



Metada Integration Modeler 2.0

<http://www.metada.com/>

- [1. Product Information](#) 3
 - [1.1. Executive Summary](#) 3
 - [1.2. Benefits](#) 3
 - [1.3. Product Features](#) 3
 - [1.4. License](#) 4
 - [1.5. Roadmap](#) 5
- [2. Development Process](#) 5
 - [2.1. Modeling](#) 5
 - [2.1.1. Types](#) 5
 - [2.1.2. Services](#) 7
 - [2.1.3. Mapping Functions](#) 9
 - [2.1.4. Value Mapping Tables](#) 9
 - [2.1.5. Routing Conditions](#) 9
 - [2.1.6. Systems](#) 10
 - [2.1.7. Input Adapters](#) 10
 - [2.1.8. Output Adapters](#) 11
 - [2.2. Validation](#) 11
 - [2.2.1. Real-Time](#) 11
 - [2.2.2. On-Demand](#) 11
 - [2.3. Generation](#) 12
 - [2.4. Autotesting](#) 13
 - [2.5. Versioning & Synchronization](#) 15
- [3. Maintenance Procedures](#) 16
 - [3.1. Metarepository Installation Procedure](#) 16
 - [3.2. Modeler Installation Procedure](#) 18
 - [3.3. Create Repository Procedure](#) 18
 - [3.4. Create Model Procedure](#) 19
 - [3.5. LDAP Authentication Setup Procedure](#) 19
 - [3.6. User Group Setup Procedure](#) 19
 - [3.7. User Authorization Setup Procedure](#) 20
 - [3.8. Backup Procedure](#) 20
 - [3.9. Disaster Recovery Procedure](#) 20

1. Product Information

1.1. Executive Summary

Enterprises may have tens or even hundreds of applications that are custom-built or acquired from third-party suppliers. The applications may operate on multiple OS platforms and can be geographically dispersed. In order to support common business processes and data sharing across the various applications, these applications need to be integrated. Implementation of a **service-oriented (SOA) integration** platform for control and management of the end-to-end integration is the way how to realize **enterprise systems integration (EAI)** for middle and large organizations.

Metada Integration Modeler applies model-driven software development approach and enables effective zero-code platform-independent integration modeling. Integration models include service definitions, service interfaces, integration flows, message routing, mapping, and transformations. With no need for programming, the tool generates configurations and/or complete program code for the selected integration/orchestration platform (e.g. **BEA AquaLogic Service Bus**, **BEA Tuxedo**, **IBM WebSphere**).

Metada Integration Modeler is a metadata management, data modeling, transformation and integration toolkit which significantly decreases the time to develop and deploy new integration components while reducing the cost and time associated with ongoing support and maintenance. It is a proven tool **for integration analysts** that allows full-scale modeling of organization's **enterprise service bus (ESB)**. It is designed for complex, heterogeneous IT environments and enables to build, deploy and manage a network of IT services by providing a complete system of record of services including contracts, implementation artifacts, and dependencies that control their use.

Metada Integration Modeler offers:

- modeling tools for platform independent description of integration
- model validation
- model testing and autotesting
- generation facilities for documentation, program code, and configurations
- model versioning and synchronization

Metada Integration Modeler is a web-based modeling environment that is always accessible to any number of users with no need to install any software on their local machines.

1.2. Benefits

Fast time to market - Metada Integration Modeler focuses on ease and speed of delivery of new IT services. This enables enterprises to meet business needs quickly and capitalize on market opportunities.

Efficient management of services - All information about systems, services, exchanged data structures, and transformations are centrally described, stored, and managed. Information is readily available to all parties and assists in business planning and business process optimization.

Quality - With wide range of validations, reports, and autotests, Metada Integration Modeler becomes a highly reliable product for managing heterogeneous IT environments.

Platform Independence - Integration models are platform-independent and may be executed in various kinds of integration technologies. It is even possible, during time, to migrate from one integration platform to another.

Low total cost of ownership - Not only the ability to expand various applications into reusable services reduces maintenance and integration costs, but also the model-driven approach used in Metada Integration Modeler that allows integration experts (not programmers) to drive development of enterprise IT services.

1.3. Product Features

To enable full-scale modeling of integration, Metada Integration Modeler provides features covering the complete integration project lifecycle. The lifecycle includes not only support for design of integration services, but also their validation, autotesting, versioning, synchronization, and deployment.

• **Web-based environment**

Any number of integration analysts may take part on development and maintenance of one central integration model or even multiple integration models at the same time. User authentication is either via enterprise LDAP or global OpenID. There is no need to install and maintain any special software on user machines - web-browser is sufficient.

• **Integration services modeling**

Service interfaces are modeled as composable and reusable types. Each type may define tree-structure of data and may be composed of other types. This allows for flexibility and ease of interface definition, while enabling effective reuse. Both XML and non-XML data formats are supported via generic transformations from/into XML.

Integration services decide what backend services will be called using **content-based message router**. When constructing a message to be sent, a message transformation may be applied. **Transformations** include mapping from one tree-structure of data to another tree-structure of data. Different mapping functions and value-mapping tables may be applied during a transformation. Mapping collections of data is supported.

- **Model validation**

Conformance of integration models to specific constraints may be regularly checked. Validations ensure production quality and consistence of models. Validations check compliance to conventions or structural constraints that need to be enforced.

- **Model versioning and synchronization**

In enterprise application integration solution it is necessary to support effective long-term development and maintenance. In production model (the one that defines the current state of the integration), only necessary fixes are usually being realized. Most of the development and changes are performed in development models, which may be migrated to production only after thorough testing. For this reason Metada Integration Modeler supports multiple versions of models that may be synchronized among one-another.

- **Services autotesting**

Each service accepts a message, does some deciding what other services to call, and does some data transformations and message formatting. All this is modeled in the integration model and needs to be repetitively tested. It is necessary that the whole integration model be regularly retested after any sort of changes. In practice it is very expensive to retest systems manually. Metada Integration Modeler provides autotesting feature that allows several automated tests to be specified for each service (usually to test various use cases). The autotest suite may be then executed at any time to check how changes in model have affected only the right integration functionalities. Autotests enable a strong grip on the integration solution.

- **Code/configuration generation**

Metada Integration Modeler currently features full code/configuration generation for **BEA AquaLogic Service Bus** (XQueries and configurations) and for **BEA Tuxedo** (C++ code). Code generation for other integration/orchestration platforms is possible.

1.4. License

License Conditions

Non-exclusive license is provided for Metada Integration Modeler, a software product, which is

- Web-based environment for modeling service-oriented enterprise IT systems integration
- Automatic generator of configurations for BEA AquaLogic Service Bus

License does not include any particular integration model. It is expected that particular integrations model will be defined by integration analysts (third party, or customer's) using the licensed Metada Integration Modeler.

License Policy

- Licensed per installation
- One installation within one organization is sufficient
- Unlimited model number and size
- Unlimited users

There are no additional costs for giving access to users from the customer organization or other cooperating organizations. The license is given for one installation and unlimited users of any kind.

