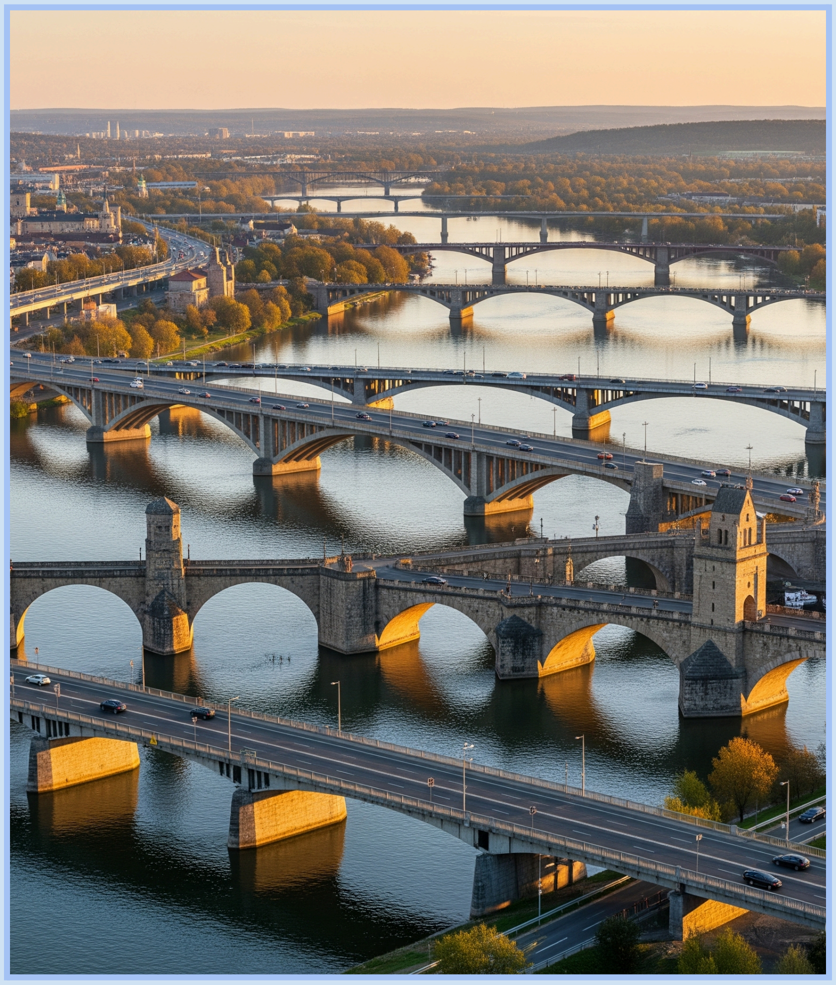
METASPEX

Metaspex Technical Insights

IV - Web Services



Introduction

Metaspex helps build batch programs that use ontologies and perform operations on database documents; calculate statistics; or perform mass updates or exports. All the benefits presented up to now are available to batch programs.

Metaspex can also produce the operational part of business applications. This is sometimes called “real time” backends. Metaspex backends can listen to HTTP-based web services, take JSON queries, and make JSON replies.

These backends are highly scalable, vertically and horizontally. More cores in the same machine, or more machines, will absorb more traffic. As a matter of fact, the binaries produced by Metaspex are record-small (300 web services can typically fit in a binary executable of a few tens of megabytes).

Metaspex accommodates monoliths (single binary) or the microservices architecture, at any granularity. This is your choice–Metaspex does not make it for you.

Additionally, in Metaspex, web services can enforce state of the art security, but this is the subject of a separate document dedicated to security matters.

Web services can be stateful with the simple use of a session token, but covering this is beyond the scope of this technical insight.

Metaspex also supports specific paginated services which “browse” through an index. They’ll be covered in a separate document.

This document concentrates purely on the definition of web services.

It’s not RESTful

The REST standard promotes a backend architecture and a naming convention whereby “everything is a resource.” Given the efforts we make to allow sophisticated ontology description as a better data modeling technique, this is not the focus of Metaspex. Metaspex concentrates on domain-oriented deep data modeling, not on listing resources and leaving everything else to code.

Metaspex HTTP web services can take virtually any user-defined JSON query message and make any user-defined JSON reply message. As a matter of fact, these are both defined as part of the ontology, as “payloads.” In terms of conciseness, the decision to not support RESTful does not sacrifice conciseness or performance.

Nothing prevents you from adopting naming conventions that are very close to RESTful ones when they make sense, as you’ll see.

