Charles University in Prague
Faculty of Mathematics and Physics

SOFTWARE PROJECT

‹XRouter›

2. Programmer's Guide

Soběslav Benda
Miroslav Cicko
Tomáš Kroupa
Petr Sobotka
Bohumír Zámečník

Supervisor: Mgr. Martin Nečaský, Ph.D.

2011

XRouter – Programmer's guide

Indended audience: programmers who maintain or extend the system
Prerequisite reading: XRouter – Definition

This document describes the implementation of the XRouter system components and subprojects on a level of detail between the architecture view and API reference. It also provides tutorials with examples how to create extensions to the XRouter system.

Implementation notes

In the this chapter you can find detailed description of the implementation of the important modules and classes, as well as some special or interesting algorithms.

XRouter Service

The main class is the `XRouterService` in the `XRouter.ComponentHosting.dll` assembly. It is implemented as a DaemonNT service (see the DaemonNT document) so it can be run as a Windows service.

During the service start-up the `OnStart()` initializes all supporting stuff (eg. TraceLog and EventLog used to simplify logging throughout the service), processes the service's DaemonNT configuration (described in the Administrator's guide) and initializes and start all the components (Broker, Processor and Gateway) which run in their own threads. As the Broker component is started it read the current configuration from the DateStore and this configuration is then provided to other components. In case of any error during the start-up sequence the service gets stopped and the `OnStop()` method is called. The error cause can be found in the log files.

When the service is stopped by the DaemonNT host the `OnStop()` method is called which stops all the components and wait for their termination.

Gateway and adapters

The gateway and base code for adapters is contained in the `XRouter.Gateway` project.

The gateway component follows the `IGatewayService` interface and is implemented in the `Gateway` class. It contains a collection of hosted adapters: `Dictionary<string, Adapter>` indexed by the adapter identifier. Each adapter must be derived from the abstract `Adapter` class which provides the API and some basic functionality.

Gateway class
Gateway is a component which manages communication of XRouter with external systems via various communication means. A gateway can manage several adapters, each specialized in one communication protocol (such as file exchange, Web services, e-mail, etc.). It works as a mediator between adapters and the broker. Incoming messages are passed from adapter to the broker for being processed and outgoing processed messages go the other way. The code in a gateway run from broker or component hosting thread. Each adapter runs in its own thread.

The gateway is started via the `Start()` method and provided with a gateway identifier and a reference to the broker (via the `IBrokerServiceForGateway` interface). For future extensibility the gateway has an identifier (for the case of multiple gateways). The adapter instances are initialized according the the gateway configuration and started here. The `Stop()` method notifies all hosted adapter to terminate.

The `SendMessage()` method is called by the broker in order to notify an adapter to send an output message, it then calls `Adapter.SendMessage()`. The `Gateway.SendMessage()` method accepts the identification of the target adapter, the message itself and some metadata for the adapter. In case the adapter communicates in a synchronous style the method returns a XML response, otherwise null.

The `ReceiveToken()` method is called by adapters which pass incoming tokens for processing to the broker. In case the adapter is synchronous it might also pass a `MessageResultHandler`, a callback which gets invoked when the message is finished and which should ensure returning a reply back to the originating adapter. Reply handlers for waiting adapters are stored in the `waitingResultMessageHandlers` dictionary.

Adapter class
This is a base class providing common code for concrete adapters. An adapter is a component of a gateway specialized in one type of communication with external systems (such as file exchange, web service, e-mail, etc.). It received incoming messages and passes them to the gateway for further processing. Also it send messagess outgoing processed messages away.

The incoming messages can be received either as XML with one of overloaded variants of the `ReceiveMessageXml()` methods, or as plain-text data with the `ReceiveMessageData()` method. All those method receive an input message and wraps it along with optional metadata into a token. The metadata might contains eg. identification of the originating adapter which can then be used for routing.

Outgoing messages can be sent with the `SendMessage()` abstract method.

An adapter is started via the `Start()` method and runs itself (the `Run()` method) in its own thread. The adapter status is indicated by the `IsRunning` property. The `Stop()` method is for stopping the adapter, it modifies the IsRunning status and calls the `OnTerminate()` method which can contains an actions specific to each concrete adapter. As the `Run()` method usually contains an infinite loop, it should frequently check the `IsRunning` status, otherwise the `Stop()` method might wait indefinitely.

Built-in adapters

XRouter comes with some built-in adapters located in the `XRouter.Adapters` project. Custom adapters should to be created within another project and put into a new assembly.

DirectoryAdapter class

The directory adapter provides file input and output within a shared file system directory. It periodically polls for new files in directories under its control. Any succesfully loaded input file is received for further processing and removed from the file system. Also outgoing messages can be saved into files in a specified directory. Each input, resp. output directory is called an input, resp. output endpoint. XML files and text files are supported. Unreadable files or directories are ignored. The adapter is asynchronous and can contain multiple named endpoints equivalent to directories. The adapter can be configured with the following parameters:

```
[ConfigurationItem("Checking interval (in seconds)", null, 0.1)]
private double checkingIntervalInSeconds;

[ConfigurationItem("Input directories", null, new[] {"In", @"C:\XRouterTest\In"})]
private ConcurrentDictionary<string, string> inputEndpointToPathMap;

[ConfigurationItem("Output directories", null, new[] {
  "OutA", @"C:\XRouterTest\OutA",
  "OutB", @"C:\XRouterTest\OutB",
  "OutC", @"C:\XRouterTest\OutC"
 })]
private ConcurrentDictionary<string, string> outputEndpointToPathMap;

[ConfigurationItem("Save only text", null, false)]
private bool saveOnlyText;
```

The `Run()` method periodically checks all the input directories (endpoint) in the interval specified in the `checkinIntervalInSeconds` parameter. In case a file is detected its content (a message) is read, passes as a token to the broker and the file is deleted. In case the file cannot be read it is ignored and check in the next iteration.

