DOI 10.1002/pmic.201100577 *Proteomics* 2012, *12*, 790–794

TECHNICAL BRIEF

jmzIdentML API: A Java interface to the mzIdentML standard for peptide and protein identification data

Florian Reisinger[1], Ritesh Krishna[2]*, Fawaz Ghali[2], Daniel Ríos[1], Henning Hermjakob[1], Juan Antonio Vizcaíno[1] and Andrew R. Jones[2]*

[1] EMBL-European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge, CB10 1SD, UK
[2] Institute of Integrative Biology, University of Liverpool, Liverpool, UK

We present a Java application programming interface (API), *jmzIdentML*, for the Human Proteome Organisation (HUPO) Proteomics Standards Initiative (PSI) mzIdentML standard for peptide and protein identification data. The API combines the power of Java Architecture of XML Binding (JAXB) and an XPath-based random-access indexer to allow a fast and efficient mapping of extensible markup language (XML) elements to Java objects. The internal references in the mzIdentML files are resolved in an on-demand manner, where the whole file is accessed as a random-access swap file, and only the relevant piece of XML is selected for mapping to its corresponding Java object. The API is highly efficient in its memory usage and can handle files of arbitrary sizes. The API follows the official release of the mzIdentML (version 1.1) specifications and is available in the public domain under a permissive licence at http://www.code.google.com/p/jmzidentml/.

Received: November 4, 2011
Revised: December 14, 2011
Accepted: December 15, 2011

Keywords:
Bioinformatics / Java API / mzIdentML / Proteomics standards initiative (PSI) / XML

Proteomics search engines use peptide mass fingerprint (MS1) or tandem mass spectrometry (MS/MS) data in conjunction with sequence databases for peptide and protein identification. The identification results and the relevant statistics can be presented in a variety of output formats. In an attempt to standardize a common output format across different search engines, the Human Proteome Organisation-Proteomics Standards Initiative (HUPO-PSI) has recently proposed the mzIdentML standard for reporting protein identification data (http://www.psidev.info/mzidentml). mzIdentML is an extensible markup language (XML) based exchange standard developed in close collaboration with instrument vendors and software developers from the global proteomics community. This paper presents a Java-based application programming interface (API), *jmzIdentML*, for the processing and creation of mzIdentML documents.

The mzIdentML format reports a peptide-spectrum match accompanied by a reference to a peptide sequence and an identifier for the corresponding spectrum present in an external file. The peptide-spectrum identification is supported by the statistical score, rank, confidence value (e.g. an *e*-value or *p*-value), and the collection of all protein sequences that contain the matched peptide. The mzIdentML file lists all the protein sequences from the primary sequence database that have been potentially identified, and the theoretically digested peptide sequences used for the peptide-spectrum matches. The peptide sequences are accompanied by their theoretical and experimental masses and the potential sites of modifications. The parameters and methods used for performing the search are specified in the relevant sections of the mzIdentML file to provide complete information about the search

Correspondence: Dr. Andrew R. Jones, Biosciences building, Institute of Integrative Biology, University of Liverpool, Liverpool, L69 7ZB, UK
E-mail: andrew.jones@liv.ac.uk
Fax: +44 151 795 4408

Abbreviations: API, Application Programming Interface; **(HUPO) PSI**, (Human Proteome Organisation) Proteomics Standards Initiative; **JAXB**, Java Architecture of XML Binding; **PRIDE**, PRoteomics IDEntifications (database); **XML**, Extensible Markup Language

*These authors contributed equally to the manuscript.

Colour Online: See the article online to view Figs. 2 and 3 in colour.

performed. A detailed description of the format can be found at http://www.psidev.info/mzidentml. The XML schema describing the structure of the mzIdentML provides the basis for the design of *jmzIdentML*. The API strictly follows the mzIdentML specifications (version 1.1) proposed by HUPO-PSI.