Payloads

Web services payloads are defined using the exact same tools as the ones used to define ontologies. They are strongly typed and contain **own** relationships and **slot**s. Metaspex automatic serialization and deserialization into or from JSON are useful in this context too. No direct handling of JSON is necessary.

Your **own** relationships become JSON sub-objects, while **slot**s become JSON attributes. And **own_list**s**, own_vector**s, and **slot_vector**s become JSON arrays.

A possibly unexpected benefit of Metaspex’s ontologies is the support of inheritance. Metaspex applications typically contain hierarchies of payloads for a variety of web services, to use them at any level of inheritance hierarchy. Updating a base payload type will automatically update all the web services using that type and the derived ones. This is key for keeping consistency between web services, and for their speedy evolution.

The support of inheritance even includes polymorphism. Metaspex supports polymorphic JSON queries and replies.

The distinction between the persistent part of an ontology and the payload part is a matter of convention. Any type defined in an ontology can be used as a web service payload, including an entire document.

Here is an example of a user-defined address payload, which should look familiar:

```
#include <string>

#include "hx2a/element.hpp"
#include "hx2a/own.hpp"
#include "hx2a/root.hpp"
#include "hx2a/slot.hpp"

using namespace std;
using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };
}
```

A web service can receive an address as a JSON query; it can also return an address as a JSON reply. The JSON serialization will look like this:

```
{"num": 42, "strt": "Metaspex Rd", "cty": "Ontology City", "st": "CA", "zip":
"55555"}
```

When an **own** relationship is used, a sub-JSON object is used with the **tag** associated with the **own** relationship.

```
    class location: public element<>
    {
      HX2A_ELEMENT(location, "loc", element, (_name, _address));
      slot<string, "name"> _name;
      own<address, "addr"> _address;
    };
```

This will serialize in JSON, as expected, as something like:

```
{"name": "Headquarters", "addr": {"num": 42, "strt": "Metaspex Rd", "cty":
"Ontology City", "st": "CA", "zip": "55555"}}
```

In case the **own** relationship's value is null, the address will simply be omitted.

We'll see that query payloads come pre-parsed in web services, ensuring complete type safety, while replies are built using the familiar **make** function (see the previous technical insight). Services only handle C++ strongly-typed ontology objects.

Payload definitions through Metaspex ontologies are a good example of "requirement is code is technical documentation." Payload definitions can be read, for instance by a JavaScript developer building a graphical user interface and implementing calls via web services to a Metaspex backend. They can clearly see which queries and replies are accepted by the web services they use, and they can rely on the fact that the backend supports them exactly.

Payload re-use, and ready-made ones

The properties of ontologies, in terms of re-use of types, apply as well to web service payloads. Once an ontology type has been defined, it can be used as a base type or as the target of an **own** relationship in a payload.

As a result, Metaspex comes with a wide portfolio of ready-made payloads that you can use, inherit from, or point at with **own** relationships. You need to make a reply using a document identifier? You need to embed in a reply an HTML5-compatible geolocation containing latitude, longitude, altitude, speed, etc.? Metaspex has these in its library.

Web service declarations

Web services are singletons declared as a global variable using the **service** template offered by Metaspex. Here is the simplest example of a web service:

```
#include "hx2a/service.hpp"

using namespace hx2a;

auto _simple_service = service<"simple">
   ([](){
   });
```

The variable _simple_service will not be used anywhere else. It is here only to implement the singleton.

The constructor of a service takes a lambda expression containing the code to execute when the service is invoked by a client.

Once this code is compiled, and installed (see below), the web server will accept HTTP web service calls to the “simple” service. The URI depends on the web server configuration, but it will look like this:

`http://domain_name/simple`

The name we chose as an argument to the **service** template, “simple”, is the last bit of the URI. You can standardize these across your application, for instance by prefixing a verb with the name of the corresponding ontology type, if it makes sense (e.g., survey_create, survey_get, survey_update, survey_delete). Just make sure that the name of the service is unique across the entire application. The application won’t start if there are homonymous services.

The domain name itself, and the port number (here the default, 80) are both under your control through familiar web server configuration. The same goes, for instance, in relation to whether or not TLS is used for encryption. The binary application compiled from the Metaspex source is independent from all that.

In this example, the service does not expect a query payload. The empty JSON object ({}) will still need to be sent by the client invoking the service.

As the lambda function returns nothing, it will systematically respond with the empty JSON object.

This service is not very useful, but it’s a start towards understanding Metaspex web services.