Source Code

- The license provides the licensee with full access to the source code.
- Access is provided to the most current version of the source via secured read-only access to the Metada Subversion repository.
- The license allows the licensee to modify the source code in order to improve the current installation, nevertheless use of the source code apart from the licensed installation is strictly prohibited. On the other hand, Metada could not be held liable for any disfunctions that were caused by such additions or modifications.
- Metada obtains full rights to any source code added to the product code-base or modified by the licensee or its contractors.

Metada, at all times, maintains full ownership of all the product source code. The right to add or modify the source code is given only to remove any potential risks that might have occurred if the chance was not given. On the other hand, Metada is ready to make any adjustments needed for the customer's project and it should not be expected that the customer will have to make any changes alone.

Generated Code / New Generators

- The generated code is not a code to be changed by the customer - it is the final code.

- To make changes in what is being generated, generation templates (XSLT templates) may be modified or added as needed.
- Generation networks may be created and modified as needed.

Support & Maintenance Policy

- After subscription to the support & maintenance plan, licensee is entitled to upgrades of the licensed software.
- Upgrades mean improvements of the modeling environment or model management tools (GUI improvements, visual modeling components, model versioning facilities, etc.).
- Upgrades of specific functionalities are not mandatory (if new functionality is required and is not part of the regular upgrading plan, it needs to be requested as a separate development).

Cooperation and Training

- Solution may be operated both on customer's or Metada infrastructure (on customer's server or "in the cloud").
- Installation/maintenance is provided off-site via VPN connection. On-site support also possible.
- Training program for integration analysts is available.

1.5. Roadmap

We have completed the 2.0 release and are beefing up for further improvements in the releases to come.

Note that the list below is subject to change. This is only a high-level map that shows the direction where the development is heading. There might be additional maintenance releases as well.

Metada Integration Modeler 2.1 (Summer 2008)

- Maintenance release (conversion from version 1.6)

Metada Integration Modeler 2.5 (Fall 2008)

- Services will have one request and one response type (instead of request/response service parameter sets)
- MW service will be able to call other MW services (and not only BE services)
- Full service flows will replace the content-based router
- Asynchronous reply targets will be removed (replaced by full service flows)
- Message transformers will become separate entities reused in flows
- Input and output channels will become system independent
- Message logging setting should be set on each service (Log request, Log response). It should be possible to override the setting on the call service action in the flow.
- Service calls should specify timeout [ms] and priority [0-9] of the calls
- Each service will have a list of "authorized systems" that may call it

Metada Integration Modeler 3.0 (Summer 2009)

- Migration to Metarepository 3.0
- Integrated documenting features (via multi-typing)
- Real-time model interpretation and autotesting

2. Development Process

Metada Integration Modeler is a model-driven environment that enables integration analysts to model, validate, test, and deploy enterprise integration solutions. This section describes the phases in which the integration model is created, what are the model entities defined, what validations are provided, and how the model may be tested and deployed.

The phases of development are the following:

- **Modeling** - Integration model is created by defining services, their interfaces, flows, and message transformations.
- **Validation** - Model is checked for structural constraints and design conventions.
- **Generation** - Artifacts are generated for the target platform.
- **Autotesting** - Automated tests are designed and executed to cover all services and their functionalities, so that when changes are done to specific services, it is easy to check that other services stay intact.
- **Deployment** - Click-of-a-button deployment of the integration solution.

The following sections describe these phases in detail.

2.1. Modeling

2.1.1. Types

Types represent data structures that are communicated among integrated systems. It is possible to define flat data structures with few fields, but also large nested tree structures of data. The type definition also allows for extensive reuse

of data structures that are often repeated, but also allows one complex type to be created with no reuse in case that reuse does not make sense (i.e. routine XSD imports). Types may be either simple or complex. Type definition contains:

- **System** - System that the type is associated with.
- **Name** - The unique name of type is used only for modeling and documentation purposes and it should obey some pre-defined naming convention to ease navigation among types (e.g. which type used for what backend). It is never used as a name of an element or data entity, since when a type is assigned, it is always given a local name.
- **Kind** - Enables distinction of simple and complex types.

Simple type represents one reusable field. Simple type contains:

- **Basic type** - Basic type specifies data type for the field. Available types are boolean, date, datetime, decimal, int, string, and xmlstring.
- **Length** - Maximum length of data in the field. For decimal type it specifies number of digits before the decimal point.
- **Decimals** - For decimal type it specifies number of digits after decimal point.
- **Date format** - Specifies format in which dates are communicated (i.e. yyyyMMdd'THHmmssSSS).
- **Fill with** - Character which will be used to fill remaining space in the value up to the specified length.
- **Justification** - Left or right justification of value in case that "fill with" is used.
- **Sample value** - Sample value for the field used for documentation.

Complex type represents a tree structure of fields and subtypes. Structure of a complex type is modeled via type properties.

Type property is either local or reference to another type. In the modeling GUI, type properties are modeled in a tree widget that visualizes the tree structure of a type. Each type property contains:

- **Prefix** - Namespace prefix to be used for the property and its subproperties.
- **Namespace** - Namespace to be used for the property and its subproperties.
- **Dive** - Defines path to the property in the tree structure of data. Repeated substructures (**Collections**) are marked with an asterisk ("**").
- **Position** - Defines relative position of type property within the tree structure of data.

Local type property represents field that is local to its type and is not reusable in other types. It is possible to design a large nested structure of a type by just using local type properties. It is good in situations where no reuse of any substructure is expected. Local type property contains all attributes that are used to define a simple type (see above), and further it contains:

- **Mandatory** - Defines that the type property value is required (now for documentation purposes only).
- **Attribute** - Defines whether the value should be in XML element or attribute (in case that XML is used).
- **Default value** - This value is used to substitute for a missing value of a field in a message being sent. Value will be formatted according to data type settings of this field. It must be processable by formatting routines. Numeric types must contain valid number, boolean type must contain true or false value, datetime type must be in yyyyMMdd'THHmmssSSS format.

Name or Dive Path	Basic Type	Attr.	Date Format	Default Value
transaction		<input type="checkbox"/>		
header		<input type="checkbox"/>		
timeStamp	datetime	<input type="checkbox"/>	yyyy-MM-dd'THH:mm:ss	
fxcVersion	string	<input checked="" type="checkbox"/>		4.0.0

Figure: Sample type properties

The above figure shows a sample type that defines a structure of nested elements. The "transaction" element contains a "header" element and "fxcVersion" attribute. The "fxcVersion" is of type string with a default value "4.0.0". Further, the "header" contains a "timeStamp" element of type datetime with the specified format.

Reference type property represents link to another type. The linked type then becomes part of the type structure. The "dive" attribute of the type property specifies where in the type structure the referenced type should be placed. The dive also specifies the local name for the referenced type. The name of the referenced type itself is not used. Both simple and complex types may be referenced. Referencing a simple type enables reuse of one field definition in several types. Referencing a complex type enables reuse of whole data structures.

