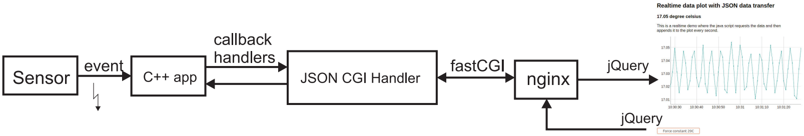
# Realtime browser JS ⇔ C++ communication

Bernd Porr

# Web server/client communication: Intro



**Realtime data plot with JSON data transfer**

**17.05 degree celsius**

This is a realtime demo where the java script requests the data and then appends it to the plot every second.

Focus on pure server (C++) ⇔ client (JS) web communication using `fastCGI` and `nodejs`.

For standard generation of dynamic content please use PHP, nodejs or other well documented frameworks.

# Web server/client communication: REST

Uniform interface: Any device connecting to the URL should get the same reply. No matter if a web page or mobile phone requests the temperature of a sensor the returned format must always be the same.

Client-server decoupling: The only information the client needs to know is the URL of the server to request data or send data.

Statelessness: Each request needs to include all the information necessary and must not depend on previous requests. For example a request to a buffer must not alter the buffer but just read from it so that another user reading the buffer shortly after receives the same data.

See `https://www.ibm.com/cloud/learn/rest-apis`

# Data format: JSON

```
{
    temperature: [20, 21, 20, 19, 17],
    steps: 100,
    comment: "all good!"
}
```

# Web servers

- NGINX: Easy to configure but very flexible web server. Pronounced “Engine-X”.
- Apache: Hard to configure but safe option
- lighttpd: Smaller web server with a small memory footprint. Pronounced “lighty”.

# Fast CGI

A fast CGI program is a UNIX commandline program which communicates with the web server (nginx, Apache, ...) via a UNIX socket which in turn is a pseudo file located in a temporary directory for example /tmp/sensorsocket.
The web server then maps certain http/https requests to this socket. An example configuration for nginx looks like this:

```
location /sensor/ {
   include        fastcgi_params;
   fastcgi_pass   unix:/tmp/sensorsocket;
 }
```

via the URL www.mywebpage.com/sensor/

# JSON C++ encoding with jsoncpp

```
virtual std::string getJSONString() {
Json::Value root;
root["epoch"] = (long)time(NULL);
Json::Value values;
for(int i = 0; i < datasink->values.size(); i++) {
    values[i] = datasink->values[i];
}
root["values"]  = values;
Json::StreamWriterBuilder builder;
const std::string json_file = Json::writeString(builder, ro
return json_file;
};
```

# JSON C++ decoding with jsoncpp

```
virtual void postString(std::string postArg) {
  const auto rawJsonLength = postArg.length();
  JSONCPP_STRING err;
  Json::Value root;
  Json::CharReaderBuilder builder;
  const std::unique_ptr<Json::CharReader>
    reader(builder.newCharReader());
  reader->parse(postArg.c_str(), postArg.c_str() + rawJsonL
  // do something with root
}
```

where `root` is a `std::map`.

# Web browser: javascript

```
function getterCallback(result) {
  var temperatureArray = result.temperatures;
}

function getTemperature() {
  $.getJSON("/sensor/:80",getterCallback);
}

function documentReady() {
  window.intervalId = setInterval(getTemperature , 1000);
}

$(document).ready( documentReady );
```