XIOS/3 Introduction Developer Tutorial

Welcome to the introductory XIOS/3 Developer Tutorial. The intent with this set of tutorials is to teach you how to set up a development environment, start a new project and be able to deploy it, along with pointers for where to learn more about specific components you need in your projects. This first XIOS/3 tutorial doesn't assume any existing developer knowledge of XIOS/3, but does require some fundamental knowledge of XML and web technologies.

Any questions regarding the tutorial itself or how to continue afterwards can be sent to tutorials@xios3.com.

The state of the Software Industry

The software industry has in the past decade seen a large architectural shift where services have moved from locally hosted to cloud hosted to microservices. This change has brought with it a new and network connected way of creating applications, where the application front-end is running on the client, but storage, security, and proprietary data and algorithms are executed within external services. While a lot of work has been done on innovation and standardization on the server side, the client side is quite fragmented, with large capability gaps to meet the quality and requirements for applications for an enterprise environment.

Front-end development today is complex, time-consuming and often a tradeoff between micro management of the UI or combining a large set of libraries and dependencies into a coherent application. Even with state-of-the-art frameworks there is no solid foundation, resulting in a constant cycle of reinventing the wheel for developers, while the functionality and features known from installed software often being out of project scope and not implemented due to resource constraints. However, there are low-code products addressing current complexities, but what they make up for in simplicity they lack in flexibility and functionality in the applications one can build.

Introducing the HTML5 Client Application Server & Application Markup

To tackle these issues, XIOS/3 has developed and patented a unique technology, enabling developers to build applications in highly abstracted Application Markup, based on XML. The Application Markup is then interpreted in a Client Application Server implemented in HTML5. The solution is a combination of the simplicity and speed of low-code with the flexibility and usability features made possible in traditional programming. XIOS/3 allows developers to focus on finishing projects instead of constantly reinventing the wheel with basic functionality, since building blocks with commonly used functionality is easily available to developers out-of-the-box.

The technology enables encapsulation and composition of UX components written specifically for XIOS/3, or in common component libraries found as open source, such as React. JavaScript application logic, protocols and data adapters connecting to server back-end logic can be wrapped into reusable building blocks where XIOS/3 application markup glues together user interfaces, application data and application logic. XIOS/3 provides advanced features such as; drag & drop, copy & paste, lasso select, keyboard shortcuts, intelligent data-binding mechanisms, multiple application and instance execution, collaboration of multiple users, transaction management, hibernation and much more.
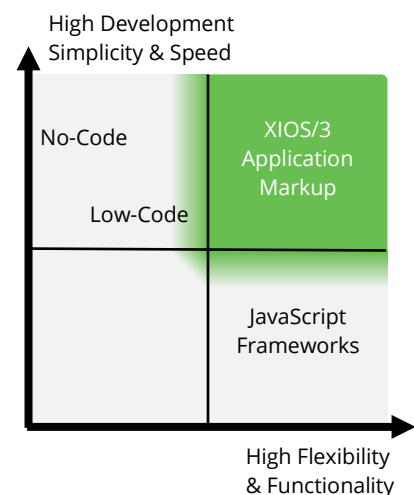
When using XIOS/3, server-side application logic can be limited to providing a Service Oriented Architecture, coordinating client communication, while providing, aggregating and persisting data used in apps. The server is then relieved from generating HTML and user interface logic. The Client Application Server contains many of the artifacts normally found in advanced server-side application servers but runs lightweight in any web browser, without any installation or plug-ins. It replaces outdated technologies like Adobe Flash, Microsoft Silverlight, and Java, while at the same time providing a more robust and abstracted environment than pure HTML or current frameworks. However, XIOS/3 is a great compliment to existing frameworks such as React, Angular and Vue.

Installation: How to get started with XIOS/3

In this section of the tutorial we will have a quick look at the XIOS/3 runtime and show how you can start it on your own computer. It is helpful if you already know how to start a webserver on your development machine.

First you need to have a copy of the XIOS/3 distribution, typically named "xios.zip". Find a good place for it on your computer, e.g. the Documents folder on Microsoft Windows, and uncompress the xios.zip file there. You should end up with a new folder named "xios". Inside of it you will find a Readme.txt file a few HTML files and another folder named "xios". This folder, with the HTML files, will be referred to as the XIOS/3 root folder from now on.

We will now need to start a web server with the XIOS/3 root folder as the web servers root folder. For this tutorial we recommend you use the Python built in web server, but any web server would work. If you are running Apple MacOS you should already have Python installed on the computer. If you are running Microsoft Windows, download and install latest Python from Microsoft Store or from https://python.org. Once you have Python installed, open a terminal shell e.g. by typing cmd and enter in the search feature, go to the XIOS/3 root folder and start the web server with the command

```
python -m SimpleHTTPServer
```

if you have Python 2 installed, or

```
python -m http.server
```

if you have Python 3 installed.

This command will open a web server on the local host on port 8000.

You can now open your Chrome web browser and go to the web site http://localhost:8000/ to see either a list of the files in the XIOS/3 root folder, or a command line interface, depending on what web server you are running. If you see a file listing, click the index.html entry to get to the command line interface.

What you are seeing is a XIOS/3 application running inside of the XIOS/3 runtime running inside of your Chrome browser. Both the entry page, the XIOS/3 runtime and the application are served from the web server running on your computer.

Note that as long as the web server is running, the contents of the XIOS/3 root folder is available to anyone who attempts to access port 8000 on your computer. You can close the server by pressing Ctrl+C in the shell window running your Python web server.

Explore CloudShell

In this section we will have a closer look at the XIOS/3 shell application and its commands.

The XIOS/3 shell works just like most command line shells do. You can type "help" to see the readily available commands, grouped in four different categories. File, Shell, System and User commands. Currently none of the file or user commands are available, as there is no file system mounted and no user account service connected. We can see the lack of filesystem by running the command "drives".

