Zend_Yadis - Pádraic Brady

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Yadis Component Proposal

Proposed Component Name	Zend_Yadis
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Yadis
Proposers	Pádraic Brady Matthew Weier O'Phinney, Zend liaison
Revision	1.0.7 - 04 July 2007 (wiki revision: 19)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

Yadis is a specification which details a protocol enabling user applications to discover a list of services a Yadis ID has available for use. Although it sounds fuzzy, it's purpose is relatively simple. Based on a valid Identity (e.g. OpenID Url, Sxip ID, XRI, Yoke) it enables an application to locate and describe that Identity's available services (e.g. authentication services, attribute exchange services, etc).

In OpenID, Yadis allows the Relying Party (say a PHP blog) to discover the User's authentication service, it's location, and it's type. In addition, it may also allow the Relying Party to discover whether a Simple Registration Extension service is available (which allow the Relying Party request private User details like email, real name, etc).

The data discovered about services is stored as an XML document called an Extensible Resource Descriptor (XRD). The Yadis Specification details the syntax for this XML document and how its data should be handled when manipulating it. This proposal captures these requirements.

Since Yadis is a standalone specification, it is extensible and allows for the discovery of any service which is tied to a specific Identity (URL or otherwise). It is a requirement for implementing OpenID 2.0 which will utilise Yadis 1.0 to enable Relying Parties locate an OpenID Provider's authentication service. It is also utilised by Sxip ID, XRI Identifiers, Light-Weight Identity (LID), Yoke and mIDm. Yadis is by no means tied to any single Identity broker or service provider.

2. References

- The Yadis Specification 1.0
- Yadis Introduction and Tutorial (Personal Blog)
- OpenID Authentication 2.0

- Ongoing implementation in subversion

3. Component Requirements, Constraints, and Acceptance Criteria

- Zend_Yadis **must** implement the Yadis Specification 1.0 including the Yadis protocol for discovering the XRDS document for an identity.
- Zend_Yadis **must** implement discovery where the Yadis ID is a valid URI
- Zend_Yadis **must** implement discovery where the Yadis ID is an i-name (XRI)
- Zend_Yadis **must** be able to parse an XRDS document into a set of Zend_Yadis_Service objects each holding URL/Type and other data capable of lookup via an XPath query on the object.
- Zend_Yadis **will** use SimpleXML for parsing XRDS documents.
- Zend_Yadis **will** support IRIs (Internationalised Resource Identifiers) once they are supported by the core Zend_Uri component of the framework.

4. Dependencies on Other Framework Components

- Zend_Service_Abstract
- Zend_Service_Exception
- Zend_Http_Client
- Zend_Uri
- Zend_Exception

5. Theory of Operation

The purpose of Zend_Yadis is to implement, in full, the Yadis Specification 1.0.

Upon receipt of any viable Identity (URL, XRI, OpenID, other), Zend_Yadis may be called upon to retrieve a list of all services available for that Identity in an iterable list ordered by priority. The definition of "priority" in this case is as defined by the XRI Technical Committee's XRI Resolution 2.0 working draft.

The initial Identity must either be a URI, or another form capable of being normalised to a URI. XRI normalisation allows for the use of the XDI XRI proxy since HTTP clients typically have no native XRI resolution support.

The normalised URI is requested by a HTTP client. Valid responses are outlines in the Yadis Specification 1.0 and include an XRD document, a response header indicating the location of a valid XRD document, or an XHTML document containing a meta tag which points to a valid XRD location.

Depending on the response, a second HTTP request may be initiated to retrieve the XRD document from it's actual location. Alternatively an Identity may serve an XRD document if a suitable "Accept" header was passed with the HTTP request.

The actual format and ordering of Services, and their constituent data, in the XRD document must follow a strict "priority" standard which (as stated by the Yadis Specification 1.0) is the OASIS XRI Resolution 2.0 specification. This rules the order in which Service objects are iterated across in the collection result object.

