

Charlie - an XML application framework

Petr Cimprich (Ginger Alliance)

1 Introduction

Charlie is an XML middleware layer providing communication between clients and data sources. It listens to requests coming from clients, evaluates the requests and provides all necessary data. Then it process the data, creates a response suitable for the client, and sends the response back to the client. It is Charlie, who takes care of processing logic and reaching real data sources. From client point of view Charlie just receives a request and returns a corresponding response.

Charlie can be employed in a variety of ways. We can use it as a simple interface manager able to present XML data in form convenient for requesting clients. This what other XML enablers are used for, but Charlie can go farther. It can also be used as a kind of application components manager or dispatcher able to communicate with XML applications or data providers and to compose responses as required. The processing logic can be separated to small entities spread over the network, which allows a flexible and modular design of what is called Charlie applications. Moreover, the applications or their components can be distributed over the network.

2 Charlie Namespace

Charlie has its own namespace with charlie:/ scheme which is used for all internal processing. Each document Charlie can return has a unique identifier in form of charlie URL. In other words, all resources are identified in terms of charlie URLs inside Charlie. Connections to the outer world are mediated by handlers and data drivers.

A handler is responsible for communication with clients. It receives requests from clients, translates request URLs to charlie URLs, and passes them to the Charlie engine for further processing. When a response comes back to a client, the handler is responsible for the opposite transformation of any URLs contained in the response, so that, links to other charlie URLs are translated to http URLs, for example. Currently, there is only http handler available, but handlers for whatever convenient protocol can be implemented.

Data drivers, on the contrary, allow Charlie to communicate with data providers and request data required during the processing. For each branch of charlie namespace there is an associated data driver, which maps internal charlie URLs to "real-world" URLs and obtains data using appropriate communication protocols. Associating data drivers is a key part of Charlie configuration. The current implementation offers 4 data drivers:

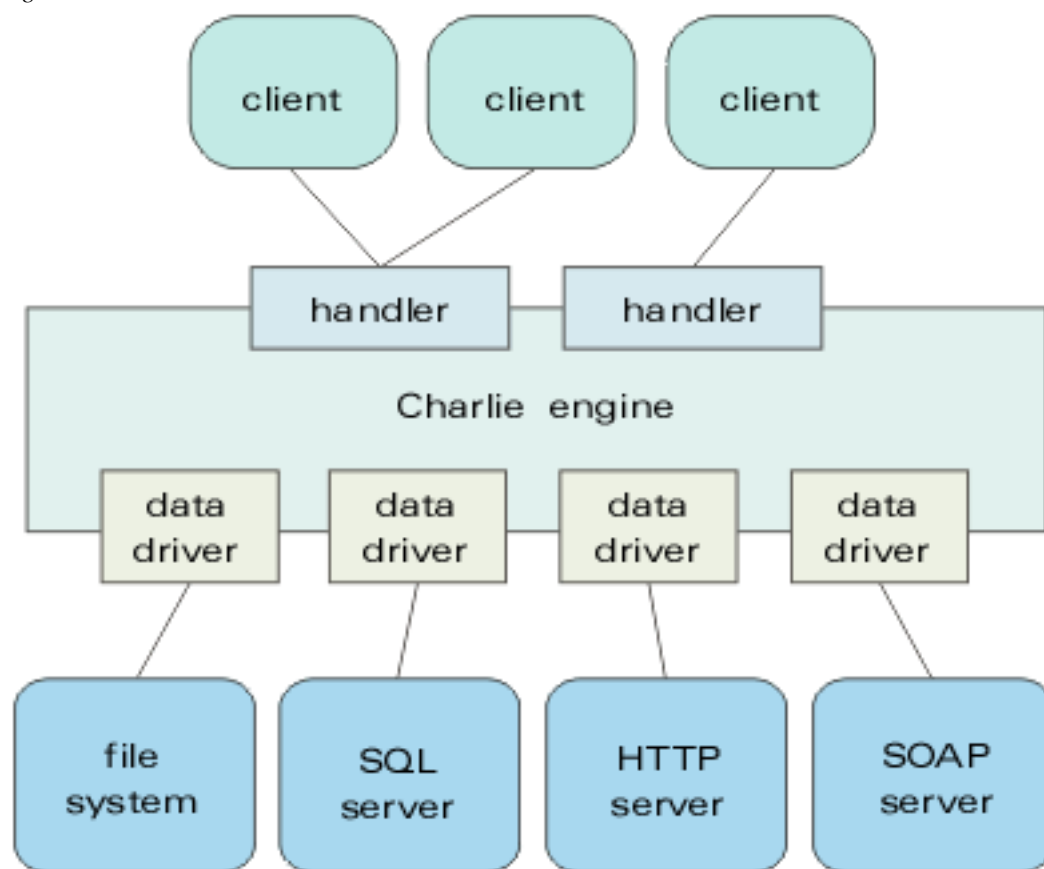
- FILE is the default data driver. It can read files from local or network file systems.
- HTTP data driver retrieves documents from http servers.
- SQL data driver transforms Charlie requests to SQL queries and returns XML documents based on results of the queries.
- SOAP data driver retrieves data from SOAP application servers.

