Catalyst EAI Gateway

Release 15.4.0.0

Aug 12, 2022

Contents

CHAPTER 1

Introduction

The *CONTACT Catalyst EAI Gateway* provides the integration of CONTACT Elements applications with ERP and other EAI systems in the company and thus forms the basis for a comprehensive orchestration of IT systems and processes using modern communication technologies such as web services.

The *CONTACT Catalyst EAI Gateway* is an integration service primarily used for coupling CONTACT Elements with ERP systems. The integration of PDM/PLM and ERP systems can be broken down into the synchronization of data and the control of processes.

The data synchronization is frequently used to transfer design data like part master records, design drawings and BOMs that are to be passed on to Production after reaching certain maturity levels. Simple examples of the integration of cross-system processes are the ordering of purchased items triggered by a status change or process planning (PP) for production planning triggered by a BOM update.

For these two tasks, *CONTACT Catalyst EAI Gateway* uses generic XML and JSON formats as well as standard protocols such as HTTP. Through flexible configuration and the provision of extension frameworks, transformation into specific data formats and the use of web services is possible. On the other hand, file-based data exchange with ASCII or CSV formats or the use of SQL adapters can also be realized.

Furthermore, *CONTACT Catalyst EAI Gateway* can be used as a tool for the one-time or regular export or import of data from/to CONTACT Elements.

This manual describes the administration, configuration and programming of *CONTACT Catalyst EAI Gateway*.

The *CONTACT Catalyst EAI Gateway* is implemented in CONTACT Elements as a service, often referred to below by the abbreviation XGW.

CHAPTER 2

Operation and Administration

2.1 Technical environment

The *CONTACT Catalyst EAI Gateway* integrates as a service into the CONTACT Elements service console. In addition to the automatically created service configuration, further instances of the service can be set up so that the integration of different systems can also be done separately from each other in terms of configuration. The configuration of further instances of the service is described in *Configuration of additional services* (page 4). In addition, if separate test, development, and production systems are used, separate synchronization services should be set up accordingly.

Depending on the communication protocol used between CONTACT Elements and the system to be integrated, additional requirements must be provided for. These can include adjusting firewalls for access to HTTP ports, sharing network drives, and setting up VPN tunnels.

Basically, the *CONTACT Catalyst EAI Gateway* is tailored to the realization of a loose coupling in order to increase fail-safety and fault tolerance in communication, to prevent data loss, and to allow the integrated systems to function largely independently of each other. If the connected system can provide corresponding error statuses and messages via its interfaces in the event of an error, this supports the *CONTACT Catalyst EAI Gateway* in tracking the synchronization state.

2.2 Installation

The *CONTACT Catalyst EAI Gateway* is implemented as an CONTACT Elements application with the name *cs.xml*. It is installed according to the administration manual. After installation and subsequent restart of the Service-Daemon, a new entry is available in the service console with the name *XMLGateway Service* (in the service configuration under `cs.xml.xmlgateway.XMLGatewayService`). If multiple instances of the gateway are to be set up, appropriate additional entries must be created here.

In addition, a configuration directory `etc/xmlgateway` is created in the instance directory with installation. A sample configuration file `xmlgateway.conf` is placed in this directory, which can be used as a starting point for a configuration.

The `mkconfig` tool provides sample configurations which can be installed and used as templates if required (see manual *Configuration with mkconfig*).

Additionally, a configuration file `etc/xgw.conf` is created in the instance directory. This file can be used like other CONTACT Elements configuration files, for example, to control the use of log files by setting the variable

CADDOK_DEBUG.

These directories and files will not be overwritten when updating the *cs.xml* application. This means that these files can be renamed and changed without losing the changes by overwriting when updating the application. But it also means that the release notes should be read when updating to integrate potential additions into the configuration files manually.

2.2.1 Configuration of the service

The operation of the *CONTACT Catalyst EAI Gateway* is controlled via the service console of the Service-Daemon. The operation of the Service-Daemon is explained in the CONTACT Elements manual *Installation and Operation* in chapter *Services/Service-Daemon*.

The following options can be configured in the service configuration:

–user The CONTACT Elements user, for which the service worker will be executed.

–language The language used for the CONTACT Elements user to be logged in. This may affect the language of feedback and error messages issued by *CONTACT Catalyst EAI Gateway*.

–configfile If this value is not set, the file `$CADDOK_BASE/etc/xmlgateway/xmlgateway.conf` is read from the *CONTACT Catalyst EAI Gateway*. This option can be used to refer to a different configuration file instead. A path specified here must be relative to `$CADDOK_BASE/etc`. So the default would be `xmlgateway/xmlgateway.conf`.

–watchdog_intervall If web services are configured with *CONTACT Catalyst EAI Gateway*, then these are typically long-running service processes waiting in the background for incoming data at their endpoints. If such a service terminates due to error conditions, the infrastructure ensures that it is restarted. In addition, each worker process started by the service must take care to detect and respond to a database connection that has been lost in the meantime. This option specifies a value in seconds for an interval at which the state of the database connection is checked. If the connection is lost, then the service terminates its worker processes. They are then restarted by the infrastructure as soon as the connection can be re-established.

The *CONTACT Catalyst EAI Gateway* is started and stopped manually via the CONTACT Elements service console after the configuration is complete, or started automatically by the *Autostart* option.

2.2.1.1 Logging

If configuration errors are recognized when starting, corresponding messages are output into the CONTACT Elements logging and the service is then ended. In such a case, logging is switched on using suitable values of the *CADDOK_DEBUG* variable, for example in `etc/xgw.conf` or in `etc/site.conf` to detect and correct the configuration errors.

In addition, logging at the level of the individual workers can be activated in the file `etc/xgw.conf`. For this purpose the variables *EAI_LOG_FILE* and *XML_LOG_FILE* can be defined. These variables refer to logging of the module *cs.erp* (responsible for general attribute mapping etc) and the module *cs.xml* described here. Typical assignments are:

```
XML_LOG_FILE = CADDOK_TMPDIR + "/XMLLOG_%(system)s.log"
EAI_LOG_FILE = CADDOK_TMPDIR + "/EAILOG_%(system)s.log"
```

Here, the name of the EAI configuration used (see manual *CONTACT Catalyst for ERP*) is included so that the log files can be recognized accordingly when using multiple integrations. In addition, another log file with attached job group name is created for each started worker process.

The two variables *EAI_LOG_LEVEL* and *XML_LOG_LEVEL* can be used to define the corresponding level of detail of the logging by symbolic names (see template of `xgw.conf`). Typical values are *INFO*, *WARNING* and *ERROR*.

2.2.2 Configuration of additional services

It is often necessary to use several instances of the gateway in parallel. This is the case, for example, when several different ERP systems are to be connected.

At the service level, the *CONTACT Catalyst EAI Gateway* consists of two components: The Service-Daemon service and worker processes (of which there are usually multiple instances). While the workers are responsible for executing the individual jobs of the gateway, it is the task of the service to coordinate the execution of the worker processes.

To create your own services based on the *CONTACT Catalyst EAI Gateway*, it is necessary to create your own service implementation and your own worker implementation. The new service must then be registered in the instance's service table.

In the following paragraphs it is assumed that an additional instance of the XML Gateway Service should be configured using the module `kunde.myxgw`. The class `kunde.myxgw.MyXGWService` will contain the service's implementation, while module `kunde.myxgw.myxgw_worker` will contain the worker implementation.

2.2.2.1 Creating a worker implementation

A *CONTACT Catalyst EAI Gateway* worker is an executable module which receives a job list as an argument, which it then executes. The standard worker implementation of the *CONTACT Catalyst EAI Gateway* is located in the `cs.xml.xgw_server` module; the `service` function defined here implements the worker. Now, to create your own implementation, the `customer.myxgw.myxgw_worker` module is created first. The content of the worker module consists of just a few lines:

```
from cs.xml import xgw_server

if __name__ == "__main__":
    xgw_server.service(name="MyXMLGateway", script="myxgw.conf", log_prefix="CGW")
```

Here a standard worker is started with its own name and its own configuration script (`etc/myxgw.conf`). Based on the name used here, it is also possible to distinguish between the configuration settings of the services in the `etc/myxgw.conf` file. The name can be queried in the variable *CDB_XGW_SERVICE*. The name of the default service is *XML Gateway*. Thus, for example, the log settings can be set differently for each service:

```
if CDB_XGW_SERVICE == "MyXMLGateway":
    CADDOK_DEBUG = "ALL.ANY:ts:log:lev=8"
elif CDB_XGW_SERVICE == "XML Gateway":
    CADDOK_DEBUG = "XGW.MSG:ts:log:lev=6"
```

The `log_prefix` argument can be used to define a prefix other than the default `XGW` for logging.

2.2.2.2 Creating a Uberserver service

The service class in this example has the fully qualified Python name `kunde.myxgw.MyXGWService`. The module containing the class is defined in the file `site-packages/kunde/myxgw/__init__.py`.

```
from cs.xml import xmlgateway

class MyXGWService(xmlgateway.XMLGatewayService):
  """Custom service implementation"""

  def __init__(self, site):
    super(MyXGWService, self).__init__(site, "MyXMLGateway",
                                       main_module="kunde.myxgw.myxgw_worker
↪")
```

The simplest implementation is to simply create a subclass of XMLGatewayService, which provides the name of the previously defined module.

It is also possible to define a version string, which will be displayed in the service console. This is achieved by overwriting the property `version_string`:

```
# ...

class MyXGWService(xmlgateway.XMLGatewayService):
  """Custom service implementation"""

  # ...

  @property
  def version_string(self):
    return 'my_version_string'
```

If the service should automatically install an entry in the service table on package installation, this can be achieved by defining and customizing the following class method:

```
class MyXGWService(xmlgateway.XMLGatewayService):
  """Custom service implementation"""

  # ...

  @classmethod
  def install(cls, svcname, host, site, password="", *args, **kwargs):
    """Install basic default configuration for this service"""
    from cdb.platform.uberserver import Services
    if not svcname:
      svcname = cls.fqpyname()
    svcs = Services.get_services(svcname, None)
    svc = None
    if svcs:
      misc.cdblogv(misc.kLogMsg, 1,
                   "Service %s is already installed (%s) " %
                   (svcname, svcs))
    else:
      super(xmlgateway.XMLGatewayService, cls).install(svcname, host, site, *args,␣
↪**kwargs)
      svc = cls._create_basic_configuration(
        svcname,
        host,
        site,
        arguments="",
        options={'--user': 'caddok',
                 '--language': 'en',
                 '--watchdog_interval': 600,
                 '--configfile': 'myxmlgateway/myxmlgateway.conf'},
        autostart=False)
    return svc
```

2.2.2.3 Service Configuration

To create a service entry for the newly created service, the easiest way is to copy the default service entry and modify it.

For the new entry, the fully qualified Python name of the service class should be specified under *service name*; this is `customer.myxgw.MyXGWService` in this example. This service should then be given its own configuration file for job configuration via the *–configfile* option, probably in its own directory where the schema and other files are also stored. For this purpose, a directory can be created parallel to the configuration directory `etc/xmlgateway`; for example `etc/myxmlgateway`, in which it is then stored.

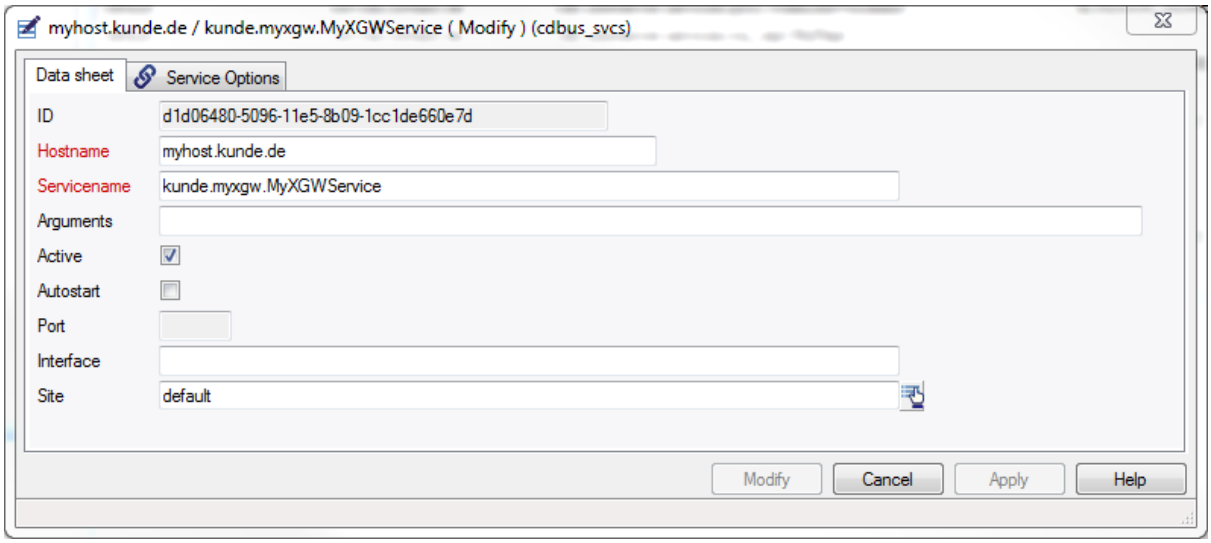

Fig. 1: The customized entry in the service table

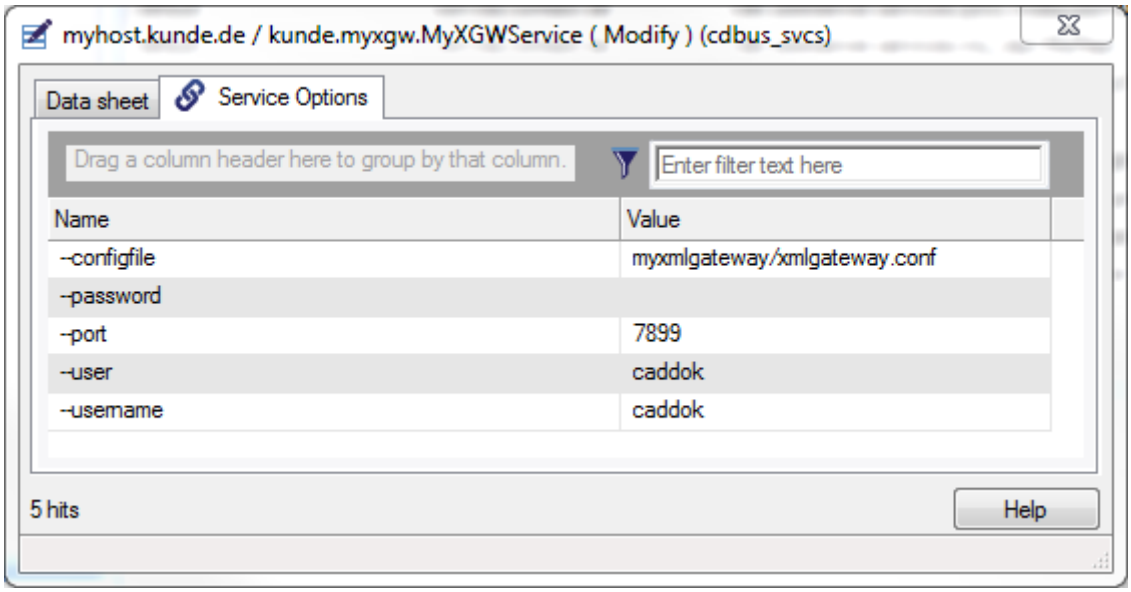

Fig. 2: The customized service options

2.3 Configuration

The configuration of the *CONTACT Catalyst EAI Gateway* is controlled by a central file. This file is specified in the respective service configuration under the *–configfile* option (see *Configuration of the service* (page 3)).

The configuration file defines a series of tasks implementing the interface functions. Tasks are, for example, the transfer of data to be synchronized with the ERP system, the receipt of messages from the ERP system, but also the execution of custom functions or the provision of a web service interface.

Individual tasks are defined in *Job configurations* and can be summarized in job groups. Each job group defines the order in which their jobs are executed and the execution time. Job groups can be executed in regular, fixed intervals or triggered by external events.

2.3.1 Job definition

Each job definition consists of a set of job-specific settings and attributes. These include the definition of various working directories and other configuration files.

The `xmlgateway.conf` configuration file consists of a series of sections where individual job groups and the respective associated jobs are defined. Within these sections, the definition of a job group consists of a general part—where global settings for all of the jobs in the group can be specified—and the special settings for each job.

An example:

```
XGW_JOBS = [
    { # job list executed in 10 min intervals
        'job_interval': 600,
        'group_id': 'regular_jobs',
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'input_dir': None,
        'output_dir': None,
        'processed_dir': '%(CADDOK_BASE)s/etc/xmlgateway/processed',
        'options': {},
        'job_list': [
            { # Einlesen part, bom als Datei
                'custom_xml_tags': 'xgw.SomeTags',
                'input_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
                'schema_files': [ 'site.xgs', 'transformations.xgs' ],
                'input_files': 'parts.xml',
            },
            # weitere Jobs dieser Gruppe..
        ]
    },
    # weitere Jobgruppen..
]
```

A job group can have a series of global attributes:

group_id an identifier ID of the job group by which it is identified. The ID must consist of an identifier that is unique across all job groups and does not contain any spaces. This ID is also used to identify the job group in the log file.

active Setting the attribute to *False* lets you disable the run of a job group. The default value used is *True*.

job_interval An interval (in seconds) that specifies regular execution of the job group. If the interval of a job group is set to 0, then it will not be triggered at regular intervals, but event-driven by the arrival of a message at a web service port (see *port*).

The attribute *job_at* (see below) should not be used if the attribute should be triggered in intervals.

port If a web service job group is defined (see above, interval=0), then each job in this group can be addressed via the URL of its endpoint, which is composed of the group's base URL and job-specific path portion. These paths are grouped under the job group's common port, which is defined using this attribute. A

worker process is provided for each web service job group defined in this way. If multiple web service job groups are configured, this can be used to define worker processes for load balancing and parallelization (see configuration example in chapter *Load balancing with multiple web service workers* (page 83)).

If only a single web service job group is defined, then the port can alternatively be defined via the service configuration.

secret If a web service job group is defined (see above, interval=0), then the endpoint set up with it can be secured via Basic Authentication. The username and password to be used can be stored in a wallet (see manual `Platform CONTACT Elements: Installation and Operation` in chapter `Management of Keys and Secrets`). When creating such a secret, the username and password to be used should be stored as colon-separated text (e.g. `username:password`). The path of the secret used can then be stored in the job group configuration under `secret`. The path should be stored in `cs.platform/customer` (as recommended in the mentioned manual) or in your own toplevel wallet. An example for a web service job group named `input-service` would be the corresponding path `cs.platform/customer/<customer>/endpoints/input-service`.

resource This option allows to include own implementations of `twisted.web.resource` resources. This allows e.g. the inclusion of own authentication methods or protocols. See section *Advanced Configuration* (page 47) for an example of how to include SOAP.

job_at If a job is to be executed at a periodic date, the attribute *job_at* can be defined instead of *job_interval*. Here, a string must be specified which corresponds to the cron syntax. The specification of such strings is described in section *Specification of the job_at attribute* (page 12).

conf_dir specifies which objects are to be transferred from the CONTACT Elements instance and which schema definition is to be used to construct the message to be transferred (see *Data transmission* (page 29)).

input_dir For jobs that expect the incoming messages in the form of files, the directory for the received files can be defined here.

output_dir For jobs that store outgoing messages as files, the output directory can be defined here.

work_dir Various jobs require the definition of a working directory (see below).

processed_dir If processed files are to be archived, the archiving directory can be specified here.

job_list A python list, in which the jobs of the group are defined. Execution of a job group meaning executing the jobs in this list in the order in which they are listed.

options A Python dictionary with options for custom jobs can be defined using this (see below)

***_dir** Additional directories can be defined for use in the job configurations; each of their names has to end in *_dir*.

These attributes can be overwritten in individual jobs (where appropriate). Furthermore, other attributes are possible in job configurations:

job_id similar to the above *group_id*, a name of the job by which the job is identified in the log file.

callable Name of a python callable. By including this attribute a custom job is defined. This job executes the callable; the job configuration is passed as a parameter. The return value of the callable controls the further processing of the job group. The following values are valid:

None signals successfull execution of the python callable and finishing of the job.

cs.xml.JobHandler.cancelJob Indicates the cancellation of this job. An exception or a program error in the callable results in this value.

cs.xml.JobHandler.cancelJobGroup This cancels running of all other jobs in the job group.

cs.xml.JobHandler.continueJob signals that the remaining execution steps are to be executed.

Finally, a message in form of an XML string can be returned, which is then used as input for further processing (see *Processing jobs* (page 11)). This leads to the same result as the returning of *cs.xml.JobHandler.continueJob* with the additional possibility to return a followup message.

If a list of jobs grouped in a job group is executed as web service (see *Web services* (page 43)), each of the separate jobs can use the function call *job.set_response("<content>")* to define a result value for the

caller of the web service. Here only the last call of this function is relevant for the actual return value. For this only jobs are considered that don't finish with a return value of *cs.xml.JobHandler.cancelJob* or *cs.xml.JobHandler.cancelJobGroup*. So if a response content already set by *job.set_response("<content>")* should not be used, the job can still be finished with a return value of *cs.xml.JobHandler.cancelJob*.

schema_files A list of configuration files, which determine the functions that the job executes. These must be located in the directory specified by attribute *conf_dir*.

input_files If the messages to be processed are passed in the form of files, the file names can be specified here as a list. The system searches for the files in the *input_dir* directory. A wildcard (*) can be used in the file names.

parameter_dict like options, defines further job parameters as a python dictionary.

custom_xml_tags Standard jobs define predefined XML messages. This attribute allows to define custom tags and implement custom messages. The attributes value should be the module which contains the implementation of the tag handler.

Section *Processing your own XML tags* (page 61) describes the definition of customer specific tags.

custom_handler This option allows to specify extension handlers for processing xml messages. While the attribute *custom_xml_tags* allows to specify new tags, this allows to extend the functionality of existing tags (e.g. `schema`, `config`). The expected value of this attribute is a python dictionary, which assigns a tag handler implementation to a tag name. This is detailed in section *Expanding existing XML structures* (page 63).

static_proc In addition to the two previously mentioned configuration options (*custom_xml_tags* and *custom_handler*), a static processor handler can be defined. This is used as a fallback if no handler could be determined for the top-level tag. In particular, this configuration option can be used if the input is not XML at all. The Python class given here, specified as FQPN, must follow the interface described in *Processing your own XML tags* (page 61) and define a function `process(self, node, options)`. The input is then passed in the `node` parameter for evaluation.

path If jobs are defined that are triggered by addressing web service endpoints, then this attribute can be used to define the path portion of the URL.

The paths of each job group are grouped under a common port, accordingly each job in a group must be assigned a unique path.

If a job should be triggered event-based, neither the attribute *job_interval* nor *job_at* should be defined.

2.3.2 Overview

The behavior of the integration module is controlled by a combination of XML messages and schema definitions. An overview of the interaction is provided below.

Basically, functions of the *CONTACT Catalyst EAI Gateway* are controlled by processing XML messages. XML messages can be data records in the colloquial sense of messages that have been sent to the *CONTACT Catalyst EAI Gateway* from an external system. But they can also be contents of schema files whose reading is triggered by the processing of a job and whose contents specify further configuration details or trigger functions.

To define the terms used in the following, here is a very brief overview of the XML components used. XML messages form a tree, via whose nodes the message content is structured and provided with semantics (tagged). The content of each node can also be attributed more specifically. Each node can contain content text and further nodes. Namespaces can be used to avoid tag name collisions.

Here is an example of an XML message with configuration data:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<config xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <destination class="cs.xml.Destinations.XMLDestination" />
  <destination class="cs.xml.Destinations.FileDestination" />
</config>
```

In the context of *CONTACT Catalyst EAI Gateway*, the root node of this tree (the top-level tag) is interpreted as the message type that defines its function.

Processing a message with this content will trigger the *config* function of the same name in the *CONTACT Catalyst EAI Gateway*. Each function is implemented by a plugin in the form of a Python class that can process the message content as desired.

A typical example is the query of data to be synchronized. It is triggered by receiving a message that passes a schema as a parameter, the contents of which describe what is to be queried. Accordingly, the message type is named *schema* in the top-level tag.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
    <object type="part" datasource="cs.xml.SharedObject.Factory" maxrecords="10">
        <field name="ART_NO"/>
        <field name="ART_VERS"/>
        <field name="DESCRIPTION"/>
        <field name="STATUS"/>
    </object>
</schema>
```

Processing a message with this content triggers the *schema* function. Here it defines that objects of the class *part* are to be transferred (*type*). The source from which this data is to be derived is specified by the parameter *datasource*. The optional parameter *maxrecords* specifies that a maximum of 10 records are to be transferred per message. Finally, a listing of the attributes to be transferred for each record follows.

The execution of this function results in the records released for ERP transfer being written to a resulting XML message, each record containing the specified fields. This message is finally given as a response to the schema message received above.

Such an outbound message may look like this (note the different namespaces: the *schema* declaration shown above is in the namespace *http://xml.contact.de/schema/xgwschema/1.0*, while the outbound *schema* message shown below is in the namespace *http://xml.contact.de/schema/xmlgateway/1.0*):

```
<?xml version='1.0' encoding='iso-8859-1'?>
<schema xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object type="part" id="706172[...]6465783d">
    <field name="ART_NO">000011</field>
    <field name="ART_VERS">A</field>
    <field name="DESCRIPTION">Schraube</field>
    <field name="STATUS">200</field>
  </object>
</schema>
```

An ERP system can now process this message and react to it, either by confirming the transfer of the data record or by reporting an error. It does this again using XML messages, which this time trigger a *response* function:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<response xmlns:xi="http://www.w3.org/2001/XInclude"
          xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object type="part" id="706172[...]6465783d" action="commit"/>
</response>
```

The transfer is confirmed using the *action="commit"* attribute.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<response xmlns:xi="http://www.w3.org/2001/XInclude"
          xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object type="part" id="706172[...]6465783d" action="error">
    <error>Error creating Material.</error>
```

(continues on next page)

(continued from previous page)

```
  </object>
</response>
```

An error message can also be passed when providing error feedback.

In this way, an asynchronous message-based communication is implemented.

The *CONTACT Catalyst EAI Gateway* functions are described in chapter *Message processing* (page 13).

2.3.3 Processing jobs

In order to process the messages described above, job configurations are created that define the conditions of the processing. Here, a job usually corresponds to a task, such as the transfer of articles, or the receipt of data from the third-party system.

The triggering of a job can be interval-controlled (as described above), in the style of a cron job at a certain repetitive time, or triggered by calling a web service. In the second case, a string must be specified which conforms to the cron syntax. The specification of such strings is described in section *Specification of the job_at attribute* (page 12). When invoked by a web service, a message can be provided when calling the corresponding web service port. For web service configuration, see *Web services* (page 43).

The processing of jobs is generally controlled by the order in which they are declared within a job group. Thus, when a job group is triggered by time or interval, an associated worker process starts, in the context of which the jobs are executed sequentially in the defined order. Individual jobs can cancel the processing of the entire group. A typical procedure for connecting legacy systems that provide data within database views can be achieved by an appropriate combination of two jobs. The first job executes a 'SELECT' in a custom function and writes the data records fetched to the working directory in the form of an XML file. If no records are found, the job instead can abort further processing. The second job is then a standard job reading in the XML files of the working directory and processing them further. Simillar systematics can be implemented by having the first job fetch transfer files via *SSH* or *FTP* from the providing server.

