Xesta Documentation

Contents

Xesta

An adjusted Java Object / Xml Document Binding

Xesta is a java to xml binding tool. Xesta balances the approach between object and document side of the binding, trying to rely on tooling and practices aligned with each of the sides.

Context.

Document information is usually more complex, updated more frequently, including both data and structure and may be more focused on small details. Documents, Xml documents, are a great way of to handle detail and persistent, isolated (not directly backed by any system) data. They are also the backbone of the pulishing and digital humanities industry, including legal texts, books, product instructions, etc.

Xml Documents are usually explored with XPath indications. While Java comes by default with XPath 1.x the Xml community rely on newer versions. Xesta annotations are defined with XPath 3.x instead of positional indications. Future XPath versions may be adopted.

On the other side, Java and object oriented tools are tend to work with more structured data, less complex but with more volume. This part require more consistency and a more structured/less flexible approach. Java tooling frecuently rely on Xml schemas when dealing with Xml documents. It is also common for Java tools to look for similar structures on both sides and rely on Xml for plain object serialization.

Xesta was presented as Fento on XML Prague 2026 and afterwards renamed as Xesta.

Getting Started with Xesta.

Dowload and configure the Xesta jar and dependencies.

Download the jar and dependencies or declare the dependency on the maven pom file.

Options:

- Declare the dependency on the pom file.

  > net.vionta.xml > xesta > 1.0.0 >

- If the version is not available on maven central you can download the last version from http://github.com/vionta/xesta/releases and install it locally.

Annotate your first class.

Xesta defines the xml document locations using Java annotations (Bind), the Xml locaitons are defined with the "expression" attribute. Classes should implement the Serializable interface.

```
@Bind(expression = "//*:CstmrCdtTrfInitn" )public class CheckNoNS implements Serializable {
```

They should also have a no args constructor.

```
public CheckNoNS() {}
```

Then you should annotate the fields.

```
@Bind(expression = "*:PmtInf/*:DbtrAcct/*:Id/*:Othr/*:Id")String debtorAccount;
```

A complete Pojo example from a class used to retrieve the value attribute from a console appender from a log4j file.

```
import java.io.Serializable;import net.vionta.xml.xesta.bind.annotation.Bind;

@Bind(expression = "/*:configuration/appender")public class LogLevelSimple implements Serializable {

@Bind(expression = "param/@value")private String level;

public LogLevelSimple() {super();}

public String getLevel() {return level;}

public void setLevel(String level)

}
```

Retrieve the document

In order to retrieve the document you use a repository. The simplest repository type is a File repository where you pass the file location.

```
FilePathRepository repo = new FilePathRepository("./cases/iso20022/
GlobalPayment.xml");GlobalPayment globalPayment = new GlobalPayment();globalPayment =
repo.load(globalPayment);
```

You can find more examples from the XmlPrague 2026 conference at http://www.github.com/vionta/xesta-samples.

Repository types.

Xesta access to documents are wrapped with the repository abstraction. In order to retrieve or update a document you need to choose from one of the repository types, provide the configuration parameters, usually the path or paths.

Once the repository is configured (its path or paths are provided) you request the repository to retrieve or update the document for you.

Path Adjustment.

The repository paths (in the supported cases) can be adjusted based on the object properties. In order to configure the adjustment the property name (separated with dots between objects) must be defined inside the key { } markers.

File/Folder based repository.

The simplest possible repository type with a base path and a file path. The base path is not adjusted based on the the object properties, while the file path can be adjusted with the {} keys.

```
FilePathRepository repo = new FilePathRepository(CommonTest.getBasePath()+"cases/iso20022/
GlobalPayment.xml");GlobalPayment globalPayment = new GlobalPayment();globalPayment =
repo.load(globalPayment);
```

Classpath repository.

A repository that retrieves files from the application classpath. This repository does not allow to create new files, also some files may be available for reading but not accesible for writing. The file path is defined with simple file path conventions separating folders with the slash "/" operator.

```
ClassPathRepository repo = new ClassPathRepository().setPath("log4j.xml");LogLevelSimple log =
repo.load(new LogLevelSimple());log.setLevel("DEBUG");repo.persist(log);
```

Mind that, depending on the running environment, the document path may or may not be accesible using the file system.