There are several commercial and open-source tools that currently support the mzIdentML standard. Most of the tools work as *converters* to facilitate the conversion of native file formats to mzIdentML format: for instance, the Mascot parser (version 2.3 supports mzIdentML version 1.0 and is currently being updated to version 1.1 - http://www.matrixscience.com/msparser.html), ProCon (conversion of SEQUEST files - http://www.medizinisches-proteom-center.de/ProCon), and libraries for the conversion of OMSSA and X!Tandem results [1]. OpenMS [2] and ProteoWizard [3] provide C++ libraries for reading and writing mzIdentML. The provision of the mzIdentML standard and file format converters will help experimentalists to compare or integrate results from different search engines, which is currently very challenging. It will also facilitate new software development by reducing the number of file formats to support. To our knowledge, *jmzIdentML* is the only API available written in Java that has been developed for mzIdentML. The API is available free of charge in the open-source domain.

jmzIdentML is developed in pure Java, thus making the API compatible across all platforms. The construction of the API is based on industry standard open-source development architectures such as Maven 2, Java Architecture of XML Binding (JAXB), Log4j, and JUnit. A ready-to-use version of the API is available for download at http://www.code.google.com/p/jmzidentml/.

An important feature of the mzIdentML format is the abundance of internal references in an XML file. The use of references reduces the redundant repetition of information in the file, while at the same time allowing similar elements to be grouped together in a structured manner. Generally, such reference resolving requires the entire file to be loaded into memory for a quick lookup. However, if the file is large, it can easily become a memory-intensive task and the processing becomes difficult on a standard computer. We have proposed an on-demand lookup mechanism to resolve internal references by using an innovative XPath-based indexer combined with a user-defined caching mechanism. The XML schema of the mzIdentML standard specifies a fixed XPath for each element in an mzIdentML file. Each time a file is opened for reading, a scanner runs through the whole file registering the start and end byte locations of all the elements in the file identified by their XPath. A memory-based data structure called *xxIndex* keeps a record of the element found in the file, their XPath, and their respective start and end byte locations in the file. An index of this kind allows us to use the whole file as a random-access file where each element can be accessed either by its name, or its XPath. The knowledge of the start and end byte locations of the relevant elements allows us to retrieve the required chunk of XML in an on-demand basis, without needing to load the complete file in memory.

Though the XPath-based random-access mechanism allows us to quickly retrieve the relevant chunk of XML file in memory, the data still need to be processed to retrieve the particular element having the reference key. The element lookup can be accelerated by applying a user-defined caching mechanism provided with the API. There are manual and automatic ways of resolving references in the data returned by *xxIndex*. The reference resolving can be requested by a Java class in two ways – (i) by specifying the string reference, and (ii) by specifying the object reference.

The string references can be resolved either by manually iterating through the data and looking for the matching element, or by automatically retrieving the required element by looking into a memory-based hash-map. Switching on the caching mechanism in the API can activate the hash-map feature. The caching takes place at the same time as *xxIndex* is making the indexes for the XPath entries while scanning the whole file. Depending on our preference for the type of elements we wish to cache, the *xxIndex* creates a hash-map for the elements of the desired type by using their string identifiers as keys, and their corresponding byte locations as values. We use the terminology for these elements as *indexed* and *ID-mapped*.

The object references requested by the Java class can be resolved by simply activating the *auto-resolve* mechanism provided by the API. When the auto-resolve option is activated for the parent element, the API automatically resolves the internal references encountered in the parent element and creates a full object reference for each reference encountered. The auto-resolve feature can seamlessly provide a complete Java model for the XML without any need for explicit searching and reference resolving, but can increase the resource requirements depending on the number of elements we have set for auto-resolve.

The caching and auto-resolve behavior of the API can be configured using an XML configuration file provided with the API. The values of *indexed*, *ID-mapped,* and *auto-resolving* can be set in the configuration file for the desired elements according to the programming requirements of a user (an example configuration is provided at http://www.code.google.com/p/jmzidentml/w/list).