Rules for integration of referenced type into a type:

- Referenced type may be inserted on any dive (even an empty one).
- Properties of the referenced type will be inserted (for each level in the data structure) as one piece (i.e. on a given position in the tree data, there will be no blending of fields from the referenced type with local properties, or properties from other types).

- Properties from referenced type will be placed on a position where the referenced type was placed.

Name or Dive Path	Type	Basic Type	Length	Decimals	Date Format
SYM_RB_TRAN_HIST_QRY_OUT	MSG_SYM_Header_reply	v			
PAGE_OUT					
ROWS_PER_PAGE		decimal			
PAGE		decimal			
TOTAL_ROWS		decimal			
ACCT_NO		string	20		
CERTIFICATE_NO		string	10		
TRAN_HIST_DTLS					
RB_TRAN_HIST_QRY_T*					
POSITION		decimal			
SEQ_NO		decimal			
BRANCH		string	6		
TRAN_DATE		datetime	19		dd.MM.yyyy HH:mm:ss
TIME_STAMP		datetime	19		dd.MM.yyyy HH:mm:ss
EFFECT_DATE		datetime	19		dd.MM.yyyy HH:mm:ss
TRAN_TYPE		string	4		
TRAN_DESC		string	140		
TRAN_AMT		decimal			
TRAN_CCY		string	3		

Figure: Sample type properties with a collection and subtypes

The above figure shows a more complex type that contains a reference to a subtype MSG_SYM_Header_reply and contains collection of RB_TRAN_HIST_QRY_T (marked with an asterisk). The referenced type itself defines a structure with three more other types.

2.1.2. Services

Services represent integration services located in an integration layer or a service bus. Two distinct kinds of services may be recognized: MW services and BE services.

MW service is an integration service that may be called by external systems and contains a content-based router that decides what target BE service will be called and how data will be mapped to it. MW service cannot call other MW services.

BE service is an integration service that represents a service offered by a specific system. BE service specifies an output channel that should be used to communicate with the system. BE service does not contain any service flow and is used purely to communicate with the target system.

Definition of an integration service contains:

- **Kind** - Selection whether service is MW service or BE service.
- **Name** - Name by which the service is identified (in runtime). Via service aliases, it is possible to identify one service under different names for different input channels.
- **Output Channel (BE only)** - Output channel specifies a communication channel through which an external service should be called.
- **Request Format Driver** - Name of the message formatting driver used to assemble the request message that is sent to the target service. Currently "xml" and "xml_all_tags" drivers are supported.
- **Reply Format Driver** - Name of the message formatting driver used to assemble the response message that is sent back to the requesting service. Currently "xml" and "xml_all_tags" drivers are supported.
- **Synchronous (MW only)** - Defines, whether this service will be processed synchronously or asynchronously. Asynchronous processing means that incoming request will be put into a queue and the requester will immediately obtain a "message sent" response. Messages in the queue are then processed separately (i.e. execution of a service flow is performed). This flag also means, that the service does not provide any data as its reply other than the "message sent" response.
- **Will Confirm (MW only)** - Specifies whether an asynchronous service will confirm status back to its caller. For synchronous call overridden to 'false', for asynchronous transactions overridden to 'true'.
- **Active** - Services may be temporarily deactivated.
- **Prefix** - Default namespace prefix of the request and response messages of this service.
- **Namespace** - Default namespace of the request and response messages of this service.

Service parameters

Each service has a service interface defined by a set of request and response parameters. The service parameter attributes are exactly the same as attributes of type properties.

Input defaults

Using input defaults it is possible to default specific values in an input message. This is useful because setting of these defaults does not have to be repeated in multiple mappings within the service.

- **Path** - Path to the field in the message which should be defaulted (set with a value if it does not have one).
- **Value** - Value that should be set.

- **Built-in function** - Function that should be executed and assigned to the field.

Called services

Called services enable modeling of a content-based routing sequence. Each branch in the sequence tests a routing condition and if this condition is satisfied, a call to the specified target service is performed.

- **Called service** - Integration service to be called.
- **Routing conditions group** - Condition that specifies if to call this branch.
- **Position** - Relative ordering of the branching sequence.
- **Request mappings** - Message mappings to be used to map request message.
- **Response mappings** - Message mappings to be used to map response message.

Request and response mappings

Each service call contains set of request mappings and response mappings. Each mapping contains the following attributes:

- **Source path** - Path in the input type (slash separated).
- **Target path** - Path in the output type (slash separated).
- **Substring from** - Number from which position in string to start.
- **Substring to** - Number to which position to take the substring. Use 0 for the remaining characters in the string.
- **Position** - Relative ordering of the mappings. It is important when several values are appended into one target field.
- **Value mapping table** - Table to be used to map values.
- **Mapping function** - Function to be applied to the input value to get the output value.
- **Join type** - Specifies what kind of join operation is used when joining currently mapped value with already present value (created by previous mapping to the same target field). If empty concatenation is performed ('concat'). Useful only when mapping more than one value to the same target field. **'concat'** - concatenates value after value already produced, **'add'** - numeric adding, **'fillin'** - first nonempty value, **'replace'** - replaces value created so far with result of this single mapping, **'temp'** - stores result of this mapping to temporary variable (does not affect value in target field).
- **Default value** - The default value is used when there is no value available for the source path. There are special values that may be used as a default: ***space** - space, ***now** - current date in standard format, ***null** - no value, ***empty** - string of zero length, ***position** - index of current iteration when creating a collection, ***value** - value created by preceding mappings to the target (only for multiple mapping to the same field), ***temp** - temporary value created by previous mapping with the 'temp' join type, ***ctx_paramA** - value of context parameter paramA.
- **Default product** - Default produced value. This value is used if mapping didn't produce any value so far. The value is not modified by value mapping table etc. It is only written to target field (join type applies).
- **Date format** - If source field has a date type, format it to the specified date format.

Source Path	Target Path
	transaction
	header
	timeStamp
	fxcVersion
	body
	us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist
	listTransactions
	Rtti
	Item
	transactionDate
	valueDate
	details
	credit
	amount
	amount
	currency
	externalLiteral
	debit
	amount
	amount
	currency
	externalLiteral
	totalNumberFlows
	pageNumber
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TIME_STAMP	transaction/header/timeStamp
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/EFFECT_DATE	transaction/header/fxcVersion
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TRAN_DESC	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TRAN_AMT	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/CR_DR_MAINT_IND	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TRAN_CCY	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TRAN_AMT	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/CR_DR_MAINT_IND	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/TRAN_HIST_DTLS/RB_TRAN_HIST_QRY_T/TRAN_CCY	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/transactionDate
SYM_RB_TRAN_HIST_QRY_OUT/PAGE_OUT/TOTAL_ROWS	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/totalNumberFlows
SYM_RB_TRAN_HIST_QRY_OUT/PAGE_OUT/PAGE	transaction/body/us.ebu.fxc.subject.accounttrnhistory-SAaccountTxHist/listTransactions/Rtti/Item/pageNumber

Figure: Sample response mapping

The above figure shows a sample mapping from one structure to another where collection of transactions (Item) is being mapped. Notable are also the double mappings to the "amount" element. Join type "fillin" is defined for them in the model (further columns not included in the picture), which means that either TRAN_AMT will be written if available, otherwise CR_DR_MAINT_IND will be used.

