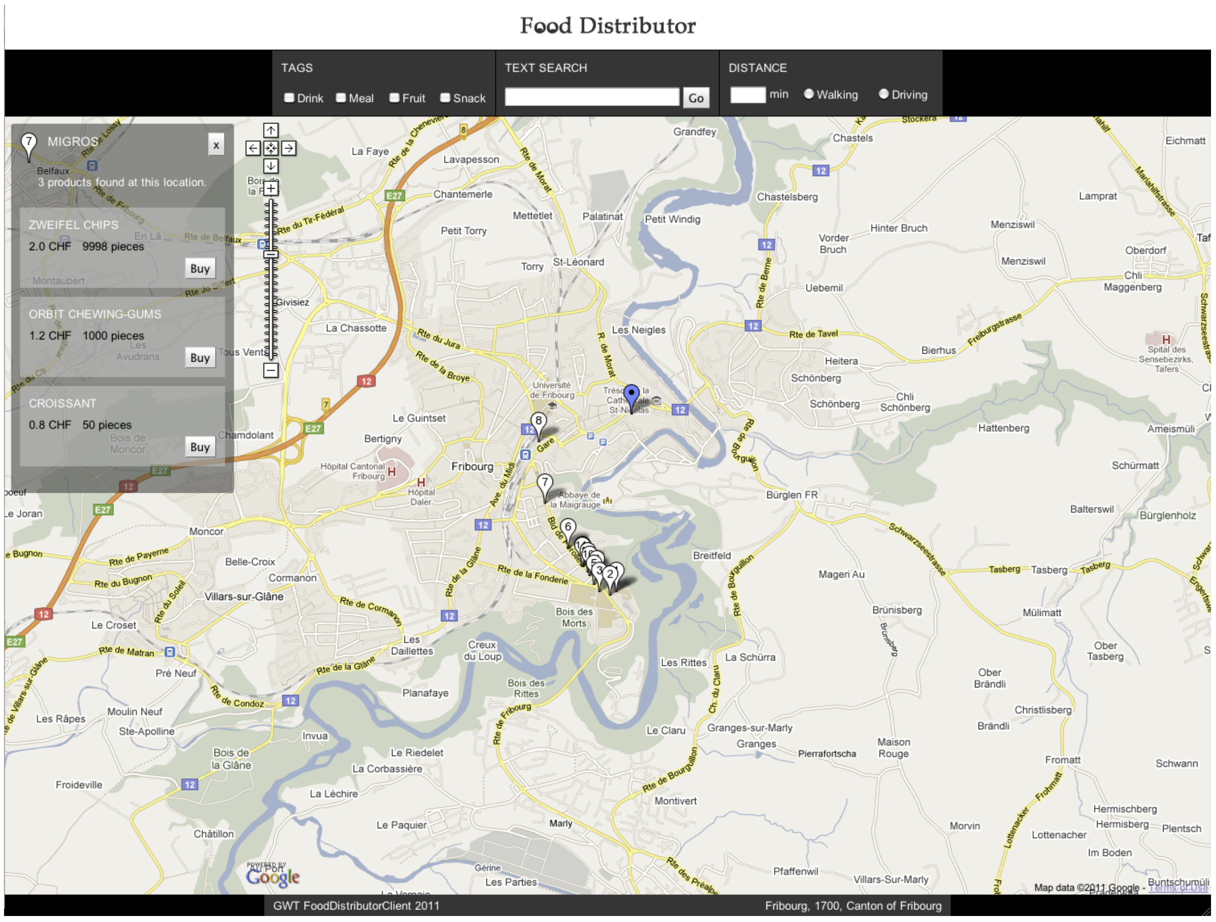
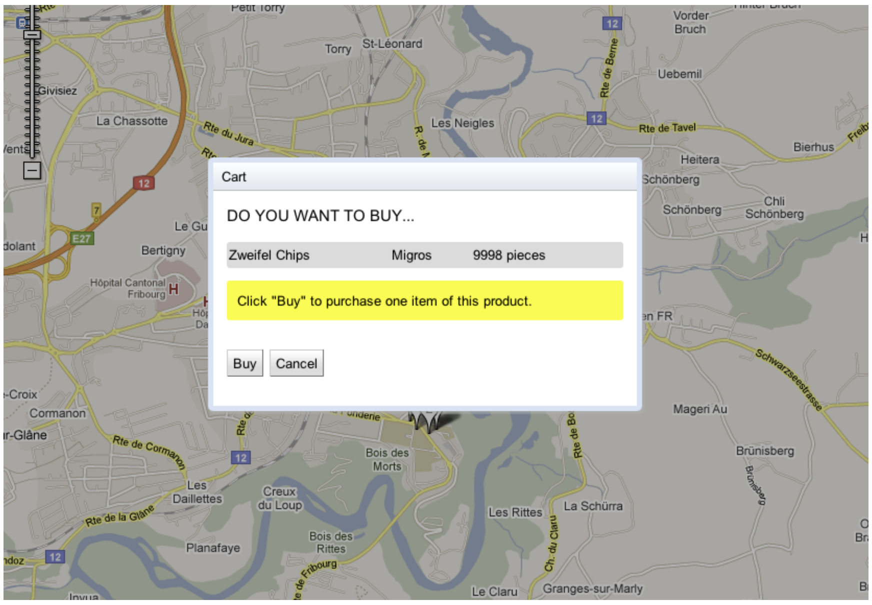


