SimpleFleet	**SimpleFleet**
	TrafficAPI

Author(s)	Athanasios Mantes (TALENT)			
Project	SimpleFleet – Democratizing Fleet Management			
Date	*Contractual:*	30.04.2013	*Actual:*	19.04.2013
Project Coordinator	Rüdiger Ebendt Deutsches Zentrum für Luft- und Raumfahrt (DLR) Tel: +49 30 67055 287 E-mail: ruediger.ebendt@dlr.de			

Abstract	The TrafficAPI provides traffic related data and functionality to SimpleFleet platform users through REST web services and efficient caching mechanisms of data provided by the TrafficStore
Keyword list	TrafficAPI, REST web services, TrafficStore, JAX-RS, Tomcat, Servlet, SQLite, SpatiaLite, JDBC,
Nature of deliverable	Report
Dissemination	Confidential

Project financially supported by	
SEVENTH FRAMEWORK PROGRAMME	**European Commission** **DG CONNECT**
Project number 296423 **FP7-ICT-2011-SME-DCL**	

Control sheet

Version history			
Version number	**Date**	**Main author**	**Summary of changes**
1.0	15.04.2013	Athanasios Mantes (TALENT)	Initial version
1.1	19.04.2013	Athanasios Mantes (TALENT)	Minor additions

Approval		
	Name	**Date**
Prepared	Manolis Koutlis (TALENT)	19.04.2013
Reviewed	All partners	19.04.2013
Authorized	R. Ebendt (DLR)	19.04.2013

Circulation	
Recipient	**Date of submission**
European Commission	19.04.2013

Table of Contents

1 Introduction 5

2 API Design Decisions 7

2.1 The Representational State Transfer architecture 8

2.2 Constraints 8

2.3 Concept 9

2.4 Guiding principles of the interface 10

2.4.1 Identification of resources 10

2.4.2 Manipulation of resources through these representations 10

2.4.3 Self-descriptive messages 10

2.4.4 Hypermedia as the engine of application state (aka HATEOAS) 10

2.4.5 Central principle 11

2.4.6 RESTful web APIs 11

3 The TrafficAPI 12

3.1 The Data Model 12

3.1.1 Node 12

3.1.2 Link 13

3.1.3 LinkTemporal 14

3.2 Resources 16

3.2.1 NodesResourceGroup 17

3.2.2 LinksResourceGroup 17

3.2.3 LinksTemporalResourceGroup 20

4 Implementation 22

4.1 JAX-RS 24

4.2 SQLite 27

4.3 SQLite JDBC Driver (Xerial) 29

4.4 SpatiaLite 29

4.5 STRtree 29

4.6 OGR Tools 30

4.7 Putting it all together 30

4.8 Enunciate 32

4.9 Servers and testbeds 33

4.10 Where is the TrafficSDK? 34
5 Conclusion and future work 35
6 References 36

1 Introduction

The purpose of SimpleFleet project is to make it easy for SMEs, both, from a technological and business perspective, to create (Mobile) Web-based fleet management applications. For this purpose, an API providing users the necessary access to traffic data and functionality is required. The TrafficAPI stands of the shoulders of the TrafficStore; it uses data provided by the TrafficStore to provide the clients with functionality.

In that sense, the TrafficAPI plays a dual role. It provides users and clients of the SimpleFleet services access to the TrafficStore data, and it exposes functionality that is deemed necessary for common operations the users of the SimpleFleet services will need to perform. In other words, in the case of a SimpleFleet services integrator, it stands between the application the integrator develops, and the SimpleFleet data pool the TrafficStore provides.

The TrafficAPI is currently hosted on a Virtual machine of the okeanos IaaS (Infrastructure as a Service) platform of the Greek Research and Technology Network, a service comparable to Amazons Elastic Compute Cloud (Amazon EC2) making migration to such a commercial service simple. It has also been tested to other linux servers (including a Windows server).

Figure 1 relates the current deliverable to the rest of core technical deliverables. Deliverable D1.1 provided information about the data sources used in the SimpleFleet project. Deliverable D1.2 (TrafficStore) provides the glue that links data collection, map-matching (described in the D2.2 deliverable) and travel time aggregation and speed-profile computation together. Several services are to be built on top of the TrafficStore such as Time-parametrized shortest-path computation (Deliverable D3.1), Business Intelligence (Deliverable D3.2), Data visualization techniques (Deliverable D3.3), and the current TrafficAPI and TrafficSDK deliverable (Deliverable D4.1).

The outline of this deliverable is as follows: Section 2 argues about the TrafficAPI design decisions made and about the reason the TrafficAPI is a REST web services based API and for using XML and JSON formatted data. Section 3 describes the TrafficAPI in detail, starting from the data model objects and advancing to resources and mount points currently provided by the API. Some usage examples are also provided. Section 4 talks about the implementation; all involved technologies and frameworks are described in good detail, arguing about the reasons they were chosen. It also contains some details about the TrafficAPI documentation process and discusses the subject of the Traffic SDK. Finally, Section 5 contains some wrap-up conclusions about the TrafficAPI development so far and discusses future tasks and procedures. Section 6 contains some references about everything discussed in this deliverable.