However, we can run "channels" to see that we have one defined communication channel "https" available. Running "apps" we can see we have one application "Shell" running, the application we are currently interacting with. Run "views" to see that we have one view displayed, the one associated with the Shell application, using the identifier "Shell:1". This is the name of the view, "Shell", followed by the instance number. Finally we can type "ps" to show that there are two processes running, one for the Shell application and one "startup.xml". This startup process was the one that set up the https channel and launched the Shell application.

In addition to the commands, it is also possible to set up macros. Some are already defined and can be listed with the command "macros". As you can see several of them simply loads another application with the "load" command. Let's try some of them.

Start with the "dev" macro, it opens a simple document viewer that displays reference documentation for XIOS/3 itself. You can look up the documentation for the Shell component itself by expanding your way down References, Components and Shell in the tree component to the left of the Help window.
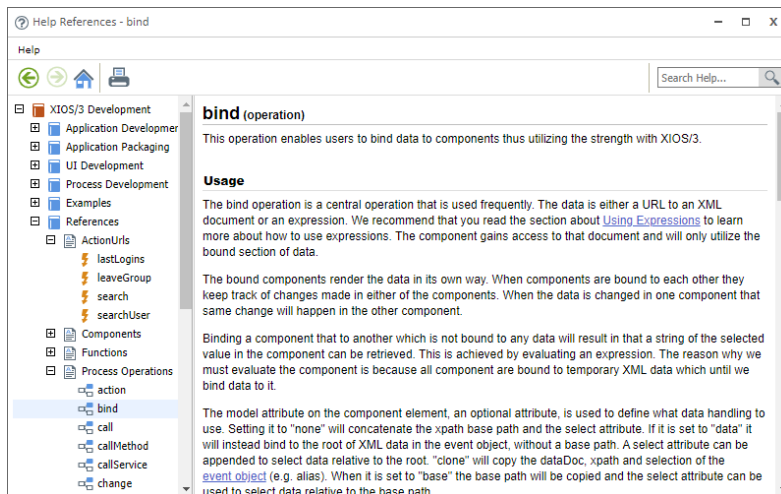
Another interesting macro to try is "system". This starts another application that show the same information we have already seen with the apps, views, ps, etc. commands.
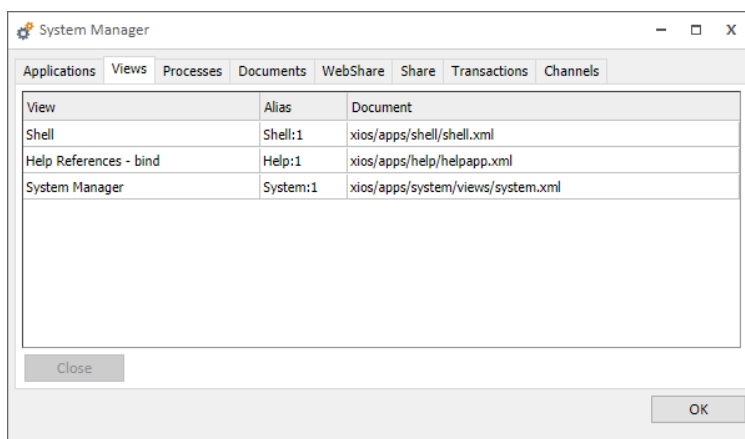
We can start another instance of the shell application using the load command, followed by the application path shown by the apps command, i.e., "load xios/apps/shell/shell.xml". This will open another instance of the shell application, but this time in a window. You can make the new window stand out a bit more by changing its background using the bgcolor command, e.g., "bgcolor #046".
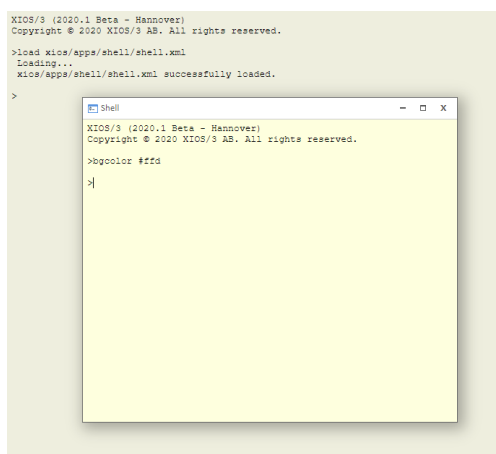
Note that we still only have one application loaded, as shown by "apps". We still have the same number of processes running, as shown by "ps". We do however have another view instance, Shell:2, running. The events that are triggered will however go into the same process code for both views.

The cloud shell itself has a bit more verbose documentation available in the help application as well. One way to access it is to simply right click anywhere in the shell and get a context menu with the "Help center…" option.

Create a View

In this section we will create a simple view and load it inside the XIOS/3 runtime. To understand this section, you should already be familiar with basic XML.

In order to start developing we need an editor that can manage editing XML files. A normal text editor like notepad would be possible to use, but there are products more suitable for development work, from sparse editors like emacs to complete environments like Microsoft Code. The more capable editors can help you verify that the syntax you are typing is correct.

In whatever editor you have chosen, start a new file in the XIOS/3 root folder named "app.xml". In the file, create a simple view tag like this

```
<view name="App" title="My App">
</view>
```

and then load the app view into the XIOS/3 runtime using the command "load app.xml". A view can be displayed in many different ways, such as taking up the entire tab, like the shell application, included in another view, or in this case displayed as a separate window. For this mode there are a number of view specific settings you can modify, to control things like the initial size of the window, minimum size of the window, if it is always on top, if it can be resized etc.

Let's start by setting a view icon using the icon attribute. This could point to any image file, but by using the icon protocol it is possible for the system to select theme-specific icon libraries and sizes.

```
<view name="App" title="My App" icon="icon://monitor_brush">
</view>
```

You can reload the currently focused view by pressing Ctrl+Alt+R on the keyboard. Note that such a reload will not at this point execute view specific process code that might be needed to properly set up the view, but for this purpose it is enough. Otherwise, you would have to reload the web page and load the view again.