The exception to this rule are the jobs of a web service job group triggered by addressing endpoints. Here, one job addressed by its own endpoint is executed at a time, regardless of other jobs defined in the job group.

Once all jobs of an interval-controlled job group are executed, the runtime of the next interval begins, at the end of which the job group is executed again.

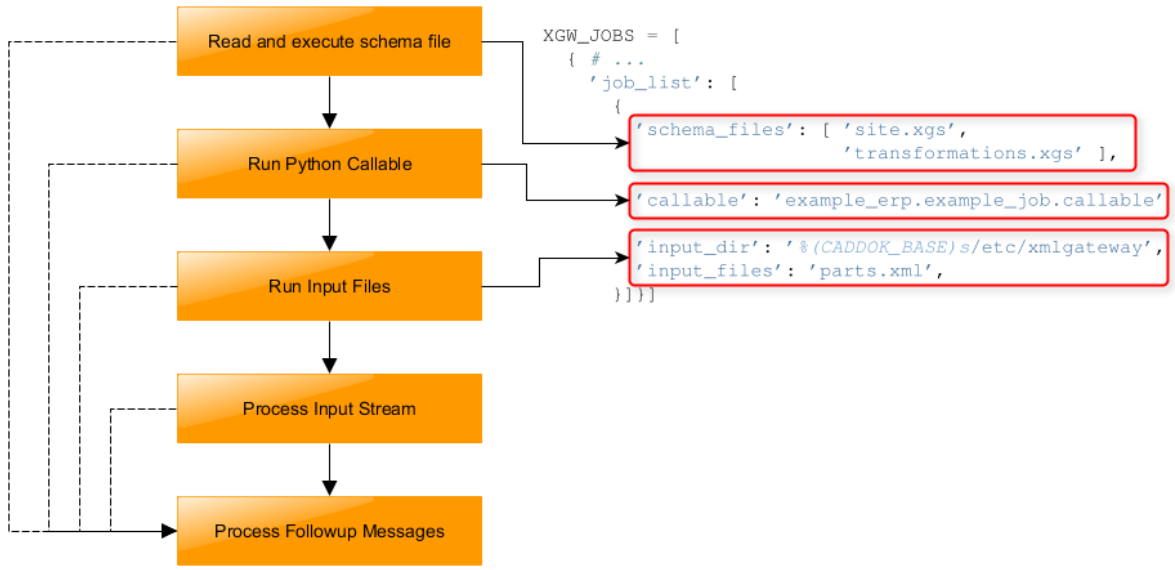

Fig. 3: The steps of job execution

Figure *The steps of job execution* (page 11) shows the individual steps in the execution of a job. Each of these steps is optional, so that, for example, either an input file or a message input via a web service endpoint can be specified. These steps are the following:

- A list of configuration files in the xml format presented above can be specified in the *schema_files* attribute. This list is initially processed by reading in all of the files in the specified order and carrying out the corresponding functions. Transformation steps for reading in other messages and transfer destinations for resulting messages can also be defined in this process. This makes it possible to convert both messages to be read in in foreign formats and messages to be read out in target formats.

  The functions that can be used in a configuration file are described in section *Message processing* (page 13).

- If a *callable* attribute is defined, the Python function specified there will now be executed. It can optionally return a message (for example generated from a record read from an external database). Such a message is saved for further processing. In addition, the job or job group can be aborted programmatically, e.g. if the prerequisites for further processing steps are not currently met.

  An example for the use of *callables* in a synchronous job triggered by web service call is shown in section *Complex Jobs* (page 45).

- If a series of files was specified with the attribute *input_files* by a list or a wildcard, these files are now read in. For each file the possibly defined transformation steps are executed. At the end of the transformation there should be a message whose type can be determined as described above. Then the corresponding function is started (e.g. *response* or *schema*). The files are deleted after successful processing to ensure that the messages are not processed more than once. The deletion of these files can also be seen as a synchronization step to the ERP system. The ERP then recognizes by the deletion that these files have been processed and new messages can be sent for processing.

- The processing of each message may result in a direct response message (for example, if a web service that provides a synchronous response has been defined as the transfer destination). Such messages are also stored for subsequent processing.

- If the job itself was triggered by calling a web service endpoint and a message was passed to it via HTTP protocol, it will now be processed analogously to the messages specified by *input_files*.

- Finally, all of the messages saved by the *callable* or as responses to the other steps are processed.

Since each of the steps is optional, configuration can be used to implement both simple jobs as well as complex workflows. The configurations for implementing more frequent applications are presented in the *Use cases* (page 28) chapter.

2.3.4 Specification of the *job_at* attribute

In addition to interval-controlled jobs and jobs triggered by incoming messages, the *CONTACT Catalyst EAI Gateway* also supports the execution of jobs on specific one-time or regularly recurring calendar dates. To achieve this, the *job_at* attribute must be defined within a job list or job instead of the *job_interval* attribute.

This attribute allows the specification of a date in a cron-job like syntax. Possible expressions are:

- Each sunday at 10:00 am
- On January, the 1st, 2016 at 1:00 pm
- Every half of an hour, monday through friday

A date is specified by six whitespace-separated field, which specify the minute, the hour, the day in the month, the month, the weekday, and the year (optionally). The values for these fields are either single values, wildcards and ranges (continuous or discontinuous). Table *The fields of the cron-job date format* (page 13) shows the possible values and special expressions which are valid for each field.

A wilcard can be used to specify a job that is executed for each value of the field. Ranges of values can be specified using `-`. A listing of values can be given using `,` and types of discontinuous ranges (e.g. each odd day in the month) can be achieved using `/`.

- To run a job on each day at 10 am, the expression `00 10 * * *` can be used.

- To run a job each even hour, the expression `00 */2 * * *` can be used.
- If the job is to be run only sundays on 10 am, the expression `00 10 * * SUN` can be used.
- If the job should be run monday through friday, the expression `00 10 * * MON-FRI` can be used.
- If the job should be run on each first or 15th of the month, the expression `00 10 01,15 * *` can be used.

Table 1: The fields of the cron-job date format

Field	optional	Values	Special Expressions
Minute	No	0-59	* , - /
Hour	No	0-23	* , - /
Day of month	No	1-31	* , - /
Month	No	1-12 or JAN-DEC	* , - /
Weekday	No	0-6 or SUN-SAT	* , - /
Year	Yes	1970-2099	* , - /

2.3.5 Message processing

The job configuration defines (as described above) a frame in which messages are determined. The actual functions of the *CONTACT Catalyst EAI Gateway* are triggered by these messages. In the following this functionality is described.

A message that triggers a function in *CONTACT Catalyst EAI Gateway* is structured in a specific format (referred to as CDBXML). It contains the name of the function to be executed and respective function-specific parameters.

Informally described, the essential requirement for such messages is that the function name is specified by the top-level tag of the XML structure. All tags of such messages should be created in the namespace "http://xml.contact.de/schema/xgwschema/1.0"; messages generated by processing the functions associated with these tags usually have the namespace "http://xml.contact.de/schema/xmlgateway/1.0". For example:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<config xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://xml.contact.de/schema/xgwschema/1.0" />
```

The other message content consists of parameters for the respective function. The format of the content is defined specifically for the individual functions. The parameters are passed as tags below the function tag in the XML message. In the *config* function listed above, for example, a *destination* parameter can be passed that defines the destination for data outputs. This can have the following appearance:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<config xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <destination class="cs.xml.Destinations.FileDestination"
               filename="%(output_dir)s/output_file.xml" />
</config>
```

The functions provided in the standard revolve around data exchange with ERP systems and automated import/export of data. Chapter *Processing your own XML tags* (page 61) describes how the *CONTACT Catalyst EAI Gateway* can be extended beyond that by implementing your own functions.

The order in which the individual standard *CONTACT Catalyst EAI Gateway* functions are usually used is shown in Figure *Typical execution order of the CONTACT Catalyst EAI Gateway functions* (page 14). In this example, the `target` function (see *Target* (page 14)) is used first to configure the connection to CONTACT Elements. Then the `config` function (see *Config* (page 14)) is used to configure various aspects of message processing in *CONTACT Catalyst EAI Gateway* itself. This includes setting up transformation pipelines for incoming and outgoing messages, and defining key fields for third-party system responses. Steps 3 and 4 then cover the actual data transfer. In step 3, data marked for transfer is sent to the external system using the `schema` function (see *Schema* (page 15)). In step 4 – based on the third-party system's response in the form of a `response` message

(see *Response* (page 15)) – the success or failure of synchronization of each object is reported to CONTACT Elements.

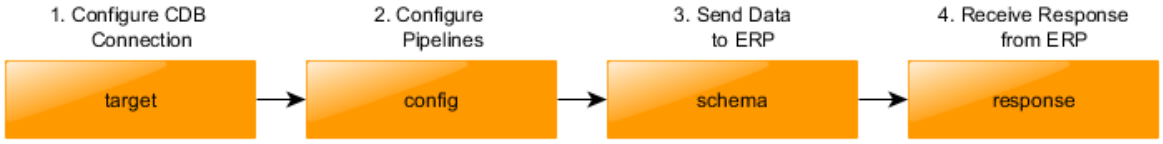

Fig. 4: Typical execution order of the *CONTACT Catalyst EAI Gateway* functions

Next, the standard functions and their parameters will be introduced in detail.

2.3.5.1 Target

Specifies the connection data for the CONTACT Elements-server. Similar to config messages target messages can be put in .xgs files and executed in a job configuration by specifying them in the *schema_files* attribute. For parameters see *Schema* (page 15).

The following attributes must be specified for the element:

name Name of the EAI configuration to be used (see manual for *CONTACT Catalyst for ERP*).

id ID of the target connection

```
<target xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        name="xmlgateway"
        id="xmlgateway">
</target>
```

The use of `target` as a top level `tag` allows to specify the connection information in an own file and to reuse this configuration in different jobs. Alternatively the target can be specified inside of the `schema` function (see *Schema* (page 15)).

2.3.5.2 Config

Config messages are mainly used to configure jobs. A common use case is the processing of incoming messages. Such messages are often sent to CONTACT Elements in a foreign format, so they have to be transformed by the *CONTACT Catalyst EAI Gateway* before processing. When configuring the job to process these incoming messages, a schema file (.xgs) can now be specified in the *schema_files* job attribute, the contents of which are a config message with the definition of transformation modules.

Parameters:

transformation defines a transformation step for incoming data by a transformation module (see *Destination modules* (page 18)). Several blocks can be enumerated. They are executed in the specified order. Thereby the result of each transformation step is used as input for the next step.

destination defines the destination of the data transfer using a destination block (see *Destination modules* (page 18)). Several blocks can be enumerated. They are executed in the specified order. Thereby the result of each transformation step is used as input for the next step.

pipeline defines a destination block, which in turn can contain nested destination blocks of its own. This can be used to define conditional or repeated pipelines.

response_object allows to define the key fields used to identify the objects in CONTACT Elements. The use of this parameter is described in section *Identification and versions* (page 38).

2.3.5.3 Schema

This function triggers the transfer of data to be transferred or exported and covers many of the main use cases of *CONTACT Catalyst EAI Gateway*.

The configuration of data to be synchronized (synchronization times, data scopes, attribute mappings, etc.) is described in the *CONTACT Catalyst for ERP* manual. The transfer to the ERP system can now be triggered by an interval-controlled job whose configuration file (attribute *schema_files*) triggers this function. Alternatively, the transfer can be triggered by the ERP system (or another external or CONTACT Elements component) by configuring the job for HTTP-triggered execution.

The *schema* function takes the following parameters (provided as child elements).

target Connection information can be defined in an own schema file. See also section *Target* (page 14).

destination Defines the destination of the data export using a destination module (see *Destination modules* (page 18)). Multiple modules can be listed. They are run in the order specified. Attributes:

class Name of the destination module

Additional attributes are listed in the documentation for the respective module.

pipeline defines a destination component, which may contain other destination components.

object specifies which objects are to be transferred from the CONTACT Elements instance and which schema definition is to be used to construct the message to be transferred (see *Data transmission* (page 29)).

Configurations using the `schema` tag are discussed in section *Schema* (page 29).

2.3.5.4 Response

Using this function an ERP system is able to respond to the data transferred from CONTACT Elements. For each transferred object success or failure can be signalled. The connected system may also provide additional parameters to CONTACT Elements. This tag can also be used to initiate a data transfer from the connected system. For details see section *Data transmission* (page 29). An example for a `response` message:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<response xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object id="706172743a[...]35373a745f696e6465783d"
          type="part"
          action="info">
    <field name="DESCRIPTION_EN">component</field>
    <field name="BRUTTO_GEWICHT">12,7</field>
  </object>
</response>
```

2.3.5.5 GetFile

This function allows access by external systems to files and documents managed in CONTACT Elements. This makes it possible to perform actions such as passing part data with links on to ERP systems without the documents themselves also being passed to the ERP system. The display of the linked documents can be implemented in the ERP system using access to an CONTACT Elements URL.

Job configurations that are set up for this function have to define a working directory using a *work_dir* attribute.

Parameters of the message:

class The class of the queried file (e.g. *document*). The desired representation of the document is specified using the *format* attribute.

key The keys for the desired document are specified using these parameters. Attributes are *name* and *value*.

An example of such a message:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<getfile xmlns:xi="http://www.w3.org/2001/XInclude"
         xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <class format="pdf">document</class>
  <key name="z_nummer" value="D-000123" />
  <key name="z_index" value="00" />
</getfile>
```

This message transfers the document with the number `D-000123` and index `00`. The content of the referenced File is stored in a buffer of type `StringIO`, which is passed to the destination pipeline. Therefore, the first element in the pipeline must be a `StreamDestination` (cf. *cs.xml.Destinations.StreamDestination* (page 22)). A complete example is given in section *Querying data from a web service* (page 80).

2.3.6 Transformation and destination definition

Messages can be transformed both when they are received from a third-party system and before they are sent. This makes it possible to influence both the data format and the transmission medium.

Transformation steps can be combined using similar methods in both input and output directions. The input steps are called transformation steps; the output steps are called destination steps. These can be defined using two *transformation* and *destination* configuration options.

There are an assortment of standard modules for this, but you can also implement your own modules.

Here is an example of the definition of destination steps:

```
<destination class="cs.xml.Destinations.XMLDestination"/>
<destination class="cs.xml.Destinations.FileDestination"
             filename="%(processed_dir)s/part.xml"
             unique="True"
             wait="False" />
<destination class="cs.xml.Destinations.HTTPDestination"
             uri="http://some.webservice.com/port" />
```

Outgoing messages are usually first generated by *CONTACT Catalyst EAI Gateway* as an XML structure and passed to the destination modules. This opens up the possibility of performing transformation steps directly on the structure tree and not only on the XML code. In the above example, no transformation is performed. Instead, a destination module is used to translate the structure into XML code (in a sense, this is also a transformation step). In the second step the XML code is written to a file in the directory configured under *processed_dir*. Such a step should be configured in if the outgoing messages are to be additionally archived.

Note: that the message structure is initially built as an XML tree does not mean that the output is limited to XML formats. The message structure can also be derived into JSON formats through appropriate standard modules. However, custom modules can also be used to write the fields of a message to columns in a database table, for example.

In the final step, the XML message is sent to a web service endpoint. If the data transfer is to be file-based instead, another *FileDestination* module can be used here accordingly.

Assuming the called web service returns a response message synchronously over the HTTP connection. To process this response, one or more additional blocks can be appended:

```
<destination class="cs.xml.Destinations.FileDestination"
             filename="%(processed_dir)s/response.xml"
             unique="True"
             wait="False" />
<destination class="cs.xml.Destinations.FollowupDestination"/>
```

The response is archived in this case. The subsequent *FollowupDestination* module causes the response to be recorded as an additional received message in job processing. Additional transformation modules are frequently inserted before the *FollowupDestination* module.

If destination modules are not defined in a schema file, modules corresponding to the following configuration are used:

```
<destination class="cs.xml.Destinations.XMLDestination"/>
<destination class="cs.xml.Destinations.FileDestination"
             filename="%(output_dir)s/xgw_output.xml" />
```

Now for the opposite direction, the transformation of incoming XML messages:

```
<transformation class="cs.xml.Transformations.XMLParser"/>
```

This block is the counterpart to *cs.xml.Destinations.XMLDestination*. It transforms an incoming XML message into a structure tree that can be processed by the *CONTACT Catalyst EAI Gateway*. If no transformation step is defined, this block is used.

Note: the `static_proc` configuration option (see section *Job definition* (page 7)) can also be used to define an input handler for messages or unstructured inputs for which no message type can be determined according to the default pattern.

Also, when receiving messages, there is an option to archive incoming messages before transformation:

```
<transformation class="cs.xml.Transformations.BackupInput"/>
<transformation class="cs.xml.Transformations.XMLParser"/>
```

If the message has been read from a file into *input_files*, it is written into the directory configured under *processed_dir* from the BackupInput module under the same name. If the message is received as a web service over HTTP or as the result from additional modules, it is saved under the name *http_input.xml*.

2.3.6.1 Complex Transformations

With the help of `<destination/>` it is possible to configure linear pipelines which transfer an XML structure generated by the gateway into the target format of a connected system. Within such pipelines it is often necessary that a transformation is applied to many elements. For example, if a `<schema/>` is generated to transfer multiple objects, it may be necessary to apply a transformation to each of these objects. To solve such, and similar scenarios in a mostly configurational way, the *CONTACT Catalyst EAI Gateway* offers the possibility to configure pipelines in which reusable components, branches, iterations and similar concepts from programming can be used.

The <pipeline/> tag

The `<pipeline/>` tag is introduced as the basic element for modelling complex transformations. It is processed as a child of the `<schema/>`, `<config/>` and `<pipeline/>` elements. The tag itself is processed by the elements `<pipeline/>` and <destination/>.

The semantics of the tag are the same as those of the `<destination>` tag, except that components may again be defined within the tag. Furthermore, if your component is not specified, a `<pipeline/>` will perform a `CompoundDestination`. That is, the input is passed to a pipeline defined by the included components. The result of this pipeline is then passed to the next component.

As a result

```
<schema>
  <destination class="cs.xml.json_destinations.ToRestDestination">
  <destination class="cs.xml.json_destinations.JSONDestination">
  <destination class="cs.xml.Destinations.HTTPDestination">
</schema>
```

and

```
<schema>
  <pipeline>
    <destination class="cs.xml.json_destinations.ToRestDestination">
    <destination class="cs.xml.json_destinations.JSONDestination">
    <destination class="cs.xml.Destinations.HTTPDestination">
  </pipeline>
</schema>
```

behave the same.

Reusable pipelines

By embedding pipeline components within a pipeline tag, it is now possible to page them out into their own files using the `XInclude` standard for `XML`. The schema definition from the above example could thus be written as

```
<schema>
  <xi:include href="pipeline.xgs">
</schema>
```

A second file `pipeline.xgs` contains the content defining the pipeline:

```
<pipeline>
  <destination class="cs.xml.json_destinations.ToRestDestination">
  <destination class="cs.xml.json_destinations.JSONDestination">
  <destination class="cs.xml.Destinations.HTTPDestination">
</pipeline>
```

If the pipeline is stored in its own file, it is thus possible to include it in other places as well.

2.3.6.2 Destination modules

A description of the default modules and their configuration options is provided below. The input and output format that each module uses has to be taken into account. It is important to ensure the output format of the predecessor matches the input format of the successor for combinations of modules. The two formats normally in use are XML structure and XML text. If an unsupported format is delivered to a module by its predecessor, this causes the data to loop through to the successor without any changes to the contents.

In addition to the blocks described in section *Linear Destinations* (page 18), which can be used to configure simple linear transformation sequences, the *CONTACT Catalyst EAI Gateway* also provides standard blocks to implement complex transformations as presented in section *Complex Transformations* (page 17). An overview of these can be found in section *Complex Destinations* (page 23).

Custom modules can be implemented in addition to these standard modules (see section *Custom destination modules* (page 64)). This gives you every possible option for manipulating incoming XML structures or data streams through program control. This includes functions for converting XML messages to XML structures using a parser.

For all destinations, further options can be defined via an `option` tag, besides those described below. The XML structure enclosed within such an `option` tag can be evaluated by custom destinations, for example. This permits to use not only simple attributes, but also structured configuration data when defining a destination configuration. For an example, see `HTTPDestination`.

Linear Destinations

cs.xml.Destinations.XMLDestination

The incoming XML structure is converted into a text stream. This module is required for storage as a file or sending.

Input XML structure

Output XML text

Parameter

encoding Default: *utf-8*

Defines the format of the resulting text.

cs.xml.Destinations.FileDestination

The incoming text stream is written to a file and passed on to a subsequent optional module without any changes.

Input XML text

Output XML text

Parameter

filename Filename with complete path. The path names from the job definition enclosed in %(*pathname*)s can be used here.

wait If the value is set to *True*, it is checked whether a file with the same name already exists. If this is the case, then it will not be overwritten, but the data transfer will be aborted. Use of this parameter allows the receiver to signal successful processing by deleting the transmission file. If there is new data for transmission in the meantime, the gateway will wait that long and only then will it write out a new file.

unique If the value is set to *True*, then the configured filename is extended by adding an addition in the form of a UUID separated by "_" (in front of the filename extension). This generates a unique filename with each export.

cs.xml.Destinations.XMLFileDestination

Combination of XMLDestination and FileDestination (see above for parameters).

Input XML structure

Output XML text

cs.xml.Destinations.HTTPDestination

The incoming data is sent to an HTTP port. This block can be used to address an external web service. The module inherits from the StreamDestination (*cs.xml.Destinations.StreamDestination* (page 22)). This also allows file-like objects to be supplied via the `output()` interface, which are then streamed.

As input arbitrary data types are accepted besides xml text (as an example see also JSONDestination).

If the http port sends response data back, these data can be handed on to an optional following component as text or as data stream (as example see FollowupDestination: *cs.xml.Destinations.FollowupDestination* (page 22)).

Input text or data stream

Output text

Parameter

uri URI of the recipient. HTTP and HTTPS protocols are supported.

secret When authenticating via Basic Authentication, a secret stored in the wallet can be specified.

key Session key

Additional parameters are supported for various E.g. SSL certificates, proxies and timeouts can be specified.

If the *secret* parameter is specified, the content will be used as the wallet path. This path should contain a username and password in the form of a colon-separated string.

How to use wallets, see the CONTACT Elements manual *Installation and Operation*, chapter *Management of Keys and Secrets*. For example, to store the username `user` and the password `abadpassword` in the path `cs.platform/customer/<customer>/endpoints/test`, the following command can be used:

```
echo user:abadpassword | cdbwallet store --stdin cs.platform/customer/my/endpoints/
↪test
```

A specific parameter is `key`. This parameter can be used to define a wallet path whose content is passed as a cookie to the called endpoint. This can be used specifically to access other CONTACT Elements instances and their REST API by passing a session ID. The ID is composed of name and value, separated by equal signs.

Example: `contact.sessionkey=12345`

In combination with Basic Authentication, this can be used to initiate a session and reuse it over multiple calls.

When configuring a HTTPDestination, instead of the parameters listed above you can also specify arguments via embedded *<option/>* tags. This way all arguments are accepted as defined in the interface definition of the python class requests.request. Using this, you can define headers, authentification, time outs, proxies, ssl certificates and all other features of the *Requests* interface. If both an attribute *uri* as well as an *url* in form of an *<option/>* are configured, the *url* is used instead of the *uri*.

An example for a http request with basic authentication:

```
<destination class="cs.xml.Destinations.HTTPDestination">
  <option name="url">http://example.org</option>
  <option name="method">GET</option>
  <option name="auth">
    <option name="secret">cs.platform/customer/my/endpoints/test</option>
  </option>
</destination>
```

Note: the text of options like `url` and `secret` must not contain line breaks or leading or trailing spaces to avoid changing the paths.

An example for a https request with ssl certificates:

```
<destination class="cs.xml.Destinations.HTTPDestination">
  <option name="url">https://example.org</option>
  <option name="cert">
      <option name="cert">/path/server.crt</option>
      <option name="key">/path/key</option>
  </option>
</destination>
```

An example for a https request using a proxy:

```
<destination class="cs.xml.Destinations.HTTPDestination">
  <option name="url">https://example.org</option>
  <option name="verify">True</option>
  <option name="cert">path/to/file</option>
  <option name="proxies">
      <option name="http">http://10.10.1.10:3128</option>
      <option name="https">http://10.10.1.10:1080</option>
   </option>
</destination>
```

An example for a http request with time out as tuple:

```
<destination class="cs.xml.Destinations.HTTPDestination">
  <option name="url">http://example.org</option>
  <option name="timeout">
      <option name="connect timeout">3.5</option>
      <option name="read timeout">10</option>
  </option>
</destination>
```

An example for a http request with timeout as float:

```
<destination class="cs.xml.Destinations.HTTPDestination">
  <option name="url">http://example.org</option>
  <option name="timeout">0.001</option>
</destination>
```

Besides this, the HTTPDestination defines a range of hooks, which allow implementation of customizing and extenstions by means of powerscript. To do so, a custom destination can be implemented inheriting from HTTPDestination and overwriting the methods described below.

`@property session(self)` This property allows access to the underlying session object of type requests.Session. Using this you can e.g. execute authentication calls in advance and then execute the actual call in the context of the authenticated session.

`prepare_auth(self, **kwargs)` This method can be used to create and return an authentication object, which has a subclass of the `requests.auth.AuthBase` class or adheres to it's protocol. The default implementation creates a HTTPBasicAuth object when an argument named *username* is provided, otherwise *None*, indicating no authentication is needed.

`prepare_http_params(self)` This method returns a dictionary, whose keys correspond to the arguments of the `Requests` class shown above. The default implementation accordingly transfers the given *<option/>* arguments into a dictionary. Often customizing is used to add custon headers. An example for this would be to set a specific content type.

`call(self)` The web service call is executed using the options defined in `prepare_http_params(self)`. Return value is the response object described in the `Requests` class.

`error_handler(self, resp)` Here, another error handler can be implemented. The default implementation checks and loggs the occurence of HTTPError exceptions.

`check_and_set_response(self, resp)` Here also the response object is provided. By overwriting this method, you can query the response for details like response code, headers or cookies. The method returns an iterator, which should iterate over the chunks of the response content, allowing to send them via `write()/flush()` interface to the following destination component. This allows to stream large contents. If a custom implementation returns `None` instead of in iterator, the `write()/flush()` interface is not used. Alternatively, the custom implementation can call the `output()` method, providing a data object. Advice: when using the `output()` method, consider the `output()` method being overwritten in the StreamDestination, which is inherited by this component. So to provide an object to the next component in the pipeline, use following code for example:

```
return super(StreamDestination, self).output(fd)
```

cs.xml.Destinations.DebugDestination

Each line of the text output is passed unchanged to an optional further block and additionally written to the log.

Input XML text

Output XML text

Parameter

lev Log level (default 5).

cs.xml.Destinations.XSLTTransformation

The XML structure is converted to a target format using a transformation written in XSL (Extensible Stylesheet Language).

Input XML structure

Output XML structure

Parameter

xslt Name and path of an XSLT file.

cs.xml.Destinations.ASCIIDestination

The data to be exported is converted to simple ASCII (CSV) text. For a configuration description, see *ASCII adapter* (page 87).

Input XML structure

Output ASCII text

cs.xml.Destinations.ResponseDestination

Jobs whose execution is triggered by a web service call require the ability to return their output as a synchronous response to the web service caller. This module can be used for this purpose.