REST based repository.

A repository that retrieves and stores the files from a REST service. The file path can also be adjusted via the object data using the {} keys on the path.

** Note: service authentication not yet developed. **

Zip file based repository.

The zip file repository is similar to the file repository but acts on files that are placed inside zip files. Both the zipfile and the content file can be ajusted via the object properties. Zip file names are not required to end in ".zip". Zip repository can be used for odt, ods, wars, etc.

```
String zipFilePattern = CommonTest.getBasePath()+"openDocument/text/fictional-
report.odt";ZipArchiveFileRepository repo =new ZipArchiveFileRepository(zipFilePattern,"content.xml");

Document1 doc1 = new Document1();doc1 = (Document1) repo.load(doc1);
```

Abstract repository.

An abstract implementation that supports extensibility in other cases. Still not available.

** Note: Functionality in development. **

Annotating Java Beans

The annotation process is intended to be as simple as possible. It is performed mostly with the Bind annotation, providing the XPath expressions and complementary configuration.

First Steps

Start by importing the Bind Class.

```
import net.vionta.xml.xesta.bind.annotation.Bind;
```

The next step is to provide an XPath expression for the main class.

```
@Bind(expression="*:appender")public class Appender implements Serializable {
```

Each annotated class **MUST**:

- Implement the Serializable interface.
- Provide a no argument constructor

```
public Appender() {}
```

Annotating Simple Properties

Properties should be also marked with the Bind annotation.

```
@Bind(expression = "@*:name")private String name;

@Bind(expression = "*:param/@value")private String level;
```

You can mark both elements, element text content or attributes. The data type is inferred from the Java type. By now the following Java types are supported:

- Integer
- Short
- Float
- Double
- Long
- Byte

For each property mapping regular setters and getters should be provided :

```
public String getName() {return name;}

public void setName(String name)
```

Providing Date Formats for dates.

Mapping netsted objects

Mapping nested objecs is quite similar to map regular properties. You just point the mapping expression to the Element that acts as the base context of the element properties.

In the following example we define a class (Location) with a nested "Group" Class. The Group class is mapped to an Xml "Band" element that has no namespace. Keep in mind that namespace declarations are not optional in XPath 3 and later.

```
@Bind(expression = "/concert-list/shows/Concert")public class Location implements Serializable {

@Bind(expression = "./event/Band")private Group group;

@Bind(expression = "./name")private String name;
```

You must add the usual setters and getters and a no argument constructor. Also the element should implement the Serialzable interface. If the Group class can be simply completed by adding the property mappings.

```
@Bindpublic class Group implements Serializable {

@Bind(expression="./name")private String name;

@Bind(expression = "./event")private Event event;
```

Upward and downward element exploration

The class element mapping acts as a base level for the property mappings but properties are not limited to the element direct child elements. XPath navigation can be defined downward, upward or using axes and filters. This opens lots of possibilities on the read side but also faces some limitations when it comes to serialization, as the algorithm may not have enought information to determine where to put the information.

Downward examples

Let's start with an open office spreadsheet. We can map a Sheet class to the first spread sheet element of the file:

@Bind(expression = "//office:body/office:spreadsheet",namespaceAlias = {"office"} ,namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:office:1.0"})public class Sheet implements Serializable {

@Bind(namespaceAlias = {"table"} , namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:table:1.0"})private List

Managing namespaces

The first thing to underline is that, while in XPath 1.0 namespace declarations were optional, and XPath 1.0 is quite common in Java and web applications, they are not optional in modern versions of XPath. In order to avoid namespace declaration you can use the * operator to mark *:your-element-name now. A query like //the-element-name won't give you results with a document like:

> First non root element > Second non root element > > >

In contrast a query like //:the-element-name will give you results on XPath 3.x and following. Also, if no namespace would be used on the document, a no namespace query on XPath 3.x will give you results.