There are a few different themes that can be used to style the applications in different ways through the theme attribute. Some useful examples are marble, the default theme, and liquid. You can read more about how to customize the view in the UI Development ► Overall Structure ► View section of the Development Manual.
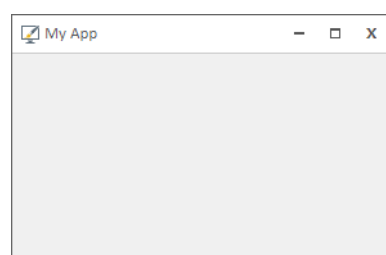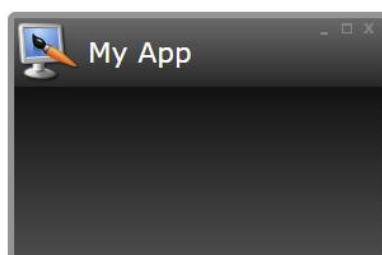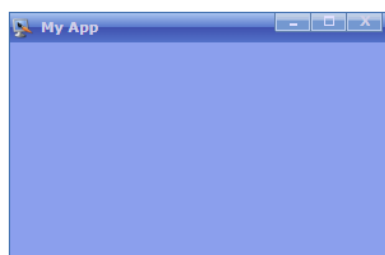
There are a few different things you can define inside the view tag. Some of it you have direct control over how it is displayed, while some of it is up to the windowing system and the active theme to decide how to render. These are things like menu bars, typically rendered at the top of the window, and icon bars, again typically rendered at the top, but could be detached or drawn along the sides or the bottom. In addition, you can also define elements that are part of the view, but not necessarily shown immediately, like context menus.

The most common item to define is however the components that are directly rendered within the view. These are placed inside a layout container, such as the panel element. That in itself doesn't do much. To see the area the panel actually covers you can try to set the bgcolor attribute on the panel element, e.g., bgcolor="red". The bgcolor attribute accept CSS color definitions. The full documentation of panel component is found in References ► Components ► Panel.

```
<view name="App" title="My App" icon="icon://monitor_brush">
  <panel bgcolor="red">
  </panel>
</view>
```
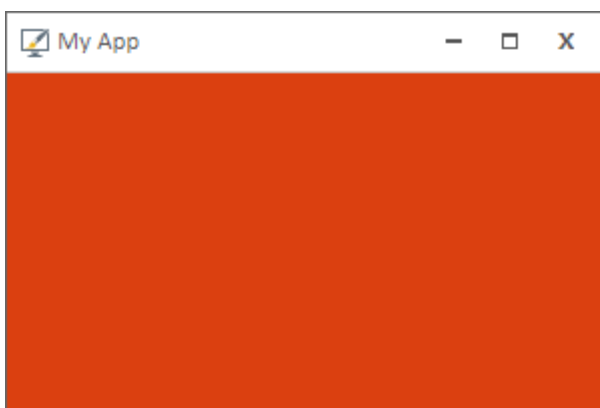

There are different kinds of components we can use inside of this panel element. Some elements such as the group, panel and container components are just used for various layout mechanisms. These are referred to as wrapper components or container components. It is possible to put panel elements inside of panel elements to group components together and control how they align horizontally and vertically to each other. Other components we refer to as atomic components. They represent specific state on the screen. One of the simplest of these is the label component. The full documentation of the label component is found in References ► Components ► Label.

```
<view name="App" title="My App" icon="icon://monitor_brush">
  <panel>
    <label size="20" default="This is a label."/>
  </panel>
</view>
```
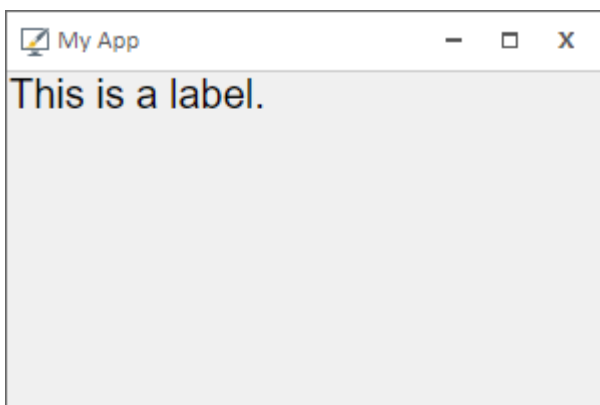
Create a Process

In this section we'll expand the App view with some simple processing code to bind a component to a data source. To understand this section, you need to have some basic understanding of XPath expressions.

In XIOS/3 applications are created by combining one or more views with one or more processes. We can create a simple application by wrapping our view in the application element. The name and icon of the application isn't directly visible to the user, but is what is shown in management views. One important setting you can do on application level is to restrict the number of instances of an application through the instances attribute. 0 means unlimited number of instances, which is the default. The full documentation of the application element and its children is found in Application Packaging ► Overall Structure.

```
<application name="My App">
  <view name="App" title="My App" icon="icon://monitor_brush">
    <panel>
      <label size="20" default="This is a label."/>
    </panel>
  </view>
</application>
```

The second part of an application is the process code. Process code is built by two things, triggers that listens for specific events and steps that are executed by triggers when the specific event happens. Let's start by looking at a trigger. The following trigger listens to the "Loaded" event sent by the "App" view. That is, when the App view has been opened and all its components have loaded it will execute the instructions in the "init" step. Triggers are documented in Process Development ► Overall Structure ► Triggers.

```
<trigger view="App" event="Loaded" step="init"/>
```