The `SendMessage()` saves a given message into a file with a specified file name or uniquely generated one.

EMailClientAdapter class

The e-mail client adapter provides asynchronous sending of outgoing messages inside e-mails via SMTP. It does NOT provide receiving messages (eg. via POP3, IMAP). It can be used eg. for e-mail notifications where a XML message is stored in the attachment of an e-mail. One adapter instance represents a single e-mail template which can be sent to multiple e-mail adresses.

The adapter is only an output adapter (only the `SendMessage()` method is implemented) and it has only a single unnamed endpoint. The adapter instance can be configured with the following parameters. In case the port is not specified (is null) a default port is used (usually 25). The default encoding is UTF-8.

```
[ConfigurationItem("SMTP host", "Host-name or IP adress of the SMTP server.", "")]
public string SmtpHost { set; get; }

[ConfigurationItem("SMTP port", "Port where the SMTP server listens.", 0)]
public int SmtpPort { set; get; }

[ConfigurationItem("Sender address", "Source address of the e-mail message
sender.", "@")]
public string From { set; get; }

[ConfigurationItem("Sender name", "Display name of the e-mail message
sender.", "XRouter")]
public string FromDisplayName { set; get; }

[ConfigurationItem("Subject", "The subject line of the e-mail message.", "")]
public string Subject { set; get; }

[ConfigurationItem("Body", "The message body.", "")]
public string Body { set; get; }

[ConfigurationItem("Recipients", "A list of e-mail addresses of recipients of this
e-mail message.", new[] { "@" })]
public List<string> To { set; get; }
```

HttpClientAdapter class

HTTP client adapter provides a simple RPC-style client which can send messages to remote web services synchronously. The content can be arbitrary XML document (typically SOAP). The client sends the XML content to a service specified by its target URI using a specified SOAP action. The request can be terminated after given timeout. After sending the request a response from the web service is returned.

The adapter is only an output adapter (only the `SendMessage()` method is implemented) and it has only a single unnamed endpoint. The adapter instance can be configured with the following parameters:

```
[ConfigurationItem("Target URI", "URI of the target web service. Example: 'http://
www.example.com:8080/path/'.", "http://localhost:8080/")]
public string Uri { set; get; }

[ConfigurationItem("Content type", "", "text/xml; charset=utf-8")]
public string ContentType { set; get; }
```

```
[ConfigurationItem("SOAP action", "", "")]
public string SOAPAction { set; get; }

[ConfigurationItem("Timeout", "Timeout in seconds.", 60)]
public int TimeOut { set; get; }
```

The HTTP request is a POST request. The messages is serialized in the encoding specified in its declaration, the default is UTF-8. The method call is synchronous and return a response from the called Web service. The reply is supposed to be a XML.

HttpServiceAdapter class

HTTP service adapter provides a simple Web service listener (server). In can receive XML messages in requests from remote clients and respond to them in the RPC style. The XML content can be arbitrary (typically SOAP).

The adapter is only an input adapter (only the `Run()` method is implemented) and it has only a single unnamed endpoint. The adapter instance can be configured with the following parameters:

```
[ConfigurationItem("Listener URI prefix", "It is composed of a scheme (http), host
name, (optional) port, and (optional) path. Example: 'http://www.example.com:8080/
path/'.", "http://localhost:8080/")]
public string Uri { set; get; }
```

The `Run()` method initializes a `System.Net.HttpListener` which acts acts a native Web service. The incoming request containing the message is passes to the broker as a token and the adapter creates a handler for the reply from the MessageFlow's terminator node which is then returned back as a HTTP response. The response ContentType is the same as in the request. In case the connection is lost during message processing the reply handler throws and exception.

Broker

The broker component is represented by `BrokerService` class in the `XRouter.Broker` namespace. It acts as a mediator between the gateway and the processor components and provides some services to them.

BrokerService class

The `BrokerService` class implements several interfaces, each one for each consumer type. The `IBrokerServiceForGateway` interface which is used by the gateway, it provides the application configuration and receives tokens the gateway. A received token is stored into the DataStore and sent to a dispatcher (class `Dispatcher`).

The `BrokerService` class also implements the `IBrokerServiceForProcessor` interface which provides methods for updating a token in database (updating message flow state, adding messages and exceptions), sending messages from the processor to adapters in the gateway and obtaining XML resources and the application configuration.

Dispatcher class

The dispatcher subcomponent is used for assigning a message flow to the token and sending the token to the processor. In case of extending XRouter to a distributed scenario where each component might run on a different machine the dispatcher becomes useful as a load balancer and to support failure recovery.

Processor

The processor component is represented by the `ProcessorService` class in the `XRouter.Processor` namespace located in the `XRouter.Processor` project.

ProcessorService class

`ProcessorService` is a multi-threaded implementation of the processor component. It manages several single-threaded processors which do the actual token processing. It holds a single message flow version according which to process the tokens.

The processor contains a thread-safe collection of tokens (`BlockingCollection`) to be processed shared by producers and consumers of tokens.

In the `Start()` method the Processor gets the configuration from the broker in which it gets the MessageFlow instructions and the number of consumer threads. The shared collection of tokens is then created and the consumers (`SingleThreadProcessor`) are started. Each consumer receives its own copy of MessageFlow so that no locking is needed as it would if it were shared. In the `Stop()` method the processor at first changes its state (`isStopping`) and waits until all the tokens are finished. Then it stops its work.

Tokens are put into the shared collection using the `AddWork()` method, but only if the processor is not in the state `isStopping` (otherwise an exception is thrown in the broker).

SingleThreadProcessor class

The `Run()` method of the `SingleThreadProcessor` consumes tokens from the shared collection.