Figure 19: List of products at location 7.

When clicking on a location, a grey box shows up on the left listing the products available at this location (see **Figure 19**). This does not pull information from the server, as all the products are already loaded and saved in an array. The map controls automatically move to the right. When there are products on stock, there is the option to buy the product. In case there are too many products to fit inside the box, pagination is used. The box, including pagination, automatically resizes, when the window height changes.

4.5.5 Buying Products



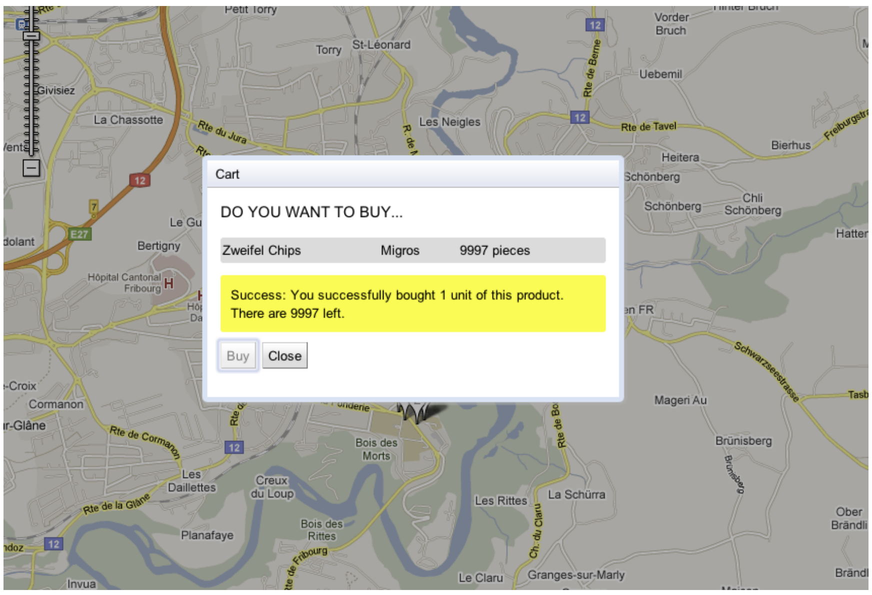
DO YOU WANT TO BUY...

Zweifel Chips	Migros	9998 pieces

Click "Buy" to purchase one item of this product.

Figure 20: Modal box for buying a product.