Service aliases

Any service may be called by various input channels. Each input channel may use different mechanism to recognize which service is to be called. For each input channel it is possible to define a name alias to be used for the service in case that it is called via the channel.

- **Input Channel** - Input channel for which the alias should be used.
- **Alias** - Name under which the service is identified in the given input channel.

Asynchronous reply targets

Asynchronous service is first called by a client. The client gets a "message sent" response and the request is put into a queue for later processing. When the request is finally processed, a backend service is called and response obtained, the response is mapped and sent based on a routing condition to a specific target service (usually on the client system), to provide answer to the asynchronous request. **Target service** - Name of a service to be called when the reply is ready.

- **Routing conditions group** - Condition that specifies if to call this specific target.
- **Position** - Relative ordering of the branching sequence.

2.1.3. Mapping Functions

Mapping functions are library functions that may be applied on values being mapped in the service mappings. The integration model contains only code and description of functions in order to be able to bind them to specific mappings. The functions themselves have to be realized in a specific programming language of the runtime framework.

- **Code** - Function identification both in model and runtime.
- **Description** - Specification of the mapping function.

Sample mapping functions

- **toUpperCase** - Converts lower case letters to upper case.
- **toMillis** - Converts date in yyyyMMdd'T'HHmmssSSS format into number of milliseconds from 1970-01-01.
- **length** - Returns length of the provided string.

2.1.4. Value Mapping Tables

Value mapping tables (VMTs) are used in mappings to translate an input value into a specified output value. Table may contain any number of input-output value pairs. To handle specific cases, such as if no value comes or the input value is not found in the table, special values (directives) may be used in the table.

Special values:

- ***null** - no value
- ***space** - one space
- ***empty** - empty string
- ***other** - value not in the table

Special values may be used in the table both as input and output values. Therefore, for example, nothing (*null) may be mapped to some specified value, or some specific value may be in turn mapped into nothing (*null). The *other is used to map any other input value (one not specifically mentioned in the table) to a specified value. There is no need to use *other on the output side, because mapping *other to *other is implicit, which means that if value is not found in the table it is sent to the output without change. Throwing such values, if needed, may be accomplished by mapping *other to *null.

#	Input Value	Output Value
1.	*other	7999
2.	0	0
3.	000	0
4.	100	6999
5.	1200	7012
6.	1400	7014
7.	1700	7017
8.	2400	7024
9.	2500	7025
10.	2600	7026
11.	300	7003
12.	4900	7049
13.	5100	7029
14.	600	7006
15.	800	7008

Figure: Sample VMT

The above figure shows a sample value mapping table that is used to map error codes. There are fourteen specific codes that are mapped to a defined value. In case that anything else comes (including null value) it will be mapped to value 7999.

2.1.5. Routing Conditions

Routing conditions are reusable conditions used in services to decide which target services to call. Routings are based on the service's input data (content-based) and therefore each routing condition matches a path in the data to a specified value or to another path.

Routing condition definition contains:

- **Input path** - Path (slash separated) on which a value may be read from input data.
- **Operator** - Operator applied in the condition (IsNull, IsNotNull, Eq, LtGt, Lt, LtEq, Gt, GtEq, StartsWith, EndsWith, Contains, ContainedIn, NotStartsWith, NotEndsWith, NotContains, NotContainedIn).
- **Condition type** - Condition is either applied to a defined value ("**val**") or to a value on another path ("**prop**").
- **Value** - Specific value or path to a value in input data (depending on condition type).
- **Substring from, Substring to** - A substring of the value on the input path may be taken.

Routing conditions are organized in groups of several conditions that are either all to be valid ("and" operator) or one is sufficient ("or" operator). Furthermore, routing condition groups may be linked as subgroups to other routing condition groups, which not only enables reuse of often used conditions, but also construction of more advanced logical operations.

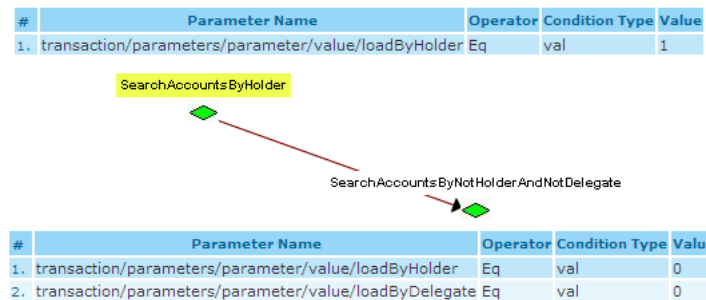


Figure: Routing condition group sample

The above figure shows a sample condition group "SearchAccountsByHolder", which defines condition that the loadByHolder property should be equal to one. The condition group then reuses the "SearchAccountsByNotHolderAndNotDelegate" condition group that defines that both loadByHolder and loadByDelegate should be equal to zero. The model also specifies (not seen in the picture) logical "or" on conditions in "SearchAccountsByHolder" and logical "and" on conditions in "SearchAccountsByNotHolderAndNotDelegate". This leads to evaluation of logical formula "loadByHolder=1 or (loadByHolder=0 and loadByDelegate=0)" in routing where this condition group is applied.

2.1.6. Systems

Systems are all enterprise systems that somehow interact with each other. Based on the interactions, systems may either play the role of clients or backends. Each system definition contains:

- **Name** - Name of the client system used for documentation purposes.

System defaults

For each system it is possible to define default settings per data type:

- **Basic type** - Data type for which the settings apply
- **Length** - Default length for the basic type
- **Decimals** - Default number of decimal digits for decimal basic type
- **Date format** - Default date format for date basic type
- **Fill with** - Default value to fill a field of the given basic type up to the specified length
- **Justification** - Default justification (left/right) for the given basic type

2.1.7. Input Adapters

An input adapter defines a communication link through which an external system may call an integration service. The input adapter definition specifies how the called service is identified in the incoming message. For each specific channel, a service may be identified under different name (alias).

- **Code** - Code that identifies the specific input channel.
- **System** - The system that communicates via this channel.
- **Generic converter class** - Name of a generic converter that transforms external data format (non-XML) from/into internal XML. This is the mechanism that allows transparent processing of non-XML data while keeping uniform transformation model.
- **Expression for service detection** - XPath 2.0 expression for service name resolution. The expression is executed on incoming message to figure out what service should process the message. This mechanism allows for arbitrary message formats without forcing all clients/channels to identify target service the same way. The service name resolution also takes service aliases into account.
- **Default element namespace** - It is possible to define the default namespace for incoming messages.

2.1.8. Output Adapters

Output adapters identify channels used to send messages to specific target systems. Output adapters are assigned to integration services (BE services) that represent calls to specific services on specific systems.

- **Code** - Output channel code recognized by the runtime framework. For each channel there is a corresponding integration adapter implementation identified by this code.
- **System** - The system that is accessible via this adapter.

