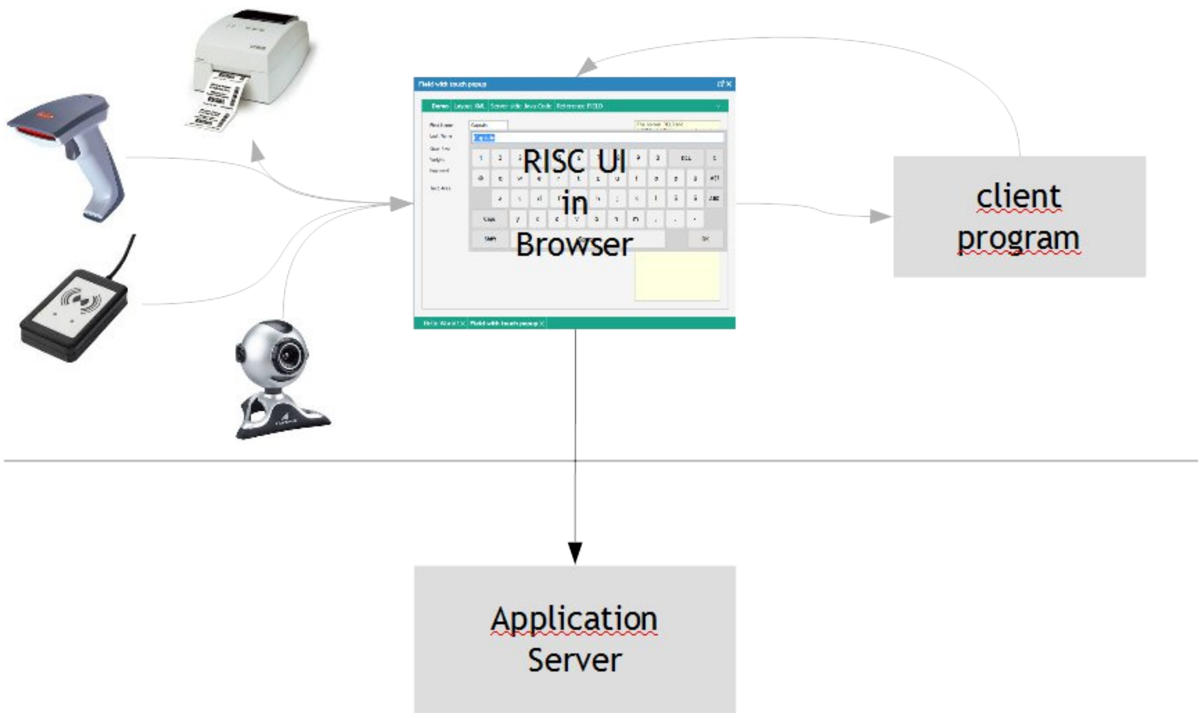
Connecting the RISC Client to “non-JavaScript-interfaces”

Motivation

In industry scenarios there is the necessity to connect the RISC client to client side sub-devices or interfaces.

Examples:

- serial / USB based scanner (which should not emulate a keyboard)
- Legic Reader for swipe cards
- direct access to client side files
- UDP messaging
- connection to an other program running on the client
- special label printers