Shortly after a token is entered into the collection it is consumed by one of the single-thread processors which then executes the next step of the token's MessageFlow. The token contains a pointer the the next MessageFlow node which is updated after the step is processed. If the token is not finished yet it gets returned back into the shared collection. During the processing a token may go in the shared collection many times and each step might be performed by a different processor thread.

Node class

Represents a single node of the message flow graph. This is a base class with common functionality for concrete message flow nodes. A node instance is specific to a single processor.

EntryNode class

Represents a special type of node that is the exclusive entry point of any MessageFlow graph. It just points on another node where the processing should begin.

CbrNode class

Represents a node in the message flow which performs content-based routing. It decides to which one of several possible nodes to go based on the contents of the token. The `SchemaTron` library is used as a validator. `SchemaTron` is our own native C# validator of the Schematron language. For more information please see the SchemaTron document.

A CBR node has one input edge and several possible output edges (it is a kind of switch). Each output edge (branch) corresponds to a single validator. As evaluation of the token CBR validates a selected document from the token with the validators in a sequence. The first validator with positive result determines the next node.

Resources referenced from Schematron schemas in validator to be included are searched for and obtained from the XRM.

In case the validated document is not valid under none of the validator the flow can continue to a node configured as default.

CbrEvaluator class
Determines the next message flow node based on the content of the input message and CBR node configuration. In other words it determines where the message should be routed to.

Each router branch corresponds to one message flow node where the processing should continue. If the input message does not branch to any branch the default branch is used.

Internally the input message (XML document) is validated using Schematron schemas. Each schema corresponds to a single branch.The CBR configuration connects each schema with a target message flow node. Branches are ordered and processed in this order. The first matching schema determines the matching branch.

The class contains nested a class, `XrmInclusionResolver`, which implements XRouter Schematron schema inclusions – it searches the referenced documents in the XRM. Thus `XrmUri` is used as its URI.

ActionNode class
Represents a node in the message flow which performs one or more actions with the processed token – in parallel.

Actions to be performed in this node are implemented as action plugins and configured specifically for each action node.

All actions of the node are held in a collection of `IActionPlugin` instances. During the processing all the actions are done in parallel (using `Parallel.ForEach`).

SendMessageAction class
`SendMessageAction` is the basic action that is provided by the MessageFlow and that enables interaction with the gateway, more specifically sending output messages during the processing.

The following parameters can be set for this action:

```
[ConfigurationItem("Target gateway", null, "gateway")]
private string targetGatewayName;
```

```
[ConfigurationItem("Target adapter", null, "directoryAdapter")]
private string targetAdapterName;

[ConfigurationItem("Target endpoint", null, "output")]
private string targetEndpointName;

[ConfigurationItem("Message", null, "token/messages/message[@name='input']/*[1]")]
private TokenSelection messageSelection;

[ConfigurationItem("Metadata", null, "token/source-metadata/file-metadata")]
[private TokenSelection metadataSelection;

[ConfigurationItem("Result message name", null, "")]
private string resultMessageName;

[ConfigurationItem("Timeout (in seconds)", null, 30)]
private int timeoutInSeconds;
```

Properties starting with the "target" prefix identify the endpoint in a gateway, or more specifically an adapter, on which method `SendMessage()` should be called. Message and metadata selections specify the subsets of the token that should be passed to the gateway. `ResultMessageName` is the name of message that may be added into the token by performing an action (ie. after calling an RPC-style Web service).

XsltTransformationAction class
Message flow action which transforms a message into another message using XSLT 1.0. The XSL transform is obtained from the XML resource manager.

The following parameters can be set for this action:

```
[ConfigurationItem("XSLT", null, "//item[@name='xslt']")]
private XrmUri xlstUri;

[ConfigurationItem("Input message", null, "token/messages/message[@name='input']/
*[1]")]
private TokenSelection inputMessageSelection;

[ConfigurationItem("Output message name", null, "output")]
private string outputMessageName;

[ConfigurationItem("Is XSLT trusted", null, false)]
private bool isXsltTrusted;
```

- `XsltUri` – identifies the XSLT in the XML Resources
- `InputMessageSelection` – specifies which subset of the token should be given to the XSLT processor
- `OutputMessageName` – identifies the result of the XSLT as a new message
- `IsXsltTrusted` – allows the usage of external scripts in the XSLT - right now it is there just for experimental reasons

`XslCompiledTransform`, the .NET native XSLT implementation, was chosen as the XSLT processor as it is an effective solution suitable for XRouter.

TerminatorNode class

Represents a node in the message flow in which processing of the token is finished. The terminator node might return one of the messages in the token as an output message (if configured so). That is important eg. for the `HTTPServiceAdapter` which has a reply handler waiting for message processing to finish in order to send the response to the caller. If the terminator is supposed to return the answer its `IsReturningOutput` property must be set to `true`.

Common

The `XRouter.Common` project contains some common core types. This includes all types used for representing configuration (mainly the `ApplicationConfiguration` class and other classes used by it. An important part of configuration is the message flow (represented by the `MessageFlowConfiguration` class). Furthermore, it contains classes for representing tokens (mainly the `Token` class), XML Resource Management (XRM), logging services and common utilities (for serialization and plugins support).

ApplicationConfiguration class

ApplicationConfiguration class is an object-oriented wrapper over configuration in xml. Xml content itself is directly accessible through property Content. All methods are just utilities to read and modify this xml in convenient object-oriented manner. Configuration includes settings for Gateway (configuration of its adapters), Processor, registered plugins (adapter types and actions), message flow and XRM content.

MessageFlowConfig class

MessageFlowConfiguration class represents configuration of message flow. It includes message flow guid, configuration of individual nodes and configuration for layout algorithm.