The steps themselves are on the high level also very simple. It is a series of processing instructions that are executed in order. It is mostly just a series of operation elements with a specific name attribute selecting what operation to perform. This allows for easy automatic validation and generation of code. The triggers and steps are placed inside of a process element that is given a name. This name is what is presented in the process manager. The step element is documented in Process Development ► Overall Structure ► Steps.

```
<process name="App Process">
  <trigger view="App" event="Loaded" step="init"/>
  <step id="init">
    <operation name="debug" value="It's loaded!" />
  </step>
</process>
```

The debug operation will, with no other parameters, write the given value into the JavaScript debug console. You can open this in Chrome by pressing Ctrl+Shift+I. By adding content to the operation element you can modify its behavior. Add the content "alert plain" to get a JavaScript alert instead, or "alert" to get a XIOS/3 alert. As you can see from the JavaScript alert, just because everything is loaded doesn't mean it is rendered to the screen yet. Documentation of the debug processing operation is found in References ► Process Operations ► debug.
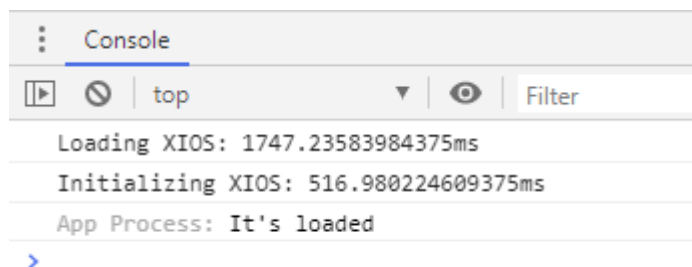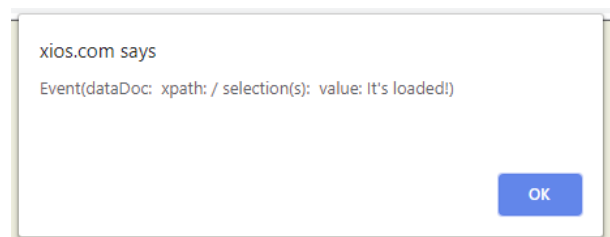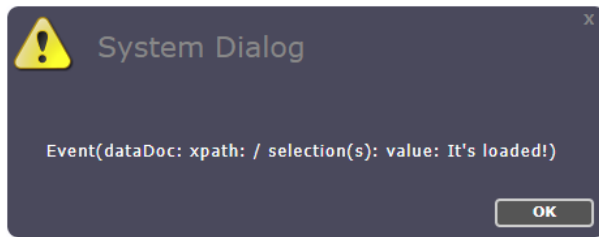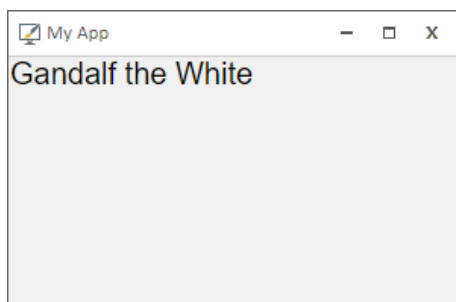
Let's make a small processing step that have the value of the label depend on a different source. For testing purposes we can create inline resources in the same document and reference them using a simple scheme of prefixing the resource name with a hash sign.

```
<resources>
  <item name="employee">
    <card>
      <name>Gandalf the White</name>
      <occupation>Wizard</occupation>
    </card>
  </item>
</resources>
```

What we have is a resource element inside the application element. In the resource element we can have multiple item elements, each with its own name. In this case we have a single inlined document under the name "employee". We can now use the "bind" operation to bind a selection from the inlined resource to a specific component. First, we need to add a name to the label component so we can reference it. Then we add the operation element with the inline resource as value. Finally, we tell the bind operation what components we want to bind to what parts of the resource, using XPath expressions to select. The bind processing operation is documented in References ▶ Process Operations ▶ bind.

```
<application name="App">
  <resources>
    <item name="employee">
      <card>
        <name>Gandalf the White</name>
        <occupation>Wizard</occupation>
      </card>
    </item>
  </resources>
  <view name="App" title="My App" icon="icon://monitor_brush">
    <panel>
      <label name="Name" size="20" default="This is a label." />
    </panel>
  </view>
  <process name="App Process">
    <trigger view="App" event="Loaded" step="init"/>
    <step id="init">
      <operation name="bind" value="#employee">
        <component view="App" name="Name" select="/card/name" />
      </operation>
    </step>
  </process>
</application>
```

Advanced Components

The label component is of course as simple as you can get, and for the amount of text the application does very little. Let's look at a more advanced example where we use an application component. Application components are components that are virtually an application in itself. As a demonstration we have wrapped the Google Map embeddable API into a XIOS/3 component.

Let's start a new file named "map.xml" that we start out very similar to the first example. A simple view with a panel, and inside the panel element we add the map element and declare a name for it. Every component that isn't just static HTML has to have a name, to be able to hook events to it. The map component is documented in References ► Components ► Map.

```
<view name="Map" title="My Map App" icon="icon://earth_location">
  <panel>
    <map name="mapComp"/>
  </panel>
</view>
```

You now have a XIOS/3 view inside a window with a Google Map embedded. You can scroll and zoom in the map, while at the same time move and resize the XIOS/3 window. Beside this dimensional data there is no information exchanged between XIOS/3 and the embedded Google Map application.
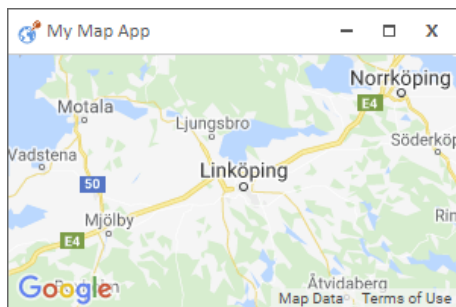
You can further customize the map element by turning visual controls off with controls="false", have it center on your location using position="fast", or setting you own map center by giving coordinates in the mapcenter attribute like mapcenter="58.3954813,15.5504102", i.e., latitude and longitude separated by a comma.