At this stage, the result may be passed to a specific handling client - for example, Zend_Openid - which will locate a compatible service type, and utilise its data in order to communicate requests to the discovered service it requires.

6. Milestones / Tasks

- Milestone 1: Implement basic service using Zend dependencies
- Milestone 2: Unit Tests and class refactoring
- Milestone 3: Debugging and Use Case testing
- Milestone 4: Documentation

7. Class Index

- Zend_Yadis
- Zend_Yadis_Xrds_Namespace
- Zend_Yadis_Xrds
- Zend_Yadis_Xrds_Service
- Zend_Yadis_Service
- Zend_Yadis_Xri
- Zend_Yadis_Exception

8. Use Cases

UC-01

Retrieve an OpenID service description **based on URL**

```
require_once 'Zend/Service/Yadis.php';
$openid = 'http://padraic.astrumfutura.com';
$yadis = new Zend_Yadis($openid);
$services = $yadis->discover();

// If an XRI check for a CanonicalID value to use as the OP Local Identifier
if($yadis->isXri()) {
    $openid = $yadis->getCanonicalId();
}

$openidProvider = null;
foreach($services as $service) {
    $types = $service->getTypes();
    if (in_array('http://specs.openid.net/auth/2.0/server', $types) ||
in_array('http://specs.openid.net/auth/2.0/signon', $types)) {
        $openidProvider = $service;
    }
}

$uris = $openidProvider->getUris();
$server = $uris[0];

// if not an XRI, and User did not provide a real OP Local Identifier (e.g. an alias)
// then check whether the Signon service provides an OP Local Identifier for us to use
if (in_array('http://specs.openid.net/auth/2.0/signon', $types)) {
    $oplocals = $service->getElements('LocalID'); // reset to the actual OpenID URL
    $openid = $oplocals[0];
}

/**
 * Data retrieved via discovery can encompass the OP Local Identifier, Server URI, and
 * in the event of an XRI, the CanonicalID value.
 */
```

UC-02

Retrieve an OpenID service description **based on XRI i-name**

```
require_once 'Zend/Service/Yadis.php';
$userOpenid = '=user.name';
$yadis = new Zend_Yadis($userOpenid);
$services = $yadis->discovery();

foreach($services as $service) {
    echo implode(', '$service->getUris()), '<br />';
    foreach($service->getTypes() as $type_url)
    {
        echo $type_url, '<br />';
    }
}
```