3 Actions

Actions play a central role in request processing. An action is simply a piece of code, which is evaluated in order to provide a response to an arriving request. The request may launch the action either direct (it specifies the action URL) or indirect way (the action is associated with the request URL in configuration).

Using functions and objects provided by the Charlie engine, actions can easily retrieve data from Charlie namespace and put them together using an XSLT processor. Actions can even call other actions, which allows a modular design of atomic actions for common tasks. Currently, actions can be written in JavaScript only, but other languages can be

Figure 1.



supported in the future. Actions can range from a simple specification of template and data to be processed together to however complicated programs.

4 Configuration and Applications

Charlie has a hierarchical three-level configuration which is an inseparable part of the engine. The configuration e.g. binds data drivers to branches of Charlie namespace and provides rules associating charlie URLs with actions. The Charlie configuration consists of three types of XML files. The basic configuration file, called global configuration, specifies rules valid globally for the entire engine and defines Charlie applications and default data drivers for each application.

The term of Charlie application refers to the first level nodes of charlie namespace tree (charlie:/my_application, for example). The second level of configuration is related to applications; each application has its profile defining all data drivers used by the application and other application specific rules. Strictly speaking, using applications and application profiles is not mandatory at all, but an application is a convenient logical unit and its profile is the best place where to put application specific rules and definitions.

The last configuration file is called directory meta-data and contains information on documents in particular directory of Charlie namespace. To obtain data required to resolve a Charlie URL these three configuration files have to be read. Please note, that the only file which has be local to the Charlie engine is the global configuration (this file is the only one, which is not read using data drivers). Application profiles can be retrieved using data drivers specified in the global configuration and directory meta-data files can be obtained using drivers specified either in the global configuration or an application profile.

5 Request Processing

The best way to explain how the request processing works is probably an example. We will follow a way of request caught by an http handler. The handler listens to http requests in form of:

```
http://host.domain.org/charlie-bin/get/path
```

Let's suppose it receives the following request:

```
http://host.domain.org/charlie-bin/get/my_application/my_dir/my_document.xml
```

The handler translates this request URL to

```
charlie:/my_application/my_dir/my_document.xml
```

passes the transformed request to the Charlie engine and waits for a response. It is the engine's turn to resolve the request and provide an answer now. This procedure requires the following sequence of steps:

1. Charlie reads its global configuration to get basic information on application called my_application. The information includes a definition of default data driver to be used to retrieve data from charlie:/my_application subtree of Charlie namespace.
2. Using this data driver Charlie retrieves the application profile. It contains definitions of all data drivers used within the application, this is why the engine already knows, that data from charlie:/my_application/my_dir subtree should be read using the FILE data driver and the URL of charlie:/my_application/my_dir/my_document.xml is translated to file://usr/local/charlie-apps/my_application/my_dir/my_document.xml.
3. The last configuration file, which Charlie has to read, is the my_dir directory meta-data file. Its URL is charlie:/my_application/my_dir/DMD.xml (or file://usr/local/charlie-apps/my_application/my_dir/DMD.xml as translated by the FILE data driver).
4. Information from configuration files allows Charlie to make several decisions:
 - Request is to be processed as an action.
 - An action charlie:/my_application/actions/common_documents.act should be launched to create a response to this request.
5. Again, configuration files are used in order to determine the real location of this action (steps 1, 2 and 3 are repeated).