Token class

Token class represents a token. It is a wrapper over xml content which constitutes token itself. All methods and properties are just utilities to read and modify this xml in convenient object-oriented manner. Token contains messages, exceptions during processing, time of creatinon, receiving by broker, dispatching and finishing and also message flow state (current processing node).

XRouter Manager service

The XRouter Manager service consists of multiple subservices for monitoring and management of a single XRouter Service and is implemented as a DaemonNT service (see the DaemonNT document).

The services contained in the XRouter Manager are:

- ConsoleServer – remote management of an XRouter Service
- Watcher – monitors an XRouter service and possibly starts it again
- Reporter – creates and sends summary reports via e-mail to the administrator

The service is implemented `XRouterManager` class in the `XRouter.Manager` namespace in the `XRouter.Manager` project (the `XRouter.Manager.dll` assembly).

XRouterManager class

The class XRouterManager is derived from DaemonNT.Service and implements its abstract methods OnStart() and OnStop().

In the OnStart() method the expected DaemonNT service configuration is processed. The configuration format is specified in the Administrator's guide document. In case the configuration is not alright an exception is thrown and the service is not started. Also the Watcher, ConsoleServer and Reporter subservices are started here. If some of them fails to start, the main service is stopped. The method OnStop() gracefully terminates all the subservices.

ConsoleServer class

The ConsoleServer class implementats and hosts an RPC-style Web service used by the XRouter GUI client. This Web service is implemented as a WCF Service with the WSHttpBinding. The only configurable parameter is the URI prefix where is listens. The service can be run within a new thread via the Start() method and stopped via the Stop() method.

The ConsoleServer Web service has the following C# interface:

```
[ServiceContract]
public interface IConsoleServer
{
  [OperationContract]
  string GetXRouterServiceStatus();

  [OperationContract]
  void StartXRouterService(int timeout);

  [OperationContract]
  void StopXRouterService(int timeout);

  [OperationContract]
  ApplicationConfiguration GetConfiguration();

  [OperationContract]
  void ChangeConfiguration(ApplicationConfiguration config);

  [OperationContract]
  EventLogEntry[] GetEventLogEntries(DateTime minDate, DateTime maxDate,
    LogLevelFilters logLevelFilter, int pageSize, int pageNumber);

  [OperationContract]
  TraceLogEntry[] GetTraceLogEntries(DateTime minDate, DateTime maxDate,
    LogLevelFilters logLevelFilter, int pageSize, int pageNumber);

  [OperationContract]
  Token[] GetTokens(int pageSize, int pageNumber);
  [OperationContract]
  void UpdatePlugins();
}
```

The GetXRouterServiceStatus() method returns the status (running, stopped, ...) of the XRouter Service instance associated with the Watcher.

The StartXRouterService() method starts the associated XRouter Service. It waits until XRouter Service status becomes Running. In case the specified timeout (given in seconds) expires it throws an exception.

The StopXRouterService() method gracefully stops the associated XRouter Service. The method waits until the XRouter Service status becomes Stopped. In case the specified timeout (given in seconds) expires it throws an exception. It also disables automatic starting of the service from Watcher, so that is enables doing a controlled shutdown. In combination with the StartXRouterService() method it enables controlling the XRouter Service running remotely (eg. from the XRouter GUI).

The GetConfiguration() method returns the current application configuration.

The ChangeConfiguration() method replaces the application configuration in the DataStore. In order to take effect of the modified configuration it is necessary to reastart the XRouter Service.

The GetEventLogEntries() method reads and selects EventLog entries according to the given criteria. The GetTraceLogEntries() works analogically for TraceLog entries. The log entries are read from files produces by the XRouter Service. Even currently locked files can be read but sometimes (especially for the TraceLog) an inconsistency might occur, since the entries are not written atomically. This could be solved by adding a new trace log storage saving the logs to a database.

The UpdatePlugins() method searches for plugins (adapters and actions) from some specific plugin directories and registers new plugins into the provided application configuration and returns the updated configuration. Any missing plugins are removed from the configuration.

Watcher class

The Watcher is the simple service, which periodically watches the status of a single associated XRouter Service instance and can automatically tries to start the service if it seems to be stopped. The automatic starting can be enabled in the contructor via the autoStartEnabled parameter. You can disable it temporarily disabled (until the next start) using the DisableServiceAutoRestart() method. On each automatic start an optional e-mail notification is sent. It is possible to get the last known status of the managed service via the ServiceStatus property. Currently, Watcher only supports the Windows service mode, not the DaemonNT debug mode.

The service is started by the Start() method. A new thread running the RunWatcher()

method is created whose task is to periodically check the XRouter Service. The Stop() method stops this thread.

The RunWatcher() method checks (approximately every second) the state of the XRouter Service (using the System.ServiceProcess.ServiceController class). If the associated service is in the Stopped state for the given time (approximately 10 seconds) it tries to start the service again (via the RestartAutomatically() method). The administrator might be optionally notified about this via e-mail.

Reporter class

Reporter is a simple server which periodically generates summary reports from the XRouter Service logs and sends them via e-mail. This is done once a day at a configured time. The reports contain mainly the number of errors and warnings in event and trace logs on the previous day.

The service is started by the Start() method in which a new thread is created, running the Run() method, whose task is to periodically check the logs of the XRouter Service and create reports. The Stop() method stops this thread.

The Run() method tries every second if it is the suitable time to send a report (if the current time is the same as the time in the configuration on a minute granularity). When a report should be sent logs from the previous day are examined for the number of errors and warnings. The report is then sent to the configured e-mail address.

EMailSender class

Provides a sender of e-mails (from templates) via SMTP. It support only the plain unencrypted SMTP without authorization. It is used in Watcher and Reporter.

ConsoleServerProxyProvider class

Represents a remote proxy to the ConsoleServer of an XRouter Manager instance.