2.2. Validation

Model validation capability in Metada Integration Modeler helps to manage the integration model quality and consistence. Validation checks that all data that are put into a model are consistent, reasonable, and complete before the code generation and deployment may proceed. Two types of validation are provided. **Real-time** validation checks user input during entering the model data and checks validity of small fragments of models. **On-demand** validation is executed after larger pieces of models are completed and checks structural consistence of models and their entities.

2.2.1. Real-Time

Real-time validation is executed during user entry of model information. Only local consistence of local data may be checked, such as:

- **Mandatory information** - checks if important data set as "mandatory" are actually present and have not been missed out, e.g.: a service should be given a name. In the GUI all mandatory fields are marked with "***". If user leaves the mandatory field blank, the "save" or "create" button is inactive for the actual page until the mandatory field is filled with data.
- **Data-type checks** - checks the data type of the entered data and give an error message if the input data does not match with the expected data type, e.g: in an input box accepting numeric data (int), if the letter 'O' was entered instead of the number zero, a warning window would appear in browser when user tries to save changes on the actual page. Until the wrong input is corrected, it is impossible to save any changes on this page. Special data type validations are done in the tree-grid widgets, where it is not possible to enter unexpected characters, e.g.: if user tries to type letter in column accepting only numeric data (int), keyboard entries are not accepted (no data are typed).
- **Maximum length checks** - check if the number of characters entered into a field does not exceed the expected maximum field length, e.g.: the name of each service can have at most 100 characters, if user types more than 100 characters, a warning window would appear when user tries to save changes on the page.

2.2.2. On-Demand

On-demand validation checks structural consistence of an integration model. This type of validation is usually executed after completion of compound parts of a model to check if model entities fulfill pre-defined constraints. The Metada Integration Modeler comes with a set of on-demand validations that may be extended based on the model policies that a particular customer would like to enforce. Such policies may include naming conventions, limits on service relations, etc.

Every user can run data validation directly in Metada Integration Modeler GUI. Results of the model validation are displayed in new browser window and contain the following information:

- Total number of validation errors detected
- Tables (one table for each model concept) listing all validation errors detected
 - **Error Source** - shows the location of the validation error with direct link to model page where the problem should be corrected
 - **Error Message** - text describing the validation error, helping user to better understand the problem cause and solution

Validation of a large integration model may take several seconds.

Pre-packaged on-demand model validations are the following:

Concept	Error Message	Rule Description
System	Duplicate System Name.	If 2 or more Systems with the same name exist
Channels	Duplicate Input Channel Code	If 2 or more Input Channels with the same code exists
	Duplicate Output Channel Code	If more than 1 Output Channel with the same code exists
Type		

Metada Integration Modeler 2.0

	Duplicate Type name under one tag	If 2 or more Types with the same name exist under one tag
	Local Type property under the complex Type should have dive defined	If complex Type has Local Type property with no Dive defined
	Basic type missing in local parameter.	If Local Type property has no Basic type defined
	Reference to type missing	If Service interface contains "ref" but no Type is referenced
	Invalid sample value for data-type "boolean"	If Sample value of Local Type property does not match the selected Basic type "boolean"
	Invalid sample value for data-type "decimal"	If Sample value of Local Type property does not match the selected Basic type "decimal"
	Invalid sample value for data-type "integer"	If Sample value of Local Type property does not match the selected Basic type "integer"
	Only one root element allowed	If there is more than 1 Type property with root name or dive
	Complex type should have some properties.	If complex Type has no Type property defined
Service (active)	Duplicate service name	If 2 or more Services with the same name exist
	Active MW service has to call some BE service.	If active MW Service does not call any BE Service
	MW service cannot call other MW service	If MW service calls other MW service
	BE service cannot call other BE service	If BE service calls other BE service
	Service cannot call itself	If Service calls itself
	Reference to type missing	If Service interface contains "ref" but no Type is referenced
	Please add service description for functional documentation.	Missing service description that is important for generation of functional documentation
Translator	Invalid mappings - bad source service paths.	If source path(s) doesn't match with the existing paths of source Service in request or reply mapping
	Invalid mappings - bad target service paths.	If target path(s) doesn't match with the existing paths of target Service in request or reply mapping
Value Mapping Table	Duplicate VMT name.	If more than 1 VMT with the same name exists
	Duplicate VMT value. The InValue has to be unique for one VMT table	If VMT has 2 or more same data InValue field
	Empty VMT.	If VMT with no Input and Output values exist
Routing Conditions	Duplicate Routing Condition Name	If 2 or more Routing Conditions with the same name exist
	Routing Condition cannot be its own subcondition	If Routing Condition is subcondition of itself
Mapping Function	Duplicate Mapping Function name	If more than 1 Mapping Function with the same name exist

2.3. Generation

Metada Integration Modeler is a model-driven software development environment focused on platform-independent modeling of integration services. The models created in the modeler have no specific settings for any integration platform, nevertheless the specifics may be added via code generation for the specific platform. Metada Integration Modeler 2.0 comes with a code generator that produces a runtime "library", including an EJB that may be placed into any Java-based integration platform (such as BEA AquaLogic Service Bus or IBM WebSphere ESB).

When model is valid (i.e. there are no validation errors), it is possible to proceed with code/configuration generation. For each integration model, Metada Integration Modeler contains a generation page which offers a button that takes care of full generation of the integration library (EAR file).

The generated EAR file contains the following:

- IntegrationServicesBean EJB
- Implementation of mapping functions
- Generic convertors (for conversion of non-XML formats)
- xqr.jar with routing and message translation XQueries
- cfg.jar with configurations (e.g. convertors and service detection configurations)
- Autotester

Further, it is possible to generate full html documentation of the integration model.

2.4. Autotesting

Automatic testing uses simple autotests (XML documents) that are used to test individual integration services. For one service there should be several autotests to test specific service usage scenarios. Each test has to test if the service properly decides which other services to call (based on the provided content) and if it properly constructs messages to the called services and in the end that the reply to the client is also properly constructed. The testing sequence is as follows:

- **Request message from client** - Specific message sample that represents data sent from a client service to the service being tested.
- **Request message to a called service** - Message that is expected to be produced by the service being tested based on the client request.
- **Reply message from the called service** - Sample message that the called service would respond with to the above request.
- [repeat above two until all services are called] - One service may call several target services. For each call there would be a request and response message in the autotest.
- **Reply message to client** - Message that is expected to be produced by the service being tested based on all the responses from the called services.

Autotests are automatically executed via autotester application during solution generation and deployment. Autotester may also be executed separately when autotests are maintained or new autotests are being created.