6. Then the action is loaded and evaluated:

```
data_req = new Request("GET", charlieURI());
data = chGetResponse(data_req);
```

A new request object is created using the URL of original incoming request (charlie:/my_application/my_dir/my_document.xml). The second line ask Charlie to get a response to this request. This again requires to read configuration and make decisions as described above. When called from actions, requests are usually processed directly; this is why the request doesn't point to another action. Instead, the my_document.xml file is looked up using the right data driver (well, this is the FILE data driver and the file is found as file://usr/local/charlie-applications/my_application/my_dir/my_document.xml, see the step 2) and its content is returned. More exactly, its content is stored to the content property of response object returned by the chGetResponse function.

```
templ_req = new Request("GET", "../templates/common_documents.xsl");
templ = chGetResponse(data_req);
glob_req = new Request("GET", "../global/common_documents.xml");
glob = chGetResponse(data_req);
```

Other two documents are retrieved. The second one is an XSLT stylesheet used to transform the first document and is obtained exactly the same way. The next document contains XML fragments inserted to all processed files and could be provided by some other, let's say, the HTTP driver. In this case, the internal URL charlie:/my_application/global/common_documents.xml might refer to http://host2.domain2.org/includes/my_application/common_documents.xml, for example.

```
sab = new Sablotron();
sab.setTemplate(templ);
sab.setDocument(data);
sab.addArg("arg:/global", glob.content());
```

Then there is a new instance of XSLT processor created and fed with data.

```
ret = sab.process();
```

The data and the template are put together. The third argument is used during the transformation. Its content is accessed as document('arg:/global') from the XSLT stylesheet. The result of transformation is returned as string.

```
response.fromSablotron(sab);
```

A response object is created with content type and content returned from Sablotron. This object is passed back to handler as the actions is completed.

7. The handler performs any "charlie-real world" translations if necessary and sends the response back to client.

The example illustrates relations between entities carrying application logic within Charlie application: configuration files, actions and XSLT transformations. This arrangement allows to keep all components simple and maintainable. There are two traditional ways how to create an XML/XSLT application. You can either employ any dedicated system, which usually doesn't allow much more than to couple data and style-sheets, or write down programs of your own for all requests, which brings obvious difficulties with scalability and maintainability. Charlie offers a solution joining better sides of the both approaches. Actions can achieve all aims of custom programs, without being unnecessary complicated for common tasks.

6 Getting Data from SQL Server

Important data aren't usually stored in text files but in database systems. This is why not even Charlie application can do without an access to SQL servers. Charlie has the SQL data driver for these purposes. Requests interpreted by the SQL driver are somewhat special. Charlie URL identifies an SQL server and a database only, while an SQL command is found in the request body. The following fragment of action shows how does it work.

```
db_data = new sqlDescriptor();
db_data.sql.push('select position.name,position.description,tasks.task');
db_data.sql.push('from position,tasks where');
db_data.sql.push('position.id_position=77');
db_data.sql.push('and position.id_position=tasks.id_position');
```

```
db_data.sql.push('order by tasks.id_task');
//db_data.addGroup('position', ['name', 'description'], ['name', 'annotation']);
xmldef = db_data.toXML();
```

A helper object sqlDescriptor is used to prepare our request. The last but one line is commented out for now, we will uncomment it later. The xmldef variable is filled with a small XML document:

```
<?xml version='1.0' encoding='iso-8859-2'?>
  <charlie-sql>
    <sql>select position.name,position.description,tasks.task
from position,tasks where
position.id_position=77 and position.id_state=2
and position.id_position=tasks.id_position
order by tasks.id_task
    </sql>
    <output method='xml' indent='yes' encoding='iso-8859-2'></output>
  </charlie-sql>
```