9. Class Skeletons

```
/**
 * Zend Framework
 *
 * These class skeletons forms part of a proposal for the Zend Framework and
 * as such are copyright to Zend Technologies USA Inc. as governed by my
 * signed Contributor License Agreement. Licensed under the New BSD License.
 */

/**
 * Zend_Yadis will provide a method of Service Discovery implemented
 * in accordance with the Yadis Specification 1.0. This describes a protocol
 * for locating an XRD document which details Services available. The XRD is
 * typically specific to a single user, identified by their Yadis ID.
 * Zend_Yadis_XRDS will be a wrapper which is responsible for parsing
 * and presenting an iterable list of Zend_Yadis_Service objects
 * holding the data for each specific Service discovered.
 *
 * Note that class comments cannot substitute for a full understanding of the
 * rules and nuances required to implement the Yadis protocol. Where doubt
 * exists, refer to the Yadis Specification 1.0 at:
 *      http://yadis.org/papers/yadis-v1.0.pdf
 *
 * @uses       Zend_Service_Abstract
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis extends Zend_Service_Abstract
{

    /**
     * Constants referring to Yadis response types
     */
    const XRDS_META_HTTP_EQUIV = 2;
    const XRDS_LOCATION_HEADER = 4;
    const XRDS_CONTENT_TYPE = 8;

    /**
     * The current Yadis ID; this is the raw form initially submitted prior
     * to any transformation/validation as an URL. This *may* allow IRI support
     * in the future given IRIs map to URIs and adoption of the IRI standard
     * and are entering common use internationally.
     *
     * @var string
     */
    protected $_yadisId = '';
```

```
    /**
     * The current Yadis URL; this is a URL either validated or transformed
     * from the initial Yadis ID. This URL is used to make the initial HTTP
     * GET request during Service Discovery.
     *
     * @var string
     */
    protected $_yadisUrl = '';

    /**
     * Holds the first response received during Service Discovery.
     *
     * This is required to allow certain Service specific fallback methods.
     * For example, OpenID allows a Yadis fallback which relies on seeking a
     * set of appropriate <link> elements.
     *
     * @var Zend_Http_Response
     */
    protected $_metaHttpEquivResponse = null;

    /**
     * A URL parsed from a HTML document's <meta> element inserted in
     * accordance with the Yadis Specification and which points to a Yadis
     * XRD document.
     *
     * @var string
     */
    protected $_metaHttpEquivUrl = '';

    /**
     * A URI parsed from an X-XRDS-Location response-header. This value must
     * point to a Yadis XRD document otherwise the Yadis discovery process
     * should be considered to have failed.
     *
     * @var string
     */
    protected $_xrdsLocationHeaderUrl = '';

    /**
     * Instance of Zend_Yadis_Xrds_Namespace for managing namespaces
     * associated with an XRDS document.
     *
     * @var Zend_Yadis_Xrds_Namespace
     */
    protected $_namespace = null;

    /**
     * Array of valid HTML Content-Types. Required since Yadis states agents
     * must parse a document if received as the first response and with an
     * MIME type indicating HTML or XHTML. Listed in order of prioroty, with
     * HTML taking priority over XHTML.
     *
     * @link http://www.w3.org/International/articles/serving-xhtml/Overview.en.php
     * @var array
     */
    protected $_validHtmlContentTypes = array(
        'text/html',
        'application/xhtml+xml',
        'application/xml',
        'text/xml'
    );

    /*
     * Array of characters which if found at the 0 index of a Yadis ID string
     * may indicate the use of an XRI.
     *
     * @var array
     */
    protected $_xriIdentifiers = array(
```

```
        '=', '$', '!', '@', '+', '('
    );

    /**
     * Class Constructor
     *
     * Allows settings of the initial Yadis ID (an OpenID URL for example) and
     * an optional list of additional namespaces. For example, OpenID uses a
     * namespace such as: xmlns:openid="http://openid.net/xmlns/1.0"
     * Namespaces are assigned to a Zend_Yadis_Xrds_Namespace container
     * object to be passed more easily to other objects being
     *
     * @param   string $yadisId
     * @param   array $namespaces
     */
    public function __construct($yadisId = null, array $namespaces = null)
    {
        $this->_namespace = new Zend_Yadis_Xrds_Namespace;
        if (isset($namespaces) && count($namespaces) > 0) {
            $this->addNamespaces($namespaces);
        } elseif (isset($namespaces)) {