XRouter GUI

The GUI application for XRouter configuration and management is implemented in the XRouter.Gui project. It is a WPF application which acts as a client of an XRouter Manager service instance. It needs to connect to the XRouter Manager in order to download or upload the configuration. The main task is to view and modify the configuration. It can also use the XRouter Manager to start and stop the XRouter Service and get its run-time status. Moreover, the application can also retrieve logs and tokens from the DataStore.

The ConfigurationManager class is responsible for providing access to configuration for GUI controls. It creates and holds a proxy to a server implementing the IConsoleServer interface located in XRouter Manager. All service operations provided by console server are accessed through this proxy.

The GUI uses the ObjectConfigurator library for reflection-based automatic creating of editor components for editing adapters and actions in the message flow. Other configuration settings are edited using manually created controls.

Perhaps the most interesting part of the GUI is the message flow editor. It uses the

SimpleDiagrammer library for displaying and editing message flow graphs.

SimpleDiagrammer

SimpleDiagrammer is an independent (it only depends on ObjectConfigurator) library for interactive visualization of oriented graphs. An important feature of this library is that an actual graph does not have to be represented in any concrete object model. Graph nodes can be objects of any type and the same holds for edges. They do not even have to implement any interface, so it is suitable to visualize existing graphs without modifying their object model. This is accomplished by providing a mapper from a real object model of a graph to a description needed to visualize it.

The user of SimpleDiagrammer must derive the GraphPresenter<TNode, TEdge> abstract base class which provides information about an actual graph. It includes methods for getting actual nodes and edges. Also, it contains the CreateNodePresenter() factory method for creating description for visualization for a given actual node. Similarly, it contains the CreateEdgePresenter() factory method for creating description for visualization for a given actual node. Any change in a graph (added/removed nodes/edges) can be notified by calling the RaiseGraphChanged() method so that SimpleDiagrammer knows that it needs to refresh the visualization.

Description of a node, the NodePresenter<TNode> class, can give each node a visualization content to be shown (the Content property). A description of an edge (the EdgePresenter<TNode> class) contains a reference to the source and the target node of the edge.

When a graph is described by classes deriving from GraphPresenter<TNode, TEdge>, NodePresenter<TNode> and EdgePresenter<TNode>, it can be displayed in a WPF Canvas by calling GraphPresenter<TNode, TEdge>.CreateGraphCanvas(). It will return a WPF control (the GraphCanvas class) which creates and manages WPF controls representing nodes and edges. Nodes and edges controls are placed on a Canvas. Location of nodes is determined by a layout algorithm (which must derive the LayoutAlgorithm abstract class) and update regularly in 50 ms intervals (20 fps).

Currently, the ForceDirectedLayout class implementes a layout algorithm simulating repulsion and attraction forces in order to position the nodes. All nodes are repulsed from each other like reversely charged particles, while edge-connected nodes are attracted like with a spring. Nodes are being moved around until the forces reach a stable state. This algorithm is inspired by a blog post available at http://www.brad-smith.info/blog/archives/129.

ObjectConfigurator

ObjectConfigurator is an independent library which makes configuration of objects easier. It allows taking a configuration of an object (values of fields and properties) and representing it in XML. Later, this XML configuration can be injected back into the object. So it can serve as a XML serializer for values of selected fields and properties. An even more important functionality of ObjectConfigurator is dynamic generation of WPF controls which allow editing the XML configuration.

XRouter uses ObjectConfigurator to (de)serialize configuration of adapters and actions and allow its editing in a GUI.

ObjectConfigurator makes configuring objects very easy. It is only necessary to mark fields and properties which should be configurable with the `ConfigurationItemAttribute` attribute. This attribute requires the name of the configuration item (which will be displayed to the user), description and a default value.

Configurator class

`Configurator`, the main class, provides following main methods:

```
public static XDocument SaveConfiguration(object sourceObject)
```

Gets the values of configration items from the given object and serializes them into an XML document.

```
public static void LoadConfiguration(object targetObject, XDocument
config)
```

Deserializes values of configuration items from the XML document and injects them into the given target object.

```
public static ConfigurationEditor CreateEditor(Type targetType)
```

Creates a WPF control for editing configuration items of the given type.

So getting a current configuration from an object and storing it in an XML is as easy as calling `SaveConfiguration()`. Similarly, reading stored values from a XML and putting them back to an object is completed with a call to `LoadConfiguration()`.

Values stored in a XML can be edited with an editor which is created by calling `CreateEditor()`. The generated editor contains textboxes, drop-down lists and other controls allowing to edit the values of the configuration items. Values can be loaded from a XML into an editor by calling `LoadConfiguration()`. Similarly, values edited in an editor can be saved to an XML by calling `SaveConfiguration()`.

ObjectConfigurator supports the following types of configuration items:
- primitive numeric types like `byte`, `int`, `double`, `decimal`, etc.
- `boolean`
- `string`
- enums
- collections implementing `ICollection<T>`
- dictionaries implementing `IDictionary<TKey, TValue>`

It is possible to support more types by implementing the `ICustomConfigurationItemType` interface which defines how to serialiaze, deserialize the item and how to create a GUI editor for a given type. Description of a new supported type is registered by adding to the collection in the `Configurator.CustomItemTypes` static property.

Configuration items can have specified validators which verify correctnes of values and the editor can display an error to the user if a validation fails. The available validators are:
- `RangeValidatorAttribute` – restricts the allowed range for numeric value
- `RegexValidatorAttribute` – restricts string to a given regular expression
- `CountRangeValidatorAttribute` – restricts collections to given minimal and

maximal count of items

More validators can be created by deriving the `ValueValidatorAttribute` base class.

SchemaTron

See the SchemaTron document for more details on the SchemaTron project.

DaemonNT

See the DaemonNT document for more details on the DaemonNT project.