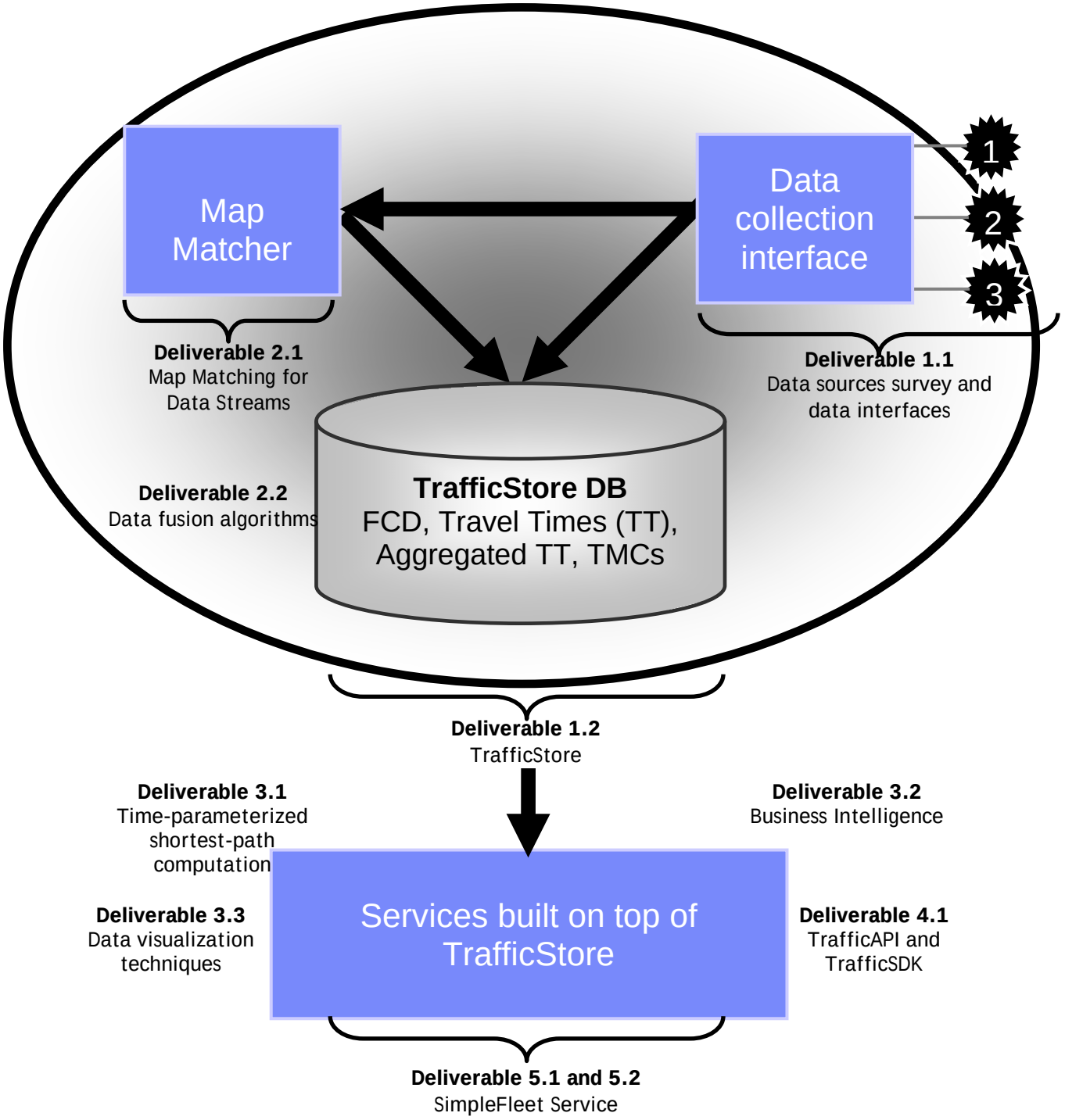

Figure 1: Core technical deliverables overview and integration

2 API Design Decisions

The main goal for the TrafficAPI is to provide all functionality needed so that all aspects of information stored in the TrafficStore can be passed on to the traffic data consumers. However, there is an important parameter that must be taken into account: the consumers/customers/users of the TrafficStore information and TrafficAPI functionality are not the same. The users can be either desktop applications, enterprise platforms, mobile devices or even webpages. Also, every user category mentioned can contain API users that are different in terms of operating system, development environment, hardware etc. So, the target for the TrafficAPI is to provide data and functionality to a large number of diverse users.

For efficiency reasons in terms of development of the TrafficAPI, it is best to handle the user diversity issue by trying to provide a "common ground" on which all users can have access to the same functionality. By that approach, maintenance is a common task for all cases, and every addition or correction passes to all API users at the same time. This means that the "common ground" that the TrafficAPI will provide, should be as independent as possible from details such as user operating system, development language or framework, etc. In other words, we need a standard way to provide functionality in the same form for each and every API user.

What comes to mind in the first place is that to provide the same information to different users/platforms, it is best to use Web standards, such as JSON or XML. By formatting the information provided by the TrafficStore using JSON or XML, one can automatically make this information usable to a multitude of platforms, application frameworks and programming language environments. These formats are so common that (at least nowadays) there isn't any programming language or application development framework that doesn't provide a standardized mechanism for parsing, serializing, and generally handling information formatted in XML or JSON. So by mainly using these formats we can achieve that the exact same information or data passed to an Iphone, a browser on Linux or a JBoss application server is easily "readable", and more importantly, without the user or developer to have to write a single line of extra code for handling these formats. All are already there in each application framework or programming environment.

So far, we have a common language between the information distributed by the API and the users. What is also required is a communication channel that can pass the information from the API down to the clients. An important detail: the TrafficAPI is used only for providing (processed or unprocessed) data or information FROM the TrafficStore TO the users. In other words, information flow is unidirectional. This has its own importance, which will be made clear in the following paragraphs.

In order to achieve the "one solution fits all" goal we described early on, we came down to the solution of using a web API. Accessing information through web API calls is a commodity for all programming languages or application frameworks. After mainly considering the cases of REST web services and SOAP implementations, we decided that for the TrafficAPI a REST web services solution should be used. The following sections analyze the REST architecture and

provide the necessary justification for choosing REST web services as the basis of the TrafficAPI.

2.1 The Representational State Transfer architecture

Representational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant web API design model. The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered in transition. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition. REST is less strongly typed than its counterpart, SOAP. The REST language uses nouns and verbs, and has an emphasis on readability. Unlike SOAP, REST does not require XML parsing and does not require a message header to and from a service provider. This ultimately uses less bandwidth. REST error-handling also differs from that used by SOAP.

Key goals of REST include:

- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

2.2 Constraints

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components free to design:

1. **Client–server.** A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.
2. **Stateless.** The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all

of the information necessary to service the request, and any session state is held in the client.

3. **Cacheable**. As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
4. **Layered system.** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. They may also enforce security policies.
5. **Code on demand (optional).** Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.
6. **Uniform interface.** The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are detailed below.

The only optional constraint of REST architecture is "code on demand". One can characterize applications conforming to the REST constraints described in this section as "RESTful". If a service violates any of the required constraints, it cannot be considered RESTful.

Complying with these constraints, and thus conforming to the REST architectural-style, enables any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

2.3 Concept

Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: presented with a network of Web pages (a virtual state-machine), the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.

REST was initially described in the context of HTTP, but it is not limited to that protocol. RESTful architectures may be based on other Application Layer protocols if they already provide a rich and uniform vocabulary for applications based on the transfer of meaningful representational state. RESTful applications maximize the use of the existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it.

Vocabulary re-use vs. its arbitrary extension: HTTP and SOAP

In addition to URIs; Internet media types; request and response codes; etc., HTTP has a vocabulary of operations called request methods, most notably:

- GET
- POST
- PUT
- DELETE

REST uses these operations and other existing features of the HTTP protocol. For example, layered proxy and gateway components perform additional functions on the network, such as HTTP caching and security enforcement.

SOAP RPC over HTTP, on the other hand, encourages each application designer to define new, application specific operations that supplant HTTP operations. This additive, "re-invention of the wheel" vocabulary — defined on the spot and subject to individual judgment or preference — disregards many of HTTP's existing capabilities, such as authentication, caching, and content-type negotiation. The advantage of SOAP over REST comes from this same limitation: since it does not take advantage of HTTP conventions, SOAP works equally well over raw TCP, named pipes, message queues, etc.

2.4 Guiding principles of the interface

The uniform interface that any REST interface must provide is considered fundamental to the design of any REST service.

2.4.1 Identification of resources

Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, perhaps, some HTML, XML or JSON that represents some database records expressed, for instance, in Swahili and encoded in UTF-8, depending on the details of the request and the server implementation.

2.4.2 Manipulation of resources through these representations

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

2.4.3 Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability.

2.4.4 Hypermedia as the engine of application state (aka HATEOAS)

Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for

any particular resources beyond those described in representations previously received from the server.

2.4.5 Central principle

An important concept in REST is the existence of resources (sources of specific information), each of which is referenced with a global identifier (e.g., a URI in HTTP). In order to manipulate these resources, components of the network (user agents and origin servers) communicate via a standardized interface (e.g., HTTP) and exchange representations of these resources (the actual documents conveying the information). For example, a resource that represents a circle (as a logical object) may accept and return a representation that specifies a center point and radius, formatted in SVG, but may also accept and return a representation that specifies any three distinct points along the curve (since this also uniquely identifies a circle) as a comma-separated list.

Any number of connectors (e.g., clients, servers, caches, tunnels, etc.) can mediate the request, but each does so without "seeing past" its own request (referred to as "layering", another constraint of REST and a common principle in many other parts of information and networking architecture). Thus, an application can interact with a resource by knowing two things: the identifier of the resource and the action required—it does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between it and the server actually holding the information. The application does, however, need to understand the format of the information (representation) returned, which is typically an HTML, XML, or JSON document of some kind, although it may be an image, plain text, or any other content.

2.4.6 RESTful web APIs

A RESTful web API (also called a RESTful web service) is a web API implemented using HTTP and the principles of REST. It is a collection of resources, with four defined aspects:

- the base URI for the web API, such as http://example.com/resources/
- the Internet media type of the data supported by the web API. This is often JSON but can be any other valid Internet media type provided that it is a valid hypertext standard.
- the set of operations supported by the web API using HTTP methods (e.g., GET, PUT, POST, or DELETE).
- The API must be hypertext driven [13]

The PUT and DELETE methods are idempotent methods. The GET method is a safe method (or nullipotent), meaning that calling it produces no side-effects.

Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs.[14] This is because REST is an architectural style, unlike SOAP, which is a protocol. Even though REST is not a standard, a RESTful implementation such as the Web can use standards like HTTP, URI, XML, etc, and that was widely used in our case.

3 The TrafficAPI

In the previous section we analyzed the decisions that led to the use of the REST Web service architecture in order to offer data access and functionality from the TrafficStore to users with different characteristics concerning application environment, operating system, target platform, etc. This section describes the functionality offered from TrafficAPI to service clients.