            require_once 'Zend/Service/Yadis/Exception.php';
            throw new Zend_Yadis_Exception('Expected parameter $namespaces to be an array; but array
appears to be empty');
        }
        if (isset($yadisId)) {
            $this->setYadisId($yadisId);
        }
    }

    /**
     * A Yadis ID is usually an URL, but can also include an IRI, or XRI i-name.
     * The initial version will support URLs as standard before examining options
     * for supporting alternatives (IRI,XRI,i-name) since they require additional
     * validation and conversion steps (e.g. Punycode for IRI) before use.
     *
     * Note: The current Zend_Uri/Zend_Filter classes have no apparent support
     * for validating Internationalized Resource Indicators (ILI) nor for
     * transforming such IRI's to a valid URI via Zend_Uri.
     *
     * @param   string $yadisId
     */
    public function setYadisId($yadisId)
    {}

    /**
     * Returns the original Yadis ID string set for this class.
     *
     * @returns string
     */
    public function getYadisId()
    {}

    /**
     * Attempts to create a valid URI based on the value of the parameter
     * which would typically be the Yadis ID.
     * Note: This currently only supports XRI transformations.
     *
     * @param   string $yadisId
     * @return  Zend_Yadis
     * @throws  Zend_Yadis_Exception
     */
    public function setYadisUrl($yadisId)
    {}

    /**
     * Returns the Yadis URL. This will usually be identical to the Yadis ID,
     * unless the Yadis ID (in the future) was one of ILI, XRI or i-name which
     * required transformation to a valid URI.
     *
```

```
     * @returns string
     */
    public function getYadisUrl()
    {}

    /**
     * Add a list (array) of additional namespaces to be utilised by the XML
     * parser when it receives a valid XRD document.
     *
     * @param   array $namespaces
     * @return  Zend_Yadis
     */
    public function addNamespaces(array $namespaces)
    {}

    /**
     * Add a single namespace to be utilised by the XML parser when it receives
     * a valid XRD document.
     *
     * @param   string $namespace
     * @param   string $namespaceUrl
     * @return  Zend_Yadis
     */
    public function addNamespace($namespace, $namespaceUrl)
    {}

    /**
     * Return the value of a specific namespace.
     *
     * @return   string|null
     */
    public function getNamespace($namespace)
    {}

    /**
     * Returns an array of all currently set namespaces.
     *
     * @return  array
     */
    public function getNamespaces()
    {}

    /**
     * Performs Service Discovery, i.e. the requesting and parsing of a valid
     * Yadis (XRD) document into a list of Services and Service Data. The
     * return value will be an instance of Zend_Yadis_Xrds which will
     * implement SeekableIterator. Returns FALSE on failure.
     *
     * @return  Zend_Yadis_Xrds|boolean
     * @throws  Zend_Yadis_Exception
     * @uses    Zend_Http_Response
     */
    public function discover()
    {}

    /**
     * Return the very first response received when using a valid Yadis URL.
     * This is important for Services, like OpenID, which can attempt a
     * fallback solution in case Yadis fails, and the response came from a
     * user's personal URL acting as an alias.
     *
     * @return  Zend_Http_Response
     */
    public function getUserResponse()
    {}

    /**
     * Run any instance of Zend_Http_Response through a set of filters to
     * determine the Yadis Response type which in turns determines how the
     * response should be reacted to or dealt with.
```

```
     *
     * @return  integer
     */
    protected function _getResponseType(Zend_Http_Response $response)
    {}

    /**
     * Use the Zend_Http_Client to issue an HTTP GET request carrying the
     * "Accept" header value of "application/xrds+xml". This can allow
     * servers to quickly respond with a valid XRD document rather than
     * forcing the client to follow the X-XRDS-Location bread crumb trail.
     *
     * @param   string $url
     * @return  Zend_Http_Response
     * @uses    Zend_Http_Client
     */
    protected function _get($url)
    {}

    /**
     * Checks whether the Response contains headers which detail where
     * we can find the XRDS resource for this user. If exists, the value
     * is set to the private $_xrdsLocationHeaderUrl property.
     *
     * @param   Zend_Http_Response $response
     * @return  boolean
     * @throws  Zend_Yadis_Exception
     */
    protected function _isXrdsLocationHeader(Zend_Http_Response $response)