XRouter.Data – database

Introduction

The Data Layer, located in the name space `XRouter.Data`, fulfills the need for persistent storage where the data for every important action made by XRouter are stored. That is important both for keeping track of history and for safely managing data needed for further processing.

XRouter itself is independent of the concrete Data Layer implementation. It communicate with it via an interface, the `IDataAccess`. In this interface there are methods for both saving data as they are created by XRouter and for getting the current data back using various filters.

What to use

While anything capable of implementing the `IDataAccess` interface might be used, there are several abilities the implementation should have in order to satisfy persistence, scalability and safety requirements, while being easy to manage. For example using files is a way to persistently store the data, however serialization and deserialization would also have to be implemented and larger amount of data would either generate a lot of files (which is not very efficient or scalable) or bring many locking issues. Any queries would have to be hard coded as the file system itself does not enable working with XML data directly. On the other hand using a database would solve many of these problems. And if one would go further and picked a database that supports XML data (thus safely storing them without changes) and querying them directly, they would of course fulfill all the given requirements.

There of course exist many such database servers, often offering functionality which could be later used for further enhancement of what XRouter has to offer. One of them is Microsoft SQL Server 2008 R2. And already having decided that XRouter would run on Microsoft platform, and having found out that thanks to the MSDN Academic Alliance license it was allowed to use the full version for free, Microsoft SQL Server was the obvious choice.

Furthermore, the developer version of SQL Server also includes the SQL Server Management Studio (a very powerful database development and management tool), great integration with Visual Studio (one might even make their own CLR types and procedures using C# code) and with Windows itself, SQL Server Integration Services (an advanced service used for ETL operations, again integrated in VS), SQL Sever Reporting Services (reporting tool with VS and web integration) and many more useful features.

Data Access Object (DAO)

There are currently two `IDataAccess` implementations – the `MsSqlDataAccess` class serves as a Data Access Object for the Microsoft SQL Server 2008 R2 and `MemoryDataAccess` (created only for development and debugging purposes) stores everything only in memory.

A directly user of the `IDataAccess` interface is `Persistence` class which provides methods for storing concrete XRouter-specific data. `MsSqlDataAccess` calls stored procedures with the help of its `ExecuteProcedure()` method. This method connects to the server, runs the specified procedure with given parameters and returns a structure which holds a `SqlDataReader` for the response and which can later close both the reader and connection to the server. Opening and closing connections to the server repeatedly is not a problem, because the server uses SQL Server Connection Pooling and so it does not actually close the previously used connections, but instead it reuses them.

Each procedure in the database that is supposed to be ran from outside has its own method in the `MsSqlDataAccess` class that only prepares parameters into a `SqlParameter` array and calls `ExecuteProcedure()` with the corresponding procedure name and parameters.

Also not that all SQL-related exceptions are propagated outside and can be either handled on a top-level catch or somewhere between.

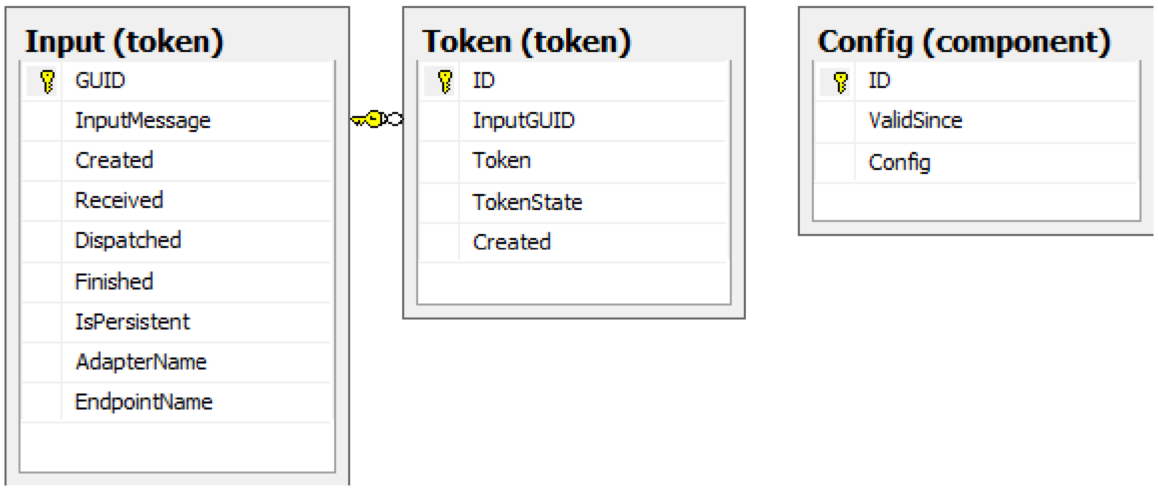
Schema

The database used by XRouter is named `XRouter` and consists of three schemas:

- *token* – holds all information about token processing done in XRouter
- *component* – contains the configuration of XRouter which describes each component and also contains the current message flow
- *executable* – houses all externally accessible procedures

Tables

- `component.Config`
    - `ID` (int) database generated identifier
    - `ValidSince` (datetime2) since when is the configuration used, valid configuration has the newest value of this column that is earlier then Now (this way whole history of configuration may be kept)
    - `Config` (XML) XML file containing complete XRouter configuration
- `token.Input`
    - `GUID` (uniqueidentifier) system generated GUID, unique for each input
    - `InputMessage` (xml) the input
    - `Created` (datetime2) when was the first token for the input created
    - `Received` (datetime2) when was the token for the input received
    - `Dispatched` (datetime2) when was the token for the input dispatched for processing
    - `Finished` (datetime2) when was the work caused by the input finished
    - `IsPersistent` (bit) should store all important changes of the token
    - `AdapterName` (nvarchar(100)) name of the adapter that got the input
    - `EndpointName` (nvarchar(100)) name of the endpoint that got the input
- `token.Token`
    - `ID` (int) database generated identifier
    - `InputGUID` (uniqueidentifier) GUID generated for the input the token was created for
    - `Token` (XML) token (used in the system)
    - `TokenState` (nvarchar(50)) state of the token
    - `Created` (datetime2) when was the token stored



Security

All parts of XRouter use the earlier mentioned DAO to access the database. The DAO itself connects to it as user `XRouter_DBAccess`. This user is allowed to log in remotely and by its only role `XRouter_DBAccess_Role` has only rights to `EXECUTE` in *executable* schema and to `SELECT` in others. Therefore the database can be modified only by implemented stored procedures in the schema *executable*.

