PRAGMA POD

Eximple

1 Technology

This document describes a simple subset of the eXaMpLe typesetting framework called eXiMpLe. This framework provides ConTeXt users an automated typesetting environment, a user interface, and a couple of xml related styles. It started as an example of how you can use ConTeXt in combination with xml, but when it became more serious, we decided to combine it with our in-house TeX scripting environment.

The eXaMpLe framework consists of several components:

exampler This is a server application that listens to a specified TCP port and handles requests for typesetting or processing data or documents.

exampleq This application watches a hot folder, sends requests to the server and handles the replies.

examplex This is one of the applications that handles the requests sent to the server. It is controlled by scripts defining sequences, processes and commands.

examplet This program combines exampleq and exampler in a more sophisticated client-server technology. It can handle multiple requests in parallel and transfer files over networks.

In this document we only describe the watchdog features. The eXaMpLe manual provides more details about how to set up clients and servers, and how to define scripts that control the process. Here we assume that ConTeXt is used for simply processing xml using predefined scripts.

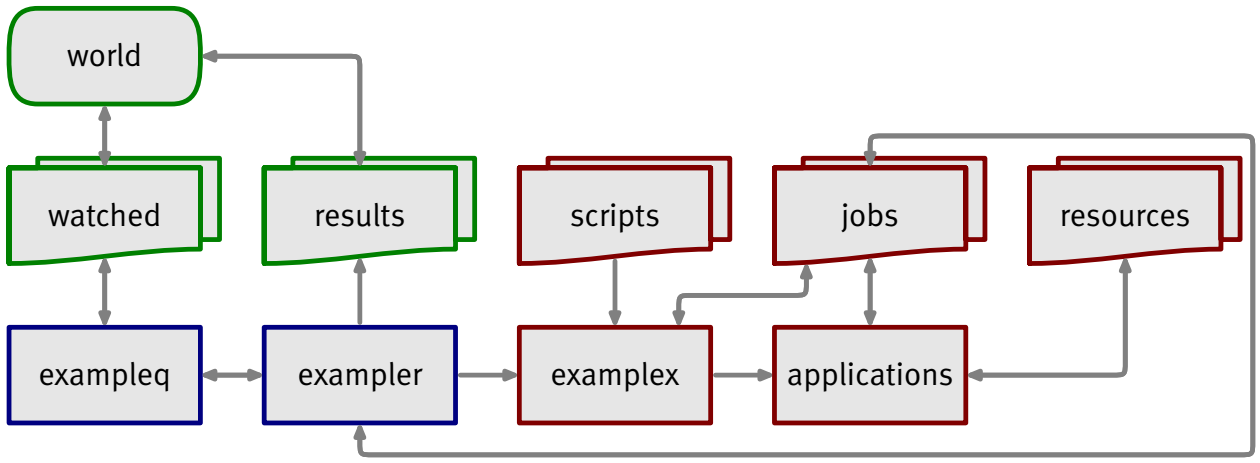

Figure 1 A rough scheme of the eXiMpLe workflow.

In figure 1 you can see how the three applications trigger each other. As soon as a request is placed in the watched folder, it is picked up by exampleq. When it is found valid, this request is send to exampler via a tcp/ip connection.[1]

[1] The exampler application is listening to a dedicated port. This mechanism permits other components of the eXaMpLe framework to communicate via an http protocol.