3.1 The Data Model

All API functionality is based on a set of objects that the REST service sends to the clients through specific Resource requests. These objects make the Data Model of the TrafficAPI and are essentially entities based on the data organization of the datasets in the TrafficStore. The most important of these objects are:

- Node,
- Link,
- LinkTemporal

There are also some other objects, for the cases where data returned to users are not in any of the above cases. These include simple wrapper objects for integers, doubles etc, a BoundingBox object representing a wrapper for a simple rectangle, etc. One can find details about these objects at the documentation of the API, which is described at the end of the Implementation section. Here, only the three basic data model objects described above will be analysed.

3.1.1 Node

A Node is a simple point on a city road network dataset, where one or many links can start or end to. The respective XML and JSON representations are the following:

XML

```
<node>
     <ID>...</ID>
     <name>...</name>
     <type>...</type>
     <active>...</active>
     <partitionID>...</partitionID>
     <cost>...</cost>
     <level>...</level>
     <parentNodeID>...</parentNodeID>
     <geometryAsWKT>...</geometryAsWKT>
</node>
```

JSON

```
{
    "ID" : ...,
    "name" : "...",
    "type" : "...",
    "active" : false,
    "partitionID" : ...,
    "cost" : ...,
    "level" : ...,
    "parentNodeID" : ...,
    "geometryAsWKT" : "..."
}
```

Most of the properties are self-explanatory, and details about these can be found in the respective deliverable about the TrafficStore database schema. Note that the geometry of the Node object (which, in the TrafficStore is a JTS geometry object) is provided in a WKT form: This is the standard way of serializing/deserializing JTS geometry objects using string representation. WKT libraries for converting string representations to geometry objects and vice versa exist for all spatial related frameworks on the various application platforms.

3.1.2 Link

The main Link object is representing a street segment entity joining two distinct points on a city road network, not interrupted or intersected by other links. The following example shows a Link object from the Berlin links dataset both in JSON and XML format.

JSON

```
{
        "ID":"377",
        "linkName": "Schöneberger Straße",
        "level":"7",
        "active":"true",
        "bidirected":"true",
        "cost":"0.0",
        "startNodeID":"26554177",
        "endNodeID":"26554180",
        "parentLinkID":"0",
        "geometryAsWKT":"LINESTRING (
                389814.52066304 5814150.61543867,
                389734.951993608 5814233.84597929)",
        "geometryLatLonAsWKT":"LINESTRING (
                22.380152243464348 52.46885619133467,
                22.37895393079904 52.469588101160184)",
}
```

Again, properties for the link are self-explanatory. Note that a link object contains two Node IDs, the IDs of the start and end Node objects for the respective link.

XML

```
<link>
    <ID>377</ID>
    <linkName>Schöneberger Straße</linkName>
    <level>7</level>
    <active>true</active>
    <bidirected>true</bidirected>
    <cost>0.0</cost>
    <startNodeID>26554177</startNodeID>
    <endNodeID>26554180</endNodeID>
    <parentLinkID>0</parentLinkID>
    <geometryAsWKT>
       LINESTRING (
          389814.52066304 5814150.61543867,
          389734.951993608 5814233.84597929)
    </geometryAsWKT>
    < geometryLatLonAsWKT>
       LINESTRING (
              22.380152243464348 52.46885619133467,
              22.37895393079904 52.469588101160184)
    </geometryLatLonAsWKT>
</link>
```

The link object contains two more properties which in fact is the WKT string representation of the geometry of the link. The API provides two WKT geometry representations: One is the WKT string for the link geometry on the dataset's native Coordinate Reference System (CRS), and the other is the WKT representation of the link in WGS84 geographic (lat-long) coordinates. This seemed to be necessary because a lot of users of the TrafficAPI may need the geometry information of the links in lat-long coordinates. Also, in the future, a functionality of transforming link geometry in other extensively used CRSs (for example Google's Spherical Mercator) may be provided.

3.1.3 LinkTemporal

The TrafficStore provides data for the entire links network of a city. But this is a static, non-changing information. The main aspect of the TrafficStore data is the spatio-temporal dimension of the road network. This means that we need to provide temporal information (in terms of traffic data) for links. To achieve this, we extended the original Link entity with some traffic related properties. Of course, this is supported by the TrafficStore's database schema. This entity is the LinkTemporal entity.

Note that the TrafficStore, in terms of traffic information for a city's road network provides two "aspects". The first is the "live" traffic data, which is the network's "live" traffic status, and the second is the "historical" data, which represent the status of the network at a specific time in the past. TrafficStore architecture covers both live and historical data with the same common parameters. Following the same guideline, the LinkTemporal entity covers communication of both live and historical traffic data for links.

An example of the link with ID =377 we saw previously from the link network of the Berlin dataset is presented below, in again, JSON and XML form:

JSON


```
{
        "ID":"377",
        "linkName": "Schöneberger Straße",
        "level":"7",
        "active":"true",
        "bidirected":"true",
        "cost":"0.0",
        "startNodeID":"26554177",
        "endNodeID":"26554180",
        "parentLinkID":"0",
        "geometryAsWKT":"LINESTRING (389814.52066304 5814150.61543867,
389734.951993608 5814233.84597929)",
        "geometryLatLonAsWKT":"LINESTRING (22.380152243464348
52.46885619133467, 22.37895393079904 52.469588101160184)",
        "speed":"34",
        "speedLimit":"60",
        "speedRatio":"57",
        "travelTime":"122",
        "timestampLatest":"1365703310300"
}
```


The extra temporal, traffic related properties are the current speed of vehicles travelling the link, the speedLimit for this link, the speedRatio (the percentage of the current speed to the speed limit of the link , a calculated field), the travel time for this link in terms of tenths of seconds, and the timestamp at which data for the specific link were collected.

XML


```
<linktemporal>
    <ID>377</ID>
    <linkName>Schöneberger Straße</linkName>
    <level>7</level>
    <active>true</active>
    <bidirected>true</bidirected>
    <cost>0.0</cost>
    <startNodeID>26554177</startNodeID>
    <endNodeID>26554180</endNodeID>
    <parentLinkID>0</parentLinkID>
    <geometryAsWKT>
       LINESTRING (
          389814.52066304 5814150.61543867,
          389734.951993608 5814233.84597929)
    </geometryAsWKT>
    < geometryLatLonAsWKT>
       LINESTRING (
              22.380152243464348 52.46885619133467,
              22.37895393079904 52.469588101160184)
    </geometryLatLonAsWKT>
    <speed>30</speed>
    <speedLimit>60</speedLimit>
    <speedRatio>50</speedRatio>
    <travelTime>138</travelTime>
    <timestampLatest>1365702110737</timestampLatest>
</linktemporal>
```

3.2 Resources

When we discussed the REST web services architecture details, we saw that Resources were the key concept of providing data and functionality to the users of a REST web service. A Resource represents a specific functionality provided by the API, and is uniquely determined by a URI containing specific Path and Query parameters. In the TrafficAPI's case we already argued that data "flows" from the server to the users/clients through simple HTTP GET HTTP requests: An API user sends a HTTP GET request to the server in order to receive some data. The kind of data or functionality that the user expects to receive is determined by the URI the user sends to the server.

Following the Data Model we described previously, the TrafficAPI resources revolve around the three main Data Model classes: the Node entity, the Link entity and the LinkTemporal entity. So, the web service resources of the API are based on:

- the NodesResourceGroup,
- the LinksResourceGroup,
- and the LinksTemporalResourceGroup.

These are described in detail below.

Note that in all cases, words contained in the URI path inside **{}** represent Path Parameters, whereas *param=value* elements of the path (following the URI path) are the Query Parameters. For example, since the SimpleFleet Project is targeted on the case of three major European cities (Athens, Berlin and Vienna), in all REST resource cases of the TrafficAPI the Path Parameter {city} is the lowercase name of the city for which data are requested by a user. So, the {city} path parameter can take three distinct values:

{city}=athens|berlin|vienna

Also, in all REST resource cases for the TrafficAPI, there is a Query Parameter that applies to all resource URI paths. This is the **xml=true|false** query parameter. This parameter simply controls the output format of the data requested by a user. Obviously, **xml=true** returns results formatted in XML. This query parameter is not mandatory, and by not including it in a HTTP GET request, all results are formatted by default in JSON. In the future, when maybe more data formats or functionality cases will be implemented, this parameter may have more (enumerated) values, instead of a boolean value. For example, the following HTTP GET request

http://… /simplefleet/rest/links/profile/athens/377/?xml=true

will return profile data for the link 377 of the Athens dataset in xml format. (We will see what profile data means in the following sections).