Indices

In order to fully utilize the power of the database server, there are several indices and keys defined on the tables. Because all the tables keep history and time is used in most queries, each table has a clustered index on that time and then something unique. That also has an added advantage of quick inserts, because new entries are physically added at the end of tables. Indices are also on foreign and primary keys which is a standard place to make them without further analysis.

Of course any further indexing should be done only after the system is in longer use by carefully evaluating its performance and looking for places where adding an index could really give the system a boost. The reason for this is of course the fact that indices are not for free and therefore should be used only where they can “cover their cost”.

Future

The control over the XRouter service can also enhanced by using SQL Server Reporting Services. The reports can be easily created in Business Inteligence Development Studio (basically part of VS 2008 which is installed along with SQL Server 2008). Each report can contain various filters the report user can pick from which are then used as parameters in the main querry (SQL or procedure call). The report also has its own graphical design that defines how the data should be displayed. The reports can there be available (for example) through the report server web page (all of it is available in the installation without further work). There the report user can run the reports he has permission for (it has its own hierarchical permission system using folders) and display or export the results in various formats (ie. Excel and PDF). Reports can also be sent via subscriptions.

In the future there is of course going to be a lot of work doing the basic maintenance, including indexing (as was already mentioned) and somehow removing the older data – either by really removing them or using partitions. Also the users of the system might take the time to define codebooks containing names of external systems, components or adapters used etc., maybe they would also be enabled to add custom data to tokens in the message flow. All of this might later be used for additional higher quality reporting (with filters using the codebooks and so much more comprehensible for the users) and more advanced analyses.

Extending XRouter

In this chapter we describe the points where XRouter is intended for being extended with plugins and also provide tutorials and some example code. More specifically you can see how to create new gateway adapters and message flow actions. Deployment of such plugins is described in the Administrator's guide. Also you can find detailed information how to create new new trace logger storages for DaemonNT in the DaemonNT document.

In addition XRouter is prepared to be modified on a deeper level. Eg. it is possible to create new implementation of the basic components or the PersistentStorage (ie. to support another database system). This is now covered here, however, as its not expected to be a typical extension scenario and it requires deeper knowledge of the XRouter internals.

Prerequisites

The prerequisites for developing XRouter extensions are the following:

- Microsoft .NET Framework 4.0 or newer
- Microsoft Visual Studio 2008 or newer (further abbreviated as MSVS)

Creating configurable properties

Usually a common extension can be parametrized, eg. with a directory name, web-service URI etc. In order to inject this configuration into a common extension instance during its initialization and to automatically create a GUI for editing the configuration, the configurable data fields or properties in the common extension must be decorated with a specific attribute.

This is done using the ObjectConfigurator library. It supports all basic CLR types, such as numbers, string, boolean, then collections, dictionaries, enumerations and custom types. XRouter supports the following custom types: `System.Uri`, `XRouter.Common.Xrm.XrmUri`, and `XRouter.Common.MessageFlowConfig.TokenSelection`.

First you have to import the `ObjectConfigurator` namespace. Then decorate each data member with the `ConfigurationItem` attribute. It may optionally contain three parameters: name, description and a default value. There are some examples:

```
[ConfigurationItem("SOAP action", "", "")]
public string SOAPAction { set; get; }

[ConfigurationItem("Timeout", "Timeout in seconds.", 60)]
public int TimeOut { set; get; }

[ConfigurationItem("Output directories", null, new[] {
            "OutA", @"C:\XRouterTest\OutA",
            "OutB", @"C:\XRouterTest\OutB"
        })]
private ConcurrentDictionary<string, string> outputEndpointToPathMap;

[ConfigurationItem("XSLT", "XRM URI of the XSLT (XPath for selecting the XSLT from
XRM)", "//item[@name='xslt']")]
private XrmUri xlstUri;
```

```
[ConfigurationItem("Input message", "XPath for selecting the input message from a token", "token/messages/message[@name='input']/*[1]")]
private TokenSelection inputMessageSelection;

[ConfigurationItem("Is XSLT trusted", "XSLT trusted", false)]
private bool isXsltTrusted;
```

This is done by reflection, it works both for field and properties and the data members might be even private.

Creating custom gateway adapters

XRouter provides some built-in adapters for exchanging messages with external systems via various protocols (eg. web-services, plain files, e-mails). If it is needed to address new communication protocols or to modify/extend the behavior of the existing adapters you can create your own custom adapters and add them to the XRouter service as plug-ins. In the following tutorial you can see in detail how accomplish that goal. We'll see how to create an example adapter.

Adapters and receiving/sending messages

As for terminology *input* messages are *received* and then processed, whereas o*utput* messages resulting from processing are *sent* away.

There are two possibilities how to structure this two functions. A single adapter type can either both receive and send messages, or the receiving and sending can be divided into two separate adapters. The decision depends on that is more suitable to a particular situation.

Preparing the environment

Obtain `XRouter.Gateway.dll` assembly from a binary release or the whole XRouter source code from the repository. In MSVS create a new project of the *Class Library* type. Add a (file) reference to the `XRouter.Gateway.dll` assembly or optionally also the `ObjectConfigurator.dll` assembly.