The XML element to Java object mapping is achieved by means of JAXB (http://www.jaxb.java.net/). The *jmzIdentML* API provides the methods for *marshalling* Java objects to XML fragments, and *unmarshalling* XML fragments to Java objects. During the processing of an XML element, the required element can be retrieved with the help of *xxIndex*. After unmarshalling the relevant XML fragment, internal references are resolved according to the options mentioned above to provide complete information to the user. The process of retrieving Java objects by resolving internal references can be cascaded to any depth, each time reading only the relevant sections of XML in memory.


```
<SpectrumIdentificationResult spectraData_ref="SID_1" spectrumID="index=285" id="SIR_10">
    <SpectrumIdentificationItem passThreshold="false" rank="1"                    A
        peptide_ref="KDLYGNVVLSGGTTMYEGIGER_1@14"
        calculatedMassToCharge="791.381" experimentalMassToCharge="791.522"
        chargeState="3" id="SII_10_1">
       <PeptideEvidenceRef peptideEvidence_ref="PE1_2_0"/>
       <cvParam accession="MS:1001328" cvRef="PSI-MS" value="5.30485096918644E-10"
           name="OMSSA:evalue"/>
       <cvParam accession="MS:1001329" cvRef="PSI-MS" value="2.00865239272489E-13"
            name="OMSSA:pvalue"/>
    </SpectrumIdentificationItem>
    ....
```

```
<Peptide id="KDLYGNVVLSGGTTMYEGIGER_1@14">                                    B
     <PeptideSequence>KDLYGNVVLSGGTTMYEGIGER</PeptideSequence>
     <Modification monoisotopicMassDelta="15.994915" location="15">
        <cvParam accession="UNIMOD:35" cvRef="UNIMOD" name="Oxidation"/>
     </Modification>
</Peptide>
```

```
<PeptideEvidence isDecoy="false" post="L" pre="R" end="312" start="291"       C
  peptide_ref="KDLYGNVVLSGGTTMYEGIGER_1@14"
  dBSequence_ref="dbseq_psu|NC_LIV_020800" id="PE1_2_0"/>
```

```
<DBSequence accession="psu|NC_LIV_020800" searchDatabase_ref="SearchDB_1" length="376"   D
      id="dbseq_psu|NC_LIV_020800">
    <Seq>MADEEVQALVVDNGSGNVKAGVAGDDAPRAVFPSIVGKPKNPG....</Seq>
    <cvParam accession="MS:1001088" cvRef="PSI-MS" value="psu|NC_LIV_020800 |
        organism=Neospora_caninum | product=actin | location=Neo_chrIb:918316-919853(+) |
        length=376" name="protein description"/>
</DBSequence>
```

Figure 1. mzIdentML code snippets for the example discussed in Case Study–I. (A) <SpectrumIdentificationItem> refers to the corresponding <Peptide> by *KDLYGNVVLSGGTTMYEGIGER_1@14*; (B) the <Peptide> element corresponding to *KDLYGNVVLSGGTTMYEGIGER_1@14*; (C) <PeptideEvidence> referenced by *PE1_2_0* in (A); (D) <DBSequence> referenced by *dbseq_psu|NC_LIV_020800* in (C).

Case study I: Information retrieval using the unmarshaller. Here, we present an example demonstrating the basic functionality of the API. Figure 1 presents snippets of an example mzIdentML file that we will use for demonstration purposes. The <SpectrumIdentificationResult> captures all the peptide identifications made from a single spectrum – listing a ranked set of peptide-spectrum matches reported by the search engine. Each individual peptide-spectrum match is reported as a child element in <SpectrumIdentificationItem>. Each ranked result is associated with a peptide sequence, referenced by a key *peptide_ref* (Fig. 1A). The *peptide_ref* points to a <Peptide> element in the document, which contains more detailed information about the peptide, such as the amino acid sequence, sites of potential post-translational modifications, etc. (Fig. 1B). In order to obtain complete information about a peptide-spectrum match, one must read the entries in <SpectrumIdentificationItem> and resolve the *peptide_ref* key to get the corresponding peptide information located in the <Peptide> element. Further, these peptides are theoretically digested from the protein sequence database used for identification. A peptide sequence can be associated with many protein sequences in the database: such references are maintained by <PeptideEvidenceRef>, where each *peptideEvidence_ref* refers to <PeptideEvidence> elements (Fig. 1C). A <PeptideEvidence> element contains information about the possible relationship between the peptide and the corresponding protein sequence such as the start and end position of the peptide in the protein sequence, the amino acids before and after the cleavage sites in the protein sequence, whether the protein sequence is a decoy sequence or not, etc. The protein sequence is referenced by the key *DBSequence_ref*. In order to retrieve the information about the matching protein sequence, the *DBSequence_ref* must be resolved from the <PeptideEvidence> (Fig. 1D).