In the following section we will see a short description of each resource and resource group. Note that the Resources and DataModel of the TrafficAPI are thoroughly described in the online web documentation of the API, located currently at:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/docs/enunciate/

At the Implementation section, we will see an example of that documentation, when Enunciate framework use will be discussed.

3.2.1 NodesResourceGroup

This group of resources provides the functionality and data related to the nodes dataset of a city. Its Resources include the following URI resource paths:

- **/nodes/{city}**
- **/nodes/{city}/{id}**
- **/nodes/{city}/category/{category}**
- /nodes/{city}

This is a resource that returns to the user all the Nodes dataset of the city specified as a Path Parameter (in XML or JSON, based in the xml Query Parameter).

/nodes/{city}/{id}

This is a resource that returns a Node from the specified city path parameter, based on a unique link id path parameter. If a Node with the specified id parameter does not exist, an Error object with a message is returned.

/nodes/{city}/category/{category}

This is a resource that returns all Nodes from the specified city path parameterthat belong to the specified category . If a category with the specified parameter does not exist, an Error object with a message is returned.

3.2.2 LinksResourceGroup

This group of resources provides data and functionality for the static links dataset of the three cities. No temporal or traffic data are included in the returned Link objects. It contains the following URI resource paths/mount points:

- /links/{city}
- /links/{city}/bbox
- /links/{city}/bbox/latlon
- /links/{city}/bounds
- /links/{city}/boundsLatLon
- /links/{city}/category/{category}
- /links/{city}/category/{category}/length
- /links/{city}/category/{category}/number
- /links/{city}/{id}
- /links/{city}/length
- /links/{city}/lengthBidirectional

- /links/{city}/lengthUnidirectional
- /links/{city}/name/{name}
- /links/{city}/number
- /links/{city}/numberBidirectional
- /links/{city}/numberUnidirectional
- /links/{city}/radius
- /links/{city}/radiusLatLon

/links/{city}

This is a resource that returns to the user all the Links dataset of the city specified as a Path Parameter (in XML or JSON, based in the xml Query Parameter).

/links/{city}/bbox

This is a resource that returns to the user the BoundingBox of the city's links dataset in the dataset's native coordinate reference system.

/links/{city}/bbox/latlon

This is the same as above, but with the difference that with this resource/mount point the BoundinBox returned is in WGS84 geographic(lat-lon) coordinates.

/links/{city}/bounds

This is a resource that returns to the user all the Links dataset of the city specified as a Path Parameter that are contained inside a bounding box. The bounding box bottom left coordinates, width and height are passed using Query Parameters: x, y, width, and height. Note that the bounding box coordinates and dimensions should be in the native coordinate reference system and units of each city's links dataset. An example follows:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/links/athens/bounds?x=485995.0&y=4230975.0&width=485&height=423

This will return all links for Athens that intersect the specified bounding box.

/links/{city}/boundsLatLon

Same as previously, but in this case the bounding box is specified in WGS84 geographic coordinates, and its dimensions are in degrees.

/links/{city}/category/{category}

This resource returns all links of the specified city's dataset that belong to the category query parameter.

/links/{city}/category/{category}/length

This resource returns the length of all links of the specified city's dataset that belong to the category query parameter.

/links/{city}/category/{category}/number

This resource returns the number of all links of the specified city's dataset that belong to the category query parameter.

/links/{city}/{id}

Returns the link of the specified city's dataset with the specified id query parameter.

/links/{city}/length

This is a resource that returns to the user a double containing the length of the links of the specified city's link dataset.

/links/{city}/lengthBidirectional

This is a resource that returns to the user a double containing the length of the bidirectional links of the specified city's link dataset.

/links/{city}/lengthUnidirectional

This is a resource that returns to the user a double containing the length of the unidirectional links of the specified city's link dataset.

/links/{city}/{name}

Returns all links of the specified city's dataset whose respective road name matches fully or partially the name query parameter given.

/links/{city}/number

This is a resource that returns to the user an integer containing the number of the links of the specified city's link dataset.

/links/{city}/numberBidirectional

This is a resource that returns to the user an integer containing the number of the bidirectional links of the specified city's link dataset.

/links/{city}/numberUnidirectional

This is a resource that returns to the user an integer containing the number of the unidirectional links of the specified city's link dataset.

/links/{city}/radius

This is a resource that returns to the user all the Links that intersect a circle whose center and radius is given by query parameters. For example this HTTP GET request:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/links/athens/radius?x=485995.0&y=4230975.0&radius=485

will return all links intersecting the circle with the given center and radius. Again, all coordinates and radius are specified usingthe dataset's native CRS and units.

/links/{city}/radiusLatLon

Same as before, but in this case all things run on WGS84 geographic coordinates and units.

3.2.3 LinksTemporalResourceGroup

This group contains resources that provide data related to the traffic situation of a city's road network. Actually, two subsets can be defined:

- Resources that provide "live" traffic link data
- Resources that provide "profile" traffic link data

As we saw in the Data model section, the link data form provided is common. The only difference is that resources for "live" data return link data that were updated during the last 15 minutes, whereas "profile" data return available traffic data for all links, even if these traffic data are older than 15 minutes (this of course means that "live" data is always a subset of the "profile" data.

In both live and profile data, resources are the same, distinguished only by the links/live/ or links/profile/ mount point subset. For example, mount points for live data resources are the following (one can replace "links/live/" with "links/profile/" to obtain the profile data mount points):

- /links/live/{city}
- /links/live/{city}/bbox
- /links/live/{city}/bounds
- /links/live/{city}/boundsLatLon
- /links/live/{city}/length
- /links/live/{city}/lengthBidirectional
- /links/live/{city}/lengthUnidirectional
- /links/live/{city}/number

- /links/live/{city}/numberBidirectional
- /links/live/{city}/numberUnidirectional
- /links/live/{city}/overSpeed
- /links/live/{city}/overSpeedLimit
- /links/live/{city}/overSpeedRatio
- /links/live/{city}/radius
- /links/live/{city}/radiusLatLon
- /links/live/{city}/underSpeed
- /links/live/{city}/underSpeedLimit
- /links/live/{city}/underSpeedRatio
- /links/live/{city}/{id}
- /links/live/{city}/bbox/latlon
- /links/live/{city}/category/{category}
- /links/live/{city}/congestion/heavy
- /links/live/{city}/congestion/light
- /links/live/{city}/congestion/mild
- /links/live/{city}/name/{name}
- /links/live/{city}/category/{category}/length
- /links/live/{city}/category/{category}/number

One can easily notice that many of the resources for the live/profile traffic data follow more or less the same mount points that were described in the static data resources. The functionality is identical, so there is no need to re-describe it here. In those cases, (and where applicable), instead of Link objects, LinkTemporal objects are returned, containing the additional information of traffic data.

The added resource mount points for the LinksTemporalResourceGroup are:

/links/live/{city}/overSpeed

Returns the specified city's LinkTemporal objects that represent links whose current live (last 15 minutes) speed is above the specified speed threshold. For profiled data latest measured speed is used.

/links/live/{city}/underSpeed

Returns the specified city's LinkTemporal objects that represent links whose current live (last 15 minutes) speed is below the specified speed threshold. For profiled data latest measured speed is used.

/links/live/{city}/overSpeedLimit

Returns the specified city's LinkTemporal objects that represent links whose current live (last 15 minutes) speed limit is above the specified speed threshold. For profiled data latest measured speed limit is used.

/links/live/{city}/underSpeedLimit

Same as before, but links under the speed limit are returned.

/links/live/{city}/overSpeedRatio

Returns the specified city's LinkTemporal objects that represent links whose current live (last 15 minutes) speed ratio is above the specified speed ratio threshold. For profiled data latest measured speed ratio is used. Again, it is noted that the speed ratio is the fraction of the link's live (or profiled) speed to the link's speed limit.