As depicted in **Figure 20**, clicking on "Buy" will fire a PUT request to the REST API, trying to buy the product. If the quantity has not changed to 0 in the meantime, this will be possible. The REST API will respond with a response, whether it has worked or not (see **Figure 21**).



DO YOU WANT TO BUY...

Zweifel Chips	Migros	9997 pieces

Success: You successfully bought 1 unit of this product.
There are 9997 left.

Figure 21: After PUT request, status message is shown.

5 Conclusion

5.1 Server side

5.1.1 Lessons learnt

Implementing a REST API in Java is an easy task, thanks to existing REST frameworks. REST is widely adopted and tools exist – in all popular programming languages – which allow developers to focus on their application and not on the REST concepts.

For Java there are a handful of capable REST frameworks. Jersey seems to me to be a good starting point. There is a good documentation and there are hundreds of helpful forum threads about possible pitfalls, error messages and tips.

Understanding the REST principles by reading books and articles seems to be a fairly easy task in the beginning. But once I started developing I quickly found myself in doubts, realising that there is more to REST than just principles. Knowing the REST theory is a start, but at the end it comes down to details and implementation. This process of decoding the theory into a concrete implementation was intelectually challenging. But only after doing so, I feel like I can really grasp the REST principles.

5.1.2 Future Improvements

At the moment the FoodDistributor API only serves a couple of use-cases. In the future it might be interesting to make more resources available.

Buying products via the PUT request is not thread-safe. Using this REST API on a bigger scale with hundreds of operations per second, this would be a serious problem. In such a scenario, a thread-safe buy transaction is a must.

Also, caching would be necessary to make the FoodDistributor more scalable.

5.2 Client side

5.2.1 Lessons learnt

Developing the client application was engaging. GWT makes rapid prototyping possible thanks to existing plugins, excellent documentation and tools like the GWT Designer. The combination between GWT and Google Maps API works flawlessly. The auto-suggestion feature in Eclipse spares the developer to consult the documentation. I have worked with GWT before, which helped me to jump right into the implementation. For a GWT beginner, getting to know the framework takes some time.

The GWTQuery plugin enables transitions and effects, which GWT is lacking until today.

Consuming a REST service is easily done with the Java backend. But this adds quite a lot of overhead to the application and increases the deployment specifications. At the moment, it is unfortunately not possible to just use JavaScript due to the SOP. Without the SOP restrictions it would be possible to create web applications, which do not rely on a server backend and run entirely in the JavaScript engine of the browser.

5.2.2 Future Improvements

First of all, the user interface could be more sophisticated. Products could be updated when the window resizes or the zoom level of the map changes. Creating a richer but equally responsive user interface would be a technical challenge.

To enrich the use-case, the functionality of the application could be extended by the possibility of using a shopping cart for buying products.

From a technical point of view it would be highly interesting to apply HTML5 features like the local database storage. This would allow a new application design, because data can be loaded step by step into the local database. The user would benefit from a more responsive interface and offline capability. It is possible to cache Google Maps, which would allow the user to use the map even when being offline.

5.3 Final Statement

To design effective REST services, I believe one has got to look at it from the client perspective. In retrospect, I should have thought about the client application first in order to create a more user-friendly REST service. The principles mentioned in Chapter 2 give a solid guideline of how to program a REST API without any practical knowledge. But the actual meaning and the priority of these principles only became clear as I began to use the API with my client application. I realised that one can implement the principles in different ways and that in reality, unlike in theory, they are blurred. For example the HATEOAS principle is theoretically easy to understand and to implement. But in practice, to provide HATEOAS, which is useful to all sorts of client scenarios is not as easy as it first seems. It comes down to usabilty questions like “what information does the client expect?”. Clients will use the API out of completely different contextes. Because of this big diversity, it is not easy to find a clear answer to the question above. Next time, instead of just starting to program the API, I will start with programming the client application first. To do so, it will be necessary to think about the REST principles more concretely.