Working with namespaces on Xesta

- Documents without namespaces. In this case you do not need to use wilcards on the mappings. Simple expressions, similar to the ones in XPath 1.0, would work:

  @Bind(expression = "/project")public class MavenProject implements Serializable {

  public MavenProject() {super();}@Bind(expression = "modelVersion")private String modelVersion;@Bind(expression = "*:artifactId")private String artifactId;...

  @Bind(expression = "./reporting/plugins", classNames = "net.vionta.xml.xesta.test.beansamples.pom.both.Plugin")private ArrayList plugins = new ArrayList();...

  In this case we are using a maven document without namespaces like:

  > 4.0.0 > com.some.library.group > somename > pom > Some Project Name > ...

- Using wildcards is another common way to query documents with namespace declarations when we don't need anything specific about any of them. In the following example we will use it with a ISO2022 file.

  @Bind(expression = "//*:CstmrCdtTrfInitn")public class CheckNoNS implements Serializable {

  public CheckNoNS() {}

  @Bind(expression = "*:PmtInf/*:PmtMtd")private String paymentMethod;

  @Bind(expression = "*:PmtInf/*:Dbtr/*:Nm")private String debtorName;

  @Bind(expression = "*:PmtInf/*:DbtrAcct/*:Id/*:Othr/*:Id")private String debtorAccount;...

  The sample file would start like this, and could be reviewed at https://developer.gs.com/docs/services/transaction-banking/iso20022samples/.

  > > > > aeoueaoueaouoea > 2024-05-10T16:10:02.017+00:00 > 1 > 510.24 > > > > > > Client-Id > ...

- Using Q{} syntax (not recommended). One only namespace for line and element could be especified using Q{} syntax. Although simple, this approach is not recommended.

  @Bind(expression = "//QCstmrCdtTrfInitn")public class Check implements Serializable {

  public Check() {}

  @Bind(expression = "QPmtInf/*:PmtMtd")private String paymentMethod;

  @Bind(expression = "QPmtInf/*:Dbtr/*:Nm")private String debtorName;...

- Using namespace alias and uris. This is the most complete and recommended way if namespaces are needed and wildcards are not enough.

  @Bind(expression = "//iso:CstmrCdtTrfInitn" , namespaceAlias = {"iso"} , namespaceUris = {"urn:iso:std:iso:20022:tech:xsd:pain.001.001.03"})public class CheckNS implements Serializable {

  public CheckNS() {}

  @Bind(expression = "iso:PmtInf/iso:PmtMtd" , namespaceAlias = {"iso"} , namespaceUris = {"urn:iso:std:iso:20022:tech:xsd:pain.001.001.03"})private String paymentMethod;

  @Bind(expression = "*:PmtInf/*:Dbtr/*:Nm")private String debtorName;

  @Bind(expression = "*:PmtInf/*:DbtrAcct/*:Id/*:Othr/*:Id")private String debtorAccount;...

More than one namespace could be added, like you could see in the following example (styles list).

  @Bind(expression="office:document-styles", namespaceAlias = {"office"} ,namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:office:1.0"})public class Document implements Serializable {

```
  @Bind(expression="office:styles", namespaceAlias =
{"office", "style"},
```

  namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:office:1.0","urn:oasis:names:tc:opendocument:xmlns:style:1.0"})private List styles = new ArrayList();...

Another example could be checked at the following property

  @Bind(expression="./style:text-properties/@fo:font-size", namespaceAlias = {"style","fo"}, namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:style:1.0", "urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"})private String fontSize;...

Like in any XPath query, wildcards and namespaces can be mixed

  @Bind(expression="./*:text-properties/@fo:font-size", namespaceAlias = {"style","fo"}, namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:style:1.0", "urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"})private String fontSize;...

Mapping collections of objects

Xml usually hold collections of elements, repeatitions of elements that we would naturally bind to java.util.List subclasses.