/links/live/{city}/underSpeedRatio

Same as before, links under the speed ratio are returned.

/links/live/{city}/congestionHigh

Returns the specified city's LinkTemporal objects that represent links whose current live (last 15 minutes) speed ratio is below 25%. Essentially this is a shortcut resource to allow the user to get straight away the links which are congested. These are the "red" road links on the TrafficStore visualizations.

/links/live/{city}/congestionMild

Same as before, but speed ratio limits are from 25% to 50%. These are the "yellow" road links on the TrafficStore visualizations.

/links/live/{city}/congestionLow

Same as before, but speed ratio limits are above 50%. These are the "green" road links on the TrafficStore visualizations.

These are the resources the TrafficAPI currently offers. Of course, it doesn't end here: At the time this report was written, more resources, related to fleet related data and functionality were added. Of course, the online documentation will always be up to date, reflecting the latest status of the data and functionality offered by the TrafficAPI.

4 Implementation

In the previous sections the API design decisions, data model and functionality were discussed. In this section, the details of the implementation of the TrafficAPI will be analysed.

At the very beginning, a major decision about the development environment of the whole API and functionality was taken. We decided to move ahead with Java-related technologies, since Talent has a more than 150 man-years experience with Java and Java-related web and desktop applications. However, this wasn't the only pro Java argument, since Java technologies offer

well established paradigms on development of such web related solutions and APIs and solutions and support on the subject is more than plenty.

Since the TrafficAPI is offered through a REST web service form, it was necessary to use a Web Application Server. We selected the latest version (7.0.37) of Apache Tomcat, which is a renowned web application server for Java EE applications.

Apache Tomcat (or simply Tomcat, formerly also Jakarta Tomcat) is an open source web server and servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java Servlet and the JavaServer Pages (JSP) specifications from Sun Microsystems, and provides a "pure Java" HTTP web server environment for Java code to run.

Apache Tomcat includes tools for configuration and management, but can also be configured by editing XML configuration files. Tomcat 4.x was initially released with Catalina (servlet container), Coyote (an HTTP connector) and Jasper (a JSP engine, which is not of our interest, since no JSP technology is used).

Catalina is Tomcat's servlet container. Catalina implements Sun Microsystems' specifications for servlet and JavaServer Pages (JSP). In Tomcat, a Realm element represents a "database" of usernames, passwords, and roles (similar to Unix groups) assigned to those users. Different implementations of Realm allow Catalina to be integrated into environments where such authentication information is already being created and maintained, and then use that information to implement Container Managed Security.

Coyote is Tomcat's HTTP Connector component that supports the HTTP 1.1 protocol for the web server or application container. Coyote listens for incoming connections on a specific TCP port on the server and forwards the request to the Tomcat Engine to process the request and send back a response to the requesting client.

However, some new features were added with the release of Tomcat 7:

A Cluster component has been added to manage large applications. It is used for Load balancing that can be achieved through many techniques. Clustering support currently requires the JDK version 1.5 or later.

A high-availability feature has been added to facilitate the scheduling of system upgrades (e.g. new releases, change requests) without affecting the live environment. This is done by dispatching live traffic requests to a temporary server on a different port while the main server is upgraded on the main port. It is very useful in handling user requests on high-traffic web applications

Tomcat 7 has also added user as well as system based web applications enhancement to add support for deployment across the variety of environments. It also tries to manage session as well as applications across the network.

Tomcat 7.x implements the Servlet 3.0 and JSP 2.2 specifications. It requires Java version 1.6, although previous versions have run on Java 1.1 through 1.5. Versions 5 through 6 saw improvements in garbage collection, JSP parsing, performance and scalability. Native

wrappers, known as "Tomcat Native", are available for Microsoft Windows and Unix for platform integration.

Using Tomcat 7, a base Servlet for hosting the API service mechanisms was setup. The next step was implementing the foundations of the REST web service. For this, a Java API was used, known as JAX-RS.

4.1 JAX-RS

JAX-RS Java API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.[1] JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.

From version 1.1 on, JAX-RS is an official part of Java EE 6. A notable feature of being an official part of Java EE is that no configuration is necessary to start using JAX-RS. For non-Java EE 6 environments a (small) entry in the web.xml deployment descriptor is required.

JAX-RS provides some annotations to aid in mapping a resource class (a POJO) as a web resource. The annotations include:

@Path specifies the relative path for a resource class or method.

@GET, @PUT, @POST, @DELETE and @HEAD specify the HTTP request type of a resource.

@Produces specifies the response Internet media types (used for content negotiation).

@Consumes specifies the accepted request Internet media types.

In addition, it provides further annotations to method parameters to pull information out of the request. All the @*Param annotations take a key of some form which is used to look up the value required.

@PathParam binds the method parameter to a path segment.

@QueryParam binds the method parameter to the value of an HTTP query parameter.

@MatrixParam binds the method parameter to the value of an HTTP matrix parameter.

@HeaderParam binds the method parameter to an HTTP header value.

@CookieParam binds the method parameter to a cookie value.

@FormParam binds the method parameter to a form value.

@DefaultValue specifies a default value for the above bindings when the key is not found.

@Context returns the entire context of the object.(for example @Context HttpServletRequest request)

JAX-RS has many implementations. We chose Jersey. Jersey is the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services. But, it is also more than the Reference Implementation. Jersey provides an API so that developers may extend Jersey to suit their needs.

```
@Path("/links")
public class LinksResource {

    @Context
    UriInfo uriInfo;
    @Context
    Request request;
    @Context
    HttpHeaders headers;

    /**
     * Returns a list of links that intersect the given bounds. If no links
     * intersect those bounds,
     * an empty list will be returned. The bounding box should be using the
     * dataset's native coordinate reference system for its definition.
     *
     * @param city The city name literal, as predefined by dataset setup.
     * @param xml  Set this to true if the responce should be in xml form.
default = false (=call will return json formed results)
     * @param x     The bounding box lower left x coordinate, in the dataset's
native coordinate reference system
     * @param y     The bounding box lower left y coordinate, in the dataset's
native coordinate reference system
     * @param width The bounding box width, in the dataset's native coordinate
reference system units
     * @param height    The bounding box height, in the dataset's native
coordinate reference system units
     * @return  A list of {@link Link} objects, or an
     * {@link gr.talent.simplefleet.entities.helpers.Error} object
     * if something failed.
     * @see {@link Link},  {@link BoundingBox}
     */

    @Path("{city}/bounds")
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Object getLinksByBounds(
            @PathParam("city") String city,
            @DefaultValue("false") @QueryParam("xml") boolean xml,
            @QueryParam("x") double x,
            @QueryParam("y") double y,
            @QueryParam("width") double width,
            @QueryParam("height") double height) {
        List<Link> links = DataManager.getInstance().getLinksAtBounds(city, new
Rectangle2D.Double(x,y,width, height), true);
        if (!xml){
            return new
ResponseBuilderImpl().type(MediaType.APPLICATION_JSON).entity(links.toArray(new
Link[links.size()])).build();
        }else{
            return new
ResponseBuilderImpl().type(MediaType.APPLICATION_XML).entity(links.toArray(new
Link[links.size()])).build();
        }
    }
    ...
```

Figure 2: A code snipped showing a data model object and the JAX-RS annotation necessary for web services implementation

After initial implementation of the data model objects and resources that were extensively described in the previous section, the communication with the TrafficStore was in turn for implementation. TrafficStore provided access to its core PostgreSQL databases for each one of the three cities, in order for the TrafficAPI implementation to pull data directly from it. In our side, a Tomcat webapp can define a context where data sources for the webapp application can also be defined. So, three data sources were defined, each one for a city. Below one can see the DataSource connecting the TrafficAPI and the TrafficStore for Vienna:

```
<Resource type="javax.sql.DataSource"
        name="SimpleFleetVienna"
        factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
        driverClassName="org.postgresql.Driver"
        url="jdbc:postgresql://1.2.3.4/mmdb"
        username="user"
        password="pass"
  />
```

This is common practice in Tomcat web applications, and, more importantly, it easily allows adding of more datasources, if the API is used for more cities.