Sample autotest

The following XML shows a sample autotest that tests a service that provides list of financial transactions (transaction history) for a given account. The client is Finantix and the backend is Symbols. Several things may be noted in the test:

- Both systems use completely different XML structures
- Collection of parameters is mapped to a non-collection
- Dates are mapped from one format to another
- Namespaces are supported
- Finantix heavily refers to implementation details (i.e. Java types)

```
<?xml version="1.0" encoding="utf-8"?>
<MLOAutoTests>
  <MLOAutoTest>
    <BEMWRequest client="FINANTIX"><![CDATA[
      <?xml version="1.0" encoding="iso-8859-2"?>
      <transaction xmlns:fx="http://www.finantix.com"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        fxcVersion="" fxsVersion="4.0.0" xmlns="http://www.finantix.com" id="_1">
        <header id="_2">
          <timeStamp>1194959398468</timeStamp>

<businessServiceCode>ua.ebu.fxc.services.accounttrxhistory_jms.VAccountTrxHistoryServiceHost#
  $getAccountInfo$ua$ebu$fxc$subject$accounttrxhistory
$$AccountTrxHistoryOutputWrapper$ua$ebu$fxc$subject
  $accounttrxhistory$$AccountTrxHistoryInputWrapper#input</businessServiceCode>
        </header>
        <parameters id="_3">
          <parameter id="_4">
            <name>input</name>
            <value id="_5">
              <accountNumber>accountNumber</accountNumber>
              <startDate>2007-08-05</startDate>
              <endDate>2007-11-21</endDate>
            </value>
          </parameter>
        </parameters>
      </transaction>
    </BEMWRequest>
  </MLOAutoTest>
</MLOAutoTests>
```

```

        <pageNumber>2</pageNumber>
        <rowsInPage>20</rowsInPage>
    </value>

<type>ua.ebu.fxc.subject.accounttrxhistory.ker.SAccountTrxHistoryInputWrapperDef</type>
    </parameter>
</parameters>
    <body id="_6"/>
</transaction>
]]>
</BEMWRequest>
<MWBERequest sync="Y" confirm="N" backend="SymbolsBS"><![CDATA[
    <?xml version="1.0" encoding="iso-8859-2"?>
    <SYM_RB_TRAN_HIST_QRY_IN>
        <HEADER_IN>
            <USER_NAME>CB1_API</USER_NAME>
            <APPLICATION>MW</APPLICATION>
            <WORKSTATION>unknown</WORKSTATION>
            <SCREEN_NO>unknown</SCREEN_NO>
        </HEADER_IN>
        <PAGE_IN>
            <ROWS_PER_PAGE>20</ROWS_PER_PAGE>
            <PAGE>2</PAGE>
        </PAGE_IN>
        <ACCT_NO>ACCOUNTNUMBER</ACCT_NO>
        <FROM_DATE>05.08.2007 00:00:00</FROM_DATE>
        <TO_DATE>21.11.2007 00:00:00</TO_DATE>
    </SYM_RB_TRAN_HIST_QRY_IN>
]]>
</MWBERequest>
<MWBEResponse><![CDATA[
    <?xml version="1.0" encoding="iso-8859-2"?>
    <SYM_RB_TRAN_HIST_QRY_OUT xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <HEADER_OUT>
            <CORR_ID>String</CORR_ID>
            <USER_NAME>String</USER_NAME>
            <APPLICATION>String</APPLICATION>
            <MSG_ID>String</MSG_ID>
            <RESULT_CODE>1</RESULT_CODE>
            <STATUS_CODE>1</STATUS_CODE>
        </HEADER_OUT>
        <PAGE_OUT>
            <PAGE>2</PAGE>
            <TOTAL_ROWS>20</TOTAL_ROWS>
        </PAGE_OUT>
        <TRAN_HIST_DTLS>
            <RB_TRAN_HIST_QRY_T>
                <TIME_STAMP>13.11.2007 00:00:00</TIME_STAMP>
                <EFFECT_DATE>21.11.2007 00:00:00</EFFECT_DATE>
                <CR_DR_MAINT_IND>D</CR_DR_MAINT_IND>
                <TRAN_DESC>String1</TRAN_DESC>
                <TRAN_AMT>100.0</TRAN_AMT>
                <TRAN_CCY>EUR</TRAN_CCY>
            </RB_TRAN_HIST_QRY_T>
            <RB_TRAN_HIST_QRY_T>
                <TIME_STAMP>14.11.2007 00:00:00</TIME_STAMP>
                <EFFECT_DATE>22.11.2007 00:00:00</EFFECT_DATE>
                <CR_DR_MAINT_IND>C</CR_DR_MAINT_IND>
                <TRAN_DESC>String2</TRAN_DESC>
                <TRAN_AMT>100.0</TRAN_AMT>
                <TRAN_CCY>EUR</TRAN_CCY>
            </RB_TRAN_HIST_QRY_T>
        </TRAN_HIST_DTLS>
    </SYM_RB_TRAN_HIST_QRY_OUT>
]]>
</MWBEResponse>

```

```

<BEMWResponse><![CDATA[
  <?xml version="1.0" encoding="iso-8859-2"?>
  <mw:transaction xmlns:mw="http://www.finantix.com" fxcVersion="4.0.0">
    <mw:header>
      <mw:timeStamp>0</mw:timeStamp>
    </mw:header>
    <mw:body>
      <mw:ua.ebu.fxc.subject.accounttrxhistory-SAccountTrxHistoryOutputWrapper>
        <mw:listTransactions>

<mw:Rtti>com.finantix.stdlib.lang::List[ua.ebu.fxc.subject.accounttrxhistory::SAccountTrxHistoryList
mw:Rtti>
      <mw:Item>
        <mw:transactionDate>2007-11-13</mw:transactionDate>
        <mw:valueDate>2007-11-21</mw:valueDate>
        <mw:details>String1</mw:details>
        <mw:credit>
          <mw:amount>0</mw:amount>
          <mw:currency>
            <mw:externalLiteral>EUR</mw:externalLiteral>
          </mw:currency>
        </mw:credit>
        <mw:debit>
          <mw:amount>100.0</mw:amount>
          <mw:currency>
            <mw:externalLiteral>EUR</mw:externalLiteral>
          </mw:currency>
        </mw:debit>
      </mw:Item>
      <mw:Item>
        <mw:transactionDate>2007-11-14</mw:transactionDate>
        <mw:valueDate>2007-11-22</mw:valueDate>
        <mw:details>String2</mw:details>
        <mw:credit>
          <mw:amount>100.0</mw:amount>
          <mw:currency>
            <mw:externalLiteral>EUR</mw:externalLiteral>
          </mw:currency>
        </mw:credit>
        <mw:debit>
          <mw:amount>0</mw:amount>
          <mw:currency>
            <mw:externalLiteral>EUR</mw:externalLiteral>
          </mw:currency>
        </mw:debit>
      </mw:Item>
    </mw:listTransactions>
    <mw:totalNumberRows>20</mw:totalNumberRows>
    <mw:pageNumber>2</mw:pageNumber>
  </mw:ua.ebu.fxc.subject.accounttrxhistory-SAccountTrxHistoryOutputWrapper>
</mw:body>
</mw:transaction>
]]>
</BEMWResponse>
</MLOAutoTest>
</MLOAutoTests>

```

2.5. Versioning & Synchronization

When an integration model is completed and migrated into production (i.e. starts to be used to integrate systems), a next round of modeling may start for the next release. Metada Integration Modeler allows management of several concurrent versions of the same integration model. One version of model may be in production, another in testing, and yet another in development. Usually it is sufficient to have two concurrent versions:

- **Version in production** - A version of an integration model that is the one used in production. Minor fixes are being done to this version in cases that errors are encountered in production environment.