** Single and multiple collections** is an important concept for Xesta. Programming approaches tend to organize collections of similar elements in its own separated bags. This is not mandatory in Java, a subinstance of List can hold instances of different objects. This is not the usual practice. Generics and other bind or serialization tools tend to the "one bag for each object type" practice. In Xml, the element name and namespace is a primary source of information and as a consequence mixed types of elements and the ordering may be really significative. We call single collection mapping to the collections where we have only one type of child on the Java side, and multiple collections where we

may map more than one object type on a java side for the same collection. Mind that using XPath we may wrap child elements from more than one element in a single Java collection and we could also take single types of java objects from an element with different child nodes. This is one of the main appealings from this technique, although it comes with a downside when we switch to serialzing.

Single Collections

Mapping a single collection is quite simple, and support several combinations:

- Mapping the element name on the child side. This is probably one of the most intuitive ways. In this case we explore a Log Level document, the appender collection points the context of the element collection to the base of the document content ("../*"). There the class type guess can be taken from the List generics declaration.

  @Bind(expression = "/*:configuration")public class LogLevel implements Serializable {

  @Bind(expression = "../*")private ArrayList appender = new ArrayList();

  In the Child Element class, we would be considering all the "appender" elements with any schema.

  @Bind(expression="*:appender")public class Appender implements Serializable {

  @Bind(expression = "@*:name",key=true)private String name;

  @Bind(expression = "*:param/@value")private String level;

  In this case, the context is pointed in the collection part and the element type on the child side.

  - Mapping the child element on the collection side. This is also a very simple approach. In the following example we would be defining the full route of a maven plugin element on the collection side. No separated context and element.

    @Bind(expression = "/project")public class MavenProject implements Serializable {

    @Bind(expression = "*:artifactId")private String artifactId;

    @Bind(expression = "./reporting/plugins/plugin")private ArrayList plugins = new ArrayList();

    On the child side we would not need to add anything at the class level, as the class type is especified using generics.

    public class Plugin implements Serializable {

    @Bind(expression="artifactId")private String id;

    - Mapping the element name on the child side. In this case we would have an annotation on the list element that states the context of the collection.

      @Bind(expression = "../*")private ArrayList appender = new ArrayList();

      On the child element side we would especify the element that should be binded to the pojo.

      @Bind(expression="*:appender")public class Appender implements Serializable {

      @Bind(expression = "@*:name",key=true)private String name;

      @Bind(expression = "*:param/@value")private String level;...

      The key atribute of the annotation states that the property value could be used as an identifier of the element. This is used in collections where we want to take into consideration moving elements, sorting, etc.

      - Providing the child class element. This is another possibility, reserved usually for the multielement collections, where we state the class names that should be inspected for element mappings.

@Bind(expression = "/*:configuration")public class LogLevel implements Serializable {

@Bind(expression = "../*", classNames = "net.vionta.xml.xesta.test.beansamples.log.Appender")private ArrayList appender = new ArrayList();...

Multiple Collections

Multiple collections are list of elements that may have more than one element child type. This is a less used practice on the Java enterprise but may make a lot of sense when working with Xml documents. Xml Element semantic and metadata capabilities enables some of the most complex and flexible use cases on data management, and the element name, schema, etc. may be used to carry significative information.

Multiple collections require that you add the child element class names to the list declaration

@Bind(expression="./:",classNames = {"net.vionta.xml.xesta.test.beansamples.list.salvora.neu1.Parameter.class","net.vionta.xml.xesta.test.beansamples.list.salvora.
ArrayList<? extends Serializable> parameters = new ArrayList();...

Where in the child objects the class mapping should state the Xml element to be mapped to each one.

@Bind(expression="/*:parameter")public class Parameter implements Serializable {

public Parameter() {}

@Bind(expression="@key")private String key;

@Bind(expression="@value")private String value;...

Collection sorting strategies

The object binding technique is actually limiting. The possibilities of what we could do or the way that we may interact with an Xml file is almost endless. Annotation strategies are small properties added to the mapping annotation in order to define specific behaviours.

Element identifying and sorting

- Bind By Key (BIND_COLLECTION_BY_KEY) In this case, the collection elements should have a mapped property that uniquely identifies the elements on the Xml side. This key is used to add, remove or sort the elements before serializing. This is the default case if a key attribute is defined.