Creating a custom adapter

Creating a skeleton of an adapter

Create a new class with the intended adapter name, eg. `MyCustomSoapAdapter`, and make it public. Then copy the following snippet:

```
using System;
using System.Xml.Linq;
using XRouter.Gateway;

namespace Foo.CustomAdapters
{
    [AdapterPlugin("My custom SOAP adapter",
      "Description of the custom SOAP adapter.")]
    public class MyCustomSoapAdapter : Adapter
    {
        protected override void Run() { }
        public override XDocument SendMessage(string endpointName,
          XDocument message, XDocument metadata) { }
    }
}
```

Now notice several things from this minimal code example:

- The `XRouter.Gateway` namesspace have to be imported since it contains the `Adapter` class and the `AdapterPlugin` attribute.
- The adapter class have to be derived from the `XRouter.Gateway.Adapter` abstract class.
- The adapter must be decorated with the `AdapterPlugin` attribute. You can provide user-friendly adapter name and description.
- The adapter class must implement two abstract methods: `Run()` and `SendMessage()` . `Run()` is called from within the base Adapter so it stays protected, while `SendMessage()` is called from outside and is thus public.

Details of implementing the Run() and SendMessage() methods

The `Run()` method has the following signature:

```
protected override void Run();
```

It is called by the base class when the adapter is being started. It is run in a new thread. Activities needed for "listening" for messages are usually done in this method – eg. creating a new channel for listening on a network port, checking for new files in a directory. Such activities usually run in an infinite loop. If there is a need to listen on multiple channel simultaneously, the adapter may create its own threads or use the Inversion of Control mechanism.

The `SendMessage()` method has the following signature:

```
public override XDocument SendMessage(string endpointName, XDocument message,
XDocument metadata);
```

It is meant for sending output messages away. In case the messages produces a reply, it can be returned back from the method. This method should not modify the adapter state. The input parameters are as follows:

- `endpointName` – Name of the adapter endpoint at which the message should be sent. In case there are no endpoints in the adapter, it can be null or empty string. The concrete behavior depends on the implementation of the particular adapter.
- `message` – The content of the message to be sent.
- `metadata` – Metadata generated during the course of processing the token. It is an optional parameter and depends on the actual implementation.

The optional return value contains the reply for the sent message or null if there is no reply.

Implementing the OnTerminate() method

You can optional implement a hook method which is called just after stopping the adapter. It can eg. close a web-service listener, free some resources, etc. It has the following signature:

```
public override void OnTerminate();
```

Inspiration from existing adapters

You can learn and inspire yourself from existing adapters. Take a look into the source repository to the files in the `XRouter/Adapters/` directory, such as `DirectoryAdapter.cs`, `HttpServiceAdapter.cs`, etc.

Deploying an adapter

An adapter is referenced from a message flow via the path to its assembly (relative or absolute). While it is possible to put the adapter assembly anywhere, it might be a good idea to place it along with the XRouter installation – either in the same directory or to a subdirectory Note that it may require Administrator privileges. Using relative path to the adapter assembly reduces the need to update all the message flow configurations when the adapter path would change.

Run-time life-cycle of an adapter

An instance of the adapter follows a particular run-time life-cycle:

- the adapter instance is created and initialized (properties set according to the configuration)
- the adapter instance is ran in a new thread via the `Run()`method
- the adapter instance does its job
- the adapter instance is stopped – the `OnTerminate()` method is called and then the adapter thread is terminated
- the adapter instance is destroyed

Creating custom MessageFlow actions

XRouter enables creating, installing and using custom MessageFlow actions. The following text describes each part of an action and shows how to create a simple one. After reading this you should be able to create your own custom MessageFlow actions. What you need before you start is already described in the common Prerequisites section.

At first run Visual Studio 2010, create a new *Class Library Project*. Add references to the assemblies: `XRouter.Processor`, `XRouter.Common`, `ObjectConfigurator`.

Creating a custom action

Create a new class and name it as your wanted action, in this case `MyCustomAction`. Replace the content of its file with following code:

```
using ObjectConfigurator;
using XRouter.Common;
using XRouter.Processor;

namespace MyCustomActions
{
    [ActionPlugin("Action identifier", "Action description.")]
    public class MyCustomActitivy : IActionPlugin
    {
        public void Initialize(IProcessorServiceForAction processorService) { }
        public void Evaluate(Token token) { }
    }
}
```

In the sample you can see these elements:

Namespaces

These three namespaces are needed:

- `XRouter.Processor` – has interface `IActionPlugin` and attribute `ActionPlugin`.
- `ObjectConfigurator` – contains external settings
- `XRouter.Common` – defines the `Token` class that carries information about processing of each input message

Class declaration

Each class used as a custom action must be annotated by the `ActionPlugin` attribute and must implement the `IActionPlugin` interface. The parameters of the attribute are a unique action identifier and a description.

```
[ActionPlugin("Action identifier", "Action description.")]
public class MyCustomAction: IActionPlugin { }
```

Implementing the body

To implement the body of a custom action you need to:

- annotate configurable properties
- implement the `Initialize()` method
- implement `Evaluate()` method

Description of the configurable properties can be found in the chapter Creating Configurable Properties.

```
public void Initialize(IProcessorServiceForAction processorService);
```

The method is called at the action initialization. The `processorService` parameter is references the processor component as viewed from an action. The method usually contains steps needed to be run before the action can function - like resource loading, interpreting the configuration (changing the action's settings according to the configuration) and storing the processing service reference.

```
public void Evaluate(Token token);
```

The method is called whenever the action is executed. As a parameter it has the token for which the action should by run. In this method we do the purpose of this action – be it further modifying the token, sending something, triggering a remote action (and probably waiting for its response), something else or more than one of the previous things.