    {}

    /**
     * Checks whether the Response contains the XRDS resource. It should, per
     * the specifications always be served as application/xrds+xml
     *
     * @param   Zend_Http_Response $response
     * @return  boolean
     */
    protected function _isXrdsContentType(Zend_Http_Response $response)
    {}

    /**
     * Assuming this user is hosting a third party sourced identity under an
     * alias personal URL, we'll need to check if the website's HTML body
     * has a http-equiv meta element with a content attribute pointing to where
     * we can fetch the XRD document.
     *
     * @param   Zend_Http_Response $response
     * @return  boolean
     * @throws  Zend_Yadis_Exception
     */
    protected function _isMetaHttpEquiv(Zend_Http_Response $response)
    {}

    /**
     * Creates a new Zend_Yadis_Xrds object which uses SimpleXML to
     * parse the XML into a list of Iterable Zend_Yadis_Service
     * objects.
     *
     * @param   string $xrdsDocument
     * @return  Zend_Yadis_Xrds|boolean
     */
    protected function _parseXrds($xrdsDocument)
    {}

}

/**
 * The Zend_Yadis_Xrds_Namespace class is a container for namespaces
 * which need to be registered to an XML parser in order to correctly consume
```

```
 * an XRDS document using the parser's XPath functionality.
 *
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis_Xrds_Namespace
{

    /**
     * Default XRDS namespaces which should always be registered.
     *
     * @var array
     */
    protected $_namespaces = array(
        'xrds' => 'xri://$xrds',
        'xrd' => 'xri://$xrd*($v*2.0)'
    );

    /**
     * Class constructor
     */
    public function __construct()
    {}

    /**
     * Add a list (array) of additional namespaces to be utilised by the XML
     * parser when it receives a valid XRD document.
     *
     * @param   array $namespaces
     * @todo    Extract namespaces (common to three classes) to new shared class
     */
    public function addNamespaces(array $namespaces)
    {}

    /**
     * Add a single namespace to be utilised by the XML parser when it receives
     * a valid XRD document.
     *
     * @param   string $namespace
     * @param   string $namespaceUrl
     * @return  void
     * @uses    Zend_Uri
     */
    public function addNamespace($namespace, $namespaceUrl)
    {}

    /**
     * Return the value of a specific namespace.
     *
     * @return   string|null
     */
    public function getNamespace($namespace)
    {}

    /**
     * Returns an array of all currently set namespaces.
     *
     * @return  array
     */
    public function getNamespaces()
    {}

    /**
     * Register all stored namespaces to the parameter SimpleXMLElement object.
     *
     * @param   SimpleXMLElement
     * @return  void
```

```
     */
    public function registerXpathNamespaces(SimpleXMLElement $element)
    {}

}

/**
 * The Zend_Yadis_Xrds class is a wrapper for elements of an
 * XRD document which is parsed using SimpleXML, and contains methods for
 * retrieving data about the document. The concrete aspects of retrieving
 * specific data elements is left to a concrete subclass.
 *
 * @uses       SeekableIterator
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis_Xrds
{

    /**
     * Current key/pointer for the Iterator
     *
     * @var integer
     */
    protected $_currentKey = 0;

    /**
     * Contains the valid xrd:XRD nodes parsed from the XRD document.
     *
     * @var SimpleXMLElement
     */
    protected $_xrdNodes = null;

    /**
     * Instance of Zend_Yadis_Xrds_Namespace for managing namespaces
     * associated with an XRDS document.
     *
     * @var Zend_Yadis_Xrds_Namespace
     */
    protected $_namespace = null;

    /**
     * Constructor; parses and validates an XRD document. All access to
     * the data held in the XML is left to a concrete subclass specific to
     * expected XRD format and data types.
     * Cannot be directly instantiated; must call from subclass.
     *
     * @param   SimpleXMLElement $xrds
     * @param   Zend_Yadis_Xrds_Namespace $namespace
     */
    protected function __construct(SimpleXMLElement $xrds, Zend_Yadis_Xrds_Namespace $namespace)
    {}

    /**
     * Add a list (array) of additional namespaces to be utilised by the XML
     * parser when it receives a valid XRD document.
     *
     * @param   array $namespaces
     * @return  Zend_Yadis
     */
    public function addNamespaces(array $namespaces)
    {}

    /**
     * Add a single namespace to be utilised by the XML parser when it receives