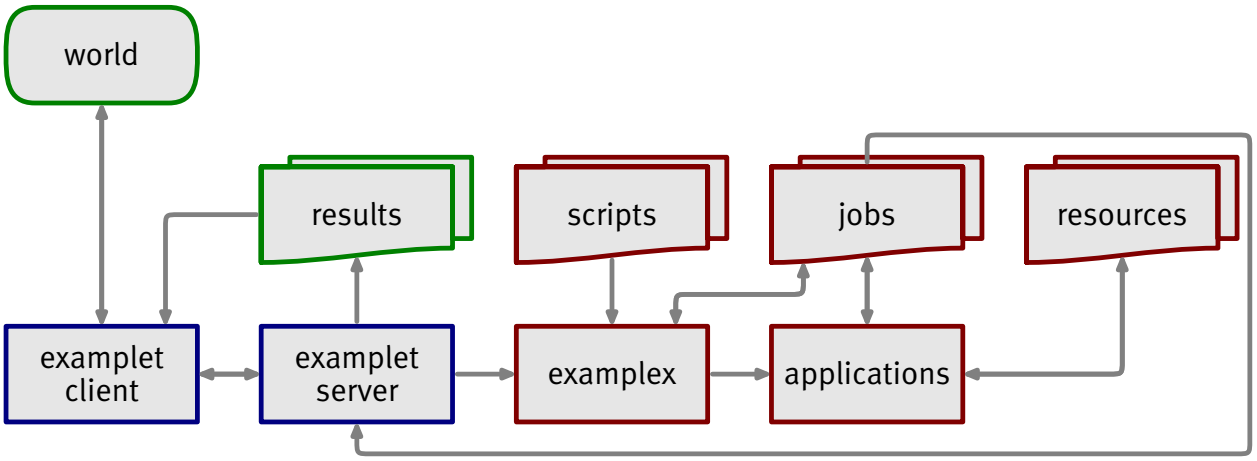

Figure 2 A rough scheme of the eXiMpLe workflow.

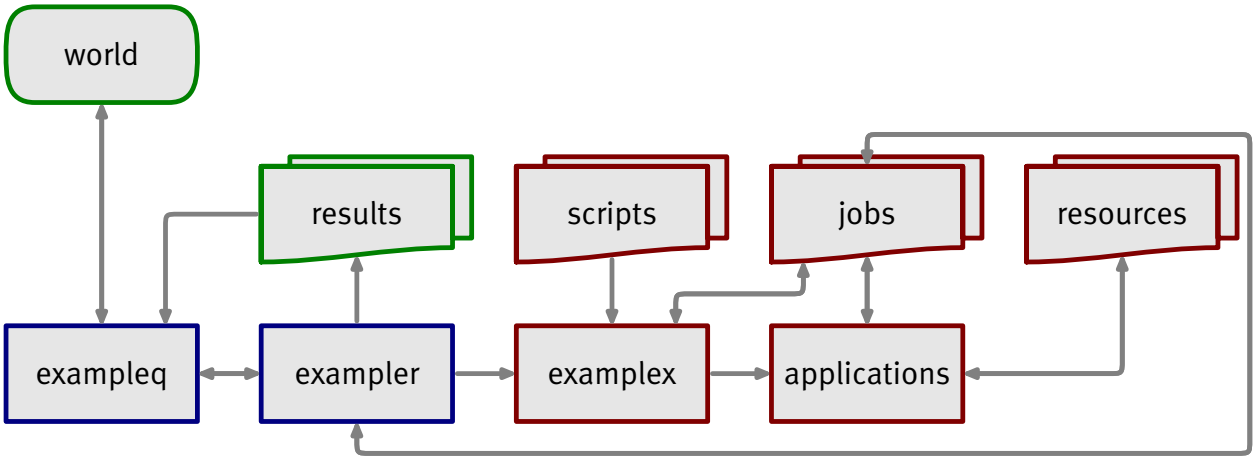

Figure 3 A rough scheme of the eXiMpLe workflow.

Before EXAMPLEQ sends the request, it may add additional information, for instance how to access the data in the watched folder. The same happens inside EXAMPLER, before it passes the request to EXAMPLEX and puts the data in the job area. When EXAMPLEX is finished, it assembles a reply describing the results. This reply as well as the results are picked up by EXAMPLER and placed in the result area. After that it is up to the application (or user) that placed the request in the watched folder to handle the results and do a proper clean up when finished.[2]

2 Framework

The communication takes place with files, encoded in XML. In the perspective of eXiMpLe we deal

[2] When EXAMPLER gets its request directly to the port, it sends the result back directly.

with a subset of the more extensive eXAMPLE specification. When consulting files, EXAMPLEQ and EXAMPLER only look into files with the suffix `exa`. Only the requests and replies are analyzed and treated as such. We use RELAX NG to describe the structure.

```
<start>
  <choice>
    <ref name="request"/>
    <ref name="reply"/>
  </choice>
</start>
```

Both applications assume that the files are valid XML, and only do a minimal validity tests. Although the specification leaves this open, for the moment an `exa:` namespace is mandate.

3 Requests

A request is defined as follows.

```
<define name="request">
  <element name="request">
    <element name="application">
      <ref name="action"/>
      <optional>
        <ref name="filename"/>
      </optional>
      <optional>
        <ref name="archive"/>
      </optional>
      <optional>
        <ref name="output"/>
      </optional>
      <optional>
        <ref name="comment"/>
      </optional>
      <optional>
        <ref name="forwarded"/>
      </optional>
    </element>
  </element>
</define>
```

All directives are encapculated in an `application` element. When the request moves through

the system, client and server blocks are added, and in other applications data blocks can be part of the game. In the EXIMPLE context, only the application block is looked at; other blocks are ignored.