The map component can use an XML document with marker elements as data and plot all the markers on the map according to each element's longitude and latitude. They could look like this example.

```
<marker lat="58.434929" lng="15.591756" title="Burger King" icon="bk.png"/>
```

The problem is however that most external data sources will have their own format, so we need a way to map the external format to the internal format. Let's look at one such source, the Swedish weather service's open API. At the following URL you can get a live XML document listing all the current and historic weather stations in Sweden, selected by type of sensor. In this example the solar radiation.

```
https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11.xml
```

The returned data looks like this, truncated after the first station element and trimmed down to only show the default namespace.

```xml
<metObsParameter xmlns="https://opendata.smhi.se/xsd/metobs_v1.xsd">
  <key>11</key>
  <updated>2020-04-17T00:00:00.000Z</updated>
  <title>Global Irradians (svenska stationer): Välj station (sedan
tidsutsnitt)</title>
  <summary>medelvärde 1 timme, 1 gång/tim</summary>
  <valueType>SAMPLING</valueType>

  <station>
    <name>Borlänge Sol</name>
    <owner>SMHI</owner>
    <id>105285</id>
    <height>164.0</height>
    <latitude>60.4879</latitude>
    <longitude>15.429</longitude>
    <active>true</active>
    <from>2008-01-01T00:00:00.000Z</from>
    <to>2020-04-17T00:00:00.000Z</to>
  </station>
```

We can see that we want the /metObsParameter/station elements where the active element contains "true", and not "false". From those hits we then want to use the longitude, latitude and name elements to generate the lng, lat and title attributes to create indata for the map. There are a number of different ways we can select this information from the external data. The most common use is to have the component select what it wants to do with the in data. We simply add the following conversion rule to the map component. The match attribute in the rule element selects what elements in the incoming data to operate on. The XML inside the rule element is outputted for every match. The brackets have their XPath expressions evaluated against the currently matched node and is expanded to the result. Thus, for example {longitude} is expanded into the element value of the longitude child element of each match. Rules are documented in UI Development ► Using Components ► Rules.

```xml
<map name="mapComp">
  <rule match="metObsParameter/station[active = 'true']">
    <marker lng="{longitude}" lat="{latitude}" title="{name}"/>
  </rule>
</map>
```

While the map component now knows how to handle the data, there is still no connection between the component and the document. To set that up, as before we need some process code, that performs a bind operation between the document URL and the component in the view, triggered when the view has loaded.

```xml
<process name="MapProcess">
  <trigger view="Map" event="Loaded" step="init"/>
  <step id="init">
    <operation name="bind"
value="https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11.xml">
      <component name="mapComp" view="Map" />
    </operation>
  </step>
</process>
```
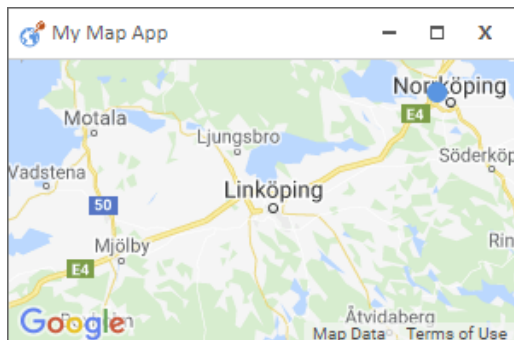
You now have an external service wrapped in one component, bound to another external service data source, integrated inside an application running inside a window within XIOS/3. Let's have a closer look at what that means. Type again the "views" command in the shell and you will get a list of the active views. It should look something like this, telling us we have two views, one for the Shell that takes up all of the background, and one for the map application. In the line for each view we can see the name of the view, followed by the instance number of the view, followed by the path to the view document.

```
Active views

Shell:1: xios/apps/shell/shell.xml
Map:1: map.xml

2 View<s>
```

The view and instance number together is important, because it allows us to reference an active view from the shell. By typing "type #Map:1" the xios shell will print out some information about the view.

```
Document: map.xml
Selection: None
Base: /application[1]/view[1]

<view name="Map" title="My Map App" icon="icon://earth_location">
 <panel>
  <map name="mapComp">
    <rule match="metObsParameter/station[active = 'true']">
      <marker lng="{longitude}" lat="{latitude}" title="{name}"/>
   </rule>
  </map>
 </panel>
</view>
```

The three first lines are not part of the document, but is an example of how all internal references in XIOS/3 looks. First there is a document, which isn't hard in this case. It is the map.xml file that we have been working on. The second part "Selection" isn't something that is relevant for views, so it will always be "None" here. The third part "Base" tells us the xpath in the document (map.xml) that points to the view element that is actually used for the view. In this case it is the first "view" element in the first (and only) "application" element in the document.

We can go one step further into the view and inspect the individual components also. In this view we only have one named component, the map component "mapComp". We can have a look at it with the command "type #Map:1#mapComp". This gives us the following reference followed by the actual XML data from the remote document.
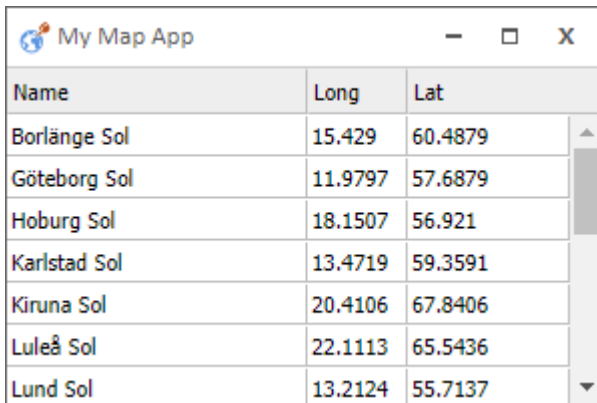
```
Document: https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11.xml
Selection: None
Base: /
```

As you can see, we get the data source that is bound to the component as result when we ask for the value of the component. The map component does however support selections, so we can go ahead and click one of the markers in the map, and type the same command again (arrow up and enter). What we get back, instead of the entire document, is just the XML element that corresponds to the selected node in the map. This is one of the key concepts in XIOS/3. Components are bound to data sources and, depending on how they are presented and viewed, represent different parts of that data when queried.