  @Bind(expression = "./reporting/plugins",classNames = "net.vionta.xml.xesta.test.beansamples.pom.both.Plugin")private ArrayList plugins = new ArrayList();...

  In this case the child element has a key attribute.

  @Bind(expression = "plugin")public class Plugin implements Serializable {

  @Bind(expression="artifactId", key = true)private String id;...

- Bind Collection by positon (BIND_COLLECTION_BY_POSITION). In this case we assume that elements can't or should not be identified and possition should be used instead. Bind by position is assumed when no key attribute is defined on the child elements.

  @Bind(expression="table:table-row", namespaceAlias = {"table"},namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:table:1.0"}, *collectionBindStrategy = SerializingMode.BIND_COLLECTION_BY_POSITION*)public class Row implements Serializable {

@Bind(collectionBindStrategy =
SerializingMode.BIND_COLLECTION_BY_POSITION,namespaceAlias =
{"table"},namespaceUris = {"urn:oasis:names:tc:opendocument:xmlns:table:1.0"})private List
cells = new ArrayList();

- Append new elements as last (BIND_COLLECTION_APPEND_LAST). This strategy assumes that the bind is made by key, but new elements are appended at the end of the xml file.
- Insert new elements as first (BIND_COLLECTION_APPEND_FIRST). This strategy assumes that the bind is made by key, but new elements are insered at the beguinning of the xml file.
- Full collection reset (BIND_COLLECTION_FULL_RESET). This strategy is equivalent to traditional object binding. It erases all the data related to the collection and serializes the information taking the object information as an argument.

Bind Annotation Reference.

The Bind annotation reference has the fo

import net.vionta.xml.xesta.bind.annotation.Bind;

- expression The XPath mapping expression that points to the selected document nodes.
- key Indicates if the attribute identifies the node element. It is only considered when the collection strategy is equal to SerializingMode. BIND_COLLECTION_BY_KEY.
- classNames [String | Default : null | Mandatory : false] The class names of a multiple collection. An element list can containt more than one single element type. Xesta looks for the mappings in all of the classes to determine the collection elements. *This feature is under developmnet*
- serializingMode
- deserializingMode
- collectionBindStrategy
- collectionDeleteUnmatched
- namespaceAlias [String | Default : null | Mandatory : false] A list of the name space alias.
- namespaceUris [String | Default : null | Mandatory : false] A list of the name space uris.
- auto If true the element name or expression is calculated from the java property name.
- DeserializingMode
  - FAIL_ON_NOT_EXISTING : 1
  - WARN_ON_NOT_EXISTING : 2
  - AVOID_ON_NOT_EXISTING : 3
- SerializingMode
  - FAIL_ON_NOT_EXISTING : 1
  - CREATE_ON_NOT_EXISTING : 2
  - SKIP_ON_NOT_EXISTING : 3
  - BIND_COLLECTION_BY_KEY : 1
  - BIND_COLLECTION_BY_POSITION : 2
  - BIND_COLLECTION_APPEND_LAST : 3
  - BIND_COLLECTION_APPEND_FIRST : 4
  - BIND_COLLECTION_FULL_RESET : 5
  - COLLECTION_DONT_DELETE_UNMATCHED : 1
  - COLLECTION_DELETE_UNMATCHED : 2

Status

Xesta has been developed as an experiment and a proposal for XML Prague 2026. First scheduled presentation will take place at the Prague Economics University on June 6. At this stage, Xesta could be considered as an alpha version. Anyhow, remarkable number of tests has been performed an executed.

Changelog

- 2026-06-08 Fento is renamed as Xesta. Due to a possible coincidence with another component the Fento Xml Binding component has been renamed as Xesta.

Acknowledgements.

- Saxonica The XPath heavy lifting is performed by the mighty tool. www.saxonica.com

References.

- XML Prague presentation The tool, named Fento at the time, was presented at Prague Xml in 2026. https://www.xmlprague.cz/day3-2026/