- **Version in development** - A future version of the model that will one day become the version in production - one that is being developed and tested.

Compare & Synchronize Functionality

When several concurrent versions of a model are managed it is necessary to synchronize changes in one version to another (e.g. to propagate the fixes done in production into the version in development). Metada Integration Modeler provides compare & synchronize functionality that enables arbitrary models to be compared between each other, and enables selected changes to be propagated from one model to another. Any two models may be compared and synchronized.

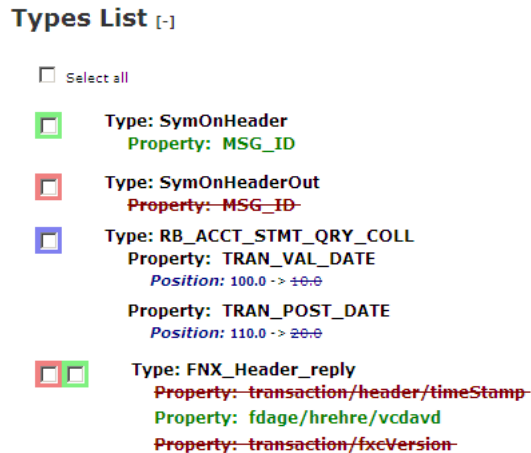


Figure: Sample compare result for types

The above figure shows a small sample of a comparison of two sample models (different versions) that have some differences between types. There are the following differences shown:

- Type SymOnHeader has a new property MSG_ID
- Type SymOnHeaderOut has MSG_ID property removed
- Two properties in type RB_ACCT_STMT_QRY_COLL changed positions
- Two properties have been removed from FNX_Header_reply and one was added.

The square checks are used to select what changes should be synchronized to the target model. Deletions are marked red, creations green, and changes blue.

3. Maintenance Procedures

This section specifies standard operating procedures for installation, setup, and maintenance of Metada Integration Modeler. This set of procedures is to be performed by responsible system administrators.

3.1. Metarepository Installation Procedure

This procedure defines steps that need to be taken to install Metada Metarepository 2.1.

Context

Metada delivers various modeling tools and metamodels, such as Integration Modeler or Frontend Modeler. These modelers share Metada Metarepository as its runtime infrastructure. Therefore, in order to use these additional tools, it is necessary to install the Metarepository first.

Procedure steps

1. Prepare environment - hardware

Prepare server hardware with at least the following specification:

- 2.8 GHz processor
- 2 GB memory
- 60 GB storage
- network card

2. Prepare environment - software

Pre-install and configure the following software:

- Windows Server 2000 or higher

- Java Development Kit (JDK) 6 or higher (<http://java.sun.com/>)
- Subversion Version Control System (<http://subversion.tigris.org/>) - Subversion is used for storage and versioning of models. Note that this software may be located on a different server and used remotely.
- Jakarta Tomcat 5.5 server or higher (<http://tomcat.apache.org/>) - Jakarta Tomcat or similar servlet container is necessary to run the Metarepository web application.
- Apache HTTP Server 2.0 or higher (<http://httpd.apache.org/>) - Apache HTTP Server is useful for secured connections (https) configuration and WEBDAV access to the Subversion VCS.

3. Download and placement

Two web application archives (wars) are provided:

- mambomde.war (<http://www.metarepository.com/downloads/2.1/mambomde.war>)
- openid.war (<http://www.metarepository.com/downloads/2.1/openid.war>)

Download the two war files from the above locations and place them into your servlet container (e.g. the webapps directory in Tomcat).

4. Configuration

There is one configuration file for the web application: "config.xml". The configuration file must be placed in the application root directory. The root directory may be specified by JVM system property "mambomde.rootdir" (default value is "c:/metada/mambomde/").

Contents of the config.xml file follow:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--
  Offline mode is default. Online mode is turned on by adding offline="false".
  If an installation is in offline mode any username is always authenticated
  (password is not checked)! Reason for this is that if someone is able to
  setup new or change mode of existing installation to offline mode. She
  already has full control over this installation is it's security is already
  compromised.
-->
<MamboMDEConfig offline="false">
  <!--
    Specifies which database XDA uses to store data. Available types are:
    "msaccess" and "mysql".
  -->
  <XDA dbtype="msaccess"/>
  <!--
    LDAP authentication settings.
  <AuthService>
    <LDAPSettings url="ldap://example.com:636/">
      <Principal>cn=%ldapuser%,ou=Users,dc=usr,dc=example,dc=com</Principal>
    </LDAPSettings>
  </AuthService>
  -->
  <!--
    Proxy selector configuration. Other way of configuring http proxy is
    to pass to JVM these command line parameters: http.proxyHost, http.proxyPort
    and http.nonProxyHosts (this Proxy selector should not be used then)
  <HttpProxy host="localhost" port="8080" nonproxyhosts="*.example.com;localhost;127.0.0.1"/>
  -->
  <!--
    Configures logging subsystem (log4j). Format of this element value is
    the same as of any log4j cofiguration properties file.
  -->
  <Logging><![CDATA[
log4j.rootLogger=WARN, console

log4j.logger.com.metada=DEBUG, mambomdefile
log4j.additivity.com.metada=false

log4j.logger.com.mambomde=DEBUG, mambomdefile
log4j.additivity.com.mambomde=false

log4j.logger.org.springframework=WARN, mambomdefile
log4j.additivity.org.springframework=false
```

```
log4j.logger.uk.ltd.getahead.dwr=WARN, mambomdefile
log4j.additivity.uk.ltd.getahead.dwr=false

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

log4j.appender.mambomdefile=biz.minaret.log4j.DatedFileAppender
log4j.appender.mambomdefile.layout=org.apache.log4j.PatternLayout
log4j.appender.mambomdefile.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
log4j.appender.mambomdefile.Directory=c:/metada/mambomde/logs
log4j.appender.mambomdefile.Prefix=mambomde.
log4j.appender.mambomdefile.Suffix=.log
log4j.appender.mambomdefile.Append=true
]]>
</Logging>