I started this bachelor thesis by mentioning that web services of Google, Twitter, Facebook and so forth were able to expand amongst other reasons because of their API’s. After doing research in this field and implementing REST services on my own, I feel that this is just the beginning. Not only big companies like the ones mentioned can spread their data by using REST services. In theory, all publicly available data should be accessible via a REST service. Like that new mashups are possible and enables content providers to spread their content over the web. An example for such a mashup could be a special reader for mobile devices, which fetches content and then displays them suitable for mobile devices. Visiting a normal web site on a mobile device as the iPhone is possible but not as effective and user-friendly as such a mobile reader.

The web how we find it today, is not yet optimised for machines to read it. It is made for human beings, using a visual browser for accessing information. On the web, data and design are melted together. From a conceptual and technical point of view this does not make sense. REST services are one way of how the web can separate between visual design and content. By doing this, the functionality and importance of the web itself will increase once more.

A
Glossary

AJAX	Asynchronous Javascript And XML
API	Application Programming Interface
CSP	Content Security Policy
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
POJO	Plan Old Java Object
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SOP	Same Origin Policy
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UI	User Interface
XML	Extensible Markup Language

B CD-ROM

On the CD-ROM attached below you will find:

- The source code of the FoodDistributor REST API.
- The source code of the FoodDistributorClient mashup.
- The binaries and sources of this documentation.
- Instructions about how to setup and to deploy the REST server and the client application.
- Various documents that were of great use during this bachelor thesis.

C References

1. **Fielding, Roy Thomas.** *Architectural Styles and the Design of Network-based Software Architectures.* s.l. : University of California, Irvine, 2000.

2. **Richardson, Leonard and Ruby, Sam.** *RESTful Web Services.* Sebastopol : O'Reilly Media, 2007.

3. **Burke, Bill.** *RESTful Java with JAX-RS.* Sebastopol : O'Reilly Media, 2009.

4. **Pautasso, Cesare, Zimmermann, Olaf and Leymann, Frank.** *RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision.* Beijing : University of Lugano, 2008.

5. **Wilde, Erik.** *Putting Things to REST.* Berkeley : School of Information, UC Berkeley, 2007.

6. **Harold, Elliotte.** POST vs. PUT. *Mokka mit Schlag » Web Development.* [Online] 8 12 2005. [Cited: 18 05 2011.] http://www.elharo.com/blog/software-development/web-development/2005/12/08/post-vs-put/.

7. **Pautasso, Cesare and Wilde, Erik.** *Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design.* Lugano : University of Lugano, 2009.

8. **ProgrammableWeb.** programmableweb.com. [Online] 29 03 2011. [Cited: 29 03 2011.] http://www.programmableweb.com/.

9. **O'reilly, Dan Kubb.** http://www.xml.com. *O'reilly xml.com.* [Online] 19 04 2006. [Cited: 30 03 2011.] http://www.xml.com/pub/a/2006/04/19/rest-on-rails.html.

10. **Java.net.** Java Jersey. *java.net.* [Online] [Cited: 30 03 2011.] http://jersey.java.net/.

11. **JAX-RS.** JAX-RS Specification. *JAX-RS.* [Online] [Cited: 30 03 2011.] http://jcp.org/en/jsr/detail?id=311.

12. **RFC 2616 Fielding, et al.** HTTP Specification Protocol Parameters. [Online] [Cited: 30 03 2011.] http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html.

13. **Java.net.** JAXB Reference Implementation. [Online] [Cited: 30 03 2011.] http://jaxb.java.net/.

14. **Ramsdale, Chris.** Large scale application development and MVP. [Online] 03 2010. [Cited: 01 04 2011.] http://code.google.com/webtoolkit/articles/mvp-architecture.html.

15. **Google.** GWT 2.2 Javadoc. *GWT Javadoc.* [Online] [Cited: 11 04 2011.] http://google-web-toolkit.googlecode.com/svn/javadoc/2.2/index.html.

Index