Finally, let's have a look at how we can modify the view without altering the process code. Let's replace the map element in the view code with the following code for the grid component. More documentation on the grid component can be found in References ► Components ► Grid.

```
<grid name="mapComp" selection="row">
  <row match="/metObsParameter/station[active = 'true']">
    <column match="name" display="." width="150" label="Name"/>
    <column match="longitude" display="." width="50" label="Long"/>
    <column match="latitude" display="." width="50" label="Lat"/>
  </row>
</grid>
```

The grid component is a component that creates a table according to how it is configured and the data bound to it. We name it mapComp so we don't have to change the process code, and we say we want to consider rows the selectable unit. Then we define what a row should look like. As before we are using the same xpath to select all "station" elements with a child element "active" with the text content "true" from the bound document. Then we create three different columns, "Name", "Long" and "Lat". For each column we match one of the station elements child elements "name", "longitude" or "latitude", and then we select the value of that node to present with the display attribute.

As before you can use "type #Map:1#mapComp" to see the bound document or the selected child element of the bound document.

Stand Alone Application

XIOS/3 is loaded within an HTML page, as we have seen previously. The contents of that HTML can be very minimalistic to start the environment and the shell application we have used so far.

```
<html>
  <head>
    <script src="xios/boot.js" charset="utf-8"></script>
  </head>
</html>
```

This is enough to have the XIOS/3 boot loader load all the relevant dependencies, initialize the system, create the elements needed to start drawing, and finally tell the application manager to load and start the shell application. The behavior of the boot loader can be controlled by the global variable "boot". To start our map application, set the "startup" property on the boot object to the path to the application.

```
<html>
  <head>
    <script>
      var boot = {
        startup : "map.xml"
      };
    </script>
    <script src="xios/boot.js" charset="utf-8"></script>
  </head>
</html>
```

This results in the application loaded without the window frame around the view, as if it was any other web application. The component is however loaded with apparently no data bound to it, regardless if the map or the grid component is used. The reason is that the application is binding an external source with the "https" protocol, but XIOS/3 has no protocols defined by default.
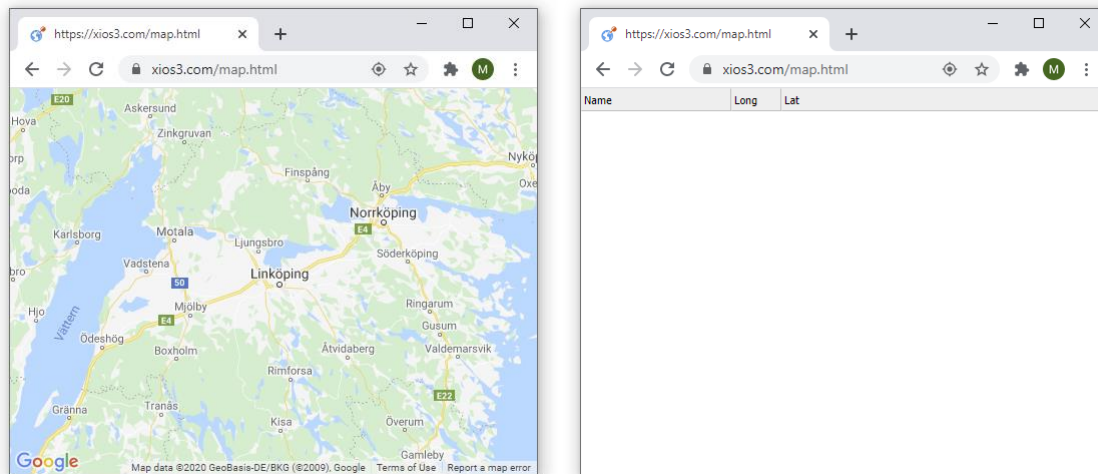
When the default shell application is started in XIOS/3 we have an "https" protocol defined in the environment as a convenience. We can look at how it is created by inspecting the default startup process "xios/config/startup.xml", which we already have seen running by typing "ps" in the default shell.

```
<process name="XIOS CLI System" description="Minimal startup into XIOS CLI"
         author="Martin Nilsson" xmlns:xlink="http://www.w3.org/1999/xlink">
  <step id="1" name="Start Shell">
    <operation name="channel">
      <name>https</name>
      <title>HTTPS</title>
      <protocol>HTTPS</protocol>
    </operation>
    <operation name="open" value="xios/apps/shell/shell.xml"/>
  </step>
</process>
```

If we look closer at this process, there are only two simple things that happens. First of all, we have a step with id "1", which as automatically executed as soon as the process is loaded. In it we first run the "channel" operation, which sets up a communication channel. This one is very simple, setting up the protocol "HTTPS" (the name of the protocol module) under the name "http" (the URL prefix to use in applications), using the presentation title "HTTPS" (the channel name presented in the UI in e.g., the system resource application). Once the channel is set up the second operation simply opens the shell application from "xios/apps/shell/shell.xml". Both the channel and the open processing operations are documented under References ▶ Process Operations.