Now if we want to modify the same service to take an address, as we defined it, and echo it back, we would specify it this way:

```
#include "hx2a/service.hpp"

#include "address.hpp"

using namespace hx2a;

auto _simple_service = service<"simple">
    ([](const rfr<address>& query){
      return query;
    });
```

When sending an address, the HTTP headers should look like:

```
POST /simple HTTP/1.0

Accept: application/json

Content-Type: application/json

Content-Length: 49
```

```
{"num": 42, "strt": "Metaspex Rd", "cty": "Ontology City", "st": "CA", "zip": "55555"}
```

With a few unimportant differences, this can be issued by a familiar curl command:

```
curl http://domain_name/simple -d '{"num": 42, "strt": "Metaspex Rd", "cty": "Ontology City", "st": "CA", "zip": "55555"}'
```

Note that in the code of the service, the address payload comes already parsed. In the case where the JSON payload is incorrect, the client will receive an exception. In the case where the values in the JSON payload do not correspond to the **slot** or **own** relationships, an error will be emitted too. This goes beyond standard JSON parsing. It causes a payload to be validated before the code of the service is executed.

The code of the service uses the query payload as a C++ object. It can do whatever it wants with it.

The reply can be assembled using the **make** function or by returning any object which is an instance of an ontology.

Consult the vast quantity of ready-made queries and replies to save you some effort, in case they fit your needs.

In general, the declaration of a service looks like:

```
#include "hx2a/service.hpp"

#include "payloads.hpp"

using namespace hx2a;

auto _my_service = service<"my_service">
    ([](const rfr<some_query_payload>& query){
      // Some code, involving databases.
      return make<some_query_reply>(/* some arguments */);
    });
```

But you are completely free to use whatever C++ code you want.

Carrying document identifiers is very common. Their type is doc_id, and they can be present as **slot** types in query or reply payloads.

Assuming that the query payload contains a **slot** of doc_id type to retrieve your ontology type T, retrieving the corresponding document from the database is done easily using something like this:

```
db::connector c("my_db");
ptr<T> t = T::get(pld->_id);
```

If the document is found, the **ptr** is non-null. No database API or query language is needed.

In case a service needs to change something in a document, assigning a new value to a **slot** in an object participating in a document, or assigning a new value to any relationship it bears, will each cause the corresponding document to be automatically identified as updated by Metaspex and automatically saved in the database where it came from.

The commit in the database is implicit at the end of the web service. No explicit commit is necessary. Every new document will be saved, and every modified document will be updated in the database it came from.

Web server modules

We said that Metaspex source code is compiled with a regular C++ compiler, such as g++ or Clang, without any code generation or preprocessing other than the C++ preprocessor.

Metaspex comes with ready-made makefiles which assign specific compilation options to compile backends. The output of the C++ compiler is called a “module” in web server parlance. It’s a binary shared object, which must be installed in a specific location of the web server you chose. (Nginx and Apache are the two supported by Metaspex.)

The choice of the target web server is made at compile time. Modules for Nginx and Apache are not identical.

Modules are “shared objects” in the sense that the binary will be loaded only once in the RAM of the operating system, like a shared library.

A Metaspex module will contain all the services you specified in the source code you compiled. You can partition your services in different sources and compile them into different binaries if you wish. As a result, depending on your architectural preferences, you can have a monolithic binary containing all the services, or multiple binaries with a partition of your choice. It is beyond the scope of this document, but Metaspex allows backends to perform web service calls to Metaspex or non-Metaspex microservices.

Security

Metaspex exceeds industry standards in terms of security. It is beyond the scope of this document to cover these important aspects. Please consult the technical insight covering security. It explains how **user**s, **affiliation**s and **role**s (among others) are managed in a very simple but very secure manner.

Conclusion

Metaspex supports HTTP web Services allowing JSON queries and JSON replies. They “sit” on top of the ontology you define, supporting an ontology-centric mode of development and ensuring type safety.

The high-level functions generated by Metaspex on each type of the ontology are used in services. They allow service definitions to concentrate on what a service should do, not how it is implemented at a low technical level, as this is entirely automated by Metaspex.

Services are hosted as modules in your preferred web server, with vertical scalability characteristics (number of processes or threads allowed) and network configuration being left to familiar web server configuration. Metaspex modules are thread-safe without any garbage collection, so vertical scalability is perfect. Horizontal scalability is achieved through regular load-balancing techniques and the excellent horizontal scalability features offered by document databases.

Services are devoid of database APIs, JSON serialization, query languages, network-related code and memory management. All these are automated.