However, since the TrafficStore is the heart of the SimpleFleet project, and it is used essentially for anything that has to do with data storage, visualization, processing and functionality, it seemed not wise to use the TrafficStore databases as the direct source of data for serving the TrafficAPI. There are many reasons for this:

1. If the TrafficAPI (or any other SimpleFleet module using the TrafficStore) pulls data from the TrafficStore at a high rate or performs complex queries, things will slow down for everyone.
2. If the TrafficAPI services are hosted on a server different from the one of the TrafficStore, a delay on transferring data between the two modules will be present. There also is some delay on encryption/decryption of PostgreSQL returned results.

For these two reasons, the TrafficAPI should have some kind of caching mechanism which will allow speedy serving of web service HTTP GET requests without constant transactioning with the TrafficStore. The ideal solution for this would be to have a small TrafficStore inside the TrafficAPI webapp, and that's what was implemented.

The TrafficStore databases contain a lot of tables used for the various modules the TrafficStore serves. The TrafficAPI only needs a subset of these tables:

1. The static link and node dataset tables
2. The live and profiled data tables
3. The log table

The first category tables are the two tables that contain the nodes and links datasets, respectfully. No temporal or traffic data are stored there. One can argue that these tables don't actually change since a change on a city's road network is not an often thing. So, to start with, these two tables can be cached for all cities by the TrafficAPI. Of course, a mechanism

that updates these tables to reflect the TrafficStore status is implemented and can be activated at any time at the TrafficAPI server, but the hard fact is that the static data don't change and should be cached.

Moving on to the traffic related tables, which are substantially smaller than the static data tables, we know that the TrafficStore updates those tables at regular time intervals (which now are around 15 minutes, but this can be configured). The last time that the tables were updated is recorded at the log table; this is a utility table with a single record containing the last time the traffic/temporal tables were updated. So, the TrafficAPI server checks on the log table, and if there is a change since the last time it pulled traffic data from the TrafficStore, it re-requests the traffic related tables. The check on the log table also happens on regular, 5 minute intervals (also configurable).

We stressed the need of caching and we argued about the static data which more or less don't change (well, they're static). But how and where to cache? At first, simple Java data structures (like ArrayLists, Sets or HashMaps) seemed to be sufficient for storing Links or Nodes. But what if in the near future complex queries were to be used by a REST web service functionality? Of course, adding more and more HashMaps or making indexed data structures for each case is not efficient. We need a database. That's why, as it was mentioned before, the ideal solution for this would be to have a small TrafficStore inside the TrafficAPI webapp.

So a database should accompany the TrafficAPI server in order to service it with data from the source, the original TrafficStore, and working as a, first line of defence, data source. However we needed to avoid setting up another full fledged DBMS on the TrafficAPI server; we needed the TrafficAPI webapp to be as small and transferrable as possible. It would be ideal if a .war package containing the TrafficAPI server could be transferred to another JavaEE container, without deploying additional database or making new configurations and setups. That's why we needed the "private TrafficStore" to be inside the TrafficAPI webapp. Naturally, this requires an embedded database. We chose SQLite.

4.2 SQLite

SQLite is a relational database management system contained in a small (~350 KB) C programming library. In contrast to other database management systems, SQLite is not a separate process that is accessed from the client application, but an integral part of it.

SQLite is ACID-compliant and implements most of the SQL standard, using a dynamically and weakly typed SQL syntax that does not guarantee the domain integrity.

SQLite is a popular choice as embedded database for local/client storage in application software such as web browsers. It is arguably the most widely deployed database engine, as it is used today by several widespread browsers, operating systems, and embedded systems, among others SQLite has many bindings to programming languages. Also, the source code for SQLite is in the public domain

Unlike client–server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is

linked in and thus becomes an integral part of the application program. (In this, SQLite follows the precedent of Informix SE of c. 1984) The library can also be called dynamically. The application program uses SQLite's functionality through simple function calls, which reduce latency in database access: function calls within a single process are more efficient than inter-process communication. SQLite stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine. It implements this simple design by locking the entire database file during writing. SQLite read operations can be multitasked, though writes can only be performed sequentially.

Several computer processes or threads may access the same database concurrently. Several read accesses can be satisfied in parallel. A write access can only be satisfied if no other accesses are currently being serviced. Otherwise, the write access fails with an error code (or can automatically be retried until a configurable timeout expires). This concurrent access situation would change when dealing with temporary tables. This restriction is relaxed in version 3.7 when WAL is turned on enabling concurrent reads and writes.

A standalone program called sqlite3 is provided that can be used to create a database, define tables within it, insert and change rows, run queries and manage a SQLite database file. This program is a single executable file on the host machine. It also serves as an example for writing applications that use the SQLite library.

SQLite is a popular choice for local/client SQL storage within a web browser and within a rich internet application framework; most notably the leaders in this area (Google Gears, Adobe AIR, and Firefox) embed SQLite. SQLite full Unicode support is optional.

SQLite also has bindings for a large number of programming languages, including BASIC, C, C++, Clipper//Harbour, Common Lisp, C#, Curl, D, Delphi, Free Pascal, Haskell, Java, Livecode, Lua, newLisp, Objective-C (on OS X and iOS), OCaml, Perl, PHP, Pike, Python, REBOL, R, REALbasic, Ruby, Scheme, Smalltalk, Tcl, Visual Basic, and JavaScript. An ADO.NET adapter, initially developed by Robert Simpson, is maintained jointly with the SQLite developers since April 2010. An ODBC driver has been developed and is maintained separately by Christian Werner. Werner's ODBC driver is the recommend connection method for accessing SQLite from OpenOffice. There is also a COM (ActiveX) wrapper making SQLite accessible on Windows to scripted languages such as JScript and VBScript. This adds database capabilities to HTML Applications (HTA).

SQLite has automated regression testing prior to each release. Over 2 million tests are run as part of a release's verification. Starting with the August 10, 2009 release of SQLite 3.6.17, SQLite releases have 100% branch test coverage, one of the components of code coverage.

SQLite, as stated at the beginning, is a relational database management system contained in a small (~350 KB) C programming library. We run on Java. So a JDBC driver was needed to access it through java code. For this job, we chose a SQLite JDBC Driver implementation**,** previously known as Xerial.

4.3 SQLite JDBC Driver (Xerial)

SQLite JDBC, developed by Taro L. Saito, is a library for accessing and creating SQLite database files in Java. This SQLiteJDBC library requires no configuration since native libraries for major OSs, including Windows, Mac OS X, Linux etc., are assembled into a single JAR (Java Archive) file. The usage is quite simple; one downloads the sqlite-jdbc library, then appends the library (JAR file) to the class path.

The SQLite JDBC driver package (i.e., sqlite-jdbc-(VERSION).jar) contains three types of native SQLite libraries (sqlite-jdbc.dll, sqlite-jdbc.jnilib, sqlite-jdbc.so), each of them is compiled for Windows, Mac OS and Linux. An appropriate native library file is automatically extracted into the OS's temporary folder, when the org.sqlite.JDBC driver is loaded by the tomcat servlet.

4.4 SpatiaLite

So far, we have the JAX-RS/Jersey implementation of the TrafficAPI REST web services, and a small, embedded TrafficStore for pulling data directly from it. This allows us to perform any SQL query necessary and to provide the results to the related HTTP GET requests. But there is still one functionality missing: spatial query support. Spatial query support comes very handy when functionality like queries about objects intersecting/are contained in bounding boxes is required. We saw earlier that the TrafficAPI supports such functionality. Let's see how this was implemented.

SpatiaLite is a spatial extension to SQLite, providing vector geodatabase functionality. It is similar to PostGIS, Oracle Spatial, and SQL Server with spatial extensions, although SQLite/SpatiaLite aren't based on client-server architecture: they adopt a simpler personal architecture. i.e. the whole SQL engine is directly embedded within the application itself: a complete database simply is an ordinary file which can be freely copied (or even deleted) and transferred from one computer/OS to a different one without any special precaution.