We can copy the first operation, the "channel" operation and paste it as is first in the "init" step in our map application. It has to be above the "bind" operation, as the "https" protocol has to be defined for it to work.

```
<application name="My Map App">
  <view name="Map" title="My Map App" icon="icon://earth_location">
    <panel>
      <map name="mapComp">
        <rule match="metObsParameter/station[active = 'true']">
          <marker lng="{longitude}" lat="{latitude}" title="{name}"/>
        </rule>
      </map>
    </panel>
```

```
      </view>
    <process name="MapProcess">
      <trigger view="Map" event="Loaded" step="init"/>
      <step id="init">
        <operation name="channel">
          <name>https</name>
          <title>HTTPS</title>
          <protocol>HTTPS</protocol>
        </operation>
        <operation name="bind"
value="https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11.xml">
          <component name="mapComp" view="Map" />
        </operation>
      </step>
    </process>
</application>
```
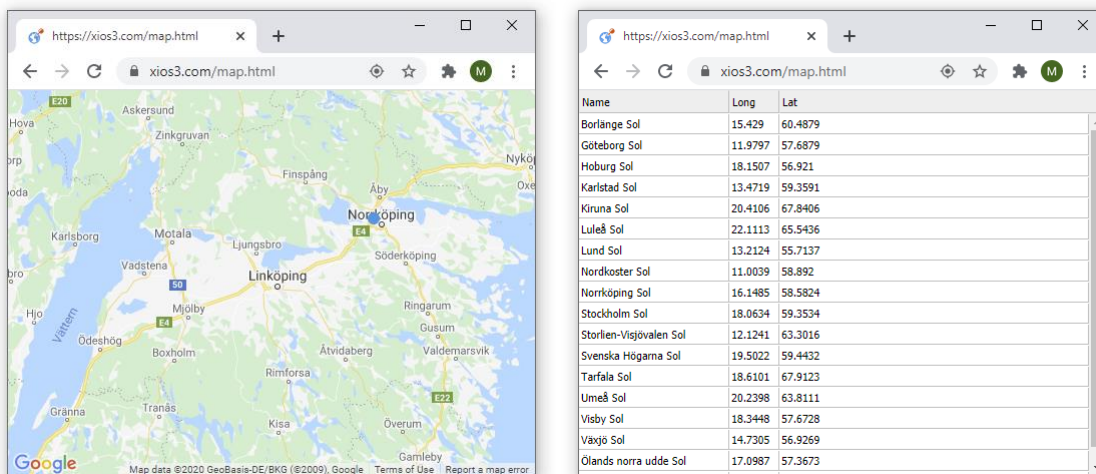
This will result in the external resource bound to the component will actually be able to load its data, since its protocol is now defined. As when previously run in a window, both the map and the grid version the full tab application will now load its bound data.

Simple Events

So far all the examples have only presented information statically without reacting to any user input, though the components themselves have been interactive. You can pan, zoom and select items on the map and you can sort and select the table. These interactions don't affect the data model or the program flow, but are internal to the presentation code of the individual components. It is a core philosophy of XIOS/3 that complexity should be encapsulated into components and data adaptors, and presentation should change automatically as data change.

But applications also need to react in more complex ways to events from users and other sources. We have already reacted to the trivial event that the application is loaded in both the label and map example. Let's expand a simple version of the label example a bit more. We add a new button element together with the label element in the panel. The panel component will by default start laying out its child components next to each other, wrapping to new lines as needed. This layout mode is called "flow". To not have the components stick right next to each other we can add the padding attribute and declare 8 pixels of padding between the components and the borders of the view.

```
<application name="My App">
  <view name="App" title="My App" icon="icon://monitor_brush">
    <panel padding="8">
      <label size="20" default="This is a label."/>
      <button width="80" name="ok" text="OK"/>
    </panel>
  </view>
```
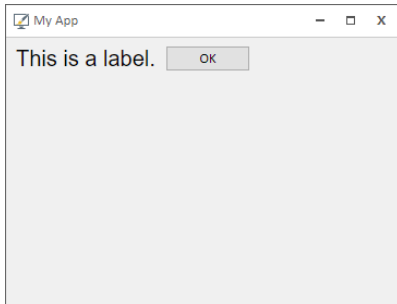
```
</application>
```

This will result in a view that looks as follows.

As you can see the button component visually reacts to mouse over and clicking, but nothing actually happens as there are nothing registered to be triggered by the component. We've already seen that we need a trigger and a step in a process section for that. Inside the application element we add a process element containing those. We set up the trigger to listen for the "Select" event from the button component and have it call the "exit" step. Inside the exit step we call the close operation and tell it to close the view by the name "App".

```
<process name="main">
  <trigger view="App" component="ok" event="Select" step="exit"/>
  <step id="exit">
    <operation name="close" value="App">
      <mode>view</mode>
    </operation>
  </step>
</process>
```

This of course makes it so the application closes if the OK button is clicked. We can add to this example further by adding a different trigger, again listening on the App view but instead on the "Escape" event. This will trigger when the escape key is pressed while this view has focus. We point to the same "exit" step, making it the target of two triggers.

```
<process name="main">
  <trigger view="App" component="ok" event="Select" step="exit"/>
  <trigger view="App" event="Escape" step="exit"/>
  <step id="exit">
    <operation name="close" value="App">
      <mode>view</mode>
    </operation>
  </step>
</process>
```

The pattern here is simple, the user performs an action, it results in a trigger which we capture, and have it execute a sequence of processing operations in a step. But it is also possible to set up a relationship between components that doesn't require any explicit processing to update one when the other changes. Let's go back to our map application and modify it a bit to have a side panel with some information in it. First we add the type attribute to the panel element, setting it to "column". This forces the components inside the panel to be laid out in columns, instead of the default "flow" type that more dynamically arrange the components depending on the size of the view. Then we add another panel element after the map element in the view XML. We name it "info", set its width to 150, add some padding so its components aren't touching the map or the window border, and tell it to layout its contents in rows.

```
<view name="Map" title="My Map App" icon="icon://earth_location">
  <panel type="column">
    <map name="mapComp">
      <rule match="metObsParameter/station[active = 'true']">
        <marker lng="{longitude}" lat="{latitude}" title="{name}"/>
      </rule>
```

```
      </map>
      <panel name="info" width="150" padding="8" type="row">
        <label default="Station:"/>
        <label name="station" bold="true"/>
      </panel>
    </panel>
  </view>
```