Input XML text or data stream

Output XML text

Parameter

content-type Data type. The default is text/xml.

cs.xml.Destinations.FollowupDestination

When a module provides a new XML message for further processing (e.g., in response to a web service call in HTTPDestination), this message can be stored for further processing by feeding it into this module.

Input XML text

Output XML text

cs.xml.Destinations.StreamDestination

If (e.g. in the context of a web service call) content other than an XML message is to be sent, then this content can be provided by a custom module as a stream in the form of a Python Filedescriptor.

The content of the descriptor (which could represent a file, socket or StringIO object) can be streamed to an HTTPDestination or ResponseDestination using this module.

Input Powerscript file descriptor

Output XML text or data stream

cs.xml.Destinations.SyncResultDestination

This module is used as the basis for custom modules that implement synchronous links to ERP systems.

The use case consists of transferring data records to be provided to an ERP system by means of a function call implemented through of a web service, for example. The ERP system from each function call returns a success message of the data update or an error message (see CAD configuration switch "PPS Single Object Transfer").

A module inheriting from this Python class can then be used as a successor of the module calling the function (e.g. HTTPDestination). The custom implementation of this module can then parse the response (e.g. in the form of an XML message) and process the success or error message by calling *self.acknowledge()* or *self.error(code, messagetext)*.

Input XML text

Output Unchanged input

cs.xml.json_destinations.ToXgwPyDataDestination

Generates a Python Data Object from the incoming XML structure, which corresponds to the format of the original XGW message. This can be passed to a JSON destination for further processing, see *The XGW-JSON Format* (page 49).

Input XML structure, which contains a `schema` or `response` message

Output Python data structure with the content of the incoming message

cs.xml.json_destinations.ToRestPyDataDestination

Generates a Python Data Object with the content of the message from the incoming XML structure. The structure of the Python Data Object is similar to the ReST API. This can be passed to a JSONDestination for further processing, see *The ReST-based Format* (page 50).

Input XML structure, which contains a `schema` or `response` message

Output Python Data Object with the content of the incoming message.

cs.xml.json_destinations.JSONDestination

Converts the incoming Python Data object into a string containing the JSON representation of the object using the Python package JSON. Thus, this destination forms the counterpart of *cs.xml.json_destinations.JSONTransformation* (page 26).

Input Python Data Object

Output String containing the JSON representation of the input

Complex Destinations

cs.xml.Destinations.DeepCopyDestination

```
<destination class="cs.psi.destinations.DeepCopyDestination" />
```

`DeepCopyDestination` renders a deep copy of the input, and passes this copy to its successor. In complex pipeline configurations it may be used as the first component in a subpipeline, to avoid transformations of the element to effect other parts of the transformation process.

Input `a` - a copyable Python object

Output `a` - a copy of this element

cs.xml.Destinations.LastElementDestination

```
<pipeline class="cs.psi.destinations.ForEach">
    <!-- ... -->
</pipeline>
<destination class="cs.psi.destinations.LastElementDestination" />
```

`LastElementDestination` receives a list of objects and passes the last object in the list to the next component. It is useful as a successor to components such as *cs.xml.Destinations.ListDestination* (page 25) or *cs.xml.Destinations.ForEach* (page 24) that produce a list of elements, if only the last element is required.

Input `[a_1, ..., a_n]` - a list of elements

Output `a_n` - the last element of the list

cs.xml.Destinations.CompoundDestination

```
<pipeline class="cs.psi.destinations.CompoundDestination">
    <destination class="foo" />
    <!-- ... -->
</pipeline>
```

`CompoundDestination` is a Destination component, which should be used as the base class for implementing custom `<pipeline/>` components. `CompoundDestination` provides the method `run_pipeline` which can be used to instantiate a pipeline from the components defined as child elements of the `<pipeline/> tag` and execute this pipeline.

A `CompoundDestination` is used to define `<pipeline/>` used in control structures as *cs.xml.Destinations.ListDestination* (page 25), SwitchDestination or *cs.xml.Destinations.ForEach* (page 24). It is also used as a base class for the implementation of these classes, and may be used as a base class for custom control structures.

Subpipeline `a -> b` - a transformation

Input `a` - input of the first `destination` in the pipeline

Output `b` - output of the last `destination` in the pipeline

cs.xml.Destinations.ForEach

```
<pipeline class="cs.psi.destinations.ForEach">
    <pipeline> <!-- sp_1 --> </pipeline>
    <pipeline> <!-- sp_2 --> </pipeline>
    <!-- ... -->
    <pipeline> <!-- sp_n --> </pipeline>
</pipeline>
```

The `ForEach` destination executes each of its pipelines – each defined by one of its components – for the provided element. The list of results from executing each pipeline is passed to the next component.

Pipelines defined in `ForEach` may use the class *cs.xml.Destinations.CompoundDestination* (page 24).

Subpipeline `[a -> b_1, a -> b_2, ..., a -> b_n]` - a list of subpipelines

Input `a` - an element

Output `[b_1, b_2, ..., b_n]` - a list of transformed elements

cs.xml.Destinations.ListDestination

```
<pipeline class="cs.psi.destinations.ListDestination">
    <destination class="..." />
    <destination class="..." />
    <!-- ... -->
    <destination class="..." />
</pipeline>
```

`ListDestination` instantiates a pipeline from the components defined in the `<pipeline/>` tag and executes it for each given object in the input list. The list of results is passed to the next component.

Subpipeline `a -> b`

Input `[a, a, ..., a]` - a list of elements of the input type of the pipeline

Output `[b, b, ..., b]` - a list of elements of the output type of the pipeline

cs.xml.Destinations.SwitchDestination

```
<pipeline class="cs.psi.destinations.SwitchDestination">
  <pipeline handle_type="p_1"> <!-- ... --> </pipeline>
  <pipeline handle_type="p_2"> <!-- ... --> </pipeline>
  <!-- ... -->
  <pipeline handle_type="p_n"> <!-- ... --> </pipeline>
</pipeline>
```

Input to `SwitchDestination` is a tuple of two values:

- A unique identifier for a type
- The object (or a list of objects), with which the pipeline is executed.

Multiple pipelines are defined in the `SwitchDestination`. Each of these must contain the unique attribute `handle_type`. The `SwitchDestination` instantiates the pipeline that matches the given identifier, and executes it on the second element of its input. The result of the pipeline execution is sent to the next component.

Subpipeline `[(handle_type=p_1) a -> b_1, (handle_type=p_2) a -> b_2, ..., (handle_type=p_2) a -> b_n]` - A list of components, that are identified by the value of their `handle_type` attribute.

Input `(String p_X in [p_1, p_2, ..., p_n], a)` - A tuple of a string and the element to be transformed.

Output `b_X` - The result of applying the pipeline, which is identified by the first element of the tuple, on its second element.

2.3.6.3 Transformation modules

The standard modules are similar to the destination modules in many respects. In this case, it is also important to ensure the output format of the predecessor matches the input format of the successor for combinations of modules.

In contrast to the destination blocks, there are no blocks for defining transfer destinations here. The final destination is always the job processing facility of the *CONTACT Catalyst EAI Gateway*. Transformation building blocks are used to log, archive and transform XML messages into CDBXML formats.

To convert foreign formats into CDBXML format, separate modules have to be implemented that parse the foreign format and that establish a CDBXML structure matching the contents. This is supported by the *cs.xml.xmllib* function library (see *cs.xml programming manual*). The implementation of these modules can be done analogous to the implementation of custom destination modules, as shown in section *Custom destination modules* (page 64).

cs.xml.Transformations.XMLParser

This module transforms the incoming XML text into an XML structure. If no transformation module is defined, this module is used automatically.

Input XML text

Output XML structure

cs.xml.Transformations.BackupInput

This module can be used to archive incoming messages for processing as well. The incoming text or data stream is written to a file and passed on to the subsequent transformation module. The file is stored under a unique filename in the directory defined by the *processed_dir* job attribute. In other words, the module should be used before *XMLParser*.

Input XML text

Output XML text

cs.xml.Transformations.ASCIITransformation

The incoming data is expected in the form of simple ASCII (CSV) text. The module transforms it into a corresponding XML message. For a configuration description, see *ASCII adapter* (page 87).

Input ASCII text

Output XML structure

cs.xml.json_destinations.JSONTransformation

Converts the incoming JSON-encoded string into a Python data structure and thus forms the counterpart to *cs.xml.json_destinations.JSONDestination* (page 23). This data structure must be further processed or converted in a subsequent module. Typically, one of the two modules described below is used for this purpose.

Input String with json-encoded data

Output A python data structure corresponding to the incoming data.

cs.xml.json_destinations.FromXgwPyDataTransformation

Converts an incoming `response` element represented as a Python data structure in *CONTACT Catalyst EAI Gateway* format to an XML structure. Thus this module is the counterpart of *cs.xml.json_destinations.ToXgwPyDataDestination* (page 23). See also *The ReST-based Format* (page 50). The result of this module can be fed into job processing just like that of the XMLParser module above.

Input Python data structure containing a `response` element in *CONTACT Catalyst EAI Gateway* format

Output XML structure representing the received message in *CONTACT Catalyst EAI Gateway* format.

cs.xml.json_destinations.FromRestPyDataTransformation

Converts an incoming `response` element represented as a Python data structure in ReST-oriented format to an XML structure. Thus, this module is the counterpart of *cs.xml.json_destinations.ToRestPyDataDestination* (page 23). See *The ReST-based Format* (page 50). The result of this module can be fed into job processing just like that of the XMLParser module above.

Input Python data structure which contains a `response` element in the ReST-like format.

Output XML structure representing the received message in *CONTACT Catalyst EAI Gateway* format.

2.3.6.4 Debugging Destination Pipelines

The *CONTACT Catalyst EAI Gateway* offers two possibilities to debug the execution of destination pipelines. First, there is the possibility to declare destination components as debugging components, whereby these components are only included in the pipeline if pipeline debugging is enabled for the corresponding job. This is described in the *Configuring debugging components* (page 28) section.

The execution of pipelines can also be monitored in the log file. While executing a destination pipelines a stacktrace of the executed components is created. This is described in section *Configuring debugging components* (page 28).

Logging Pipeline Execution

The gateway logs calls to each component during pipeline execution. Logged data are:

- Executed destination class
- The xml attributes with which the class has been instantiated
- Executed destination method (`output`, `write` or `flush`)
- the object, which is passed as input to the destination (in case of logging `output` and `write`)

Logging of the pipeline execution occurs on log level 6. During the execution each call to a pipeline component is logged immediately. If an error occurs during pipeline execution the job execution is cancelled, and a stacktrace of the pipeline execution is written to the log.

Defining Handlers for custom data types

In order to log objects being transformed during pipeline execution, it is necessary to serialize these to human-readable text. The default is to call the Python function str.

However, since this does not always lead to the desired result (for example, the objects used in *CONTACT Catalyst EAI Gateway* to represent XML do not provide a serialized form) a custom handler can be registered for each type.

Custom Handlers can be registered by using the decorator `cs.xml.destination_debugger.custom_handler`. The function `cs.xml.destination_debugger.apply_custom_handler` can be used to invoke the appropriate handler for a given object. This allows to handle data types as collections, which require to recursively invoke custom handlers.

The following example demonstrates the definition of a custom handler, that allows to serialize python sequences.

```
from cs.xml.destination_debugger import custom_handler
from cs.xml.destination_debugger import apply_custom_handler

@custom_handler(list)
def custom_handler_list(my_list):
    serialized_items = [apply_custom_handler(i) for i in my_list]
    return ', '.join(serialized_items)
```

Custom Handlers need to be defined in a module, which is loaded when the worker is starting.

The *CONTACT Catalyst EAI Gateway* already defines custom handlers for the following types:

- lxml.etree._Element
- xmllib.lxml_tree
- tuple
- list

- dict
- set

If required callables for other types can be registered.

Configuring debugging components

The method explained in the previous section requires studying debugging information within the log. Since this is not always desired, there is also the alternative option of declaring pipeline steps as debugging steps. These will then only be executed if pipeline debugging is enabled for the job in question.

A useful way to debug with this is to add *XMLFileDestination* components marked as debugging to the pipeline, which logs intermediate steps of the transformation in files.

The follow XGW schema file contains an example on how an additional destination has been introduced to the pipeline to generate the file `debug_file.xml`.

```
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
     xmlns:xi="http://www.w3.org/2001/XInclude">
  <destination class="cs.xml.Destinations.XMLFileDestination"
               filename="%(work_dir)s/debug_file.xml"
               debug="True"
               unique="True" />
  <destination class="cs.xml.Destinations.XMLFileDestination"
               filename="%(work_dir)s/output_file.xml"
               unique="True" />

  <object type="part" datasource="cs.xml.SharedObject.Factory">
    ...
  </object>
</schema >
```

The attribute `debug="True"` prevents the component from being instantiated if debugging is not activated for the executing job.

To activate debugging, a parameter can be passed to the job as follows:

```
XGW_JOBS = [    {'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
     'input_files': None,
     'job_interval': 10,
     'group_id': 'example_grp',
     'job_list': [{'active': True,
                   'job_id': 'Import Shared object 1',
                   'debug_destinations': 'True',
                   'position_id': 0,
                   'schema_files': ['site.xgs', 'job_conf.xgs']}],
     'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway/work'}]
```

When configuring pipelines with debugging-destinations it is important to ensure that these are runnable when debugging is activated, as well as when debugging is deactivated. For each debugging destination in the pipeline it must be ensured, that both the input to the destination, as well as its output can be processed by the following destination. An easy way is to ensure that the debug destination does not modify its input.

2.4 Use cases

The following describes how the most common use cases can be implemented using the configuration options presented: transferring data to ERP systems, retrieving online info from ERP systems, providing interfaces for file retrieval (document links), providing web service interfaces, and using the *CONTACT Catalyst EAI Gateway* as an export tool for mass data extraction or report generation.

2.4.1 Data transmission

This chapter describes the use case of data exchange between CONTACT Elements and the ERP system. The concepts can be applied analogously to data exchange with other external systems.

The use case can be roughly divided into the following exemplary steps:

- Design engineering data are recorded in CONTACT Elements.
- After approval, the production-relevant data are transferred to the ERP system.
- In the ERP system, more data is added (for example, foreign-language designations, supplier relationships) and transferred back to CONTACT Elements.

The definition of data classes to be transferred, transfer times, field definitions and mappings is described in the Catalyst administration manual. The *CONTACT Catalyst EAI Gateway* now comes into play in this use case when data in CONTACT Elements have reached a synchronization state defined in this way. Here, job definitions are used to specify which data is to be fetched from CONTACT Elements by a job, and in which form it is to be transferred to an ERP system.

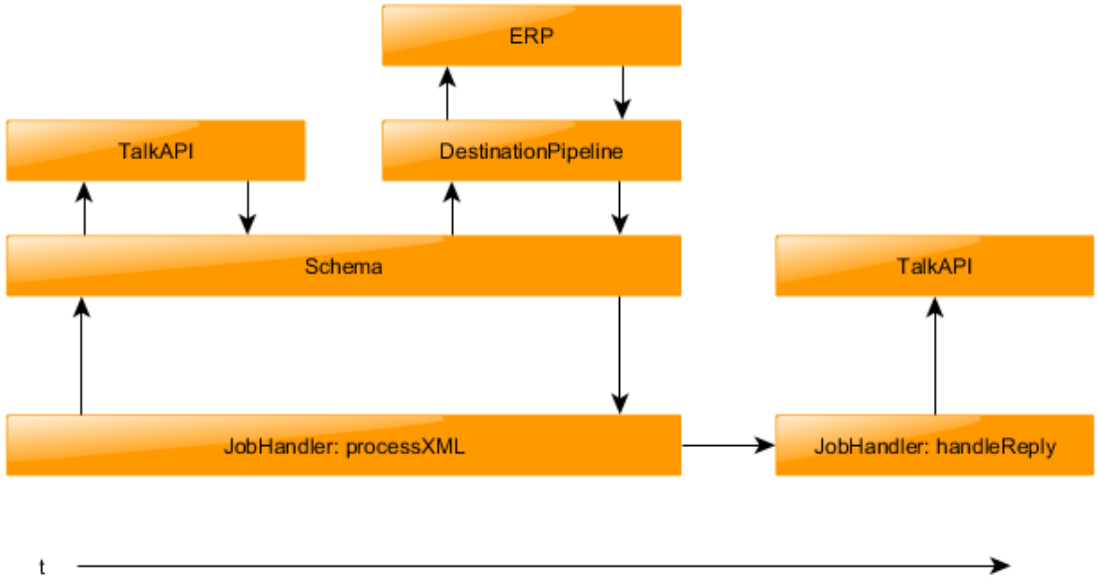

Fig. 5: Basic Process of Data Transfer with synchronous replies of the ERP-System

Figure *Basic Process of Data Transfer with synchronous replies of the ERP-System* (page 29) shows the fundamental interaction of the various components which participate in transferring data by HTTP. Normally a job first calls the function *Schema*, which will determine the objects to be synchronized by using Catalyst or SQL. These objects are then sent through the destination pipeline, which transforms, serializes and finally sends the data to the connected system. The connected system may then send a response, which is again transformed by a pipeline and processed as a followup message. If the transformation yields a *response* message, the result contained in the message is sent to CONTACT Elements.

2.4.1.1 Schema

By executing the schema function, a schema object is generated. The schema definition specifies the structure and scope of the data to be output. It includes the following configuration attributes, which can be specified in a schema message or in associated configuration files.

Target Defines from which configuration of integrated systems the synchronization configuration is obtained. The *CONTACT Catalyst EAI Gateway* obtains from this configuration, among other things, the definition of the data to be synchronized and their field mappings.

Data source Determines the data source for exportable data. For using the integration infrastructure of CONTACT Elements (Catalyst), *SharedObject* data sources for data to be synchronized and *Related* for relation structures of this data (for example, BOMs) are available. Moreover, data sources exist for direct database queries formulated directly in SQL; you also have the ability to implement your own data sources (see *Custom data sources* (page 57)).

Object Object elements define the transfer of data. They contain the data class of the data to be transferred, its data source and the format to be output including the fields to be transferred. Definition of an object schema with this element results in the specified data source being used to receive and output corresponding objects. For example, using the *SharedObject* data class causes the job to connect to the CONTACT Elements server defined under *Target* and query it for all objects of the specified class intended for transfer.

Filters Filters can be used for modifying the XML structure according to further requirements. By doing so, it is possible to filter individual elements (fields of an object or elements of a relation. Moreover, elements of the XML structure can be modified or added. *Filters* (page 52)).

Action For the synchronization of objects with ERP systems, various actions can be defined in addition to the normal transfer (see manual *CONTACT Catalyst for ERP*). For each of these actions a separate transfer scope (fields and relations) can be defined. The respective action is named by the attribute 'type'. Possible actions are:

put If no distinction should be made between different actions during transfer, the type "put" can be used. If an <action> element should be completely omitted when defining a schema and the fields and relations to be transferred are directly defined under <object>, then a "put" action is implicitly defined.

create Fields and relations that are defined within this action are used only when transferring an object for the first time.

modify If in further transfers, different field and relation scopes are to be transferred than at initial creation, then this can be defined via this action.

action_<n> In the Catalyst ERP configuration, further actions can be defined. When running this action, the exportable field and relation scopes can be defined here, where the respective action number has to be added to the names (for example: "action_3").

lock If an object should be disabled in the ERP system, then this action can define the exportable fields.

unlock As for *lock*, but for unlocking an object in the ERP system.

In the Catalyst attribute configuration you also have the ability to distinguish between a new creation and a modification by accessing the field element "CDB::put_flag" in computed attributes, each of which is assigned the values "create" or "modify".

Field This element defines the data fields to be transferred. The name of the respective field is stored in the *name* attribute of the element. If, instead of a field name, the placeholder "*" is used, all fields that are defined in the Catalyst attribute configuration for the class of the object to be transferred are taken into account.

Related Related elements are used to model 1:n relations. Recursive compositions can nest related elements in any depth desired. In this manner, during an export the corresponding CAD documents can be assigned to a part, for example, and, in turn, to the editors as 1:n relations.

2.4.1.2 Definition of the Message format

In this chapter, the format of the message schema, it's contents and structure are described.

Details for the schema format

This schema format can be interpreted as follows:

1. A schema is a message (message element) that contains *0-n* object elements.
2. Object elements contain *0-n* filter elements, *0-n* action elements, *0-n* field elements and *0-n* related elements.
3. A related element has a *name* attribute and contains *0-n* object elements.

Message generation based on Catalyst

The following example shows such a schema message:

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- -\*- XML -\*- -->
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="target.xml"/>
  <object type="part" datasource="cs.xml.SharedObject.Factory">
    <field name="IDENTNR"/>
    <field name="GEWICHT"/>
  </object>
</schema>
```

The “IDENT NO” and “WEIGHT” fields of the “part” object type are exported via the synchronization mechanism of Catalyst. A prerequisite for this kind of export is that the pertinent fields in the ERP configuration be configured by Catalyst for the “part” object type.

An example of an export in accordance with this schema definition is provided here:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object id="706172743a[...]35373a745f696e6465783d"
                type="part">
        <field name="IDENTNR" attribute_type="2">1017657</field>
        <field name="GEWICHT" attribute_type="2">0</field>
    </object>
    <object id="706172743a[...]36393a745f696e6465783d"
                type="part">
        <field name="IDENTNR" attribute_type="2">1214469</field>
        <field name="GEWICHT" attribute_type="2">0</field>
    </object>
</schema>
```

The exported objects contain unique transaction IDs in the “id” attribute to be able to identify synchronized objects.

The enclosed fields respectively contain attributes describing the field names and their types as defined in the EAI field configuration of Catalyst. The type attribute has a numeric value with following meaning:

Value	Meaning
0	String
1	Float
2	Integer
3	Boolean
4	Date

Relationships (<related> elements)

<object> elements can be assigned further elements via <related> elements. This way structures of any depth can be created by recursive composition (the assigned objects can contain <related> elements again, etc.).

The way a relationship is evaluated is determined by the source used (attribute `datasource`). While an `object` element usually uses the `cs.xml.SharedObject.Factory` data source, which delivers objects released via Catalyst to be shared, special data sources are used for deriving objects related to such objects, which are introduced in the following sections.

The definition of a `related` element is similar to defining an `object` element:

```
<related type="example_class"
         name="example_name"
```

(continues on next page)

(continued from previous page)

```
        datasource="my_datasource">
  <related>
    ...
  </related>
  <field name="example_field" />
</related>
```

The `related` element needs to specify a `datasource` and a `type` attribute, similarly to `object` elements. Additionally an optional `name` attribute can be provided, which describes the relation exported. If provided, it will be exported instead of the `type` attribute.

The `related` element may contain field definitions, that specify the fields to be exported. It may also contain further `related` elements.

Data Source for BOMs

A typical example of using <related> elements is the transfer of part masters with associated BOM. The following example shows a corresponding schema:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="target.xml"/>
  <object type="part"
          datasource="cs.xml.SharedObject.Factory">
    <field name="IDENTNR"/>
    <field name="GEWICHT"/>
    <related type="bom_item"
             name="bom"
             datasource="cs.xml.BOM.Factory">
      <field name="POSITIONSNR"/>
      <field name="COMP_NR"/>
      <field name="AMOUNT"/>
    </related>
  </object>
</schema>
```

<Related> elements have two attributes *name* and *type*, which define the name of a related structure and the data type of the elements it contains. The type *BOM* is used here as data source for the <related> structure. This is a special data source for the derivation of parts lists. The position elements here have the special type *bom_item* (see manual *CONTACT Catalyst for ERP*).

Here is an example for a corresponding export:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object id="706172743a[...]35f696e6465783d"
             type="part">
    <field name="IDENTNR">1017657</field>
    <field name="GEWICHT">0</field>
    <related name="bom">
      <object type="bom_item">
        <field name="POSITIONSNR">10</field>
        <field name="COMP_NR">199969</field>
        <field name="AMOUNT">2</field>
      </object>
      <object type="bom_item">
        <field name="POSITIONSNR">20</field>
        <field name="COMP_NR">102176</field>
        <field name="AMOUNT">1</field>
```

(continues on next page)

(continued from previous page)

```
      </object>
      <object type="bom_item">
        <field name="POSITIONSNR">30</field>
        <field name="COMP_NR">120991</field>
        <field name="AMOUNT">1</field>
      </object>
    </related>
  </object>
</schema>
```

Relationships via `cdb_relships`

In addition to the special case of BOMs, other relationships defined in CONTACT Elements can also be exported. To do so, module `cs.xml.datasources.related` provides the datasources `NaryRelshipFactory` and `UnaryRelshipFactory`. The related objects are determined by executing the kernel operations `navigate_Relship` and `naviaget_OneOnOne_Relship`. As a consequence export is not restricted to attributes, which are defined in the table of the object class. Different attribute types, e.g. joined attributes, can also be used in the scheme.

Here is an example for exporting a hypothetical part/supplier relationship:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="target.xml"/>
  <object type="part"
          datasource="cs.xml.SharedObject.Factory">
    <field name="IDENTNR"/>
    <field name="GEWICHT"/>
    <related type="cust_lieferanten"
             datasource="cs.xml.datasources.related.NaryRelshipFactory"
             relship="cust_part2manufacturers">
      <field name="MANUFACTURER_NR"/>
      <field name="MANUFACTURER_NAME"/>
    </related>
  </object>
</schema>
```

And here is an example for a corresponding export:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object id="706172743a[...]35f696e6465783d"
              type="part">
    <field name="IDENTNR">1017657</field>
    <field name="GEWICHT">0</field>
    <related name="manufacturers">
      <object type="cust_lieferanten">
        <field name="MANUFACTURER_NR">000127</field>
        <field name="MANUFACTURER_NAME">Firma 1</field>
      </object>
      <object type="cust_lieferanten">
        <field name="MANUFACTURER_NR">000129</field>
        <field name="MANUFACTURER_NAME">Firma 3</field>
      </object>
    </related>
  </object>
</schema>
```

SQL-based definition of relations

Finally, data source structures formulated generally by SQL commands can be output by using the *QueryFactory*. An example of this is the transfer of part masters with the corresponding drawings. A corresponding schema:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="target.xml"/>
  <object type="part"
          datasource="cs.xml.SharedObject.Factory">
    <field name="IDENTNR"/>
    <field name="GEWICHT"/>
    <related type="document"
             name="documents"
             datasource="cs.xml.SQL.QueryFactory">
      <query basetable="zeichnung">
        <![CDATA[
          SELECT * FROM zeichnung
          WHERE teilenummer='REFERER[teilenummer]'
          AND t_index='REFERER[t_index]'
        ]]>
      </query>
      <field name="Z_NUMMER"/>
      <field name="Z_INDEX"/>
    </related>
  </object>
</schema>
```

The SQL query in the <related> element can reference attributes of the higher-level data object in the syntax REFERER[<attribut_name>]. Thus the data retrieval references the “part number” and “t_index” attributes of the part master that is one level up. If the ‘REFERER[part number]’ character string is in fact to be used for selecting, the character string can be prefixed with the letter “r” (for “raw”):

```
WHERE teilenummer=r'REFERER[teilenummer]'
```

Here is an example for a corresponding export:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <object id="706172743a[...]35f696e6465783d"
             type="part">
    <field name="IDENTNR">1017657</field>
    <field name="GEWICHT">0</field>
    <related name="documents">
      <object type="document">
        <field name="ZNUMMER">0199969</field>
        <field name="ZINDEX" />
      </object>
      <object type="document">
        <field name="ZNUMMER">0199865</field>
        <field name="ZINDEX" />
      </object>
      <object type="document">
        <field name="ZNUMMER">0228430</field>
        <field name="ZINDEX" />
      </object>
    </related>
  </object>
  <object id="706172743a[...]45f696e6465783d"
             type="part">
    <field name="IDENTNR">1292763</field>
```

(continues on next page)

(continued from previous page)

```
    <field name="GEWICHT">0</field>
    <related name="documents">
      <object type="document">
        <field name="ZNUMMER">0188064</field>
        <field name="ZINDEX" />
      </object>
    </related>
  </object>
  <object id="706172743a[...]03834303a745f696"
            type="part">
    <field name="IDENTNR">1330840</field>
    <field name="GEWICHT">0</field>
  </object>
</schema>
```

The corresponding documents (if present) are assigned to the part masters (object type "part"); these are grouped by <related> tags, which in turn each contain 1:n <object> elements.