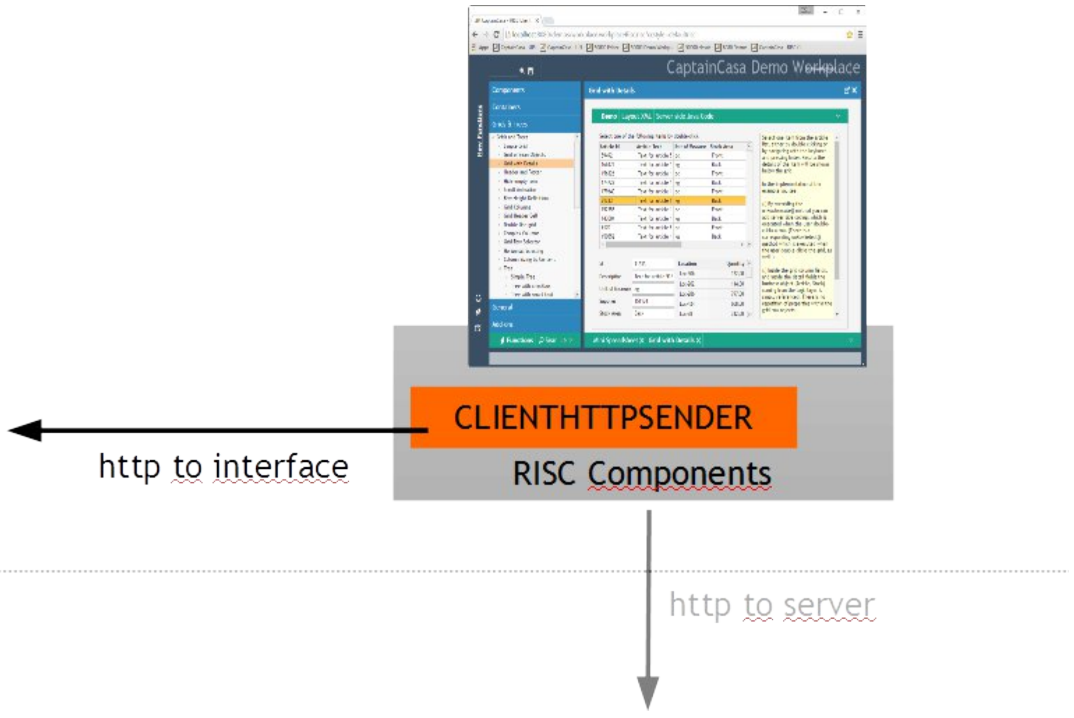
JavaScript does not provide functions to communicate to such interfaces directly.

Architecture

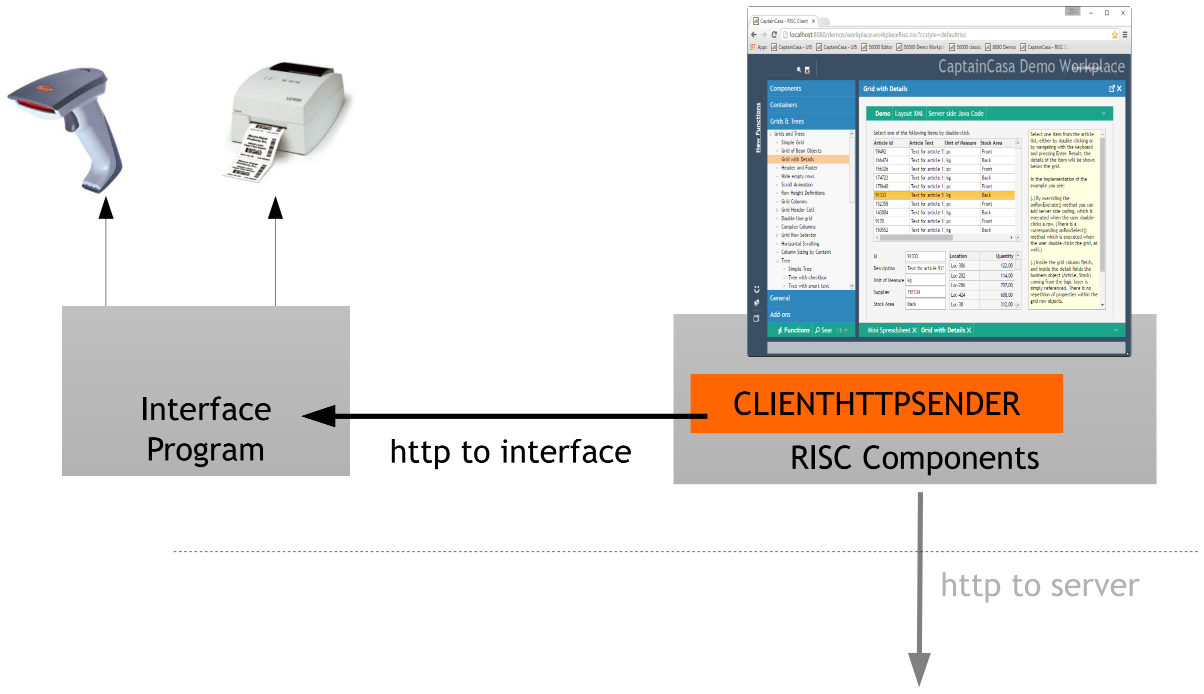
What we do not want to do: we do not want to “enhance” the browser by adding some plugins or extensions, that provide a possibility to connect to the native level from JavaScript. Such architecture comes with a high probability of failing in the real world of multiple browser with multiple version on multiple operating systems.

So what's our approach?

The solution is quite simple (and we are not the only ones doing this...): the only possibility which allows the browser to talk to others is to use an http-connection. So we provide an invisible component in the RISC component library, that is able to communicate to another program via http. The component's name is CLIENTHTTPSENDER.