If we start this application we get the map as before, but we get a blank area to the right of the map with the "Station:" label on it. To make it display the name of the selected station in the map we simply add the following to end of the init step.

```
<operation name="bind" value="#Map#mapComp">
  <component view="Map" name="station" select="name" model="selection"/>
</operation>
```

This little snippet binds the map component to the label component, but in a different way than what we've seen before. First off, the expression in the value attribute to the bind operation is something we haven't seen before. It simply points to the mapComp component in the Map view and sets this as the source for the list of components to be bound to it. Our only component is the station component in the Map view. From the current value of the map view, which either is the entire document or a specific station element when there is a selection, we are going to pick out the element named "name". Finally, we are telling xios that we want to keep tracking the selection of the mapComp component into its bound document. So instead of being bound to whatever section of the document was selected at the time of binding, and only be updated when the document updates, we are instead bound to the live selection of the mapComp component. Whenever the user changes selection in the map the binding for the label will update its data.
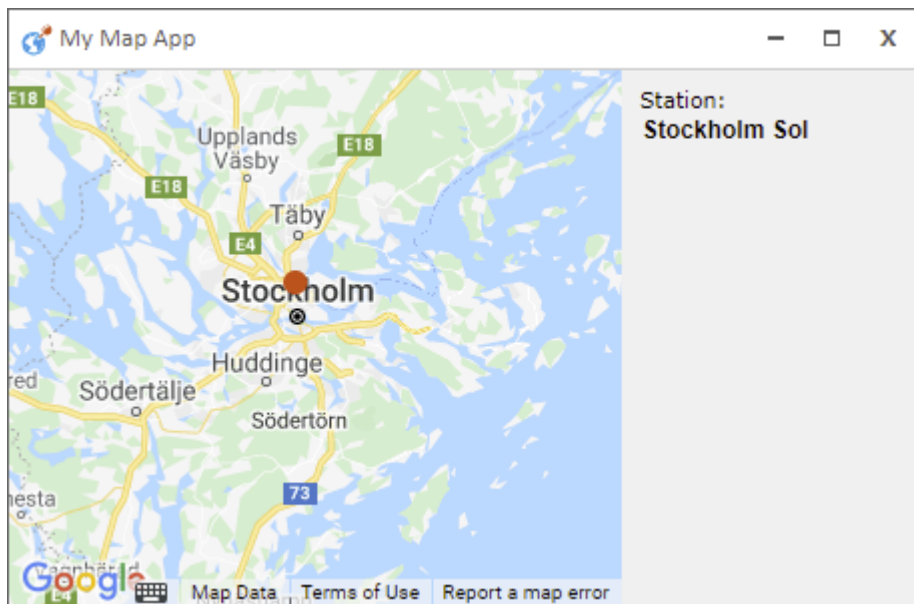
There is a small simplification we can do by replacing the two label components with the following single component, creating almost identical result.

```
<label label="Station:" name="station" bold="true"/>
```

The label attribute is actually handled by the panel component. It will look at all its child components and if they have a label attribute set it will create a label outside of the actual component.

Next we want to present some actual data form the sensor. Let's first have a look again at what data is actually bound to the component. As described earlier we can look at the data associated with the selected station on the map by typing "type #Map:1#mapComp" in the command line. If you do that for the "Stockholm Sol" from the example above you will get a return value similar to the following, with some elements removed for brevity.

```
<station>
  <name>Stockholm Sol</name>
  <owner>SMHI</owner>
  <ownerCategory>SMHI</ownerCategory>
  <id>98735</id>
  <height>41.0</height>
  <latitude>59.3534</latitude>
  <longitude>18.0634</longitude>
  <active>true</active>
  <from>2008-01-01T00:00:00.000Z</from>
  <to>2021-10-22T04:00:00.000Z</to>
</station>
```

The list of solar radiation sensor station was fetched from the following API URL where the notation is alternating key name and value, i.e. version 1.0 and parameter 11 (solar radiation).
https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11.xml

To get the sensor data we need to add what station and time period we are interested in. For this "Stockholm Sol" with station id 98735 and the latest available measured hour that URL is the following.
https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/11/station/98735/period/latest-hour/data.xml
Which contain something similar to the following, again with lots of elements removed.

```
<metObsSampleData xmlns="https://opendata.smhi.se/xsd/metobs_v1.xsd">
  <updated>2021-10-22T05:00:00.000Z</updated>
  <value>
    <date>2021-10-22T05:00:00.000Z</date>
    <value>-0.28</value>
    <quality>Y</quality>
  </value>
</metObsSampleData>
```

So what we want to do is have a click on a station create a binding between that station's sensor data document and a label component that can present the return data. First and simplest is to add a new label component to the view code.

```
<label label="Radiation:" name="radiation" bold="true"/>
```

Then we turn to the process code. First we set up a new trigger for select events from the map component.

```
<trigger view="Map" component="mapComp" event="Select" step="select"/>
```

Then we create a select step that dynamically creates a URL and binds it to the map component. For the URL he need to get hold of the id element from the map selection. That means selecting the xpath "id" from the "mapComp" component in the "map" view, which all together becomes the expression #map#mapComp#id. Then from the result we should bind the data in the /metObsSampleData/value/value to the "radiation" component. Note that the expression in the curly brackets are evaluated at the time of binding, i.e. when the selection happens.

```
<step id="select">
  <operation name="bind" value="https://opendata-download-
metobs.smhi.se/api/version/1.0/parameter/11/station/{#Map#mapComp#id}/period/latest-
hour/data.xml">
    <component view="Map" name="radiation" select="metObsSampleData/value/value"/>
  </operation>
</step>
```