In addition, the parameters for the <query> definition can be set beyond this. The *record_type* attribute defines whether field mappings are executed in CONTACT Elements in accordance with the attribute configuration (default) or whether the "raw" database contents are to be made available. The second case is achieved by the *record_type="raw"* attribute setting. Depending on this, the names of the database attributes (for "raw") have to be used in the schema definition as field names, otherwise the field names of the field definitions in the configuration.

The database table for the query can be defined using the *basetable* attribute. Instead of the complete SQL statement, this requires specifying only the selection condition:

```
<query basetable="zeichnung">
  teilenummer='REFERER[teilenummer]' and t_index='REFERER[t_index]'
</query>
```

This definition is supplemented internally by such a SQL statement:

```
SELECT * FROM zeichnung where teilenummer='<teilenummer>' and t_index='<t_index>'
```

Setting parameters of SQL queries

In addition to the options shown, there is another way to set parameters of queries. Variables that are substituted during runtime can be used in the schema file.

```
SELECT * FROM angestellter WHERE personalnummer = '%(personalnummer)s'
```

The syntax for representing the variables here is %(<Variablenname>)s' (this is a character substitution from a dictionary in Python). Accordingly, the above form means that the character string `%(personalnummer)s` is to be replaced by the value of the parameter with the same name.

These parameters are primarily intended to be passed in the form of arguments to an HTTP URL (ex: *http://my.server:7488/query_user?persno=027*). See *Web services* (page 43) for a description related to web services.

However, a corresponding dictionary can also be defined in the configuration. To do so, an *http_args* parameter has to be created in the job definition and a Python dictionary has to be assigned to it:

```
{
  ..
  'http_args': {'pers_no': '027'},
  ..
}
```

Another option is to define parameters within custom code (for example, in custom job implementations). This is done by calling up *set_http_arguments()*:

```
def my_custom_job(options):
  options.set_http_arguments(persno='027')
```

2.4.1.3 ERP system replies

The connected ERP system can, in turn, send a series of messages to CONTACT Elements. These can include both replies to data transmissions and independent messages triggered by internal ERP workflows.

By default, there are four different message types:

- Replies that confirm the error-free processing of data synchronization
- Error messages caused by data synchronizations
- Data synchronization initiated by the ERP system
- A combination of its own synchronization and reply

The message format corresponds largely to the format for transferring data from CONTACT Elements (see above). Such messages are embedded in a response tag rather than in a schema tag. The message content consists of a series of object tags that have a type attribute for describing the class and an ID attribute for identifying the object. In addition, each object has an action attribute that informs about a respective use case. Actions are *commit*, *error*, *info* and *cominfo*.

A *commit* message only has to contain the empty object tags (it can also contain field tags, which are then ignored). An *error* message looks similarly, but, in addition, can contain an error tag with an error description. For example:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<response xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object id="706172743a[...]35373a745f696e6465783d"
            type="part"
            action="commit"/>
    <object id="706172743a[...]36393a745f696e6465783d"
            type="part"
            action="error">
        <error>Error creating ERP-Material.</error>
    </object>
</response>
```

In an incoming *commit* message, the object is marked as synchronized in CONTACT Elements. Additionally, this can trigger follow-up status changes or consequential actions can be implemented in PowerScript (see Catalyst administration manual). Corresponding actions can be set up in response to *error* messages.

The ERP system can transmit data records to CONTACT Elements using the *info* message. This can entail complete data records from the ERP. These are created as new data records in CONTACT Elements or modified if they already have been transferred. Just as in transferring to the ERP, the content is transmitted via field tags. For posting, the standard operations *CDB_Create* or *CDB_Modify* are used, so that PowerScript extensions can also be executed. In the PowerScript code, by evaluating the variables *xtc.batch* or *ctx.cad_system*, it is possible to check whether a new creation or change is executed manually or within the context of an ERP integration.

This function can be used in the same manner for adding data in objects that previously had been transferred from CONTACT Elements. The addition of data can also be directly transferred when confirming a successful processing. The first case (addition of data from the ERP system) therefore provides additional information to CONTACT Elements. Accordingly, the associated action is called *info*. The combination of *commit* and *info* is called *cominfo*. For example:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<response xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object id="706172743a[...]35373a745f696e6465783d"
            type="part"
```

(continues on next page)

(continued from previous page)

```
            action="info">
        <field name="DESCRIPTION_EN">component</field>
        <field name="BRUTTO_GEWICHT">12,7</field>
    </object>
    <object id="706172743a[...]36393a745f696e6465783d"
            type="part"
            action="cominfo">
        <field name="DESCRIPTION_EN">engine</field>
        <field name="BRUTTO_GEWICHT">1250</field>
    </object>
</response>
```

Inversely to the transfer to the ERP system, the mappings of field names to CONTACT Elements attributes must also be defined here. Often you want to assign the same field names differently when transferring to the ERP and when replying. For example, a field should be defined as a composite of several attributes when transferred to the ERP. However, composite attributes cannot be used via transfer from the ERP. Therefore, in the field configuration of the EAI system configuration in CONTACT Elements, the definitions for the transfer out and the response from the ERP system can be distinguished. In addition to a default view, further views can be defined in which corresponding variants of the respective attribute mappings are created (see manual for *CONTACT Catalyst for ERP*).

After the *CONTACT Catalyst EAI Gateway* has processed the response from the ERP system, it in turn can generate a response that can optionally be sent back to the ERP system synchronously. This is particularly useful if the communication takes place via HTTP web services. This response contains an element *"result"* for each object sent in the original response of the ERP system, which contains information about the processing of the response of the ERP system.

The *"result"* element contains an attribute *"code"* for this purpose, which specifies the result of the processing, as well as an error message as text content. If successful, the *CONTACT Catalyst EAI Gateway* responds to a message with the *"code" 0*:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<response xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object id="706172743a[...]35373a745f696e6465783d"
            type="part"
            action="info">
        <field name="DESCRIPTION_EN">component</field>
        <field name="BRUTTO_GEWICHT">12,7</field>
        <result code="0">Action commit: Result: OK.</result>
    </object>
    <object id="706172743a[...]36393a745f696e6465783d"
            type="part"
            action="cominfo">
        <field name="DESCRIPTION_EN">engine</field>
        <field name="BRUTTO_GEWICHT">1250</field>
        <result code="5">Action cominfo: Error: update failed.</result>
    </object>
</response>
```

If processing of the answer from the ERP system fails, a value unequal to zero is returned.

If the option *custom_handler* (see *Expanding existing XML structures* (page 63)) is used to extend the format of replies, the reply sent back to the ERP system (in response to the reply of the ERP system) can be extended by custom results. To achieve this, the custom handler should invoke the function *cs.xml.ReplyHandler.set_result(node, code, text)*. This function can be called after processing each object element, to insert a custom result node for each object, which contains a value indicating success or failure of the operation as well as an additional description. How to configure a custom handler for tags is described in section *Expanding existing XML structures* (page 63).

The response message generated by the *CONTACT Catalyst EAI Gateway* is processed by the configured destination pipeline, if configured. Here the message can be modified by suitable blocks and finally saved as a file (*cs.xml.Destinations.FileDestination*), or returned to the caller (*cs.xml.Destinations.RepsonseDestination*).

Identification and versions

The default case to identify objects received by replies in CONTACT Elements is to use IDs that have been transmitted to the ERP system beforehand. Data that originally is transferred from the ERP system does not have such IDs. In the case of asynchronous communication, for example, the ID of the original message may no longer be available when sending the reply. Thus multiple ways are available to forgo or determine IDs.

First, it is possible to transfer the key fields, rather than an ID, of the desired objects in the message. Here it is required to mark the key fields in the configuration. For this purpose, in the associated job definition, you have to specify a *Config* message in the section *schema_files*. In this Config message, you have to specify one or more *response_object* tags. Using a *type* attribute, each of these tags determines a class and defines the key fields for the messages that concern the data records of that class. To do so, *field* tags with a respective *name*- and *key* tag are used. The *key* tag defines whether the field is a key field. Example:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<config xmlns="http://xml.contact.de/schema/xgwschema/1.0">
    <response_object type="part">
        <field name="TEILE_NR" key="True"/>
        <field name="TEILE_INDEX" key="True"/>
    </response_object>
</config>
```

If this configuration is used, then respective entries have to be created in the mapping configuration of the attribute for mapping these field names to CONTACT Elements attributes (see Catalyst administration manual). In the example, the field names *TEILE_NR* and *TEILE_INDEX* for the class *part* would be mapped to the CONTACT Elements attributes ‘ part number‘ and *t_index*. If such a configuration has been set up, the object of an *info* message can be identified correspondingly without *id*:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<response xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object type="part"
            action="info">
        <field name="TEILE_NR">127001</field>
        <field name="TEILE_INDEX">01</field>
        <field name="DESCRIPTION_EN">component</field>
        <field name="BRUTTO_GEWICHT">12,7</field>
    </object>
</response>
```

The ERP system often does not use version or revision numbers, so that here only one number identifies the data record. Under such circumstances, from CONTACT Elements you have to initialize a version number or, in case of a modification, you have to determine the “correct” version.

For this purpose, first there is an option, addressing the most frequent case, to transfer part masters with an original version (empty version field). In such a case, the ERP system can send a *commit* variant with the action *pcommit* and only transfer the part number in the message.

This option makes it possible to transfer data from the ERP system for the purpose of creating within CONTACT Elements.

For use cases in which the ERP system does not transfer its own data, but instead replies asynchronously to transferred data and no longer knows the ID at the time the reply was generated, the following options are available.

If it has been verified that at all times, only exactly one version of a data record is transferred from CONTACT Elements to the ERP system (for example, through a workflow configuration, which allows only one released version and ensures that all other versions are either blocked or drafts), then that version can automatically be determined and added to the incoming reply message. For this purpose, the CAD configuration switch “PPS Teileindex” has to be set to “True”. Then, when a reply message for parts or documents with the configuration of the key fields as referenced above is received without a version field, the version is determined that is marked for synchronization and had been transferred to the ERP system previously. This version is added to the message. Despite the name of the CAD configuration switch, this method also works for documents.

Another option for determining the ID is to implement a transformation module. If they do not administer the version information necessary for the PDM/PLM systems, ERP systems often also have their own message formats. Under such circumstances, a transformation into the above-mentioned CDBXML format has to take place. During this step, missing data such as the version information can also be determined, which can then be added to the resulting message. Of course, defining a clearly formulated rule for determining the version is required. If this cannot be provided, the ERP system has to supply a version attribute or an ID.

If a version has been determined based on rules, in addition to the option of inserting missing key fields into the CDBXML message and flagging the configuration as a key during the transformation step, it also possible to compute the ID and paste it as an attribute into the <object> tag.

The ID is composed of class names and key fields and values:

```
from cs.xml.SharedObject import SharedID

class_name = "part"
id = SharedID.sharedID(class_name, {"teilenummer": "000123", "t_index": "00"})
```

For program-controlled generation of CDBXML messages, see the *CONTACT Catalyst EAI Gateway* programming manual.

Identification of data in response messages

If the message formats described in the previous section are used to send messages of type *commit*, *info* or *cominfo* to CONTACT Elements addressing existing objects, an adjacent ID is generated while processing these messages. This ID can be used to identify the corresponding object in CONTACT Elements. If responses are configured to be sent (e.g. when using web services), all of the *"object"* tags within these responses are outfitted with these IDs. This results in generating answers as shown in *Beispiel für die XML-Repräsentation von Rückmeldungen* (page 37). This makes it possible to identify the objects modified by processing the messages. Besides that, it enables writing a custom destination selecting the processed object within CONTACT Elements and, using this data, inserting additional fields into the answer.

If an *Info* message refers to an object to be created newly in CONTACT Elements, this function is crucial—especially if number generation is configured to take place within CONTACT Elements. In this case, the external system probably has no means of identifying the new object. Only if a response is given containing this ID, a subsequent identification is made possible.

Here as an example a code snippet is shown inserting the corresponding key fields into the response:

```
from cs.xml import xmllib, Destinations

class KeyDestination (Destinations.Destination):
    def output(self, tree):
        xml = xmllib.xml(namespace)
        for obj in tree:
            id = SharedID.parsedID(obj.get("id"))
            obj.append(xml.element("class", id.type())
            for k in id.keys():
                obj.append(xml.element("key", id.key(k), name=k))
        super(KeyDestination, self).output(tree)
```

2.4.1.4 Additional details on data transfer

Further configuration options can be used to specify a number of other details about the form of data transfer, which have less to do with the general configuration of the *CONTACT Catalyst EAI Gateway*, but more with the properties of the data exchange with an ERP system. These details are summarized in the following chapter.

Synchronization of part data

Synchronizing article characteristics and BOMs are two common special aspects when synchronizing part data. They are supported with additional configuration options.

Note: the classification discussed below is the old characteristics bars. For Universal Classification applications, please contact your account manager.

If a part classified using a generic group is prepared for the transfer, then the characteristics are provided like additional attributes for attribute configuration. For each characteristic, the name it is configured under in CONTACT Elements is provided a "**sml_**" prefix in this process. This means a "Diameter" characteristic could be addressed by an attribute name of *sml_Durchmesser* in the attribute configuration.

If assembly parts are transferred together with their BOM, then you frequently need to take into consideration that the component parts addressed in the BOM positions have to be announced in the ERP system beforehand.

This can be achieved by using configurations to ensure that the assemblies can be converted to a share status only if their components are synchronized with the ERP system.

One alternative is the ability to formulate relationship dependencies in the ERP configuration in Catalyst (see Catalyst administration manual). If a dependency such as this is defined, then the system determines all of the objects described by this rule before the synchronization and their synchronization status is checked. The export of the assembly part is then canceled with an error message ("All dependent objects have not yet been synchronized").

This configuration option can be used the same way for other relationships.

Synchronizing document data and files

A speciality of the transfer of document data is that the associated files are also to be transferred here. Usually links are embedded in the transmitted document data sets for this, so the desired files can be obtained over the REST API. Alternatively, files can be embedded in the message to be sent or provided within a dedicated exchange directory.

This procedure can be adapted to other requirements using custom destination modules. For example, there is often a requirement to embed files in the message contents or to transfer them to the target system using other protocols. This can be achieved using your own destination modules that read in file contents and embed them in the message encoded in Base64 or that transfer files to the ERP server using services such as FTP and insert a suitable link into the master data message.

As with all master data, the configuration is imported using a <object> tag (with type="document" in this case). It can also have a *single_file* attribute. If this attribute is set, then the system checks that all of the subsequently listed file types (see below) exist in the specified order and only the first one found is checked out and inserted into the message.

Inside the <object> tag, in addition to the usual <field> and <relation> tags, <export_file> tags can be inserted. Each of these tags defines a file to be transferred. Attributes are *path* (defines the export directory) and *format* (defines the format and can be used if exactly one file of a certain format exists. Usually converted formats like pdf or jpg are transferred as view formats). If a file is defined without specifying the format, then this will cause the respective primary file to be transferred.

Each of the exported files is stored in the export directory under a unique filename generated from a UUID. This ensures that several different files with the same name do not overwrite each other if they are checked out at the same time. Instances of different files with the same name can occur if the file name does not contain any version information but there are several versions of the associated document.

Here is an example of a schema for document export:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0">
    <object type="document"
            datasource="cs.xml.SharedObject.Factory"
            single_file="False">
```

(continues on next page)

(continued from previous page)

```
        <field name="DOC_NR"/>
        <field name="DOC_VERS"/>
        <field name="DOC_DESCR"/>
        <export_file path="%(export_dir)s" format="pdf"/>
        <export_file path="%(export_dir)s" format="jpg"/>
    </object>
</schema>
```

Here is an example of a resulting message (the export directory is defined as `"C:/temp/"` here). The assigned files are listed in *filename* tags. In each case, the content of the tag is formed from the path and name of the file in the export directory, where it is available for additional transfer steps or can be accessed directly by the ERP system. In order to be able to identify the file, additional attributes are stored in the *filename* tag. The format is stored under *type*. The *docid* attribute contains the *cdb_object_id* of the document; *oid* contains the corresponding *cdb_object_id* for the *cdb_file* data record. You can use these two attributes to identify and assign any file. The original filename is also available in the *filename* attribute.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns:ns0="http://xml.contact.de/schema/xmlgateway/1.0">
    <object type="document"
                id="706172743a[...]35f696e6465783d">
        <field name="DOC_NR">127001</field>
        <field name="DOC_VERS">01</field>
        <field name="DOC_DESCR">component</field>
        <filename type="pdf"
                  docid="35f696e6465783d"
                  oid="706172743a"
                  filename="127001-01.pdf">
            C:/temp/706172743a35f696e6465783d.pdf
        </filename>
    </object>
</schema>
```

Example: Several PDF files are assigned to one document. One of them is an export from a CAD document; the others contain additional documentation. The file derived from the CAD document is intended to be transferred. The other PDF files are not intended to be transferred; they are removed from the resulting message. A filter that the gateway applies to each element of the message is implemented to accomplish this. The filter checks all of the *filename* elements, removes all but the desired one and renames that file. Files can be selected for synchronization using any criteria by adjusting the filter conditions of this example.

```
from cs.xml import SchemaParser, Filter
from cdb import sqlapi


def filter_condition(doc_id, file_id):
    # select primary file
    cad = sqlapi.RecordSet2("cdb_file", "cdbf_object_id='%s' and
        cdbf_primary='1'" % doc_id)[0]
    # select pdf file
    my_file = sqlapi.RecordSet2("cdb_file", "cdb_object_id='%s'" % file_id)[0]
    # check if pdf is derived from primary
    return (my_file.cdbf_derived_from == cad.cdb_object_id)


class Myfilter (Filter.Filter):

    def OnChild(self, element):
        if element.tag != SchemaParser.XGW_filename:
            return element
        else:
            file_type = element.get("type")
```

(continues on next page)

(continued from previous page)

```
            doc_id = element.get("docid")
            file_id = element.get("oid")
            filename = element.get("filename")
            if filter_condition(doc_id, file_id):
                return element  # nicht ausfiltern
            else:
                return None     # ausfiltern
```

Response behavior of the ERP system

After sending a message to an ERP system, CONTACT Elements normally assumes that a response is sent by the ERP system as a result. If a detail-based data exchange has been agreed, two jobs are typically defined to implement this at regular intervals: The first job attempts to find response files from the ERP system, the subsequent one carries out the data export from CONTACT Elements.

The rationale for this order is due to the fact that data records where processing has not yet been confirmed by the ERP system are exported again at each interval. If the interval is set to a value such as 10 minutes, then a file with the exported data is generated when starting the interface for the first time (the initial job run to search for ERP response files cannot succeed at this point yet).

After the interval elapses, the job for searching for a response file is run again. If the ERP system has since acknowledged processing the file exported by CONTACT Elements, then all of the acknowledged data records are marked as synchronized. The job run immediately afterwards for exporting data will now only export newly released data.

If the jobs were defined in the opposite order, then all of the data that was already been exported would be exported again first. This could then be acknowledged immediately due to reading in the waiting response, resulting in duplicating the transfer.

If the ERP system does not acknowledge the data transfer operations within the configured intervals, then the system assumes there is an error and the data is transferred again. This behavior can be changed in several ways.

When the "PPS Keine Antwort" CAD configuration switch is set to "TRUE", exporting the message for data transfer is immediately taken as successful synchronization. The exported data is marked as synchronized immediately without waiting for a response from the ERP system.

Alternatively, the User Exit component *pps_share* can be implemented (see Catalyst administration manual). In combination with the option of defining a data attribute as a transfer counter, you can implement behavior that omits a data record from further transfer after a certain number of attempts or after receiving an error message and sends an automated notification to an administrator instead.

Synchronous data exchange

You frequently have the possibility of communicating with the ERP system more directly than when using asynchronous data exchange. If objects in the ERP system can be modified using web service interfaces, this can be initiated using the *HTTPDestination* destination module. If the ERP system provides other proprietary interfaces, then you can frequently implement your own destination module that configures the parameters of this interface with the exported data.

Cases like this are supported by the option to export data records in a step-by-step process. This function is enabled by setting the "PPS Single Object Transfer" CAD configuration switch to "TRUE". This generates a message for each individual object to be exported instead of packaging them all into one shared message. By implementing a custom destination module, the feedback for each of these messages is booked in CONTACT Elements synchronously (see *Custom destination modules* (page 64)).

2.4.2 Web services

This chapter describes the option of using the *CONTACT Catalyst EAI Gateway* for implementing web services, i.e. for implementing services provided to external systems using standard protocols.

2.4.2.1 Overview

Two properties of the *CONTACT Catalyst EAI Gateway* are combined to implement these kinds of functions: The option of using events entering through web endpoints triggering jobs by receiving HTTP messages and the option of implementing your own message formats and functions.

This allows the implementation of use cases such as:

- Receiving messages from ERP systems (or ESB middlewares etc)
- Implementing reports or data searches where the results are transferred in HTML, XML or JSON format to a portal
- Providing functions external systems can use to create data in CONTACT Elements or trigger processes.

In contrast to REST API, which allows individual objects to be queried or modified directly, the message-based web services described here are used to enable encapsulated system communications without having to disclose to the external systems detailed information about the individual elements, attributes, etc. to be modified in CONTACT Elements on order to process the messages.

In particular, by using the JSON destinations, communications based on appropriate JSON protocols can also be implemented.

2.4.2.2 Event-triggered jobs

Event-triggered jobs are defined in the job configuration in the form of job groups where the run interval is set to 0:

```
XGW_JOBS = [
    { # job list executed in 10 min intervals
        ...
    },
    { # job configuration for http input
        'job_interval': 0,
        'port': 7488,
        'secret': 'cs.platform/customer/test/endpoints/webservice',
        'group_id': 'http_jobs',
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.http',
        'input_dir': None,
        'output_dir': None,
        'processed_dir': '%(CADDOK_BASE)s/etc/xmlgateway.http/processed',
        'input_files': None,
        'job_list':
        [
            {
                'job_id': 'Update from ERP',
                'path': '/partupdate',
                'schema_files': [ 'key_fields.xgs', 'site.xgs',],
            },
            {
                'job_id': 'http view file',
                'path': '/viewfile',
                'custom_xml_tags': 'xgw.CustTags',
                'schema_files': [ 'viewfile.xgs' ],
                'work_dir': '%(CADDOK_BASE)s/tmp/'
            },
```

(continues on next page)

(continued from previous page)

```
            {
                'job_id': 'http state change',
                'path': '/statechange',
                'custom_xml_tags': 'xgw.statechange',
                'schema_files': [ 'site.xgs', 'statechange.xgs' ]
            }
        ]
    }
]
```

In addition to setting the *interval* attribute to 0, it is notable that the directory attributes *input_dir* and *output_dir* as well as the *input_files* attribute have been set to *None*. Since the message exchange is not file-based, these attributes can be left blank. Nevertheless, it can make sense to fill *processed_dir* with content in order to be able to log incoming and outgoing messages as well.

Two attributes are important in the configuration sections of each job: *path* and *port*. The path attribute goes into the URL of the endpoint through which the job can be triggered. The entire URL is composed of the hostname of the server or, if multiple hostnames are used, the hostname specified in the service configuration, the port specified in the port attribute, and the path. If the *CONTACT Catalyst EAI Gateway* is installed on a host named *xml-service.local* and defined as port 7488, the first job defined in the example (Update from ERP) can be addressed at the URL *http://xml-service.local:7488/partupdate*.

In contrast to *path*, the *port* attribute can only be used when configuring a job group. I.e., the paths of all jobs in a group are grouped under a common port and must be unique accordingly.

If the CONTACT Elements instance uses secure connections via SSL/TLS (see CONTACT Elements administration handbook), then the endpoints defined here are also encrypted. In the example shown above the URI *https://xml-service.local:7488/partupdate* should be used then. The options described in the CONTACT Elements administration handbook (*ssl_certificate_file*, *ssl_ca_certificate*, *ssl_certificate_key_file*, *ssl_dhparam*) can also be used for the service *cs.xml.xmlgateway.XMLGatewayService*, if it should use it's own certificates. If these options are not defined explicitely for this service, it will use the global instance certificates.

If access to such a function is to be secured by *Basic Authentication*, then an attribute *secret* can be specified in the job configuration. This can be used to address a secret stored in a wallet as described in the CONTACT Elements administration manual. This should be composed as a colon separated username and password. When the URI is accessed, the data is then retrieved with reference to the *CDBXML Gateway Services* realm. The use of SSL/TLS implies the use of *Basic Authentication*. The use of a secret is not optional then.

In previous versions of this manual, synchronous and asynchronous jobs were referred to. This distinction is no longer made; accordingly, the *immediate* attribute is now ineffective. In previous versions of *CONTACT Catalyst EAI Gateway*, two worker processes were always used, which were responsible for synchronous and asynchronous job processing. Among other things, this had to do with the internal communication of the gateway to CONTACT Elements processes, which had limited multithreading capability. In the current version, an independent worker process is always used for each job group. Asynchronous or parallel processing can therefore be achieved by distributing web service jobs to job groups accordingly. It should be noted that the end points of different job groups must be addressed via their respective different ports. In chapter *Load balancing with multiple web service workers* (page 83) is an example of a (reverse) proxy configuration of the Apache server to provide the ports of different job groups under a common endpoint. In this way, it is also possible to scale the function of a job by using multiple worker processes.

2.4.2.3 Jobs for simple message processing

First, an example of the job configuration for a simple job:

```
...
{
    'job_id': 'Response from ERP',
    'path': '/response',
    'schema_files': [ 'site.xgs',]
```

(continues on next page)

(continued from previous page)

```
},
...
```

This configuration provides only one endpoint where a message can be passed to the *CONTACT Catalyst EAI Gateway* - in this case the response to an ERP data synchronization. The message is sent to the resulting URI by an HTTP/POST command.

The function to be executed is determined here exclusively by the message content. I.e. such a job configuration could be used also for the delivery of other than the intended messages. However, it usually provides the framework for a special application (e.g. by specifying a coupling configuration through *site.xgs* and by specifying transformation steps. The endpoint configured with this should therefore also only be used for the intended messages. If required, this can also be ensured technically by using an appropriate transformation module.

An incoming message at an end point like this is acknowledged by a "200 OK" HTTP status code immediately.

2.4.2.4 Complex Jobs

Beyond that further possibilities are available. In addition to the message content, further parameters can be passed, the function to be executed can be specified independently of the message, and an incoming message can also be omitted entirely. This is useful, for example, if only one action is to be triggered in CONTACT Elements or if it is sufficient to pass parameters encoded in the HTTP call.

You can specify the function to be activated in several ways if it is not defined by an incoming message, including the two following options. As one option, a pseudo message featuring the desired function in its top-level tag and stored in a file in the configuration directory can be defined in a *schema_files* job parameter. As another option, the function can be specified directly in the form of a Python function using the *callable* job parameter.

Parameters can be passed to a job by appending their names and values to the URI separated by a question mark according to the HTTP standard. A Python dictionary containing these parameters can be retrieved from the job options using the *job_options.http_arguments()* function.

In addition, path elements beyond the path defined in the job configuration can be passed when calling the URI. For example, the job configuration *"path": "my_func"* is activated by a URI *http://localhost:7488/my_func/*. If the associated function is called by a call from *http://localhost:7488/my_func/opt1/opt2/*, the additional *opt1/opt2* portion of the path can be queried and evaluated using the *job_options.http_path()* function.

The input of the call (a message or a file uploaded using HTTP/POST or HTTP/PUT) can be queried using the *job_options.http_input()* function. Finally, the entire request is available in the form of an object of *twisted.web.http.Request* type using the *job_options.http_request()* function.

The implementation of a function can return various information to the caller. The *job_options.set_status_code(code, message=None)* function can be used to set an HTTP status code and optionally a status text. This information will be reported back to the caller by the HTTP protocol after the function has been executed (example: 404 - "not found").

The *job_options.set_response(contents)* function can be used to directly specify the contents of a response message to be returned. This can be useful for small responses, but normally the mechanisms of the *CONTACT Catalyst EAI Gateway* (ResponseDestination) should be used instead. The *job_options.set_content_type(type)* function can be used to specify the MIME type of the response if something other than *text/xml* is supplied.

Often the content of the response of a function call can be generated by standard functions without having to implement custom functions for it. The standard defined functions *Schema* (e.g. together with a query data source) and *GetFile* cover e.g. database queries and file queries (see chapter *Message processing* (page 13)).

In case of a schema function the format of the response message is defined by the schema definition, in case of GetFile it is the content of a file. Now it must be defined where this message or the file content should go. This is defined as usual by a destination configuration. In this use case, however, the *CONTACT Catalyst EAI Gateway* is not an actuator that sends a message to a configurable recipient, but a reactor that sends a response back to its caller. This is exactly what can be achieved by using the *ResponseDestination* destination module. This module only works in job configurations that are triggered by an HTTP call and has no effect otherwise. The only

parameter of this module is the optional *content-type* attribute, which should only be configured if the response has its own format or is transformed by upstream destination modules.

If the desired function cannot be configured by a default function, a custom function has to be implemented, as described above. Such a function provides any degree of freedom when generating responses. For example, an XML tree can be built and transformed by a series of destination components. This is shown in the following example, which implements a variant of the classic "Hello, World".

The example is called up by sending an HTTP/GET request to the URI *http://localhost:7488/greet?name=some_name*. A response should be made by an XML message with the form

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<greetings xmlns="http://xml.company.com/schema/greetings/1.0">
    <hello name='some_name'>
       some_name
    </hello>
</greetings>
```

The name transferred as a parameter is output as both a tag attribute and tag content for the sake of completeness of the example.