At the receiving side of the http interface there is any program that has the task to connect to the subdevices and to transfer messages between the subdevices and http:



The direction of the http connection clearly is driven by the RISC client – this is the one to request data from the client side interface program. Of course you may use long polling in order to let the http request wait until some event is available within the interface program.

By default a browser client can only open up some communication to the server it was loaded from. With up to date browser installation supporting CORS (cross origin resource sharing) the browser may also talk to other web servers – including the interface program, which typically directly runs on the client.

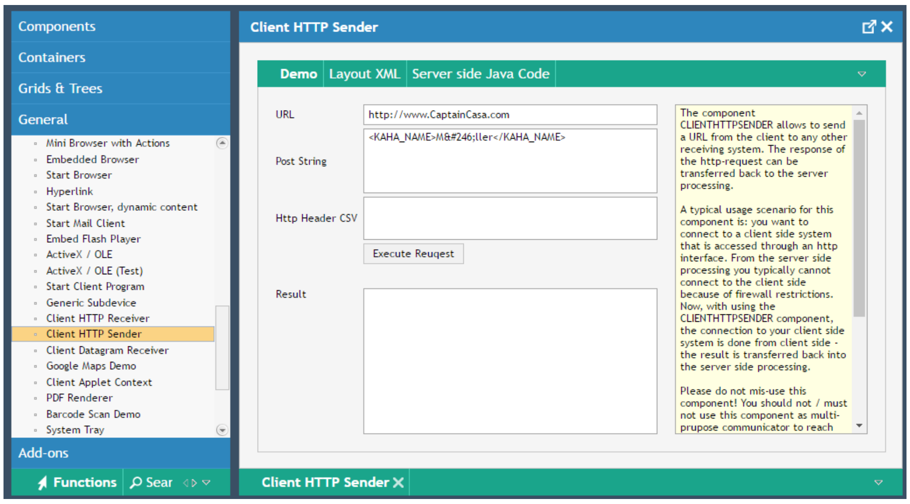
The component CLIENTHTTPSENDER

The component CLIENTHTTPSENDER is a quite simple one. You pass the following information by using certain attributes:

- URL: the URL to connect to (“http://localhost:8080/...”)
- URLPOSTDATA, optional: The POST-string (“param1=value1&param2=value2&...”) that you want to send with the URL
- HTTPHEADERPARAMS, optional: some http header parameters
- URLCALLBACK, optional: definition if the response of sending the URL is transferred back to the server (calling the method defined in ACTIONLISTENER) or not. The default is “false”, this means you have to explicitly switch it on by setting the value “true”.
- TRIGGER: trigger for sending the URL

When activating the TRIGGER attribute of the component, then the component will open up a connection (via XMLHttpRequest). The result of the request is received and – if attribute URLCALLBACK is set to “true” - is transferred to the server side using the ACTIONLISTENER of the component (event “BaseActionEventClientHttpSend”).

Please check the example in the demo workplace:



URL: http://www.CaptainCasa.com

Post String: <KAHA_NAME>M&#246;ller</KAHA_NAME>

Http Header CSV:

Result:

The component CLIENTHTTPSENDER allows to send a URL from the client to any other receiving system. The response of the http-request can be transferred back to the server processing.

A typical usage scenario for this component is: you want to connect to a client side system that is accessed through an http interface. From the server side processing you typically cannot connect to the client side because of firewall restrictions. Now, with using the CLIENTHTTPSENDER component, the connection to your client side system is done from client side - the result is transferred back into the server side processing.

Please do not mis-use this component! You should not / must not use this component as multi-prupose communicator to reach

Developing a client-side Interface Program

The CLIENTHTTPSENDER is a generic component – it just transfers a message from the RISC client to an other program and delegates the response back into the server side processing. There is no predefined semantics of how data transferred should look like.

You now “only” have to develop an interface program that opens up a little web server on client side, so that you can send http-requests using the CLIENTHTTPSENDER component. Using Java the default way of using such program is to write a servlet and to embed it into some small servlet engine like Tomcat or Jetty, that you then install on client side.