Figure 2 presents the class layout for the main Java classes used in this example. Whenever a reference needs to be replaced by an actual Java object, the unmarshaller automatically invokes the reference-resolving mechanism. The unmarshaller uses the *xxIndex* to find the byte location of the XML snippet containing the relevant element. In this example, the XML code containing the <Peptide> element with the ID *KDLYGNVVLSGGTTMYEGIGER_1@14* was *looked up* when the reference *peptide_ref* was encountered while processing <SpectrumIdentificationItem>. A similar mechanism was adopted for resolving *DBSequence_ref* in <PeptideEvidence>. After unmarshalling the retrieved XML portion into a Java object, the API seamlessly provides a complete object model to the end user. The arrows in Fig. 2 explain the reference-resolving mechanism used for this example. Figure S1 in the Supporting Information contains the example Java code using the API to process an example file.

Case study II: Information presentation using the marshaller. The API is not only useful for parsing an mzIdentML

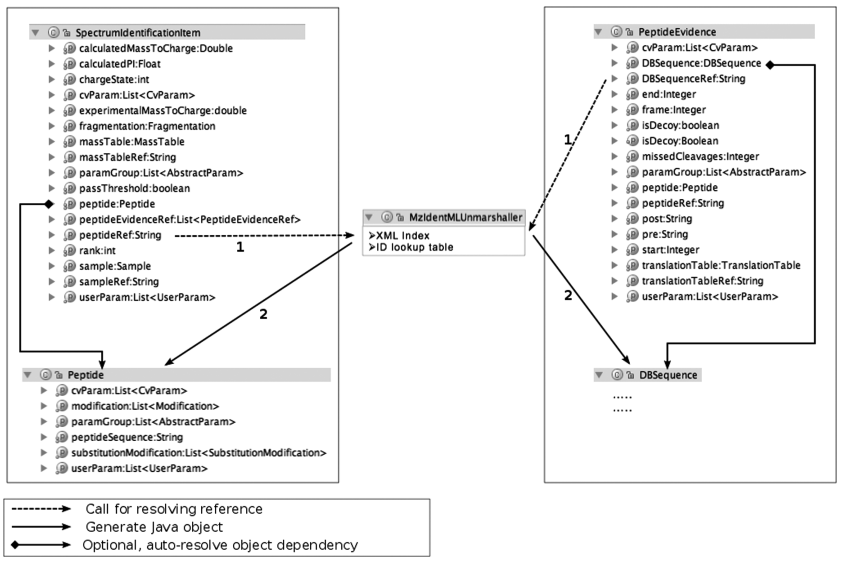

Figure 2. The reference resolving mechanism of *jmzIdentML* for the Case Study–I. The string references (PeptideRef in the left panel and DBSequenceRef in the right panel) are resolved in two steps. First, the index created by *xxIndexer* is looked up to find the relevant information about the referenced element, and second, the referenced element is modeled as a Java object and returned to the user. There is also a mechanism for automatically resolving the object dependencies when the auto-resolve option is switched on.

document for information retrieval, but can also be used to produce new mzIdentML documents. The API provides a marshaller interface to end-users to convert plain Java objects into the corresponding mzIdentML elements in XML format, without the user having to understand the XML schema for the required elements. The marshaller interface is particularly useful in writing *converters*, where it is desired to convert a file from its native format to mzIdentML. We show the usefulness of the marshaller by explaining two converters we developed for the output files produced by the search engines – OMSSA and X!Tandem (beta release). The Java source code for the converters is available in the open-source domain at http://www.code.google.com/p/mzidentml-parsers/.