Here is the job definition:

```
...
{
    'job_id': 'hello, world',
    'path': '/greet',
    'callable': xml.greet,
    'schema_files': [ 'destinations.xgs' ]
},
...
```

The implementation is in the xml.py Python module. It takes advantage of the *xmllib* module to prepare an XML tree for the response output. Finally, the response is passed to the destination modules by the *job_options.output()* function.

```
# xml.py

def greet(options):
    from cs.xml import xmllib

    who = "stranger"
    args = options.http_arguments()
    if args.has_key("name"):
        who = args["name]

    xml = xmllib.xml(namespace="http://xml.company.com/schema/greetings/1.0")
    root = xml.element("greetings")
    hello = xml.element("hello", who, name=who)
    root.append(hello)
    return options.output(xml.tree(root))
```

The *xmllib* module builds up an XML structure. Accordingly, an *XMLDestination* transforming the structure into an XML stream is used as the first destination module. This is followed by the *ResponseDestination* mentioned before, which sends the XML stream to the caller as a response.

```
<!-- destinations.xgs -->
<destination class="cs.xml.Destinations.XMLDestination" />
<destination class="cs.xml.Destinations.ResponseDestination" />
```

Readers who are interested can insert another transformation step for converting the output to HTML can be inserted here.

2.4.2.5 Advanced Configuration

In addition to the configuration options described so far, you can also store your own implementation of a `twisted.web.resource.resource` class. This allows e.g. the implementation of a custom `render()` method, the use of site wrappers for other authentication methods or even the use of protocol implementations from the Python environment. One example is the SOAP implementation `spyne`, which can be included as a `twisted` site (see below).

For configuration, a web service job group must be created as described above. In addition to the `job_interval` (0) and `port` attributes, the `resource` attribute must also be defined. The value must be a string that defines the FQPN of a Python function. This function is called as site factory with the arguments `job` and `args`:

```
def factory(job, args):
    return resource()
```

Here `job` is a dictionary containing the job definition, `args` is a dictionary describing the context of the worker. It contains this data:

service_id The service name of the service for which this worker is configured. Normally this is `cs.xml.xmlgateway.XMLGatewayService`. Only if additional services are configured (e.g. to connect different ERP systems), other names are passed here.

group_id The name of the job group that will be executed by this worker.

port The port number where this worker provides its endpoint.

username If basic authentication is configured, the username by which a default login is possible is passed here. Custom implementations can also implement their own authentication scheme.

password If basic authentication is configured, the password by which a default login is possible is passed here. Custom implementations can also implement their own authentication scheme.

configfile Name and path of the configuration file for this service.

2.4.2.6 Example: SOAP

The Python module `spyne` implements a SOAP service and can be integrated into *CONTACT Catalyst EAI Gateway*. It can be installed using `pip install spyne`.

The configuration basically consists of a job group containing a `resource` attribute:

```
XGW_JOBS = [
{
    'group_id': 'webservice',
    'job_interval': 0,
    'port': 7488,
    'conf_dir': u'%(CADDOK_BASE)s/etc/xmlgateway',
    'resource': 'webservice.factory',
    'job_list': [
        {
            'job_id': 'soap',
            'active': True
        }
    ]
}]
```

A corresponding factory method should be implemented in a client module (here `webservice.py`):

```
import spyne
from spyne import rpc
from spyne.protocol import soap
from spyne.server import twisted
```

(continues on next page)

(continued from previous page)

```
def factory(job, args):
    return twisted.TwistedWebResource(application)
```

A corresponding application that provides the example service `say_hello` can then be implemented like this:

```
class HelloWorldService(spyne.ServiceBase):
    @rpc(spyne.Unicode, spyne.Integer, _returns=spyne.Iterable(spyne.Unicode))
    def say_hello(ctx, name, times):
        for i in range(times):
            yield 'Hello, %s' % name


application = spyne.Application([HelloWorldService],
    tns='http://your.namespace.local',
    in_protocol=soap.Soap11(validator='lxml'),
    out_protocol=soap.Soap11()
)
```

After starting the *CONTACT Catalyst EAI Gateway* the *WSDL* can now be retrieved and the service function consumed e.g. using the Python module `zeep`:

```
import zeep

client = zeep.Client("http://localhost:7488/?wsdl")
response = client.service.say_hello("stranger", 10)
print response
```

2.4.3 Export and Import of JSON-based formats

The `cs.xml.json_destinations` module provides some building blocks to work with JSON-based data representations within the *CONTACT Catalyst EAI Gateway*.

The basis for the conversion between data structures and their JSON representation is the Python package `json`. The pipeline modules *cs.xml.json_destinations.JSONDestination* (page 23) and *cs.xml.json_destinations.JSONTransformation* (page 26) provide the possibility to use the functionality of that module within the pipelines. For more details on converting data structures to JSON, see https://docs.python.org/2/library/json.html#encoders-and-decoders. As input or output, these modules get Python data structures, which are converted to or from JSON by the module. Additional pipelines allow to create corresponding Python data structures from *CONTACT Catalyst EAI Gateway* messages or to convert these data structures to *CONTACT Catalyst EAI Gateway* messages.

E.g. to use the components `custom.MyFormatDestination` and `custom.MyFormatTransformation` for transforming XML-Gateway messages, the following pipelines would be configured:

```
<destination class="custom.MyFormatDestination" />
<destination class="cs.xml.json_destinations.JSONDestination" />

<transformation class="custom.MyFormatTransformation" />
<transformation class="cs.xml.json_destinations.JSONTransformation" />
```

In addition to these basic building blocks, the module also already provides a destination and a transformation for each of the formats explained below. The format presented in *The XGW-JSON Format* (page 49) is a format that corresponds as closely as possible to the format of XML-based gateway messages. The format presented in *The ReST-based Format* (page 50), on the other hand, is based on REST API.

2.4.3.1 The XGW-JSON Format

The `ToXgwPyDataDestination/FromXgwPyDataDestination` blocks allow the generation of JSON-based messages, which follow the format of *CONTACT Catalyst EAI Gateway* messages in their structure.

Format of the object list

These messages contain one JSON object, which defines the attribute `objects`. `objects` is a list of CONTACT Elements objects, each of which is itself represented by a JSON object.

```
{ "objects": [ { ... }, ... ] }
```

These objects contain the following attributes:

- `id`: The shared ID assigned to the object by the *CONTACT Catalyst EAI Gateway* (JSON string).
- `type`: The type of the object, which usually corresponds to the CONTACT Elements classname (JSON String).
- `fields`: A list of JSON objects, which correspond to the synchronized fields of the object
- `related`: A list of JSON objects, each of which represents a type of relation being synchronized

Additionally, response elements being sent to CONTACT Elements should contain the following elements (cf. *ERP system replies* (page 36)).

- `action`: A string that represents the action to be executed (`info`, `commit`, `cominfo`, `error`)
- If the specified action is `error`, this attribute can be used to provide a message

Response messages from CONTACT Elements to the connected system require this additional field:

- `result`: JSON object, which holds the result of the executed CONTACT Elements operation.

Format of fields

Each field in the list of fields is a JSON object containing the attributes `name` and `value`:

```
"fields": [
  {
    "name": "TEILE_NR",
    "value": "000001"
  },
  ...
],
```

Format of relations

Relations contain the attribute `type` (type of the relation), as well as a list of objects, with which the synchronized is related by this type of relation.

```
"related": [
  {
    "type": "bom",
    "objects": [
         ...
    ]
  },
  ...
]
```

Format of results

Contains the attributes `code` and `message`.

```
"result": {
  "code": 0
  "message": "Action error (Error text 'Key Error'): Result: OK."
}
```

2.4.3.2 The ReST-based Format

The `ToRestPyDataDestination/FromRestPyDataDestination` blocks allow the generation of JSON-based messages, which follow the format of REST API in their structure.

The default behavior is to include the gateway-specific fields with a "xgw:" prefix. Fields are also included directly as attributes into the object. This behavior can be customized by adding the following two attributes to the destination.

- `content`: the declaration `content="raw"` leaves out all meta attributes and puts in only the configured fields.
- `format`: when setting the CAD configuration switch *PPS Single Object Transfer*, each message contains only a single object. According to this, the JSON structure's frame, the attribute `objects` und the contained list is not needed. Declaring `format="single"` allows to only include the JSON structure of the next object to be synchronized.

The list of relations is included here following the structure of REST API, except that objects are contained directly within the list of relations.

```
{
  "objects": [
     {
          "xgw:type": "part",
          "xgw:id": "...",
          "xgw:action": "",
          "xgw:result": {
              "code": "0",
              "message": "Action error (Error text 'Key Error'): Result: OK."
          }

          "system:relships": {
              "relships": {
                  "bom_item": [
                      {
                          "BAUGRUPPE": "000001",
                          "xgw:type": "bom_item",
                          "xgw:id": "..."
                          "system:relships": {
                              "relships": {}
                          },
                          "EINZELTEIL": "000000"
                      }
                  ]
              }
          },

          "COMBINED": "000001_"
          "TIX": "",
          "TNR": "000001",
     }
  ]
}
```

When defining the destination like this:

```
<destination class="cs.xml.json_destinations.ToRestPyDataDestination"
             format="single"
             content="raw"/>
```

the output of the example would look like this:

```
{
    "COMBINED": "000001_"
    "TIX": "",
    "TNR": "000001",
}
```

Now, choosing adjacent field names allows to define a JSON output which can directly be passed into a consumer's REST API via `HTTPDestination`.

CHAPTER 3

Advanced options for adaptation

The following chapter describes a number of ways to influence the operation of *CONTACT Catalyst EAI Gateway* beyond the configurational ways presented. This is made possible by extensions implemented in PowerScript, which can be integrated into the processes at various points.

Filters can be used to modify the data that is output. These filters can be used at defined places in the data schema in order to filter out, modify or add data based on self-defined criteria.

Furthermore, separate data sources can be defined in PowerScript to specify data for the export.

If in the *CONTACT Catalyst EAI Gateway* own *functions* or applications are to be implemented, this can be done by implementation of appropriate extensions. These extensions can define and process own message formats.

Finally, schema files can be configured so that they can be used more flexibly.

By implementing separate destination and transformation components, separate transformation steps can be implemented for incoming messages and transfer destinations can be implemented for outgoing messages.

3.1 Filters

Filters provide the option of further modifying the output of exported objects by calling up program code in PowerScript for each object to be exported. However, filters can also be used to execute actions based on the individual exported objects. Sequences of filters can be defined; this leads to the respective filters then being sequentially applied to the respective object.

For example, a simple filter that modifies the export so that the exported ID number is provided with a prefix "XGW-SYNC-ID-". Here is the schema file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="target.xml"/>
    <object type="part" datasource="cs.xml.SharedObject.Factory">
        <!-- ****************************** -->
        <filter class="xml.MyFilter.PrefixFilter"/>
        <!-- ****************************** -->
        <field name="IDENTNR"/>
        <field name="GEWICHT"/>
```

(continues on next page)

(continued from previous page)

```
    </object>
</schema>
```

The filter is located under the <object> element. The "class" attribute indicates the *full qualified python name* of the PowerScript module and the class from which the program code for the filter comes. In this case, the Python search path (for example, in `instance/site-packages`) has to include a `xml/MyFilter.py` module by defining a *PrefixFilter* class. Here is an example for the corresponding PowerScript module:

```
#!/usr/bin/env python
# File:     MyFilter.py
# Author:   caddok
# Creation: 09.03.05
# Purpose:  Versieht IDENTNR-Werte mit einem Präfix

import re
from cs.xml import Filter
from cs.xml.SchemaParser import XGW_field, XGW_object, XGW_related, make_field

class PrefixFilter(Filter.Filter):
    """
    Versieht Objekte mit einem Präfix

    Der Filter versieht den Inhalt von <field>-Elementen mit dem
    Präfix, soweit das Attribut "name" des Elements den Wert
    "IDENTNR" hat.
    """

    def OnChild(self, node):
        if node.tag == XGW_field:
            if node.get('name') == "IDENTNR":
                if node.text:
                    node.text = "XGW-SYNC-ID-" + node.text
        return node
```

Here is an example for the resulting output:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object id="706172743a7[...]5723d313031373635373a745f696e6465783d"
                type="part">
        <field name="IDENTNR">XGW-SYNC-ID-1017657</field>
        <field name="GEWICHT">0</field>
    </object>
    <object id="706172743a74[...]723d313231343436393a745f696e6465783d"
                type="part">
        <field name="IDENTNR">XGW-SYNC-ID-1214469</field>
        <field name="GEWICHT">0</ns1:field>
    </object>
</schema>
```

3.1.1 Implementation of custom filters

Custom filters are implemented by defining a subclass of *Filter.Filter*. *Filter.Filter* defines six methods. Three methods permit access during the generation of XML elements:

- `OnStart(self, node)` This method is applied immediately when starting to process an element.
- `OnChild(self, node)` This method is applied for every child element.
- `OnEnd(self, node)` This method is applied at the end of processing the element.

Additionally, there are three expanded methods in which the source object is also available:

- `OnStartExt(self, node, transferrable)` This method is applied immediately when starting to process an element.
- `OnChildExt(self, node, transferrable)` This method is applied for every child element.
- `OnEndExt(self, node, transferrable)` This method is applied at the end of the processing of the element.

Here *transferrable* is a Python object of type *TalkObjectFacade* that represents the record to be transferred from CONTACT Elements and provides a simple interface. With *type_name()* the object class of the dataset can be queried. With *items()* a list of tuples is queried. Each tuple contains the attribute name of an EAI attribute defined for transfer (see Catalyst administration manual *CONTACT Catalyst for ERP*) and the associated value. The value represents the converted format for transfer to the ERP system. The *_get(cdb_attribute)* method can be used to retrieve the unchanged attribute values from CONTACT Elements. The attribute parameter used here is not the EAI attribute name, but the name of the database attribute in CONTACT Elements.

Thus these expanded methods can be used both to define filter criteria based on further properties of the transferred objects and to enrich the output with more information.

When building up the XML structure, each of these methods is called up once or multiple times for each of the inserted elements. Each element is regarded at first as a potential container for embedded elements. For a new element, therefore, the *OnStart* method is called up at first (or *OnStartExt*). Then any subelements are exported. The same procedure is carried out for each of the subelements. Then the *OnEnd*/*OnEndExt* method is called up. If the object was generated as a subelement of a higher-level element, the *OnChild*/*OnChildExt* method is called up now for this element before it is embedded into the higher-level element. Accordingly, the *OnStart*/*OnStartExt* method was called up beforehand for the higher-level element.

This makes it possible to access any level while generating the XML tree.

An example is the exporting of a data record object (object tag), which contains multiple field tags. When such an object tag is generated, *OnStart(node.tag == XGW_object)* is called up at first; that is, it is transferred to a data record whose type can be identified by *node.tag* as object (*XGW_object*). Now its individual data fields are exported. This time *OnStart(node.tag == XGW_field)* is called up for each field. Since a field has no subelements, this is immediately followed by calling up *OnEnd(node.tag == XGW_field)*. Now the field is appended to the object and, accordingly, *OnChild(node.tag == XGW_field)* is called up at first. Once all fields are exported this way, then *OnEnd(node.tag == XGW_object)* is called up. The object generated this way can be a subelement of a relation, so that it is followed by calling up *OnChild(node.tag == XGW_object)* accordingly, etc.

All three methods have in common that they must return an object of type *xmllib.element*, usually the argument, or *None*. If *None* is returned, the corresponding element is filtered out of the output. In addition, elements (or even whole subtrees of the XML structure) can be generated and returned. This replaces the originally intended element in the resulting message with the returned one. In this way, filters can be used to both filter out elements and insert custom elements.

The three extended methods get in addition to the *node* element of type *xmllib.element* a parameter *transferrable* of type *cs.xml.SharedObjec.TalkObjectFacade*. In the same way they must return a tuple (node, transferrable) as return value. Here, a tuple (*None*, transferrable) must be returned accordingly to filter out the element.

The contents of the individual elements can be accessed through their members. Besides *node.tag* for the tag name this is *node.text* for the content text. The attributes can be retrieved by *node.get('attributname')* (see also Python module *cs.xml.xmllib*).

One (or more) of these methods can now be overwritten to implement the function for the custom filter. For the prefix filter, the *OnChild* method was overwritten. The first line contains the condition

```
if node.tag == XGW_field:
```

Thus assurance is provided here that the filter operates only at the level of <field> elements, since otherwise *node* is returned unchanged. The condition in the next line

```
if node.get('name') == "IDENTNR":
```

specifies that only elements whose “name” attribute has the value “IDENT NO” should be processed. The values of attributes of *node* should generally be accessed via *node.get()*, since this type of access does not cause an error if the element has no attribute. In the line:

```
if node.text:
```

there is a check to see whether the element contains text. The text contained in an element is stored in the *text* attribute. This is modified in the next line:

```
node.text = "XGW-SYNC-ID-" + node.text
```

, so that the prefix is pasted into the text. In conclusion,

```
return node
```

returns the element. In this regard it is important for the `return` statement to be at the top indentation level within the method, so that it is always run through, even if the overlying conditions are not fulfilled. This ensures that this element is not filtered out by accident.

Now we present a more complicated example of filters to clarify the principle. This time parts will be exported with documents again, but with these additional requirements:

1. Part masters will receive an additional <field> element, which specifies the number of documents.
2. Documents will receive an additional <field> element, which specifies the sequence number within the exported list of drawings for this document.
3. All text is to be converted into uppercase letters.

The format mentioned requires two filters:

1. One filter that counts the documents to determine the sequence number and total number. We call this DocCountFilter.
2. One filter that converts the text within the fields into uppercase letters; it is called UppercaseFilter.

Both filters are required for the <object> elements at the top node and for the corresponding <related> elements, therefore, they have to be received into the schema file at both places. Here is the corresponding schema file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="target.xml"/>
    <object type="part"
            datasource="cs.xml.SharedObject.Factory">
        <filter class="MyFilter.DocCountFilter"/>
        <filter class="MyFilter.UppercaseFilter"/>
        <field name="IDENTNR"/>
        <field name="GEWICHT"/>
        <related type="document" name="documents"
                 datasource="cs.xml.SQL.QueryFactory">
            <filter class="MyFilter.DocCountFilter"/>
            <filter class="MyFilter.UppercaseFilter"/>
            <query basetable="zeichnung"><![CDATA[
                SELECT * FROM zeichnung
                WHERE teilenummer='%(teilenummer)s'
                AND t_index='%(t_index)s'
                ]]>
            </query>
            <field name="ZNUMMER"/>
            <field name="ZINDEX"/>
        </related>
    </object>
</schema>
```

At first, a filter is developed for counting the drawings. A *DocCountFilter* class is created for this:

```
class DocCountFilter(Filter.Filter):
    """
    Counter for documents associated with an article
    """
```

The counter for documents is initialized with 0 as soon as handling for the <related> element of a part is begun; the *OnStart* method is used to do so:

```
def OnStart(self, node):
    if (node.tag == XGW_related and node.get("name") == "documents"):
        DocCountFilter.counter = 0
    return node
```

The *OnChild* method is used to count up the document counter:

```
def OnChild(self, node):
    if node.tag == XGW_object and node.get("type") == "document":
        DocCountFilter.counter += 1
        currentField = Element(XGW_field, name = "LAUFENDE_NR")
        currentField.text = "%s" % DocCountFilter.counter
        node.append(currentField)
    return node
```

The lines

```
currentField = Element(XGW_field, name = "LAUFENDE_NR")
currentField.text = "%s" % DocCountFilter.counter
node.append(currentField)
```

generate a new <field> element with the name “LAUFENDE_NR”, assign the value of the counter to it, and add this element under *node*.

When the part is done being edited, editing of the underlying documents in the <related> element also has to be completed. Therefore the current revision of the counter corresponds to the total number of documents for this part; thus the corresponding field can be inserted now. This takes place via the *OnEnd* method at the end of editing the <object> element for the part.

```
def OnEnd(self, node):
    if (node.tag == XGW_object and
        node.get("type") == "part"):
        counterField = Element(XGW_field, name = "ANZAHL_ZEICHNUNGEN")
        counterField.text = "%s" % DocCountFilter.counter
        node.insert(2, counterField)
    return node
```

The filter for conversion into uppercase letters is relatively simple. Each element is checked to see whether it is a <field> element; if it is, the text in it is converted:

```
class UppercaseFilter(Filter.Filter):
    """
    Convert all text content to uppercase
    """
    def OnChild(self, node):
        if node.tag == XGW_field:
            if node.text:
                node.text = string.upper(node.text)
        return node
```

Here is the entire module:

```
#!/usr/bin/env python
# File:     MyFilter.py
# Author:   caddok
# Creation: 09.03.05
# Purpose:  Beispiel für Einsatz von Filtern

import string

from cs.xml import Filter
from cs.xml.SchemaParser import XGW_field, XGW_object, XGW_related, make_field

class UppercaseFilter(Filter.Filter):
    """
    Convert all text content to uppercase
    """
    def OnChild(self, node):
        if node.tag == XGW_field:
            if node.text:
                node.text = string.upper(node.text)
        return node

class DocCountFilter(Filter.Filter):
    """
    Counter for documents associated with an article
    """

    def OnStart(self, node):
        if (node.tag == XGW_related and node.get("name") == "documents"):
            DocCountFilter.counter = 0
        return node

def OnChild(self, node):
    if node.tag == XGW_object and node.get("type") == "document":
        DocCountFilter.counter += 1
        currentField = xml.element(XGW_field, name = "LAUFENDE_NR")
        currentField.text = "%s" % DocCountFilter.counter
        node.append(currentField)
    return node

def OnEnd(self, node):
    if (node.tag == XGW_object and node.get("type") == "part"):
        counterField = xml.element(XGW_field, name = "ANZAHL_ZEICHNUNGEN")
        counterField.text = "%s" % DocCountFilter.counter
        node.insert(2, counterField)
    return node
```

3.2 Custom data sources

It can happen that the data to be exported cannot be obtained via Catalyst or a SQL query. In this case, you have the option of defining a custom data source that provides the required data.

An example of this is the exporting of documents, during which a selected file is to be stamped and made available to the target system at the same time. The stamping procedure produces a new file that is not to be stored in CONTACT Elements, and therefore is a good example of a custom data source.

First we will take a look at the corresponding schema file. In doing so, we will limit ourselves to defining just one object type, which contains the document number (z_nummer), and document index (z_index) data fields as well as the path of the stamped file in the file system. Here is the corresponding schema file:

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- -\*- XML -\*- -->
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="target.xml"/>
    <object type="document" datasource="MyModule.DocumentFactory">
        <field name="z_nummer"/>
        <field name="z_index"/>
        <field name="pathname"/>
    </object>
</schema>
```

What is critical here is the line:

```
<object type="document" datasource="MyModule.DocumentFactory">
```

which specifies "MyModule.DocumentFactory", which is the data source for the data to be exported. Implementation of the corresponding function requires a `DocumentFactory` function at first, which returns an object for generating the data to be exported:

```
def DocumentFactory(schema, element):
    return ToplevelDocGenerator(schema, element)
```

All corresponding functions have to have the signature mentioned above. The function calls up the constructor of the `ToplevelDocGenerator` class. This class, which facilitates generation of the data that is to be provided, has to inherit from the `cs.xml.Interfaces.IDataSource` class. Depending on whether the provided data is to be exported at the top node of the export in <object> elements or within <related> elements, the generating class has to implement one of two methods:

- `go()`, if objects are to be provided at the top node,
- `generate(filterchain)`, if <related> elements are to be generated.