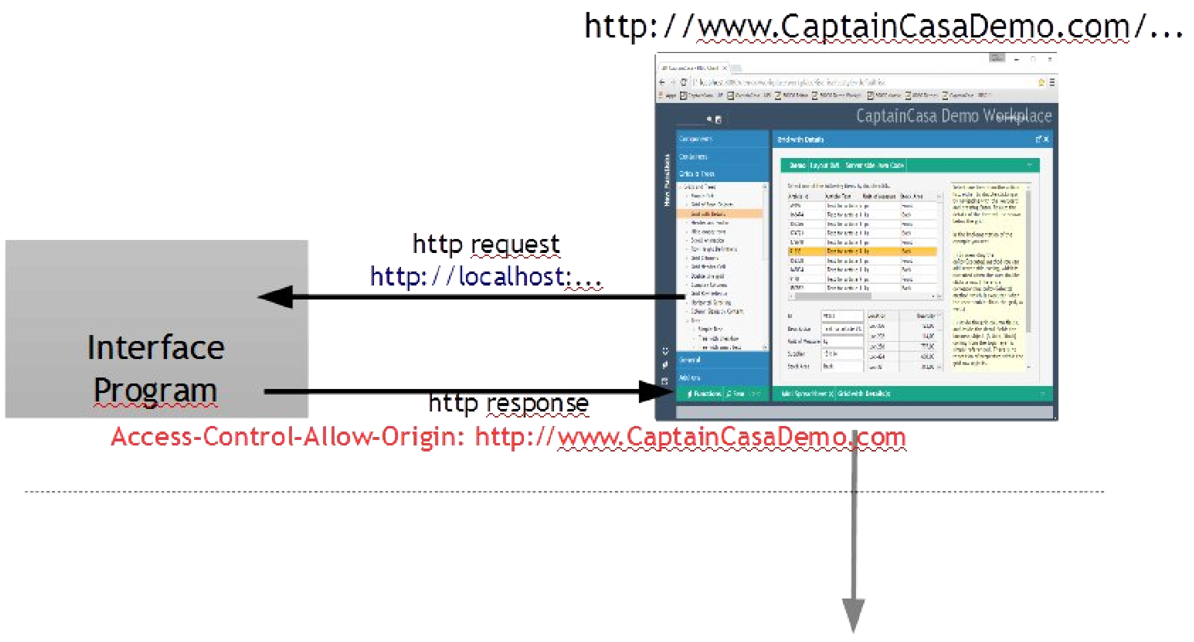
CaptainCasa has written such interface program as show case, you may obtain the whole code on request.

There are some issues to be pointed out - that are listed in the following chapters.

Implementing CORS so that RISC Client can talk to multiple servers

The interface program needs to explicitly allow the page that is loaded within the browser (i.e. the “RISC dialog”) to use the response that is sent by the interface program. This is done by setting the http-repsonse header parameter “Access-Control-Allow-Origin” and passing the original domain of the page. This sounds complex, but is not at all!

Take a look onto the following image:


http://www.CaptainCasaDemo.com/...
Interface Program
http request
http://localhost:...
http response
Access-Control-Allow-Origin: http://www.CaptainCasaDemo.com
CaptainCasa Demo Workplace

The situation described in the image is:

- In the browser a RISC-dialog is loaded from “http://www.CaptainCasaDemo.com/ccdemos/xyz.risc?ccstyle=defaultrisc” - so the original domain is “http://www.CaptainCasaDemo.com”
- As consequence the domain ”http://www.CaptainCasaDemo.com” has to be added to the repsonse header of all response messages that are returned by the interface prorgram, so that the local interface programm allows the browser to also use it together with the page domain.

A hard coded way to do this within you servlet processing would be:

```
public class LocalGatewayServlet extends HttpServlet
{

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException
    {
        try
        {
            resp.setHeader("Access-Control-Allow-Origin",
                               ”http://www.CaptainCasaDemo.com”);
            ...
            ...
            ...
```

You may implement CORS within your servlet in a quite generic way:

```
public class LocalGatewayServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException
    {
        doPost(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException
    {
        try
        {
            String pageDomain = req.getHeader("origin");
            if (pageDomain != null)
            {
                checkSendingPageDomain(pageDomain);
                resp.setHeader("Access-Control-Allow-Origin",pageDomain);
            }
           ...
           ...
           ...
```