An action is a reference to an EXAMPLE script to be used by EXAMPLEX. In this respect it must be a proper filename, preferable consisting of a–z, 0–9, and/or –. Spaces are a bad idea, as are special characters, and using them is not advised, if only to be portable between platforms.

```
<define name="action">
  <element name="action">
    <text/>
  </element>
</define>
```

When applicable, the action is applied to the filename provided by:

```
<define name="filename">
  <element name="filename">
    <text/>
  </element>
</define>
```

This file or an archive containing this file and other files needed, can be put in the watched folder. Such an archive is identified by:

```
<define name="archive">
  <element name="archive">
    <text/>
  </element>
</define>
```

Again, only safe filenames are advised (a–z, 0–9 and/or –) and the suffix must be known EXAMPLER, since this application will unpack the archive in the jobs folder.

The output element specifies the (prefered) name of the reply file.

```
<define name="output">
  <element name="output">
    <text/>
  </element>
</define>
```

You can add a comment to the file. This comment is discarded in the process.

```
<define name="comment">
  <element name="comment">
    <text/>
  </element>
```

```
</define>
```

The data encapsulated in `forwarded` is passed on to the reply file without further analysis.

```
<define name="forwarded">
  <element name="forwarded">
    <text/>
  </element>
</define>
```

4 Reply's

A reply has the following construct:

```
<define name="reply">
  <element name="reply">
    <ref name="status"/>
    <optional>
      <ref name="result"/>
    </optional>
    <optional>
      <ref name="comment"/>
    </optional>
    <optional>
      <ref name="forwarded"/>
    </optional>
  </element>
</define>
```

The `status` element has (for the moment) either the value `ok` or `error`. In future versions, `Error` may become more explicit, so you can best use `ok` to determine a valid run.

```
<define name="status">
  <element name="status">
    <text/>
  </element>
</define>
```

The optional `result` element specifies the name of the result file, if needed with a path. As in any path in this workflow, the path separator is a forward slash (/).

```
<define name="result">
  <element name="result">
    <text/>
```

```
  </element>
</define>
```

The system may generate a comment.

```
<define name="comment">
  <element name="comment">
    <text/>
  </element>
</define>
```

The data encapsulated in `forwarded` in the request, ends up in the reply in a similar way.

```
<define name="forwarded">
  <element name="forwarded">
    <text/>
  </element>
</define>
```

5 Example

An example of a request is the following:

```
<exa:request>
  <exa:application>
    <exa:action>zorgboek</exa:action>
    <exa:filename>pilot204.xml</exa:filename>
    <exa:comment>a test job</exa:comment>
    <exa:forwarded>whatever non-exa-thing you want</exa:forwarded>
    <exa:output>reply.exa</exa:output>
  </exa:application>
</exa:request>
```

When everything goes all right, you can expect a reply like:

```
<exa:reply>
  <exa:status>ok</exa:status>
  <exa:result>result.pdf</exa:filename>
  <exa:forwarded>whatever non-exa-thing you want</exa:forwarded>
</exa:reply>
```

If you have access to the job folder, you will notice that sometimes the request file has more information in it. This is because each step may add its information. You can find more on this in the eXAMPLE manual.

6 Installation

The name and location of the request path (watched folder) depends on the configuration files. In this example the path is set to `/example/requests`. Since EXAMPLEQ recurses into the sub directories, the following set up is valid. A subdivision permits you to prevent name clashes.

```
/example/requests/hagen
/example/requests/otten
/example/requests/vanmarle
```

These subdivisions can also be found in the reply and job paths. Whenever `/example/replies` is defined as result path, EXAMPLER will make sure that a user can find the result under his/her subpath:

```
/example/replies/hagen
/example/replies/otten
/example/replies/vanmarle
```

These paths are created automatically. You will find the reply files here, but depending on the configuration, you will find the results one level deeper; the exact name can be found in the `result` element. In this workflow, unique names are generated that conform to the following specification:

```
/example/.../username/yyyymmdd-hhmmss-ms-pid
```

The user (or controlling application) needs only access to the request and reply paths. When a request is processed, its file is renamed.

7 Colofon

Author	Hans Hagen
Version	December 14, 2002
Copyright	PRAGMA POD, Hasselt NL
Internet	www.pragma-pod.com
Email	j.hagen@xs4all.nl