</MamboMDEConfig>
```

5. Run

Run the servlet container (e.g. startup a Tomcat server) and navigate to [http://\[server\]/mambomde/](http://[server]/mambomde/). Metarepository start page appears with no repositories loaded.

6. Load metarepository models

In the "Load Existing Repository" box enter "<https://svn.metada.com/m21/rep>" as the repository url and press the "load" button (no login needed). The loaded repository contains metamodels needed to create other repositories and models.

After loading the Metarepository models the installation procedure is completed. Please, see further procedures for creating new repository or loading modeler specific repositories (such as Metada Integration Modeler repository).

In case that an update to m21 repository is provided, it is possible to make an SVN update. In such a case navigate to "M2.1 Repository" in the top menu, then go to "Model Control" in the left menu, select models to update in the list of models, and finally press the "SVN Update" button.

3.2. Modeler Installation Procedure

This procedure specifies how to load Metada Integration Modeler 2.0 into already installed Metada Metarepository.

Context

Metada Integration Modeler uses Metada Metarepository as its runtime environment, where it runs as a repository that contains an integration metamodel (im_mta_v2_0) and generation network (im_genet_v2_0).

Procedure steps

1. In running Metarepository instance navigate to [http://\[server\]/mambomde/](http://[server]/mambomde/) page.
2. In the "Load Existing Repository" box enter "<https://svn.metada.com/im/rep>" as the repository url and press the "load" button. Login and password is required (provided as part of the license or the support and maintenance agreement).
3. When loaded, the repository appears in the repositories box as "im_rep". Navigate to this repository.
4. Navigate to the "IM Repository" category in the top menu and then to "Model Control" in the left submenu.
5. Select im_mta_v2_0 and im_genet_v2_0 models, and press the "SVN Checkout" button.

After successful checkout of the metamodel and generation network, the Metada Integration Modeler is ready to be used. At this time it is possible to load existing repository with integration models or create a new one (please, see further procedures).

The metamodel and genet may be updated as needed from the Model Control by selecting the models and pressing "SVN Update".

3.3. Create Repository Procedure

This procedure specifies how to create new repository where integration models will be stored and versioned.

Context

Metada Integration Modeler supports multiple repositories with integration models. Each repository is versioned in Subversion version control system.

Procedure steps

1. Create new Subversion repository in your Subversion instance (e.g. "esb").
2. In the created repository create new "rep" directory.

3. Check, that this repository is accessible via browser by navigating to url of the Subversion instance (e.g. [https://\[server\]/svn/esb/rep](https://[server]/svn/esb/rep)).
4. In running Metarepository instance navigate to [http://\[server\]/mambomde/](http://[server]/mambomde/) page. Make sure that "im_rep" is loaded (see Modeler Installation Procedure).
5. In the "Load Existing Repository" box enter the repository url (e.g. [https://\[server\]/svn/esb/rep](https://[server]/svn/esb/rep)) and press the "load" button.
6. When loaded, the repository appears in the repositories box as "esb_rep". Navigate to this repository.
7. Go to "Model Control" on the right side of the menu bar.
8. Go to "User Groups" and create user groups with users that will need access to the repository.
9. Go to "Systems" and create system "rep" and set R to "all" and RW to an administrators group.

After this procedure your new repository will be ready for models to be created (see further procedures).

3.4. Create Model Procedure

This procedure specifies how to create new integration model in existing integration repository.

Context

Metada Integration Modeler supports multiple repositories with integration models. Each repository is versioned in Subversion version control system.

Procedure steps

1. Create new directory in your Subversion repository (e.g. "test").
2. Check, that this repository is accessible via browser by navigating to url of the Subversion instance (e.g. [https://\[server\]/svn/esb/test](https://[server]/svn/esb/test)).
3. In running Metarepository instance navigate to [http://\[server\]/mambomde/](http://[server]/mambomde/) page. Go to your integration repository (e.g. "esb_rep").
4. Go to "Model Control" on the right side of the menu bar.
5. In the "Create New Model" box type model name (e.g. "test") and metamodel name "im_mta_v2_0", and press create.
6. The new "test" model appears in the list of created models.
7. To synchronize the new model with Subversion, select the model and press the "SVN Checkout" button.
8. Go to "System" in the left menu and edit the new "test" model - change authorization to groups that should have access to the model.
9. Go to "Menu" in the left menu and create "root" menu category. Then create "test" menu category with model URL "test". In the root menu category place the test category as a subcategory. Write descriptions of the categories.

After this procedure your new integration model is ready for modeling and is accessible from the repository menu.

3.5. LDAP Authentication Setup Procedure

In case that there is a company active directory or LDAP server, it is possible to use it for user authentication to the Metada Integration Modeler. This procedure specifies how to configure the product to access the identity server.

Procedure steps

1. Obtain LDAP url (e.g. <ldap://example.com:636/>)
2. Obtain LDAP principals for the users (e.g. `cn=%ldapuser%,cn=Users,dc=division,dc=example,dc=com`)
3. Open `C:\metada\mambomde\config.xml` file
4. Uncomment the `<authservice>` element and configure LDAP, for example:

```
<LDAPSettings url="ldap://example.com:636/">
  <Principal>cn=%ldapuser%,cn=Users,dc=division,dc=example,dc=com</Principal>
  <Principal>cn=%ldapuser%,cn=ExternalUsers,dc=division,dc=example,dc=com</Principal>
</LDAPSettings>
```

5. Save the configuration file and restart the "Metada Integration Modeler" Windows service

3.6. User Group Setup Procedure

This procedure defines how to create a new user group and authorize it to specific models.

Context

It is good practice not to assign users directly to models and authorize them only via user groups. Access to individual models is then provided to users through user groups they are assigned into. A user group is a collection of users with the same needs and privileges. If a user group is assigned to a model, all users in this group get the same level of access. There are the following access levels for each model:

- **Read** - User may only read model contents.

- **Read-Write** - User may read and write model contents.

Procedure steps

1. Navigate to the Model Control and then to User Groups menu
2. Click on "Create" button under the list of existing user groups
3. Define "Name (code)" for the new user group and add at least one existing User ID to "Members" field.
4. Press "Create" button create the new user group
5. Add user group to specific models either with Read or Read-Write privilege

3.7. User Authorization Setup Procedure

This procedure specifies how to give specific users access (authorization) to specific integration models.

Context

Authentication is provided by OpenID or LDAP, and therefore any user may log into the application, nevertheless to see any models, users have to be authorized to see them. Users are not assigned to models directly, but are assigned to user groups. User groups are then linked to models either to have read-only or read-write access.

Procedure steps

1. In Model Control navigate to "User Groups" section
2. Add user to an existing user group or create a new user group (see User Group Setup Procedure)

3.8. Backup Procedure

This procedure specifies how to setup mechanism for regular automatic storing of models.

Context

Backups are a way to protect the investment in the modeling efforts. Regular backup to different secured locations ensures that models will be recoverable even in case of hardware failures, software bugs, viruses, human actions, or natural disasters. The most valuable assets created in Metada Integration Modeler are integration models. These models should be regularly checked-into version control (Subversion). What needs to be backed-up then are the version control data.

Procedure steps

1. Create operating system job that runs regularly (e.g. once a day at night)
2. Configure the job to check-in all changes in all models
3. Configure the job to pack and copy the whole Subversion repository to a different location (e.g. Amazon S3)

3.9. Disaster Recovery Procedure

This procedure specifies what to do in case of complete or partial hardware or software failure that causes loss of integration models or Metada Integration Modeler disfunction.

Procedure steps

1. In case of complete destruction - first execute the installation procedures
2. In case of Subversion repository destruction, load and recover the full repository from the latest backup
3. Reload the models from Subversion repository

Ver: mtda_doc, 2008-07-28+02:00