SpatiaLite extends SQLite's existing spatial support to cover the OGC's SFS specification. It isn't necessary to use SpatiaLite to manage spatial data in SQLite, which has its own implementation of R-tree indexes and geometry types, but in order to do advanced spatial queries and support multiple map projections, SpatiaLite is needed.

So, we extended SQLite with SpatiaLite in order to be able to perform spatial queries to out TrafficStore, However, this wasn't the first choice. A small, memory efficient RTree cache was also used and evaluated.

4.5 STRtree

STRtree is a query-only R-tree created using the Sort-Tile-Recursive (STR) algorithm, for two-dimensional spatial data.

The STR packed R-tree is simple to implement and maximizes space utilization; that is, as many leaves as possible are filled to capacity. Overlap between nodes is far less than in a basic R-tree. However, once the tree has been built (explicitly or on the first call to #query), items may not be added or removed. This doesn't affect our case, since the static data r-tree build is

an one-off cost at the start on the TrafficAPI server, and the traffic data r-trees are built when traffic data are updated. In any case, building a STRtree is fast; for 10000 links this is less than 10 milliseconds. STRtrees are used only for link-related data, not nodes. Also, STRtree doesn't keep the actual link object, just a reference to it (the ID). That makes it very memory-efficient.

Some words on the creation of the TraficAPI server's "private" TrafficStore. This was done by "transforming" the original TrafficStore's tables to SQLite/SpatiaLite ones using the GDAL spatial data processing library's OGR Tools.

4.6 OGR Tools

The OGR toolkit is a subkit of the FW Tools Toolkit. The FW Tools Toolkit is a toolkit available in both Linux and Windows executable form as well as source code form. It has several command line tools, like OgrInfo, Ogr2Ogr, etc. We used Ogr2Ogr.

This is a command line tool that converts one Ogr defined data source to another Ogr data source. Ogr supports multiple data formats: ESRI Shapefile, MapInfo Tab file, TIGER, s57, DGN, CSV, DBF, GML, KML, Interlis, SQLite, ODBC, ESRI GeoDatabase (MDB format), PostGIS/PostgreSQL, MySQL . In our case, PostgreSQL tables were converted to SQLite ones.

4.7 Putting it all together

At the beginning, a skeleton JavaEE servlet web application was created using Eclipse application development IDE. This servlet application contained the necessary code for loading native libraries about JDBC operations and SQLite/SpatiaLite functionality.

The next step was to write the code related to the Data Model objects. Following that, Jersey was used to create Resources and mount points using those data model objects. Since mount points and data model objects skeletons were ready, we started writing the actual code that accessed data sources to return data to HTTP GET calls. For that, creating the "local TrafficStore" was necessary.

To achieve that, we used OGR Tools to generate our own SQLite files for each one of the three cities. The three SQLite files were placed under the WEB-INF directory of the web application. Next, a context file for the web application was created; this file contained the definition of the three datasources, one for each city.

The heart of the TrafficAPI functionality is an Java class called CityRepository. For each city, a different CityRepository was created automatically just by reading the datasources defined in the web application context. This mechanism enables us to add more CityResource objects easily in the future in case more cities will be supported by the SimpleFleet services. These CityRepository objects also contain information about each city's dataset native coordinate reference system, to enable spatial conversions, and other metadata related to database table names prefixes and suffixes, configuration times to poll the TrafficStore and update traffic data, etc. Then, resource implementation code is completed so as to use CityRepository objects to pull data and support functionality specified by the TrafficAPI.

Finally, each CityRepository object opens a connection to the respective local SQLite file, and everything is in place for responding to all HTTP GET requests defined by the API specs. Note that the mechanism is common for all cities; we don't use a separate server for each city. Instead, from the respective Path Parameter of each HTTP GET request, the server "understands" the city for which this request is destined to, and uses the respective CityRepository object to respond. Schematically, the server architecture supporting the TrafficAPI functionality can be seen in the following image:

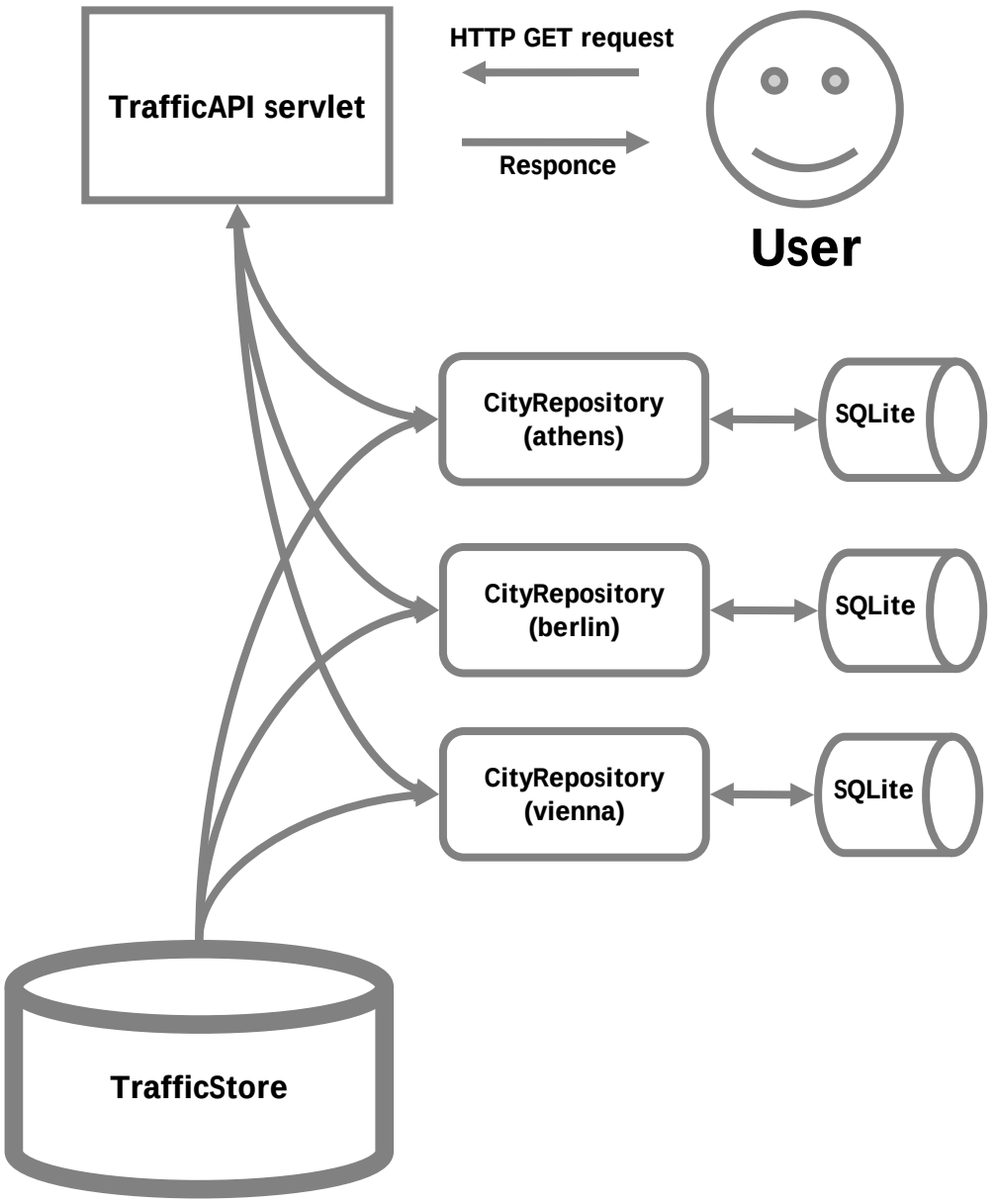

Figure 3: TrafficAPI implementation architecture

To automatically produce documentation about the TrafficAPI functionality, resource mount points and data model objects, we used the Enunciate REST web services framework.

4.8 Enunciate

Enunciate is an engine for dramatically enhancing a Java Web service API.

It's logic is quite simple. As soon as a REST Web service API is developed and implemented Enunciate is attached to the API's buid process. By that the Web service API is boasting some pretty impressive features:

- Full HTML documentation of REST services, scraped from the code JavaDocs.
- Client-side libraries (e.g. Java, .NET, iPhone, Ruby, Flex, AJAX, GWT, etc.) for developers who want to interface with your API.
- Interface Definition Documents (e.g. WSDL, WADL, XML-Schema, etc.)