There are two Java classes *Omssa2mzid* and *Tandem2mzid* in *mzindentml-parsers* for producing mzIdentML compatible outputs. *Omssa2mzid* takes an OMSSA xml file (.omx) as input; whereas, *Tandem2mzid* converts the X!Tandem specific .xml output file. To implement these converters, we have used *jmzIdentML* API along with two other Java-based open-source libraries, OMSSA parser [4] and XTandem parser [5]. While on the one hand, the OMSSA parser and XTandem parser facilitate the reading and parsing of the native input files, *jmzIdentML* on the other hand allows the parsed snippets to be automatically converted into relevant mzIdentML elements. The information in the native files is parsed and read as plain Java objects by the OMSSA parser and XTandem parser; the Java objects in turn are accessed to retrieve information relevant for constructing relevant Java objects provided by *jmzIdentML* interface. Finally, the marshaller is called to convert the newly constructed Java objects into mzIdentML snippets. Figure 3 shows the schematic of the workflow. A description of the methods used in *Omssa2mzid* can be found in Fig. S2 in the Supporting Information.

We have presented an open-source Java API, *jmzIdentML*, for the recently released mzIdentML standard. The two case

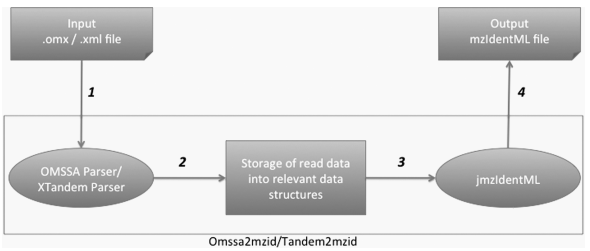

Figure 3. The flow of data in the format converters. (1) Parse the input OMSSA (.omx) or X!Tandem (.xml) file using appropriate parser; (2) Create internal data structures to store the parsed information; (3) Create jmzIdentML supported Java objects from the internal data structures; (4) Marshal the Java objects into XML snippets and produce an output mzIdentML document.

studies show the utility of the API for parsing as well as producing mzIdentML documents. Additionally, the API has also been adopted in a new version of the tool PRIDE Converter [6] to facilitate the upload of mzIdentML data into the PRIDE database. The source code and binaries for the API are freely available on Google code.

We gratefully acknowledge funding that has supported this work, including BBSRC grants (BB/H024654/1) and (BB/G010781/1) to A.R.J. F.R. and D.R. were supported by the Wellcome Trust (grant number WT085949MA). J.A.V. is supported by the EU FP7 grants LipidomicNet (grant number 202272) and ProteomeXchange (grant number 260558).

The authors have declared no conflict of interest.

References

[1] Wedge, D. C., Krishna, R., Blackhurst, P., Siepen, J. A. et al., FDRAnalysis: a tool for the integrated analysis of

tandem mass spectrometry identification results from multiple search engines. *J. Proteome. Res.* 2011, *10*, 2088–2094.

[2] Sturm, M., Bertsch, A., Gropl, C., Hildebrandt, A. et al., OpenMS - an open-source software framework for mass spectrometry. *BMC Bioinformatics* 2008, *9*, 163.

[3] Kessner, D., Chambers, M., Burke, R., Agus, D., et al., ProteoWizard: open source software for rapid proteomics tools development. *Bioinformatics* 2008, *24*, 2534–2536.

[4] Barsnes, H., Huber, S., Sickmann, A., Eidhammer, I., et al., OMSSA Parser: an open-source library to parse and extract data from OMSSA MS/MS search results. *Proteomics* 2009, *9*, 3772–3774.

[5] Muth, T., Vaudel, M., Barsnes, H., Martens, L., et al., XTandem Parser: an open-source library to parse and analyse X!Tandem MS/MS search results. *Proteomics* 2010, *10*, 1522–1524.

[6] Barsnes, H., Vizcaino, J. A., Eidhammer, I., Martens, L., PRIDE Converter: making proteomics data-sharing easy. *Nat. Biotechnol.* 2009, *27*, 598–599.