The action continues:

```
req = new Request("GET", 'charlie:/expert2/sql/');
req.content(xmldef);
position_data = chGetResponse(req);
```

A new request object is created using Charlie URL associated with a SQL server in configuration. The XML definition created above is inserted to the request body. Then the request is evaluated and the response is stored in a variable. The position_data variable now contains the following string:

```
<?xml version="1.0" encoding="iso-8859-2"?>
  <data>
    <result>
      <fields num="3">
        <field id="0" name="name" />
        <field id="1" name="description" />
        <field id="2" name="task" />
      </fields>
      <row>
        <name>Driver</name>
        <description>Drives vehicles.</description>
        <task>Driving cabs</task>
      </row>
      <row>
        <name>Driver</name>
        <description>Drives vehicles.</description>
        <task>Driving coaches</task>
      </row>
      <row>
        <name>Driver</name>
        <description>Drives vehicles.</description>
        <task>Driving bikes</task>
      </row>
    </result>
  </data>
```

Charlie SQL data driver also can group data so that the resulting document is closer to natural tree structure than this table based document. If uncomment the line calling `sql.addGroup` method now, we get somewhat different result:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<data>
  <result>
    <fields num="3">
      <field id="0" name="name" />
      <field id="1" name="description" />
      <field id="2" name="task" />
    </fields>
    <position level="0">
      <name>Driver</name>
      <annotation>Driver is can drive vehicles.</annotation>
      <row>
        <task>Driving cabs</task>
      </row>
      <row>
        <task>Driving coaches</task>
      </row>
      <row>
        <task>Driving bikes</task>
      </row>
    </position>
  </result>
</data>
```

The SQL data driver also allows to rename fields, which makes it possible to get results with the same DTD from different tables. Once having data from database in form of XML document, it can be passed to Sablotron and processed the same way as any other XML data.

7 Links

Resolution of request can involve even more steps than the presented request processing example. The configuration allows to define links for each Charlie namespace document. In this case a stack of Charlie URLs is created for an incoming request and the engine looks for a document using data drivers appropriate for individual URLs from the stack until the document is found. Obtained data then can be stored to locations where selected links point to.

```
<file name="my_doc.xml" link-index="1" primary="yes" store="yes">
charlie:/local_dir/my_doc.xml
  <link primary="no" index="2" @store="no">charlie:/remote_web/doc08.xml</link>
</file>
```

Due to this simple link definition in a directory meta-data file, the engine tries to find the `my_doc.xml` document on the local storage. If not successful, it retrieves the document from a remote http server and stores it on the local disk. The next time this document is wanted, it is found within the local file system, unless its validity expires in the meantime.

In terms of Charlie namespace, links introduce a kind of mirroring between different branches of the namespace. The capability to define links allows an easy implementation of useful scenarios, such as a locally cached remote application or a semi off-line application. Entire applications can be distributed or updated from remote servers.

8 Conclusion

Charlie has been designated as XML middleware in this article, but it doesn't say much on its execution context. Actually, it is designed to run in different contexts from the beginning. Charlie can run either on server (in the same

manner as common server-based XML application frameworks), within an independent proxy-like layer (which might be expected from what is called middleware), or even on client. Since Charlie applications are as transparent as possible, they should be portable from one context to another without modifications. The above mentioned capability of caching seems to be particularly interesting in the client or LAN-proxy context.

Charlie represents rather an environment for distribution of applications than an XML publisher or application server. However, it can be successfully employed as a simple middleware layer between your data sources and xml/html clients. Current Charlie-based applications (www.portalek.com, www.istp.cz) employs Charlie on server side, however the significance of Charlie enabled clients is supposed to grow up in the near future.

Charlie is still in an early stage of its life. The current version offers an engine providing all above described features, an http handler and basic data drivers. The engine is implemented as a set of perl packages using more or less independent components such as JavaScript interpreter, expat (an XML parser) or Sablotron (an XSLT processor). The Charlie project is an Open Source; for more details see www.gingerall.com.