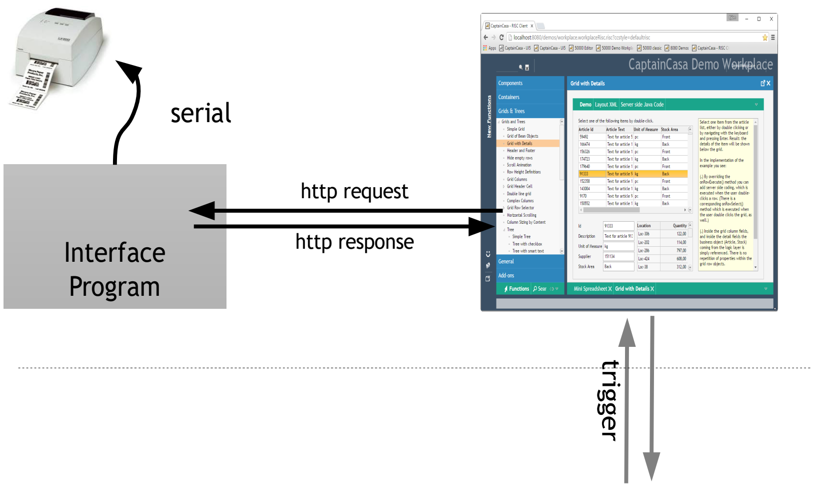
In the code the domain of the page is dynamically detected by using http-request header parameter “origin”. The domain is set as valid domain for the response.

Of course you need to pay attention: by using this code you open up your local interface program to anyone! So you should at least have some additional code to check the origin!

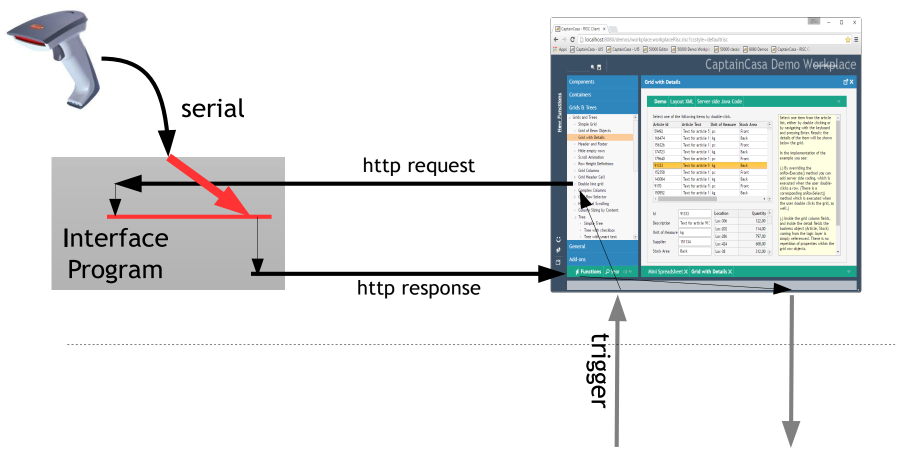
Sending Information <=> Waiting for Events

When communicating from the browsers CLIENTHTTPSENDER component to the client side interface program there are two different scenarios:

- You want to pass certain information to the client side.
  - Example: you have a certain printer connected via serial interface. So the client side interface program is triggered by the CLIENTHTTPSENDER, within the http message the information what to print is passed.



  - In this case it's quite simple: you send the message to the client side interface program by triggering the CLIENTHTTPSENDER component. The result of the processing is sent back to the server side.
- You want to wait for a certain event that occurs on client side.
  - Example: you have a barcode scanner connected via serial interface.
  - In this case you have to establish some type of “long polling” communication to your client side interface program:



Implementing some kind of long polling means:

- The CLIENTHTTPSENDER gets triggered and calls the client side interface program.
- The client interface program does not immediately send the response but it waits for a certain event. The waiting can be implemented by some thready synchronization (Object.wait()).
- The event of scanning is received in the client side interface program and releases the thread that is currently waiting (Object.notify()/ Object.notifyAll()).
- The thread that now is released sends the response. As consequence the CLIENTHTTPSENDER component talks to the server side.
- The server side immediately triggers the CLIENTHTTPSENDER component again, so that a new connection is built up to wait for new events.

Dependent on the “quality of receiving events” you need to think carefully about some issues:

- It can happen that the http request to the client side interface program is aborted or timed out. In this case the server gets notified but receives some error information: the server side event of type “BaseActionEventClientHttpSend” has a corresponding method “getSuccess()” which returns “false” if there was communication problem on client side.
- It may happen that there is an event triggered by the scanner just at this point of time when the http response was given and the response is transferred to the server side. When establishing a new connection from the browser to the client side interface program then it's already to late to catch the event.
  In this case you have to first write the event information in some queue and pick up the queue correspondingly – so that no event information gets lost.