In our example, <object> elements are to be exported, therefore the `go()` method has to be implemented. Then the implementation of the class looks as follows:

```
class ToplevelDocGenerator(Interfaces.IDataSource):

    def __init__(self, schema, element):
        self._type = element.get('type')
        self._schema = schema
        self.records = sqlapi.RecordSet("zeichnung",
                                        "z_nummer like 'K_%'",
                                        addtl = "order by z_nummer")

    def go(self):
        for record in self.records:
            self._schema.feed(DocumentFacade(self._type, record))
```

The constructor buffers only its arguments and saves the results of a database query as an attribute. This database query determines which documents are to be exported. The `go()` method then iterates over the results list of the query and uses this to generate a separate object of the `DocumentFacade` class for each record. This class has to implement the interface of the `cs.xml.Interfaces.IDataSource` class, which is defined as follows:

```
class ITransferObject:
    """Interface definition for transferrable objects fed into a
    Schema."""

    def type_name():
        """Should return a typename (same as <object type="...">)"""
        return name

    def id(self):
```

(continues on next page)

(continued from previous page)

```
        """Returns an object id."""
        return None

    # stripped-down dict-like interface to access attributes
    def items(): pass

    def __getitem__(key): pass
```

The `type_name()` function has to return a type. This is stored in the export as a value of the "type" attribute. The `id()` function can be used to generate identifiers. The default implementation does not return any identifier. The `items()` function returns a list of attribute/value pairs; a corresponding attribute/value pair has to be included in the returned list for the "name" attribute of each <field> element. Finally, `__getitem__()` implements dictionary access to the relevant attributes. Here is an example for implementing the `DocumentFacade` class:

```
from cs.documents import Document

class DocumentFacade(Interfaces.ITransferObject):

    def __init__(self, type_name, record):
        self._type_name = type_name
        self._record = record
        export_dir = "//SERVER/Export_Dir"
        format = "pdf"
        doc = Document.ByKeys(record.z_nummer, record.z_index)
        filename = doc.getExternalFilename(format)
        self.dst_filename = os.path.join(export_dir, filename)
        doc.checkoutFile(dst_filename, format)
        ### Die folgende Funktion steht als Beispiel für
        ### das Stempeln der Datei:
        stamp_file(self.dst_filename)

    def items(self):
        for key in self._record.keys():
            yield key, self.dict[key]
            yield "pathname", self.dst_filename

    def type_name(self):
        return self._type_name

    def __getitem__(self, key):
        if "pathname" == key:
            retVal = self.dst_filename
        else:
            retVal = self._record[key]
        return retVal
```

The type name is set in the constructor, the path of the document's file representation is determined and stored in the `dst_filename` attribute. In addition to the key/value pairs of the `record` object's dictionaries, the `items()` method also returns the pair ("pathname", self.pathname). Accordingly, the `__getitem` method was also adapted so that a value is returned also for the "pathname" key, namely, the path determined in the constructor. The `type_name()` method returns only the value of the `_type_name` attribute set in the constructor. `id()` is not defined at all; accordingly, the default implementation is used, which does not set any identifier.

The defined classes and methods are stored together in the module `MyFilter.py`. Here is the entire module:

```
#!/usr/bin/env python
# File:     MyModule.py
# Author:   caddok
# Creation: 02.12.05# Purpose:

from cdb import sqlapi
```

(continues on next page)

(continued from previous page)

```
from cs.xml import Interfaces
from cs.xml.SchemaParser import XGW_field, XGW_object, XGW_related, make_field
from cs.xml.xgwlib import xgwlog
from cs.documents import Document

class DocumentFacade(Interfaces.ITransferObject):

    def __init__(self, type_name, record):
        self._type_name = type_name
        self._record = record
        export_dir = "//SERVER/Export_Dir"
        format = "pdf"
        doc = Document.ByKeys(record.z_nummer, record.z_index)
        filename = doc.getExternalFilename(format)
        self.dst_filename = os.path.join(export_dir, filename)
        doc.checkoutFile(dst_filename, format)
        ### Die folgende Funktion steht als Beispiel für
        ### das Stempeln der Datei:
        stamp_file(self.dst_filename)

    def items(self):
        for key in self._record.keys():
            yield key, self.dict[key]
        yield "pathname", self.dst_filename

    def type_name(self):
        return self._type_name

    def __getitem__(self, key):
        if "pathname" == key:
            retVal = self.dst_filename
        else:
            retVal = self._record[key]
        return retVal

class ToplevelDocGenerator(Interfaces.IDataSource):

    def __init__(self, schema, element):
        self._type = element.get('type')
        self._schema = schema
        self.records = sqlapi.RecordSet("zeichnung",
            "z_nummer like 'K_%'",
            addtl = "order by z_nummer")

    def go(self):
        for record in self.records:
            self._schema.feed(DocumentFacade(self._type, record))

    def DocumentFactory(schema, element):
        return ToplevelDocGenerator(schema, element)
```

The result for the given example is the following export:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
    <object type="document" >
        <field name="z_nummer">KA_OW</field>
        <field name="z_index" />
        <field name="pathname">//SERVER/Export_Dir/KA_OW-.pdf</field>
    </object>
    <object type="document">
```

(continues on next page)

(continued from previous page)

```
        <field name="z_nummer">K_BO_1</field>
        <field name="z_index" />
        <field name="pathname">//SERVER/Export_Dir/k_bo_1.pdf</field>
    </object>
    <object type="document">
        <field name="z_nummer">K_BO_2</field>
        <field name="z_index" />
        <field name="pathname">//SERVER/Export_Dir/k_bo_2.pdf</field>
    </object>
</schema>
```

3.3 Processing your own XML tags

In addition to the built-in functions, the *CONTACT Catalyst EAI Gateway* can be extended by any other functions and use cases. For this purpose, separate function-specific messages are defined for each case. The respective message reception triggers the function and the message content parameterizes it. The type of processing depends on the top-level element in the incoming XML structure.

For each top-level element of an XML file, there is a class with a `process()` method that implements the processing of the message. Processing classes for default functions (for example, processing of schema and target files) are in the `cs.xml.XmlTagClasses` module.

Custom processing classes can be defined in job configurations by creating the *custom_xml_tags* attribute and assigning a module name to it. Then the classes which implement custom functions and process corresponding function messages have to be implemented in this Python module (which can be stored, for example, in the instance in the `site-packages` directory).

The following provides a simple example of processing XML files using a custom processing class.

The processing of messages with addresses serves as the example. Only one address per message is to be stored within an <adress_element> element; name, street address, zip code and city are to be stored in it. Here is an example of a corresponding file:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<address_element>
    <name>Johannes Schmidt</name>
    <strasse>Norderstrasse 21</strasse>
    <plz>28359</plz>
    <stadt>Bremen</stadt>
</address_element>
```

The top-level element in this file is <address_element>. The name of the corresponding processing class follows by convention from the prefixed character string `XGS_` (XML gateway schema) and the name of this element, thus the processing class is called `XGS_address_element`. The class has to inherit from `cs.xml.XmlTagClasses.XgwTag` and set the name of the top-level tag to be processed in the *self.qname* attribute in its constructor:

```
from cs.xml.XmlTagClasses import XgwTag

class XGS_address_element(XgwTag):

    def __init__(self):
        self.qname = "address_element"
```

The processing is implemented in the `process(node, options)` method. It gets as parameter a *cs.xml.xmllib.Element* object of the top-level tag, which is used to access the structure and content of the message, and the Python dictionary of the job options. The resulting output can be output or passed to the destination modules by the *options.output()* function.

In our example, an object of the `Address` class is to be created that is initialized with the data from the XML message. For the sake of simplicity, this object is merely output; in a real application, this class could be used to create the address in the database, or something similar. Here is the example of implementation:

```
#!/usr/bin/env python
# -*- Python -*-
#
# Copyright (C) 1990 - 2012 CONTACT Software GmbH
# All rights reserved.
# http://www.contact.de/

from cs.xml import SchemaParser, SharedObject
from cs.xml.xgwlib import xgwlog
from cs.xml.XmlTagClasses import XgwTag, registerClasses

class XGS_address_element(XgwTag):
    class Address:
        def __init__(self):
            self.name = ""
            self.street = ""
            self.zip = ""
            self.city = ""

        def __repr__(self):
            return ("Adresse:\n"
                    "========\n"
                    " Name:%(name)s\n"
                    " Strasse:%(street)s\n"
                    " Postleitzahl:%(zip)s\n"
                    " Stadt:%(city)s"
                    % self.__dict__)

    def __init__(self):
        self.qname = "address_element"

    def process(self, node, options):
        addr = self.Address()
        for subElement in node:
            if subElement.tag == "name":
                addr.name = subElement.text
            elif subElement.tag == "strasse":
                addr.street = subElement.text
            elif subElement.tag == "plz":
                addr.zip = subElement.text
            elif subElement.tag == "stadt":
                addr.city = subElement.text
        options.output(str(addr))

classDict = registerClasses(__name__)
```

For the class to be able to be used, it is absolutely necessary for the line

```
classDict = registerClasses(__name__)
```

to stand at the end of the file after the class definition, otherwise the gateway cannot assign the top-level element <address_element> from incoming messages to the implementation. When the above message is received, the gateway generates the following output:

```
Adresse:
========
Name:Johannes Schmidt
Strasse:Norderstrasse 21
```

(continues on next page)

(continued from previous page)

```
Postleitzahl:28359
Stadt:Bremen
```

If no other destination components are defined, the output is written into a `output_dir/output.xml` file now.

3.4 Expanding existing XML structures

Section *Processing your own XML tags* (page 61) described how separate message types can be implemented using a separate tag handler using the *custom_xml_tags* option. It is often easier to implement requirements by using existing processing classes and adding configuration options or functionality. This path is supported by the *custom_handler* option. This makes it possible to add your own message elements (tags) to existing message types.

An extension like this can be implemented using three methods: *start_<tag>_childs(node)*, *process_<child_tag>(child_node)* and *end_<tag>_childs(node)*. The PowerScript module with the implementation of these methods is specified in the `xmlgateway.conf` file or the corresponding configuration file in the *custom_handler* option:

```
'custom_handler': {'config': 'customer.my_module.my_class'}
```

This causes the system to supplement XML structures with the top-level *config* tag by having the system search for and call corresponding methods when parsing those structures in the specified class.

For implementing custom handler classes the base class *cs.xml.SchemaParser.CustomTagHandler* is provided.

```
class CustomTagHandler(object):
    def __init__(self, node, options, default_handler):
        pass

    def call_default_handler(self, child):
        pass
```

As an example: Configuration messages with the top-level <*config*> tag recognize the <*transformation*> and <*destination*> tags. They are to be supplemented with the addition of another <*option*> tag. Thus, XML structures of this type with this addition look like this:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<config xmlns:xi="http://www.w3.org/2001/XInclude"
          xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <destination class="cs.xml.Destinations.FileDestination"
               filename="%(output_dir)s/output_file.xml" />
  <option>myOption1=Value1</option>
  <option myOption2="Value2"/>
</config>
```

Without the specified configuration addition, the *cs.xml.ConfigHandler.Config* class is used to evaluate these structures. It implements the internal *_handle_destination()* and *_handle_transformation()* methods to evaluate these two tags. Due to the configuration addition, the *start_config_childs()*, *process_option()* and *end_config_childs()* methods are now called in the *customer.my_module.my_class* class as well, if they are implemented.

Caution: A *process_destination()* method can also be implemented here. If this is done, this module would then replace the *cs.xml.ConfigHandler.Config._handle_destination()* that is normally called. This effect can be used to modify default behavior, but it can also lead to unwanted side effects. Normally, implementations for new expanding tags should be used this way for this reason.

Within these methods, the *cs.xml.xmllib.xml_element* object describing the associated tag is passed as argument. In the extension methods its members can be evaluated. In addition to *node.tag* for the respective tag name (which

is already known from the method name), this is *node.text* for the content text. The attributes can be retrieved by *node.get('attributname')* (see also Python module *cs.xml.xmllib.xml_element*).

In the example above, access to *node.text* in the first call for *process_option(node)* results in the content "myOption1=Value1". In the second call, the access to *node.get('myOption2')* results in the content "Value2". These options and their values could now be inserted into the job dictionary passed in the Init method in order to use them later in other extensions.

3.4.1 Modification of the default behaviour of TagHandler

As mentioned in the previous section, the behaviour of a *TagHandler* processing the child elements of its tag can be customized by implementing the corresponding method *process_<tag>* in a custom handler; To modify the behaviour of SchemaParser when processing destination elements the custom handler must define the method *process_destination(node)*. The default behaviour when calling a custom handler is, that the replacing method will be called instead of the original implementation.

If the original behaviour should also be executed, the instance method *TagHandler.call_default_handler(node)* can be called. Like a super call in inheritance hierarchies, this method allows to call the replaced implementation from the custom handler. Note that – in this case – the custom handler needs to be derived from *CustomTagHandler*.

An example for this approach is given in chapter *Extending Response Handler* (page 85)

3.5 Custom destination modules

As described in *Destination modules* (page 18), destination modules that can receive an XML message in the form of a text stream or an XML structure as an input (or both) can be implemented.

Two interfaces have been defined to accomplish this (see *cs.xml.Interfaces.IXMLTransformation* and *cs.xml.Interfaces.IDataDestination*). The input of a module depends on the position where it is inserted in the list of destination modules. If an XML structure is returned as an input, then it is passed using the *self.output(tree)* function. The content of an XML message is passed in the form of several individual text strings (not necessarily line-by-line) using the *self.write(input_line)* function. The end of an XML message is indicated by a *flush()* function call.

The implementation of a module should be derived from *cs.xml.Destinations.Destination*. A Python dictionary with configuration attributes for the current job is passed to each module via its constructor (*self.__init__(attrs, options, output)*) using the *options* parameter. This allows additional, module-specific configuration options to be passed to the module. The *attrs* parameter is used to pass a dictionary containing the attributes of the <destination> tag from the configuration. A module that expects its configuration via the attributes of the <destination> tag can use this to call them.

Each module has the option to modify its input, to output to a destination and/or to output to the subsequent module specified in the configuration. Passing on information to the subsequent module is done by calling the respective inherited function. This does not require calling the same function that was used to pass the input. For example, a module can transform a passed XML structure into a text stream. It will then implement the *output()* function, which transforms the XML structure passed in the *tree* parameter and outputs the result using the *super(Classname, self).write()* function. After completing the transformation, it has to call *super(Classname, self).flush()* accordingly. Optionally, it can also output the XML structure by calling *super(Classname, self).output(tree)*. Depending on the configuration, this results in a downstream module being able to receive and process the structure or stream.

The XML structure passed by *output()* corresponds to structures typically generated by the Python module *cs.xml.xmllib*. Using the APIs documented there, the structure can be iterated and modified. Alternatively, a completely custom structure can be built and passed to the next module in each case.

A module that replaces the <field> tags of the CDBXML structure with tags that have the respective field names as tag names is shown here as an example. In this example, a separate structure is set up as a copy of the input XML structure where the <field> tags are replaced in each case. At the same time, the use of a separate XML namespace is shown:

```
from cs.xml.Destinations import Destination
from cs.xml import SchemaParser
from cs.xml import xmllib

class MyDestination (Destination):

  def __init__(self, attrs, options, output):
      super(MyDestination, self).__init__(attrs, options, output)
      MY_NAMESPACE = "http://xml.my.company/schema/my_schema/1.0"
      self.xml = xmllib.xml(MY_NAMESPACE)

  def copy_tag(self, tag):
      tagname = unicode(SchemaParser.tagname(tag.tag))
      content = unicode(tag.text) if tag.text else ""
      attrs = {}
      for key, value in tag.attrib.items():
          attrs[unicode(key)] = unicode(value)

      if tagname == "field":
          tagname = attrs["name"]
          del attrs["name"]

      new_tag = self.xml.element(tagname, content, None, **attrs)
      return new_tag

  def copy_level(self, old_root, new_root):
      for i in old_root:
          next_elem = self.copy_tag(i)
          new_root.append(next_elem)
          if len(i) > 0:
              self.copy_level(i, next_elem)

  def output(self, tree):
    root = tree.getroot()
    new_root = self.copy_tag(root)
    self.copy_level(root, new_root)
    tree = self.xml.tree(new_root)
    tree.write(self, encoding="UTF-8", pretty_print=True)
    self.flush()
```

Thus, this module is also an example of receiving a structure (by implementing *output()*) and outputting a text stream (by calling *write()* and *flush()*). *write* is called indirectly here: The *cs.xml.xmllib.tree().write()* function receives an object usable as a file descriptor as a parameter. Here, *self* is passed to it, resulting in *self.write()* being called each time "Filedescriptor" is written.

In other words, the module is used in a destination configuration to write the output into a file:

```
<destination class="my_module.MyDestination" />
<destination class="cs.xml.Destinations.FileDestination"
             filename="%(output_dir)s/output.xml" />
```

Custom *transformation* components can be implemented similarly to custom *destination* components. Though the interfaces which are defined by the methods *output*, *write* and *flush* are the same, the parent class for *transformations* should be `cs.xml.Transformations.XMLTransformation`.

These classes can – like section *Debugging Destination Pipelines* (page 27) mentioned – be used in a *transformation* pipeline.

CHAPTER 4

Tutorials and examples

In the following some functions of the *CONTACT Catalyst EAI Gateway* are explained by means of examples. In section *Setting up the object transfer with the CONTACT Catalyst EAI Gateway* (page 66) the setup of the *CONTACT Catalyst EAI Gateway* and the realization of the object derivation with it is discussed. It is therefore also a good introduction to the topic. Section *Interface to query objects* (page 75) shows how to use *CONTACT Catalyst EAI Gateway* to implement a web service for querying user data from CONTACT Elements.

4.1 Setting up the object transfer with the *CONTACT Catalyst EAI Gateway*

This chapter describes the general procedure to implement the transfer of articles and documents using the *CONTACT Catalyst EAI Gateway*.

In the course of this tutorial, the derivation of various object types from CONTACT Elements into a fictitious system *example_erp* is shown. This will be used to demonstrate various aspects of transferring data using *CONTACT Catalyst EAI Gateway*:

- Setting up a coupling based on *CONTACT Catalyst for ERP* for use with the *CONTACT Catalyst EAI Gateway*
- Export of data using the `object` and `related` elements, as well as transformation of the outgoing data using a configurable transformation pipeline
- File-based transfer of data

Note: To do this tutorial the packages `cs.vp` and `cs.documents` need to be installed in addition to `cs.xml` and `cs.erp`, as these are required for synchronizing items and documents.

4.1.1 General setup in CONTACT Elements

Before the actual synchronization logic can be configured, some general configuration steps have to be performed first. The Catalyst component must be activated in CONTACT Elements (see the Catalyst administration manual), and the ERP system to be connected must be made known.

After installation of *CONTACT Catalyst EAI Gateway* a sample configuration can be installed first, which will be used in the following. The installation is done with the tool `mkconfig` (see also corresponding manual). `mkconfig` can install the configurations of different Catalyst applications, so in the command line, the desired package is specified. Example:

```
powerscript -m mkconfig --package cs.xml list
```

As a result, the list of the supplied configurations is output. In the following the transfer of articles and documents is to be realized, therefore the appropriate configurations `part` and `doc` are installed:

```
powerscript -m mkconfig --package cs.xml --configs part,doc install
```

The following section also describes how to activate Catalyst manually. However, this has already been done by importing the above-mentioned sample configuration.

4.1.1.1 Setting up the EAI system in CONTACT Elements

After installing the package, the Catalyst component must be activated, unless this has been done in the course of another use of the `cs.erp` package. To do this, modify the `ppst` property in CONTACT Elements under `Administration/Configuration -> Configuration -> Properties`. This property initially has the value `false`. If it is set to `true` instead, the component is enabled.

Now the sample system *example_erp* can be made known to the EAI configuration. For this purpose, a new entry is added under `Administration/Configuration -> Administration -> Integrations -> ERP Configuration -> EAI System`. Figure *Flow chart for the Synchronization of BOMs* (page 67) shows the corresponding dialog.

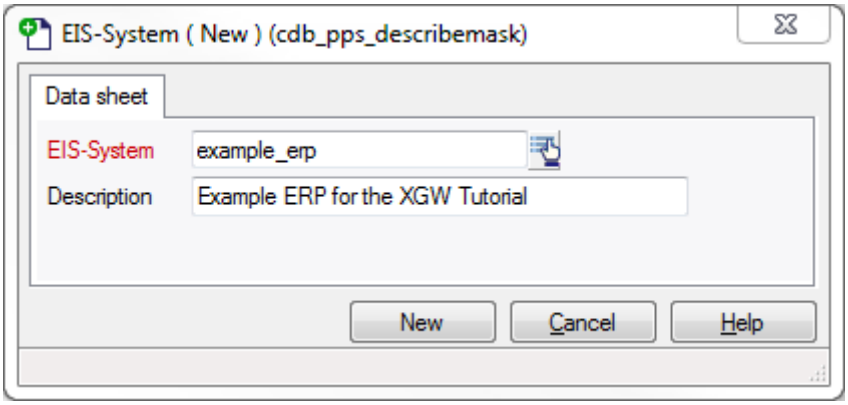

Fig. 1: Flow chart for the Synchronization of BOMs

Many aspects of the synchronization with ERP systems are controlled by CAD configuration switches, so a basic number of such switches now need to be set to allow data transfer.

CAD configuration switches are set in CONTACT Elements (`Administration/Configuration -> Administration -> Integrations -> CAD-configuration -> General`). At least the switches shown in table *CAD-configuration switches 1* (page 67) must be set.

Table 1: CAD-configuration switches 1

Name	Value	CAD-System
PPS Abgleich aktivieren	TRUE	example_erp

4.1.1.2 Configuration of the gateway

Now the *CONTACT Catalyst EAI Gateway* must be configured to use the correct ERP system (the *example_erp* configured above).

A configuration file was already created during installation of the package *cs.xml*. It is found in the instance directory under `etc\xmlgateway\site.xgs` and has the following content:

```
1 <?xml version='1.0' encoding='ISO-8859-1'?>
2 <target xmlns:xi="http://www.w3.org/2001/XInclude"
3         xmlns="http://xml.contact.de/schema/xgwschema/1.0"
4         name="xmlgateway" />
```

The target definition (see also *Message processing* (page 13)) here defines the ERP system to be matched.

The system to be connected is `example_erp` in our case; line 4 must therefore be adapted:

```
name="example_erp"
```

4.1.2 Synchronization of parts

As mentioned in the *General setup in CONTACT Elements* (page 66) section, setting up item transfer is divided into the basic configuration, which is done in CONTACT Elements, and the creation of a corresponding transfer job in the context of *CONTACT Catalyst EAI Gateway*.

4.1.2.1 EIS configuration of `part`

In order to synchronize a specific object type (independent of the issues of transfer via *CONTACT Catalyst EAI Gateway*), it must be defined in the EAI configuration of the system. For this purpose, the object is added in the EAI system configuration in the tab `Objects for transfer`. Figure *PPS configuration of class part* (page 68) shows the datasheet for the configuration of class `part`.

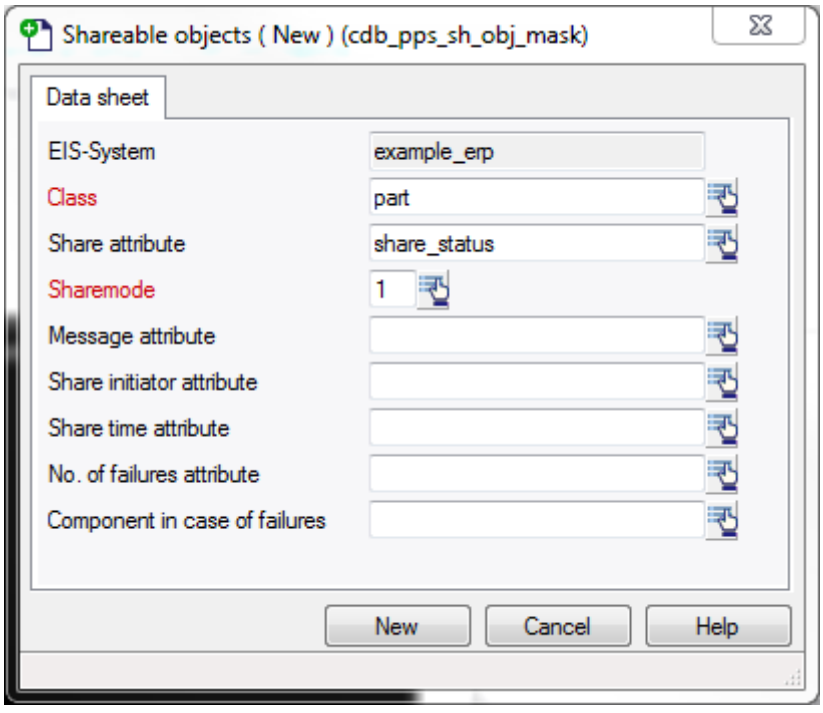

Fig. 2: PPS configuration of class `part`

After this, if not already done after the steps from *General setup in CONTACT Elements* (page 66), the server must be restarted. If the configuration has been done correctly up to this step (regardless of the steps of the gateway configuration), the class `part` should now be displayed in the EAI status panel.

By specifying the attribute `share_status` as a synchronization attribute, it is used by *CONTACT Catalyst for ERP* to determine when an object should be transferred. The EAI configuration allows to set predicates as transfer

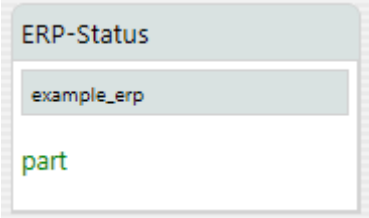

Fig. 3: E-link panel ERP status after configuration of class `part`.

conditions. They are checked when changes are made to an object, to accordingly set the transfer attribute if necessary, and thus mark the object for synchronization, for example.

The next step is to define a sharing condition for the class `part`. Objects of this class should be synchronized, when their lifecycle status is `200` (Released). Figure *Sharing condition for class part.* (page 69) shows the data sheet with the appropriate sharing condition. The module `cs.vp.items` has the predefined predicate *part_ERP freigegeben* on the relation `teile_stamm`, which checks if the field `status` has the value `200`.

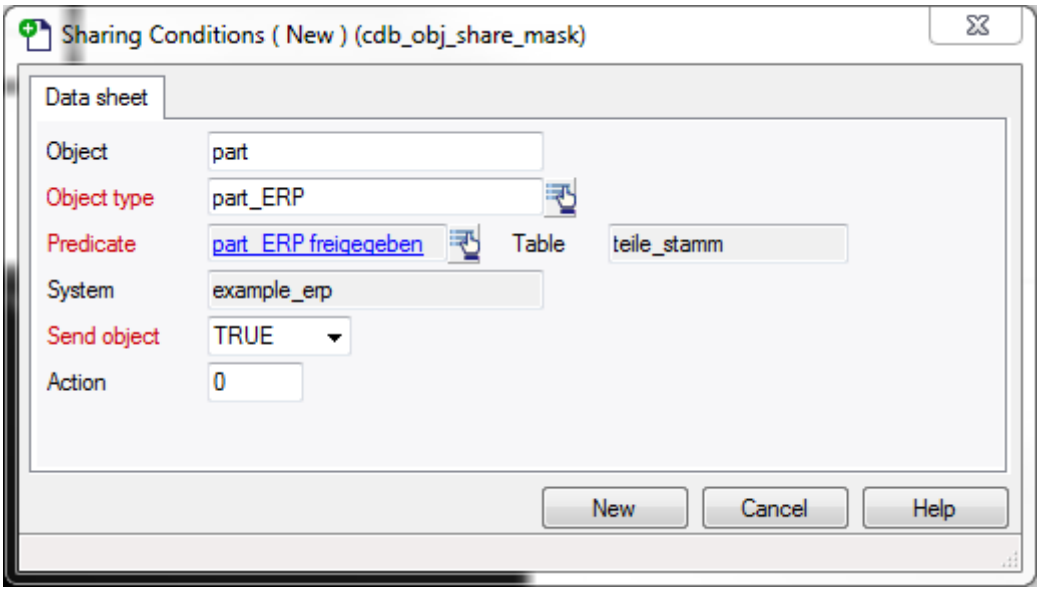

Fig. 4: Sharing condition for class `part`.

If now an item is created and switched to the status `Released`, it should be displayed in the EAI status panel as an object to be synchronized. However, when this object is now transferred, it would not contain any data; in order for the attributes of an object to be transferred, they must be explicitly specified as part of the PPS field configuration.

The synchronization of fields is configured in the EIS configuration tab *Attributes*. For each object class the fields that are transferred need to be defined here. Additionally fields can be assigned a new name. Figure *Dialog to configure fields for EAI Systems* (page 70) shows a field configuration, that synchronizes the field `teilenummer` of class `part` using the alias `artikelnummer`.

For each field a data type can be assigned; `teilenummer` uses the type `string`. Additionally, a default value can be mentioned. If the attribute is configured as `-`, the synchronization logic will always use the default value. This is useful for attributes that have no correspondence in CONTACT Elements.

As a last step we introduce fields for the article's name and type: these will be defined with attribute `-`. The field configuration tab should now look similar to figure *Field configuration of class part in example_erp* (page 70).

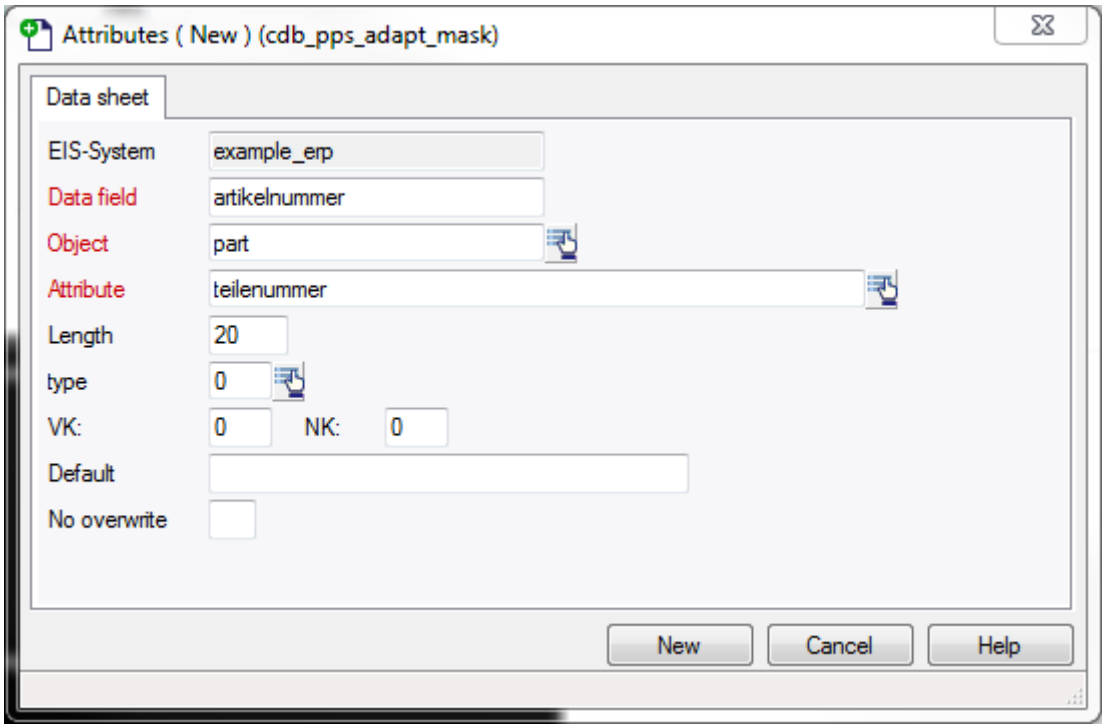

Fig. 5: Dialog to configure fields for EAI Systems

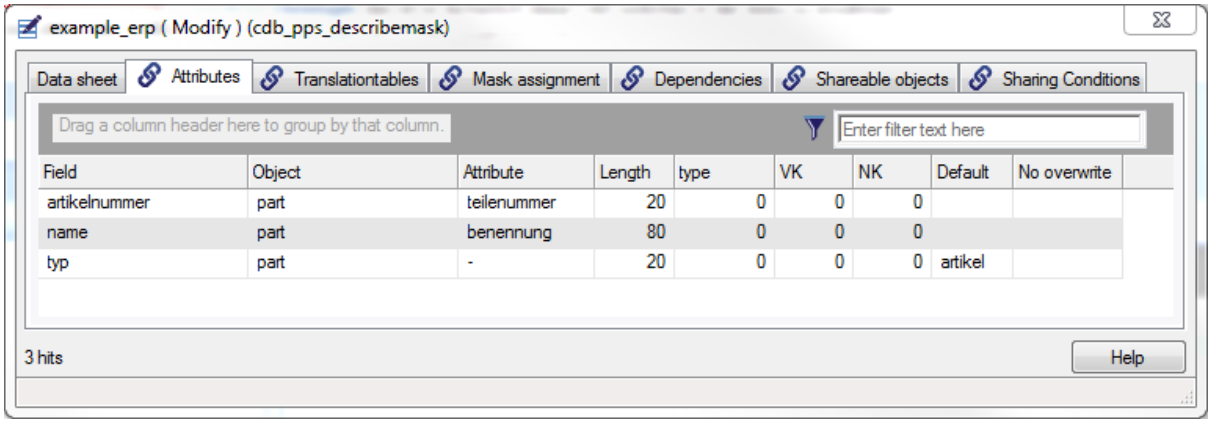

Fig. 6: Field configuration of class `part` in `example_erp`

4.1.2.2 Setting up a job for article synchronization

After installation, the default configuration in the `etc/xmlgateway/xmlgateway.conf` file already contains a sample job that implements article synchronization. The job is executed every 600 seconds and performs the functions defined in the `schema_file` `site.xgs` and `job_conf.xgs`.