Notice that the actual runtime of the web application supporting the REST web services implemented isn't affected. Enunciate is primarily a build-time tool; it's intent is to "get out of the way" so Jersey or Metro or CXF or JBoss-WS can do their job processing Web service requests.

Some of these successful applications of Enunciate are:

- Development of a Web service API, where documentation is updated (including fully-up-to-date client libraries) as part of the build process.
- iPhone/iPad application development where the Web service API is invoked via Enunciate-generated Objective-C libraries.
- AJAX application development via GWT where the Web service API is invoked via Enunciate-generated GWT libraries which include GWT-RPC invocations or even Enunciate-generated JSON overlays via REST/JSON invocation.
- Flex application development where the Web service API is invoked via Enunciate-generated ActionScript libraries.

We used Enunciate just for automatically producing documentation for the REST web services of the TrafficAPI, but it is clear from the above that it can be used for many other things, including generation of client libraries for the TrafficAPI web services. This hasn't escaped our attention, should there be a need for that functionality in the future of the project.

An example of the documentation produced by Enunciate for the TrafficAPI web services can be seen below. The entire Enunciate documentation for the TrafficAPI is located at the following link:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/docs/enunciate/

/links/{city}/bounds

Mount Point: /rest/links/{city}/bounds

GET

Returns a list of links that intersect the given bounds. If no links intersect those bounds, an empty list will be returned. The bounding box should be using the dataset's native coordinate reference system for its definition.

Parameters

name	description	type	default
city	The city name literal, as predefined by dataset setup.	path	
xml	Set this to true if the responce should be in xml form. default = false (=call will return json formed results)	query	false
x	The bounding box lower left x coordinate, in the dataset's native coordinate reference system	query	
y	The bounding box lower left y coordinate, in the dataset's native coordinate reference system	query	
width	The bounding box width, in the dataset's native coordinate reference system units	query	
height	The bounding box height, in the dataset's native coordinate reference system units	query	

Response Body

element:	(custom)

A list of Link objects, or an gr.talent.simplefleet.entities.helpers.Error object if something failed.

Figure 4: An example of a Resource mount point documentation generated by Enunciate

4.9 Servers and testbeds

The TrafficAPI can currently be accessed under the following URL:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/rest

One needs to just add the respective resource mount point path to this URL (with the proper parameters) to access all data and functionality described in the TrafficAPI section. As mentioned at the Introduction section, the TrafficAPI is currently hosted at a Ubuntu server VM, running at the Okeanos server, (the same server where the TrafficStore and the related databases are located). This setup favours fast communication between the TrafficStore and the TrafficAPI server.

However, the TrafficAPI server has been tested to other servers, including a Scientific Linux platform, a Fedora Platform and even a Windows located Tomcat. In all cases it performed well, and most importantly, no needs in code or webapp setup were needed. This proves that the TrafficAPI webapp with its respective support files can be easily transported to any Tomcat server as a war file, without the need of extra configuration.

4.10 Where is the TrafficSDK?

In the SimpleFleet project documentation and roadmap, along with the TrafficAPI, development of a Traffic SDK is mentioned. What is a Traffic SDK? It was defined as a set of libraries necessary for clients in order to take advantage of the TrafficAPI functionality for different clients.

At the API design decisions section, we argued about the choice of the TrafficAPI to be a REST web services based API. With this decision, all clients, no matter what OS or application/programming framework they use for developing SimpleFleet client applications, can access the same API without the need of a client-specific library, since HTTP GET request-response mechanism is common functionality across all platforms, and JSON/XML formatted data can be handled easily. This makes the creation of client libraries, more or less, unnecessary.

Note that the Deliverable 4.2 is about an application framework built to access SimpleFleet functionality, i.e., the TrafficStore and related functionality (as described in Work Packages 2 and 3). This deliverable differs from D4.1 in that it should provide incomplete pieces of software which, with minimal effort, can be extended to full programs.

These "incomplete pieces of software" can be seen as a kind of platform-specific SDKs, especially when provided in the form of some library or collection of libraries. Using that point of view, the consortium has agreed that it makes more sense to provide an SDK in the form of a loose collection of extendable software templates/sample programs, plus some libraries containing basic functions, all for the purpose of facilitating the development of mobile apps. For example, some "wrapper" functions and code could be provided in order to improve handling of the specific HTTP GET requests and responses the TrafficAPI uses, or to provide a more elaborate error handling, or to map Data Model objects in JSON or XML to plain objects in each development environment (POJOs for Java, etc).

This means that the description of a "TrafficSDK" (in terms of a collection of libraries used for developing fleet/traffic related mobile apps) will be provided in Deliverable 4.2. Finally, all the TrafficAPI functionality together with the libraries of the TrafficSDK will be used in the Task 5.3. Its goal is the development of an "iFleet app demonstrator" or, as stated in the DoW, the "creation of a simple fleet management app that uses the iPhone, both, as a sensor platform and as a user interface". A mobile demonstrator will be implemented for the iPhone using the TrafficApps framework. The purpose of this demonstrator will be to showcase the SimpleFleet framework as well as to evaluate the system, i.e., the use of smartphones as interfaces and positioning sensor. Testing will be carried out with existing fleet management customers to provide feedback on relation to the quality of existing solutions.

We argued that the functionality provided by the TrafficAPI is based on the HTTP request-response model, and that in most cases it covers most needs by various platforms. However, some basic libraries, incomplete pieces of example code or wrapper functions and data model objects will also be provided where needed in the future. Also, this may lead to providing extended functionality, currently not covered by the TrafficAPI REST web services. An example; the SimpleFleet project may, in the future include functionality such as notification

of clients about TMC messages. Although this may be also covered by some kind of HTTP "polling", development of client libraries for the same purpose may be also necessary. Such added functionality is something that the consortium will probably decide if it is necessary for the project goals.

5 Conclusion and future work

In this deliverable an attempt was made to describe the TrafficAPI design and implementation procedure so far. Design decisions were analyzed and implementation technologies were described in good detail. Of course, the TrafficAPI has just started to be used by the project partners, and it is not complete; after studying the way it is used, more things may be added to it, and more functionality will be supported in the future.

The main point that one needs to focus to about the TrafficAPI is that it is build to the logic that "one size fits all". Using the REST web services architecture for providing functionality through common HTTP GET requests, coupled with widely used data formats such as XML or JSON, the target that it can be used by any OS, platform, application development framework, or programming language environment is achieved. In the months which follow, several new services are to be implemented and concluded on top of the TrafficStore, affecting the TrafficAPI and pushing us to provide the related functionality and improving the existing one. Still, the fact that the architecture chosen is solid and "universal" will surely enable us to come easily through all necessary changes and additions made to the TrafficStore, or to the requested functionality by the SimpleFleet services users.

6 References

[Enunciate] A REST framework implementation. Online at: http://enunciate.codehaus.org/

[JAX-RS] Online at: http://jax-rs-spec.java.net/

[Jersey] Java JAX-RS reference implementation. Online at: http://jersey.java.net/

[JTS] A Java Geometry library. http://www.vividsolutions.com/jts/JTSHome.htm

[ogr2ogr] Geometry utility tools. Online at: http://www.gdal.org/ogr2ogr.html

[okeanos] ~okeanos. Online at: https://okeanos.grnet.gr/home/

[REST] A web service architecture. Online at: https://en.wikipedia.org/wiki/Representational_state_transfer#RESTful_web_APIs

[SpatiaLite] Spatial extension for SQlite. Online at: http://www.gaia-gis.it/gaia-sins//

[SQLite] An embedded database. Online at: http://www.sqlite.org/

[SQLite-jdbc] A JDBC Driver for SQLite. Online at: https://bitbucket.org/xerial/sqlite-jdbc

[STRTree] An R-Tree implementation. Online at: http://www.vividsolutions.com/jts/javadoc/com/vividsolutions/jts/index/strtree/STRtree.html

[Tomcat] Servlet container. Online at: http://tomcat.apache.org/