```
XGW_JOBS = [
  {'job_interval': 600,
   'job_list': [{'active': True,
                 'job_id': 'Shared object',
                 'schema_files': ['site.xgs', 'job_conf.xgs']
                }],
    'input_files': None,
    'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
    'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway/work'
  }
]
```

`site.xgs` contains the configuration for the connection and has been customized in section *General setup in CONTACT Elements* (page 66).

In the next step we configure a function which realizes the synchronization of articles: so we use the file `job_conf_part.xgs` in this directory. Here we want to use the gateway function `schema` (see *Schema* (page 15)), which reads marked objects from CONTACT Elements for synchronization and processes them through a destination pipeline.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude" />
```

To achieve this, the `object` and `destination` parameters must be defined by specifying appropriate child elements.

Let us first define an `object` element. In the sample configuration, the file `part_object.xgs` is included, where such an object is entered. Object elements define, in the context of a schema, which objects and fields are to be synchronized from CONTACT Elements. The element first needs a data source to read the data from. We use `cs.xml.SharedObject.Factory`, which reads data to be synchronized using Catalyst. As type we specify `part` – the class name of article. In addition, the fields of the objects to be synchronized must be specified as child elements with the tag `field`. Here we can simply insert the fields of the PPS field configuration from Figure *Field configuration of class part in example_erp* (page 70).

```
<object datasource="cs.xml.SharedObject.Factory" type="part">
   <field name="artikelnummer" />
   <field name="name" />
   <field name="typ" />
</object>
```

Alternatively, a wildcard `*` can be used as a `name` to synchronize all configured fields.

Using `destination` elements it is now possible to specify what should happen to the objects defined for transfer. Section *Destination modules* (page 18) shows the blocks that are provided by default by the *CONTACT Catalyst EAI Gateway*. In order to transfer objects using files the block `cs.xml.Destinations.FileDestination` can be used here. This block expects an input stream (type XML text) as described in the documentation; however, the input to the destination pipeline is an XML structure, which must be serialized first. This function is achieved by the `cs.xml.Destinations.XMLDestination` block.

The destination pipeline is created by instantiating each component in the order they are defined. The following configuration instantiates a pipeline, whic streams XML messages into files.

```
<destination class="cs.xml.Destinations.XMLDestination" />
<destination class="cs.xml.Destinations.FileDestination"
```

(continues on next page)

(continued from previous page)

```
              filename="%(output_dir)s/artikel.xml"
              unique="True" />
```

In the installed sample configuration, a `JSON` block is used instead, which should be removed and not used for this experiment.

Next we define the directory `output_dir` in the job-configuration of the file `xmlgateway.conf`. Furthermore, we need to create the configured directory, and introduce the newly defined job-function. The file `xmlgateway.conf` should now look like this:

```
XGW_JOBS = [
  {'job_interval': 600,
   'job_list': [{'active': True,
                 'job_id': 'Shared object',
                 'schema_files': ['site.xgs', 'job_conf_part.xgs']
                }],
    'input_files': None,
    'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
    'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway/work',
    'output_dir': '%(CADDOK_BASE)s/etc/xmlgateway/output'
  }
]
```

The file `job_conf_part.xgs` should look like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">

  <destination class="cs.xml.Destinations.XMLDestination" />
  <destination class="cs.xml.Destinations.FileDestination"
               filename="%(output_dir)s/artikel.xml"
               unique="True" />

  <object type="part" datasource="cs.xml.SharedObject.Factory">
     <field name="artikelnummer" />
     <field name="name" />
     <field name="typ" />
  </object>
</schema>
```

If the *CONTACT Catalyst EAI Gateway* service is now started, the article created above and marked for transfer should be written to a file in the `etc/xmlgateway/ouput` directory.

4.1.2.3 User-defined transformation of articles

In the previous sections, we created an EAI configuration that implements the transfer of objects of class `part` for the sample system *example_erp*. The export is done to files that correspond to the transfer scheme defined by the *CONTACT Catalyst EAI Gateway*; thus, after the transfer, the item in question has the following form:

```
<?xml version='1.0' encoding='UTF-8'?>
<ns0:schema xmlns:ns0="http://xml.contact.de/schema/xmlgateway/1.0">
  <ns0:object id="706172743a7465696c656e756d6d65723d3030303030333a745f696e6465783d"
              type="part">
    <ns0:field name="artikelnummer">000003</ns0:field>
    <ns0:field name="name">Testteil</ns0:field>
    <ns0:field name="typ">artikel</ns0:field>
  </ns0:object>
</ns0:schema>
```

Now, assume that the system requires the data in a different format, which would represent the object from above as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<records xmlns="http://example.erp.org/schema">
  <record>
    <artikelnummer>000003</artikelnummer>
    <name>Testteil</name>
    <typ>artikel</typ>
  </record>
</records>
```

To convert the data before they are written to a file, an intermediate component needs to be introduced. This component should be put before the serialization step `XMLDestination`.

In order for the XML gateway to find this new component it needs to be defined in the instance's Python path: we create a python package `example_erp` in the instance's site-packages folder. In the package we create a module `destinations`.

```
$ ls inst\site-packages\example_erp
__init__.py
destinations.py
```

The destination component can now be defined in the module `destinations`. Using this component we can execute arbitrary modifications of the XML structure. The component will inherit from the base destination class `cs.xml.Destinations.Destination` and implement the method `output`; the default implementation of this method simply redirects it's input to the next destination. The skeleton for creating a custom destination component is:

```
from cs.xml.Destinations import Destination


class ExampleERPDestination(Destination):
    def output(self, tree):
        super(ExampleERPDestination, self).output(tree)
```

Now the desired transformation can be implemented using the *cs.xml.xmllib* API:

```
 from cs.xml.Destinations import Destination
 from cs.xml import xmllib


 class ExampleERPDestination(Destination):

     def output(self, tree):
         ns_erp = 'http://erp.example.org/schema'
         xml =  xmllib.xml(ns_erp)

         ex_records_node = xml.element('records')

         for cdb_record_node in tree.root:
             ex_record_node = xml.element('record')

             for cdb_field_node in cdb_record_node:
                 if xmllib.strip(cdb_field_node.name()) == 'field':
                     ex_field_node = xml.element(
                         cdb_field_node.attrib['name'],
                         cdb_field_node.text)
                     ex_record_node.append(ex_field_node)
                 else:
                     ex_record_node.append(cdb_field_node)

```

(continues on next page)

(continued from previous page)

```
            ex_records_node.append(ex_record_node)

        super(ExampleERPDestination, self).output(
            xml.tree(ex_records_node))
```

The destination should now be included into the pipeline:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">

  <destination class="erp.destinations.ExampleERPDestination" />
  <destination class="cs.xml.Destinations.XMLDestination" />
  <destination class="cs.xml.Destinations.FileDestination"
               filename="%(output_dir)s/artikel.xml"
               unique="True" />

  <xi:include href="part_object.xgs" />
</schema>
```

The result of the export should now look as in the example above, if the steps above have been correctly implemented. Details about the API of the class `Destination` are provided in section *Custom destination modules* (page 64).

4.1.3 Synchronization of documents

The EIS configuration of documents can be done analogous to the configuration of class `part` explained in section *Synchronization of parts* (page 68). The different object types as well as corresponding predicates are configured for synchronization. The status attribute for documents is `z_status`.

4.1.4 Synchronization of relations

To synchronize related objects, the previously introduced `object` element can be extended by `related` child elements. Using a `related` tag documents associated with an article can be exported together with the article. The `related` element requires the type of relation and a data source.

Possible use cases for the `related` element are exporting associated documents or the positions of a BOM.

The following example shows how components of a BOM can be exported by using the datasource `cs.xml.SQL.RelshipFactory`.

```
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">

  ...

  <object type="part" datasource="cs.xml.SharedObject.Factory">
    <field name="*" />
    <related type="bom_item"
             name="related_positions"
             datasource="cs.xml.SQL.RelshipFactory"
             relship="cdbqc_part2subparts">
      <field name="*"/>
    </related>
  </object>
</schema>
```

`related` elements are usually defined in `object` elements. There they are used to export objects which are in some way related to the object exported by the surrounding element. The different types of relations are, e.g.,

BOM and components, relations defined via `cdb_relships` or arbitrary relations defined by a SQL query. The element inserted in the example above exports the components of a BOM by using the `cdb_relships` relation `cdbqc_part2subparts`.

It is also possible to insert `related` elements into `related` elements.

The different possibilities on how to use the `related` element are explained in section *Relationships (<related> elements)* (page 31).

4.1.5 Troubleshooting

This section lists typical errors, which are often made when configuring the data transfer with *CONTACT Catalyst EAI Gateway* and shows possible solutions.

4.1.5.1 Objects are not marked for synchronization

Objects not being marked for synchronization when they should be can have different reasons: First you should check wether a predicate is configured correctly. When using user defined predicates the database relation must be used, while for `part` and `document` often these names are used instead of the relation name.

Since predicates are cached by the server process, the CONTACT Elements client needs to be restarted after predicates have been changed.

4.2 Interface to query objects

4.2.1 Overview

A generic interface between different systems is frequently required to implement integration steps. One of these interfaces is implemented with the help of the *CONTACT Catalyst EAI Gateway* in the example below. This interface is intended to allow an external system shared access to data within CONTACT Elements.

Two functions can be implemented using the interfaces configured in the following example:

1 Querying a list of all users known in the system

2 Querying detailed information on a specific user and the roles assigned to the user

4.2.2 Interface

These interfaces are implemented based on the *CONTACT Catalyst EAI Gateway*. The properties of the *CONTACT Catalyst EAI Gateway* are used to accomplish this:

- Providing an HTTP interface through which messages can be received in CDBXML format
- Being able to run commands embedded in messages on the CONTACT Elements data stock using search queries
- Providing the results as messages in CDBXML format again.

The HTTP interface in use can be adapted to your own needs using configurations and with the help of PowerScript customization. By default, authentication mechanisms are used to control access to the data accessible via these kinds of interfaces.

4.2.3 Format

The basic schema of the CDBXML format has been kept clear and straightforward. The respective selected objects are provided in a list enclosed by a top-level tag (e.g. <schema>). The data for each object is also enclosed in an <object> tag that has a "type" attribute that reflects the object type (in this example, "user" or "role"). The values of the individual pieces of data themselves are each listed in <field> tags and named using a "name" attribute. The data types are omitted by default (for data type-specific formatting see below, attribute mapping) and can be inserted into the schema as needed.

4.2.4 Example

An example for using the completely configured interface is listed below. Any HTTP client that supports the POST or PUT methods can be used to run it. One example of a client is the Firefox "RESTClient" add-on.

The first step is querying all users from CONTACT Elements—this does not require an input message. Instead, the query is triggered by a simple HTTP/GET command to the interface. The URL of this interface in the example configured below is *http://xgw.contact.de:7488/users*.

Calling this command returns the following result:

```
<?xml version="1.0" encoding="utf-16"?>
<ns0:schema xmlns:ns0="http://xml.contact.de/schema/xmlgateway/1.0">
    <ns0:object type="user">
        <ns0:field name="lastname">caddok</ns0:field>
        <ns0:field name="firstname" />
        <ns0:field name="beruf">System-Administrator</ns0:field>
        <ns0:field name="personalnummer">caddok</ns0:field>
        <ns0:field name="abt_nummer">IT</ns0:field>
    </ns0:object>
    ...
    <ns0:object type="user">
        <ns0:field name="lastname">Enver</ns0:field>
        <ns0:field name="firstname">Julia</ns0:field>
        <ns0:field name="beruf">Engineer</ns0:field>
        <ns0:field name="personalnummer">Enver, Julia</ns0:field>
        <ns0:field name="abt_nummer">Engineering</ns0:field>
    </ns0:object>
    ...
    <ns0:object type="user">
        <ns0:field name="lastname">Wu</ns0:field>
        <ns0:field name="firstname">Yang</ns0:field>
        <ns0:field name="beruf">Manager</ns0:field>
        <ns0:field name="personalnummer">Wu, Yang</ns0:field>
        <ns0:field name="abt_nummer">Procurement</ns0:field>
    </ns0:object>
</ns0:schema>
```

In the second step, the detailed information on a user is to be retrieved at this point, Yang Wu in this example. A corresponding message is prepared to do this:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<user xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <field name="lastname">Wu</field>
  <field name="firstname">Yang</field>
  <field name="beruf">Manager</field>
  <field name="personalnummer">Wu, Yang</field>
  <field name="abt_nummer">Procurement</field>
</user>
```

This involved simply copying all of the field tags of the desired object into the enclosing <user> tag. The user tag is a custom tag described in the Configuration section. Only the *personalnummer* key attribute is required, however (see Configuration).

In order to send the message (to the URL *http://xgw.contact.de:7488/user* this time), an HTTP/POST command is run this time to transport the message contents.

The response now contains the assigned roles:

```
<?xml version="1.0" encoding="utf-16"?>
<ns0:schema xmlns:ns0="http://xml.contact.de/schema/xmlgateway/1.0">
    <ns0:object type="user">
        <ns0:field name="personalnummer">Wu, Yang</ns0:field>
        <ns0:field name="name">Wu, Yang</ns0:field>
        <ns0:related name="roles">
            <ns0:object type="role">
                <ns0:field name="role_id">Documentation</ns0:field>
            </ns0:object>
            <ns0:object type="role">
                <ns0:field name="role_id">Procurement</ns0:field>
            </ns0:object>
            <ns0:object type="role">
                <ns0:field name="role_id">Procurement: Manager</ns0:field>
            </ns0:object>
        </ns0:related>
    </ns0:object>
</ns0:schema>
```

4.2.5 Configuration

After installation the *CONTACT Catalyst EAI Gateway* is set by default to use the port 7488, so that the base URL is *http://localhost:7488*.

For access to the service (XML Gateway Services) Basic Authentication with the username *caddok* and example password *badpassword* is used in the following.

The configuration of the *CONTACT Catalyst EAI Gateway* is stored in the `$CADDOK_BASE/etc/xmlgateway/xmlgateway.conf` file. The following entry has to be added to this file:

```
{ # job configuration for http input
    'job_interval': 0,
    'group_id': 'http_users',
    'port': 7488,
    'secret': 'cs.platform/customer/xml-test/endpoints/test',
    'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.beispiel',
    'job_list':
    [
        {
            'job_id': 'http users',
            'path': '/users',
            'schema_files': [ 'users.xgs' ],
        },
        {
            'job_id': 'http user',
            'path': '/user',
            'custom_xml_tags': 'xgw.users',
            'schema_files': [ 'user.xgs' ],
        }
    ]
}
```

The secret addressed above must now be created within a wallet. The username and password must be separated by a colon. So for the username `caddok` and the example password `badpassword`, the string `caddok:badpassword` should be specified. This can be achieved e.g. by the following command:

```
echo caddok:badpassword | cdbwallet store --stdin cs.platform/customer/xml-test/
↪endpoints/test
```

The specified configuration creates two interfaces under the */users* and */user* path components. The first interface (users - query all users) is configured in the `users.xgs` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
  <destination class="cs.xml.Destinations.XMLDestination" />
  <destination class="cs.xml.Destinations.ResponseDestination"/>
  <object type="user" record_type="raw" datasource="cs.xml.SQL.QueryFactory">
    <query basetable="angestellter">
      select * from angestellter
    </query>
    <field name="lastname" />
    <field name="firstname" />
    <field name="beruf" />
    <field name="personalnummer" />
    <field name="abt_nummer" />
  </object>
</schema>
```

Three properties of the interface are configured here: The data source is a SQL query (*select * from angestellter*), the field attributes to be output from all of the data records or objects found this way are listed and the output is defined as a response to an HTTP request in XML format using destination modules. In this context, note that the "raw" database values are output due to the definition of the "raw" record type and the listing of the field attributes. Attribute mapping provides another option (see below).

The second interface (`user.xgs`) is configured accordingly:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template xmlns="http://xml.contact.de/schema/xgwschema/1.0"
          xmlns:xi="http://www.w3.org/2001/XInclude">
  <destination class="cs.xml.Destinations.XMLDestination" />
  <destination class="cs.xml.Destinations.ResponseDestination"/>
  <object type="user" record_type="raw" datasource="cs.xml.SQL.QueryFactory">
    <query basetable="angestellter">
      select * from angestellter where personalnummer='%(key)s'
    </query>
    <field name="personalnummer" />
    <field name="name"/>
    <related type="role" name="roles" record_type="raw"
             datasource="cs.xml.SQL.QueryFactory">
      <query basetable="cdb_global_subj">
        select * from cdb_global_subj where subject_id='%(key)s'
      </query>
      <field name="role_id" />
    </related>
  </object>
</template>
```

Unlike the Users interface, a special data record is selected here and a relation is defined using the <related> tag where the objects of the relation are output appended to the relation. As an alternative to this method of defining a relation by specifying an SQL statement, relationships defined directly in CONTACT Elements can be used as well (by using a *RelationshipFactory* as *datasource*).

This configuration is enclosed in a <template> tag (a custom tag). The custom implementation initiated using this tag (see below) results in no direct query being carried out. Instead, this configuration is stored as a template that has to be supplemented by other data before it is carried out. An example for a corresponding message was shown above.

The definitions of the custom tags (template and user) are generated in the PowerScript file `xgw/users.py` (see

`xmlgateway.conf`, *custom_xml_tags*). This also contains the code that creates the template and that fills out the following after receiving a user message:

```
from cs.xml import SchemaParser, ConfigHandler
from cs.xml.XmlTagClasses import XgwTag, registerClasses

class XGS_template(XgwTag):
    """
    XGS_template is triggered by the schema file. It creates a
    template schema process which is executed later, when its
    variables are filled in by XGS_user.
    """
    qname = None

    def __init__(self):
        self.qname = SchemaParser.QName(SchemaParser.XGS, "template")

    def process(self, node, options):
        schema = SchemaParser.Schema(node, options)
        options.schema_template = schema

class XGS_user(XgwTag):
    qname = None

    def __init__(self):
        self.qname = SchemaParser.QName(SchemaParser.XGS, "user")

    def process(self, node, options):
        u = User(node, options)
        u.process()

class User (ConfigHandler.Config):
    """
    class user is triggered by incoming messages with top level 'user' tag.
    It is used like a configuration option and extends Confighandler.Config by
    _handle_field: the user fields of the incoming message are used as
    configuration options in this case. The key field (login) is placed in the
    'http_args' variable, which is used in the template/schema class.
    Thus by now executing the schema process, the previously missing variable
    in user.xgs ('key') can be substituted.
    """

    def __init__(self, node, options):
        self._key_fields = {}
        ConfigHandler.Config.__init__(self, node, options)

    def _handle_field(self, node):
        key = node.attrib["name"]
        value = node.text
        if value:
            value = value.strip()
        else:
            value = ""
        self._key_fields[key] = value

    def process(self):
        if not self._key_fields.has_key("personalnummer"):
            return
        self._options.http_args = {'key': self._key_fields["personalnummer"]}
        self._options.schema_template.go()
        self._options.schema_template.write()

classDict = registerClasses(__name__)
```

In the *XGS_template* class, the template tag is announced to the system and the fact that a configuration enclosed in such tags is to be read in (*SchemaParser*) and saved (*options.schema_template*) is specified in the code.

The *user* tag is defined in the *XGS_user* class accordingly. The associated *process()* method is triggered by the arrival of a message with one of these user tags and its content is read in by the *User* Config class. The *_handle_field()* method of the class is called for each field tag of the received message. It reads the respective name and value into a dictionary. Finally, the key field (*personalnummer* in this case) is inserted into the template by the *process()* method and the query is carried out. A property of the schema class is used here to be able to add additional arguments to the configuration in the context of an HTTP call. The places at which these arguments are to be substituted are marked with a *%(key)s* placeholder in the `user.xgs` configuration file.

Exception handling is left out in this example. In the *User.process()* method, the system only queries whether the required key for identifying the desired user is contained in the received message. Instead of the return statement, an error message can also be generated as an XML response for the caller here.

4.2.6 Attribute mapping

In this example, the listed field values have been transferred directly (*raw*) from the database. In order to achieve independence from the database names, symbolic field names for the fields to be transferred can also be defined (see the Catalyst administration manual). When using this approach, the list of the field names to be transferred can be replaced by a <field name="*" /> placeholder. This results in all of the symbolic field names defined for an object type being transferred with their content so that the selection of fields to be transferred can be configured in the CONTACT Elements interface.

4.2.7 Summary

Based on the example, you can easily define interfaces that can be used to integrate external systems via standard HTTP and XML protocols. By following the examples shown, you can easily adapt and expand these interfaces to suit your needs.

4.3 Querying data from a web service

This example implements a web service that provides files stored in CONTACT Elements by using the `getfile` element (cf. *GetFile* (page 15)).

Section *Querying per getfile messages* (page 80) shows how an object can be exported by sending a `getfile` message to the XML gateway per HTTP. After that, section *Querying per URL* (page 81) shows, how a custom handler for `getfile` messages can be implemented, to allow the specification of document data via URL.

4.3.1 Querying per `getfile` messages

At first we create a job configuration similar to the one in section *Interface to query objects* (page 75):

```
{ # job configuration for http input
     'job_interval': 0,
     'group_id': 'export_files',
     'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.beispiel',
     'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway/work',
     'job_list':
     [
         {
             'job_id': 'export files',
             'path': '/documents',
             'immediate': True,
             'schema_files': [ 'site.xgs', 'job_conf_file.xgs' ],
         }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

The file `site.xgs` initializes the connection to CONTACT Elements. The file `config.xgs` initializes a destination pipeline to send files:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<config xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <destination class="cs.xml.Destinations.StreamDestination" />
  <destination class="cs.xml.Destinations.ResponseDestination"/>
</config>
```

The stream object generated by the `config` function will be transformed to a stream, which is sent back to the caller by the `ResponseDestination`.

Let us now consider the gateway function `getfile`. This specifies the document to be transferred in the `class` and `key` parameters. Therefore we cannot hardcode it inside the pipeline. Instead, it is sent by the caller. Therefore, the pipeline must be initialized to return it through the `config` function.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<getfile xmlns:xi="http://www.w3.org/2001/XInclude"
         xmlns="http://xml.contact.de/schema/xmlgateway/1.0">
  <class format="pdf">document</class>
  <key name="z_nummer" value="D-000123" />
  <key name="z_index" value="00" />
</getfile>
```

By sending such messages, any files can be exported from CONTACT Elements via HTTP.

4.3.2 Querying per URL

An alternative to specifying the document in the `getfile` message is to use the URL for that purpose. This section demonstrates how key fields and format can be provided in the URL.

To introduce this functionality, it is necessary to introduce a new tag definition, which we will define in `xgw.documents`. It will be called `docfile`. First we need to extend the job definition to point to the definition of the new tag handler.

```
{ # job configuration for http input
     'job_interval': 0,
     'group_id': 'export_files',
     'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.beispiel',
     'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway/work',
     'job_list':
     [
         {
             'job_id': 'export files',
             'path': '/documents',
             'immediate': True,
             'custom_xml_tags': 'xgw.documents',
             'schema_files': [ 'site.xgs', 'job_conf_file.xgs' ],
         }
     ]
}
```

The module `xgw.documents` must contain a new `XgwTag` definition and a `TagHandler`.

```
from cs.xml.XmlTagClasses import XgwTag, registerClasses
from cs.xml import SchemaParser, ConfigHandler
```

(continues on next page)

(continued from previous page)

```
class XGS_docfile(XgwTag):
  qname = None

  def __init__(self):
    self.qname = SchemaParser.QName(SchemaParser.XGS, "docfile")

  def process(self, node, options):
    vf = DocFile(node, options)
    vf.transfer()

class DocFile (SchemaParser.TransferFile, ConfigHandler.Config):
  def transfer(self):
    super(DocFile, self).transfer()
```

The basic properties of the tag handler `DocFile` are defined via inheritance: To implement file transfer we inherit from `SchemaParser.TransferFile`. Additionally `ConfigHandler.Config` is used to be able to parse destination elements directly in the `DocFile` element.

When processing a `getfile` element in *CONTACT Catalyst EAI Gateway*, the following code is normally executed, which processes the `class` and `key` parameters, setting the format and key fields:

```
def _handle_class(self, node):
    self.file_conf._export_format = node.attrib["format"]

def _handle_key(self, node):
    name = node.attrib["name"]
    value = node.attrib["value"]
    self.transferrable._set(name, value)
```

Our `TagHandler` should read these values from the called URL. If we use the gateway as a web service we can read the path of the URL by calling `self._options.http_path()`. The URL paths which are used by TagHandler should have the following form.

`/documents/<z_nummer>@<z_index>?format=<format>`

Let's extend the function `transfer` as follows:

```
1  def transfer(self):
2       args = self._options.http_arguments()
3
4       # Extract Document Number and Index
5       path = self._options.http_path().split('/')
6       if len(path) == 0:
7           raise Exception('Invalid Doc path')
8
9       docrev = path[0].split('@')
10      if len(docrev) != 2:
11          raise Exception('Invalid Doc path')
12
13      self.transferrable._set('z_nummer', docrev[0])
14      self.transferrable._set("z_index", docrev[1])
15
16      # Determine preferred format
17      if "format" in args:
18          self.file_conf._export_format = args["format"][0]
19      else:
20          self.file_conf._export_format = 'pdf'
21
22      self._options.set_content_type("application/octet-stream")
23
24      super(DocFile, self).transfer()
```

In lines 5-14 the key of the document is read from the path. After that the requested format is read from the URL parameters in lines 17-22.

Now the file `job_conf_file.xgs` can be modified as follows:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<docfile xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://xml.contact.de/schema/xgwschema/1.0">
  <destination class="cs.xml.Destinations.StreamDestination" />
  <destination class="cs.xml.Destinations.ResponseDestination"/>
</docfile>
```

Now documents can be read just using a URL. E.g., the URL `http://erp.example.org:7488/documents/D-000123@00?format=pdf` returns the document with number `D-000123` and index `00` as a PDF.

4.4 Load balancing with multiple web service workers

If web service job groups are defined in the *CONTACT Catalyst EAI Gateway* configuration file, then a permanently running worker process is started for each of these job groups. Each of these workers waits for the URL of one of its endpoints to be addressed (each endpoint is defined by the `path` option of a job in the group). As soon as an endpoint is addressed, this leads to a synchronous execution of the function behind it and, if necessary, to the return of the response. Only then the process listenes again for further input. So if many requests are sent to the endpoints of one worker at the same time, the system queues them and processes them sequentially.

This bottleneck can be avoided by defining a number of job groups and thus a corresponding number of worker processes. On the one hand, this makes it possible to distribute different endpoints to their own workers respectively, and on the other hand, it also makes it possible to define a number of workers for one endpoint.

4.4.1 Configuration in *CONTACT Catalyst EAI Gateway*

This is made possible in the first place by the `port` job group option, which places each job group on its own port. Multiple job groups with the same job paths (as well as different ones) can thus be addressed via their respective different ports.

The procedure for splitting multiple endpoints into their own workers looks like this: each of the jobs in a previously shared web service job group is placed in its own job group. The example from *Interface to query objects* (page 75) then looks like this:

```
XGW_JOBS=[
    { # job configuration for users list endpoint
        'job_interval': 0,
        'group_id': 'http_users',
        'port': 7488,
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.beispiel',
        'job_list':
        [
            {
                'job_id': 'http users list',
                'path': '/users',
                'schema_files': [ 'users.xgs' ],
            }
        ]
    },
    { # job configuration for http single user details
        'job_interval': 0,
        'group_id': 'http_user_details',
        'port': 7489,
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway.beispiel',
```

(continues on next page)

(continued from previous page)

```
        'job_list':
        [
            {
                'job_id': 'http user details',
                'path': '/user',
                'custom_xml_tags': 'xgw.users',
                'schema_files': [ 'user.xgs' ],
            }
        ]
    },
    [...]
]
```

The two endpoints can then be addressed using the URLs `http://serverhost:7488/users` and `http://serverhost:7489/user` and thus be run in parallel.

4.4.2 Configuration for load balancing in *CONTACT Catalyst EAI Gateway*

The same applies if the processing of a single job is to be distributed among several parallel worker processes. The job group configuration of the endpoint in question must be duplicated to the number of workers desired, with each group assigned its own port.

```
XGW_JOBS=[
    {
        'group_id': 'webservice_1',
        'port': 7488,
        'job_interval': 0,
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'job_list': [
            {
                'job_id': 'webservice_1',
                'path': '/test',
                'schema_files': [ 'webservice.xgs' ]
            }
        ]
    },
    {
        'group_id': 'webservice_2',
        'port': 7489,
        'job_interval': 0,
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'work_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'job_list': [
            {
                'job_id': 'webservice_2',
                'path': '/test',
                'schema_files': [ 'webservice.xgs' ]
            }
        ]
    }
]
```

It should be noted that with this configuration the functions behind the endpoint are executed in parallel. This should be taken into account when modifying shared data (for example, when checking out document files to a shared temporary directory). If necessary, it should be ensured that each of these distributed job groups has its own working directories or is not otherwise sharing data.

4.4.3 Configuration in the Web server

To achieve HTTP(S) load balancing with the configuration shown above, established solutions can be used. Well-known solutions for this are listed in the CONTACT Elements manual *Installation and Operation*. In the following, a configuration for the Apache server with the module `mod_proxy_balancer` is described as an example. Already with a few lines, it provides an alternating load balancing over all ports of the previously configured worker processes, if the mentioned modules are loaded.

```
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

Now a balancer cluster can be defined, which contains the ports defined above. The load balancing method `lbmethod=byrequests` distributes each incoming request to the next member respectively.

```
<Proxy balancer://eai_cluster>
       BalancerMember http://host:7488/
       BalancerMember http://host:7489/
       ProxySet lbmethod=byrequests
</Proxy>
```

Finally, a (reverse) proxy configuration is created making the cluster accessible under a common endpoint. The example exposes the balancer cluster of the web service `/test` configured above under the endpoint `/endpoints/test`.

```
ProxyPass /endpoints balancer://eai_cluster
ProxyPassReverse /endpoints balancer://eai_cluster
```

With this configuration, all incoming requests to the `/endpoints` URL or paths below it are forwarded alternating to the respective next node in the configured cluster. Accesses to `http://host/endpoints/test` are thus alternately distributed to the `webservice_1` and `webservice_2` job groups.

For more configuration options, please refer to the official documentation for `mod_proxy_balancer`.

4.5 Extending Response Handler

The default behavior of the *CONTACT Catalyst EAI Gateway* ignores fields when processing responses from the ERP system, if they have no equivalent in the CONTACT Elements side field configuration of the ERP system. This example demonstrates how the TagHandler for *response* elements can be extended to reject the response in such a case.

To include a custom handler, a corresponding Python module must first be created, which must be included in the job configurations to be changed. In this example, we use the path *$CADDOK_BASE/site-packages/example/xgw.py* for the module of the custom handler. Within the module, the appropriate class definition can then be created.

```
from cs.xml.SchemaParser import CustomTagHandler

class HandleUndefinedFields(CustomTagHandler):
    def process_object(self, object_):
        self.call_default_handler(object_)
```

This must now be made known in the job configuration of the *CONTACT Catalyst EAI Gateway* by specifying the fully qualified Python name:

```
{'response': 'example.xgw.HandleUndefinedFields'}
```

When a response arrives, the behavior of the *CONTACT Catalyst EAI Gateway* does not change, but when processing the object elements, the custom handler is now employed. Now the individual fields can be checked within the handler. Let's first introduce a function that checks if a field name exists for a given object class within the EAI field configuration. For this we use the class *cs.erp.protocol.Protocol*:

```
def is_field_defined(protocol, type_, name):
    for field_def in protocol.AllFields:
        if field_def.get_Class() == type_ and field_def.get_Name() == name:
            return True

    return False
```

The function checks, if the list of fields contains an entry corresponding to the provided field.

The function can then be used to create a list of undefined fields.

```
from cs.xml.SchemaParser import TagHandler, tagname
from cs.xml import SharedObject
from cs.xml.xmllib import XQ as QName

def get_tag_name(node):
    try:
        return tagname(QName(node.tag))
    except:
        return node.tag

def get_undefined_fields(object_):
    protocol = SharedObject.SharedObject.sharedObject().system.get_protocol()

    # Generate a list of undefined fields
    undefined_fields = []
    for field in object_:
        if get_tag_name(field) == 'field' and \
            not is_field_defined(protocol, object_.attrib['type'], field.attrib[
↪'name']):
            undefined_fields.append(field)

    return undefined_fields
```

If the list contains any entries, the tag handler will create and append an appropriate error message. If the list is empty (all fields are defined) the default behaviour of the tag handler is invoked.

```
class HandleUndefinedFields(CustomTagHandler):
    def process_object(self, object_):
        undefined_fields = get_undefined_fields(object_)

        if undefined_fields:
            set_result(object_, 99,
                       'Object has undefined fields: %s'
                       % ', '.join(['\'%s\'' % field.attrib['name']
                                    for field in undefined_fields]))
        else:
            self.call_default_handler(object_)
```

CHAPTER 5

ASCII adapter

The ASCII adapter is a component of the *CONTACT Catalyst EAI Gateway* that enables data exchange by means of text files. Here, many details of the format such as field lengths, sequences, field separators, etc. can be defined configuratively. Thus data sets can be processed e.g. as tabular text lists or as CSV files (comma separated values).

Despite the name *ASCII*, processing text files is not limited to that character format. Naturally, special characters such as those in *iso-8859-1* and *cp1252* formats can also be used. The important aspect here is ensuring the correct format conversion between CONTACT Elements and the linked ERP system (see *cs.xml.Destinations.ASCIIDestination* (page 89)).

This adapter replaces the standalone ASCII gateway present up to CIM Database 2.9.7. In the event of a migration, the field and formating configurations can be taken over, but a separate job configuration has to be created. Examples are provided below for orientation in this regard.

Processing of these formats is implemented using a transformation and a destination module (see Chapter *Debugging Destination Pipelines* (page 27)).

Both modules use a shared configuration that describes the format to be processed and generated.

The content below starts by describing the shared configuration where the properties of the field displays are specified. The two modules and their specific configuration options are then described.

5.1 Message configuration

5.1.1 Defining the field format for messages

The *PPS XGW Format* CAD configuration switch normally controls the way data fields to be transferred are displayed. You can activate a special compatibility mode (see *Configuration of the field conversion* (page 88)) to enable an additional conversion of the field formats. This involves specifying the representation within the ASCII format using the *PPS ASCII Format* CAD configuration switch. This CAD configuration switch contains a value created by adding the partial values for the desired properties. The following properties can be controlled (*Field formats* (page 88)):

Table 1: Field formats

Value	Not added	Added
1	Left-aligned character string	Right-aligned character string
2	Left-aligned numbers	Right-aligned numbers
4	Numbers without leading zeros	Numbers with leading zeros
8	Floating point character is a period	Floating point character is a comma
16	Fixed field lengths	Variable field lengths
32	DD.MM.YYYY date format	YYYYMMDD date format
64	Return character allowed in fields	Return character prohibited in fields
128		Exponential notation
256	Variable number of decimal places	Fixed decimal places
512		No separator for floating point

A manual calculation is provided here for use as an example. Suppose we want to get the following display:

- Left-aligned character strings of fixed length
- Right-aligned numbers with configured number of decimal places, leading zeros and a period as the separator.
- Date field in YYYYMMDD format

The result is that the values 2 (right-aligned numbers), 4 (numbers with leading zeros), 256 (numbers with fixed decimal places) and 32 (date format) have to be added together. However, the character strings are to be displayed left-aligned, so the associated value of 1 should not be added. This results in a total of 294, which has to be entered in the CAD configuration switches.

5.1.2 Separator

The individual fields and data records can be delimited by separators as necessary. This simplifies both legibility and machine processing of the generated files. The field separator is defined by the *PPS ASCII Feldtrennzeichen* CAD configuration value in this process; the separator between data records is defined by the *PPS ASCII Satztrennzeichen* CAD configuration value.

The CR (carriage return), NL (new line) and TAB (tabulator) ASCII control characters can be implemented using the following flags: *"\n"* (NL), *"\r"* (CR) and *"\t"* (TAB). This lets you perform actions such as separating a data record by tabs and outputting data records into separate lines using CR+NL.

Typically, commas (*","*) or the *"|"* character are used as field separators and *"\r\n"* (CR+NL) is used as a record separator.

5.1.3 Configuration of the field conversion

In order to achieve extensive compatibility, the ASCII adapter can also carry out its own conversion of the field contents into the ERP format. This property should be used only if an ASCII gateway configuration is being migrated from CIM Database 2.9.7.

This conversion is controlled using a *converter* attribute in the job configuration. The following values can be configured in this attribute:

PPSTalk A special field conversion is carried out based on Catalyst. This requires a connection to the CONTACT Elements server, i.e. a *target* configuration has to be specified in the job definition (e.g. using a `site.xgs` configuration file). This process evaluates the *PPS ASCII Format* CAD configuration switch.

NewConversionType This option activates additional configuration options (see *PPS ASCII True Format*, *PPS ASCII False Format*, *PPS ASCII True Output* and *PPS ASCII False Output* in table.cadkonf_ascii) to control the formatting for boolean values. Here, the first two switches (*PPS ASCII * Format*) specify strings that are to represent a True or False value in CONTACT Elements. If these values are found in fields defined with the field type *boolean* (see manual *CONTACT Catalyst for ERP*), then they are replaced by the corresponding representation from *PPS ASCII * Output*.

An example:

```
XGW_JOBS = [
    { # job list executed in 10 min intervals
        'job_interval': 600,
        'group_id': 'regular_jobs',
        'conf_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
        'processed_dir': '%(CADDOK_BASE)s/etc/xmlgateway/processed',
        'job_list': [
            { # Abgleich part, bom
                'output_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
                'schema_files': [ 'site.xgs', 'destinations.xgs' ],
                'converter': 'PPSTalk',
            },
            { # Einlesen part, bom als Datei
                'custom_xml_tags': 'xgw.SomeTags',
                'input_dir': '%(CADDOK_BASE)s/etc/xmlgateway',
                'schema_files': [ 'site.xgs', 'transformations.xgs' ],
                'input_files': 'parts.xml',
                'converter': 'NewConversionType',
            },
        ]
    }
]
```

If the job attribute *converter* is not defined, then the normal field formatting of the *CONTACT Catalyst EAI Gateway* is used (controlled by the CAD configuration switch *PPS XGW Format*). This procedure should normally always be used.

5.2 Transformation modules

In addition to configuring field formats, the input and output formats of transformation modules have to be defined, such as the field order. This information was stored in a `messages.conf` configuration file in the ASCII gateway up until CIM Database 2.9.7. They have to be converted to a corresponding XGW schema in the course of a migration.

5.2.1 cs.xml.Destinations.ASCIIDestination

The *cs.xml.Destinations.ASCIIDestination* destination module is configured using a schema definition (see *Schema* (page 15)). This is used to transform outgoing XML messages into ASCII or CSV formats.

Here is an example of a destination module configuration:

```
<schema xmlns="http://xml.contact.de/schema/xgwschema/1.0"
        xmlns:xi="http://www.w3.org/2001/XInclude">
    <destination class="cs.xml.Destinations.ASCIIDestination"
                 output_coding="ISO-8859-15"/>
    <destination class="cs.xml.Destinations.FileDestination"
                 filename="%(work_dir)s/%(output_file)s"
                 wait="True"/>
    <object type="part"
            datasource="cs.xml.SharedObject.Factory"
            record_type="raw">
        <field name="PART_NAME"/>
        <field name="teilenummer"/>
        <field name="t_index"/>
        <field name="SHARE_STATUS"/>
        <field name="st_gewicht"/>
        <field name="cdb_mdate"/>
```

(continues on next page)

(continued from previous page)

```
        <field name="has_bomflt_attr"/>
        <field name="CDB::Control::"/>
        <related type="part_of"
                 datasource="cs.xml.BOM.Factory"
                 number_of_records="True"
                 record_type="old">
            <field name="*"/>
        </related>
    </object>
</schema>
```

5.2.1.1 Encoding for text files

In many cases, the linked ERP system uses different character encoding than CONTACT Elements, requiring the outgoing text files to be converted into the ERP system's encoding. The character format used by CONTACT Elements is normally defined by the database configuration and can be checked in the `instance/etc/dbtab` file.

If the character encoding differs from that expected by the ERP system, the 'output_coding' attribute of the destination module can be used to specify the format for outgoing text files. Suppose the linked ERP system uses 'ISO-8859-15' encoding while CONTACT Elements uses the 'cp1252' format. Setting 'output_coding' to 'ISO-8859-15' configures appropriate encoding conversion.

If the 'output_coding' attribute is not defined, no character conversion takes place and CONTACT Elements encoding uses the corresponding database for data transmission.

5.2.1.2 Configuration for messages

For transmitting data, the *cs.xml.Destinations.ASCIIDestination* module uses a configuration file similar to the definition of a schema file for an XML message (see *Schema* (page 29)). This defines which fields are to be transferred in which order. If the "*" placeholder is used in the field element instead of a field name, all of the fields defined in the ERP attribute configuration for the class of the transferred object are transferred in alphabetical order.

In addition to these schema configurations, there are two optional parameters for the *cs.xml.Destinations.ASCIIDestination* module: *number_of_records* and *CDB::Control::<Feldname>*.

number_of_records If *number_of_records* is set to "True" in a related element, the overall number of related objects to be transferred is determined and specified as the last field of the assigned object after a field separator.

CDB::Control::<Feldname> If a field name is specified in this form, the field contents are then output unchanged, bypassing the conversion functions. This allows the output of escape sequences such as \n and \r to partition a data record further using formatting this way. This could be done by defining a field containing a default assignment with the desired escape sequences. This field can be recorded included in the output schema at the desired point using "CDB::CONTROL::<>".

Normally, the processed text stream is passed from *cs.xml.Destinations.ASCIIDestination* to additional destination modules such as *cs.xml.Destinations.FileDestination* (see *Debugging Destination Pipelines* (page 27)) where the destination directory and filename are defined. In addition to this, the "PPS Verzeichnis" and "PPS Filename:<Objektart>" (s. table.cadkonf_ascii) CAD configuration switches remain available for compatibility reasons. These can be used to define the directory and names of the output files.

5.2.2 cs.xml.Transformations.ASCIITransformation

The *cs.xml.Destinations.ASCIITransformation* transformation module is configured using a Config definition (see *Schema* (page 15)). Similar to the schema definition, object-specific field orders are defined in this definition.

In this case, however, they are defined within <*response_object*> tags for better differentiation. This transforms incoming ASCII or CSV files into XML messages.

Here is an example of integrating the ASCII transformation module with configuration data:

```
<config xmlns="http://xml.contact.de/schema/xgwschema/1.0">
    <transformation class="cs.xml.Transformations.ASCIITransformation"
                    input_coding="cp1252"/>
    <response_object type="part">
        <field name="TEILENUMMER" key="True"/>
        <field name="t_index" key="True"/>
        <field name="HOEHE"/>
        <field name="st_gewicht"/>
        <field name="cdb_mdate"/>
        <field name="has_bomflt_attr"/>
        <field name="CDB::ERROR_FLAG" />
    </response_object>
    <response_object type="bom_item">
        <field name="TEILENUMMER" key="True" />
        <field name="t_index" key="True" />
        <field name="HOEHE"/>
        <field name="st_gewicht"/>
        <field name="cdb_mdate"/>
        <field name="benennung" />
        <field name="has_bomflt_attr"/>
    </response_object>
    <response_object type="document">
        <field name="z_nummer" key="True" />
        <field name="CDB::T_INDEX" key="True" />
        <field name="CDB::ERROR_FLAG" />
    </response_object>
</config>
```

5.2.2.1 Decoding for text files

The character format of incoming text files can be converted for the same reason as the encoding for outgoing text files (see *cs.xml.Destinations.ASCIIDestination* (page 89)). The conversion is configured using the optional *input_coding* parameter of the ASCII transformation module.

The encoding of the incoming format is defined here to enable conversion into the CONTACT Elements format accordingly (normally the database format defined in the `instance/etc/dbtab` file).

Failure to specify this attribute means the character format will not be converted.

5.2.2.2 Configuration for messages

At the beginning of each data record, the transformation module expects the object class, such as *part*, followed by a space. This is followed by the data as defined in the <*response_object*> elements for the corresponding *type* object type. For example, a file for confirming the synchronization of three articles, a BOM position and two documents could appear as follows:

```
part |Art-000073|04|commit
part |Art-000081|01|commit
part |Art-000081|01|commit
bom_item |Teil-000181|01|commit
document |Z-0002004|01|commit
document |Z-0001034|01|commit
```

A special CDB::ERROR_FLAG field should be defined in the <*response_object*> definition. This field marks the position in the text files where the type of return messages from the ERP system is distinguished.

The following values can be passed in incoming data records for this field:

commit The message is a commit message. A confirmation is booked in CONTACT Elements indicating that the synchronization was successful.

pcommit The object class is not part; nevertheless, a synchronization of parts is to be confirmed by the message. This often makes sense if information is being saved from the ERP system as a 1:N table for the part master. In addition to confirming the synchronization of parts, the class data is imported into CONTACT Elements just like info (see below).

info The message contains data from the ERP system that is to be updated or newly entered in CONTACT Elements.

cominfo A confirmation of the synchronization is booked in CONTACT Elements and the data is imported (commit and info at the same time)

error The message contains an error message. The system reads up to the defined record separator and passes the error message to an error attribute if one is defined.

If the CDB::ERROR_FLAG field is not defined, the message type is treated as *info*. In the field configuration for the response message, you can specify if defaults are to be used in CONTACT Elements or if empty fields in the message are to overwrite existing data in CONTACT Elements. CONTACT Elements ignores empty date fields when sending.

If a field is defined starting with *CDB::<field name>* and set to a length of *0* in the corresponding field configuration, this prevents a field separator from being read in or searched for before processing the following field. If a field separator is used, this configuration achieves that the two fields before and after this special field, as an exception, are not expected to be divided by the field separator. This configuration can not be used if dynamic field lengths are configured.

If a field is a key field in CONTACT Elements, it has to be marked by the *key="True"* attribute in the Config schema. This allows CONTACT Elements to identify the data record again (also see *ERP system replies* (page 36)).

It is possible that individual parts are referenced in the parts list that have already been written to a transfer file, but still do not yet exist in the ERP system. If different files have been defined for the adjustment of the object types part and bom_item, the job for the part file should therefore be processed in the job configuration of the *CONTACT Catalyst EAI Gateway* before the job for the bom_item file. If the same transfer file was defined for both object types, it is to be expected that parts are referenced in a parts list whose master data were written into the file only later. To avoid this case see also xgw-appl-data-details.

5.3 CAD configuration switches

The following CAD configuration switches control additional configuration aspects of the ASCII adapter. To a large extent, they are available for compatibility purposes to simplify the migration from the ASCII gateway from CIM Database 2.9.7.

Table 2: CAD configuration switches in the ASCII adapter

Name	Value	Remark
PPS Verzeichnis	Directory	Specifies a directory used for exchanging data files when using the ASCII adapter if this is not controlled using destination modules. If variable names are defined in the job configuration, variables can be used by using the Python placeholder %(VAR_NAME)s.
PPS Filename:<Objektart>	STRING	Name of the file where the data records of an object type or class are written. These fields are written in the directory specified by "PPS directory". If the same filename is used for various object classes, the data records are inserted successively into that file.
PPS ASCII Feldtrennzeichen	STRING	Defines a character string that separates the fields of a data record.
PPS ASCII Satztrennzeichen	STRING	Defines a character string that separates the data records in the ASCII file.
PPS ASCII Format	INTEGER	Format in which the data fields are written (see *Defining the field format for messages* (page 87)
PPS ASCII True Format	STRING	Character string from CONTACT Elements for representing a True value.
PPS ASCII False Format	STRING	Character string from CONTACT Elements for representing a False value.
PPS ASCII True Output	STRING	Character string that represents a True value in the ERP system (e.g. "YES")
PPS ASCII False Output	STRING	Character string that represents a False value in the ERP system (e.g. "NO")

CHAPTER 6

CAD configuration switches

The following CAD configuration switches control the behavior of the *CONTACT Catalyst EAI Gateway*.

Table 1: CAD configuration switches in *CONTACT Catalyst EAI Gateway*

Name	Value	Remark
PPS XGW Format	`INTEGER`	Specify the data format for ERP conversion in *CONTACT Catalyst EAI Gateway*. The default value is 16. If 16 is selected, spaces that precede or follow values are removed when evaluating strings.
PPS Info nur Änderung	TRUE,FALSE	The "TRUE" value prevents new records from being created in CONTACT Elements when processing files from the ERP system with an "info" or "cominfo" action; only changes to existing records are allowed. The default value is "FALSE".
PPS Keine Antwort	TRUE,FALSE	If a value of "TRUE" is set here, objects to be synchronized are marked as synchronized immediately after being written out without waiting for the corresponding acknowledgment from the receiving ERP system. "FALSE" causes objects to be synchronized to be marked as waiting for response until the ERP system has acknowledged receipt in a corresponding response file. The default value is "FALSE".
PPS Teileindex	TRUE,FALSE	Setting this switch automatically populates the part index for "commit" and "cominfo" actions (see *ERP system replies* (page 36)).
PPS Single Object Transfer	TRUE,FALSE	Data records are transferred synchronously if this switch is set. The destination modules then have to return an immediate response (see *Additional details on data transfer* (page 39)).
PPS XGW No Attribute	TRUE, FALSE	A Value of "TRUE" prevents "attribute_type" from being output. This can be used to generate a compatible output, e.g. when an external parser cannot cope with the new attribute.