     * a valid XRD document.
     *
```

```
     * @param   string $namespace
     * @param   string $namespaceUrl
     * @return  Zend_Yadis
     */
    public function addNamespace($namespace, $namespaceUrl)
    {}

    /**
     * Return the value of a specific namespace.
     *
     * @return   string|null
     */
    public function getNamespace($namespace)
    {}

    /**
     * Returns an array of all currently set namespaces.
     *
     * @return  array
     */
    public function getNamespaces()
    {}

    /**
     * Fetch an array of all XRD nodes contained by the document. Technically
     * we're only allowed to use the final XRD element if more than one exists.
     */
    protected function _getValidXrdNodes(SimpleXMLElement $xrds)
    {}

    /**
     * Register all namespaces to a SimpleXMLElement. Need to repeat this
     * step for each new SimpleXMLElement we get as the result of another
     * XPath query.
     */
    protected function _registerXpathNamespaces(SimpleXMLElement $element)
    {}

    /**
     * Sort all elements in a list by priority in accordance with the rules
     * defined by Clause 3.3.3 of the XRI Resolution 2.0 Specification with
     * the aim of establishing order descending of highest priority.
     *      http://yadis.org/wiki/XRI_Resolution_2.0_specification
     *
     * @param   Zend_Yadis_Service $service
     */
    protected function _sortByPriority(array $elements)
    {}

}

/**
 * The Zend_Yadis_Xrds_Service class is a wrapper for Service elements of an
 * XRD document which is parsed using SimpleXML, and contains methods for
 * retrieving data about each Service, including Type, Url and other arbitrary
 * data added in a separate namespace, e.g. openid:Delegate.
 *
 * This class extends the basic Zend_Yadis_Xrds wrapper to implement a
 * Service object specific to the Yadis Specification 1.0. XRDS itself is not
 * an XML format ruled by Yadis, but by an OASIS proposal.
 *
 * @uses       Iterator
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis_Xrds_Service extends Zend_Yadis_Xrds implements Iterator implements Iterator
{
```

```
    /**
     * Establish a lowest priority integer; we'll take the upper 2^31
     * integer limit.
     */
    const SERVICE_LOWEST_PRIORITY = 2147483647;

    /**
     * Holds the last XRD node of the XRD document as required by Yadis 1.0.
     *
     * @var SimpleXMLElement
     */
    protected $_xrdNode = null;

    /**
     * The Yadis Services resultset
     *
     * @var array
     */
    protected $_services = array();

    /**
     * Constructor; Accepts an XRD document for parsing.
     * Parses the XRD document by <xrd:Service> element to construct an array
     * of Zend_Yadis_Service objects ordered by their priority.
     *
     * @param   SimpleXMLElement $xrds
     * @param   Zend_Yadis_Xrds_Namespace $namespace
     */
    public function __construct(SimpleXMLElement $xrds, Zend_Yadis_Xrds_Namespace $namespace)
    {}

    /**
     * Implements Iterator::current()
     *
     * Return the current element.
     *
     * @return Zend_Yadis_Service
     */
    public function current()
    {}

    /**
     * Implements Iterator::key()
     *
     * Return the key of the current element.
     *
     * @return integer
     */
    public function key()
    {}

    /**
     * Implements Iterator::next()
     *
     * Increments pointer to next Service object.
     */
    public function next()
    {}

    /**
     * Implements Iterator::rewind()
     *
     * Rewinds the Iterator to the first Service object
     *
     * @return boolean
     */
    public function rewind()
    {}
```

```
    /**
     * Implement Iterator::valid()
     *
     * @param  integer $key
     * @return boolean
     */
    public function valid($key = null)
    {}

    /**
     * Add a service to the Service list indexed by priority. Assumes
     * a missing or invalid priority should be shuffled to the bottom
     * of the priority order.
     *
     * @param   Zend_Yadis_Service $service
     */
    protected function _addService(Zend_Yadis_Service $service)
    {}

}

/**
 * The Zend_Yadis_Service class represents a service parsed from the
 * XRD node of a Yadis 1.0 XRDS document.
 *
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis_Service
{

    /**
     * Holds the Service node parsed from a Yadis XRDS document as a
     * SimpleXMLElement object.
     *
     * @var SimpleXMLElement
     */
    protected $_serviceNode = null;

    /**
     * Holds the Service node parsed from a Yadis XRDS document as a
     * SimpleXMLElement object.
     *
     * @var SimpleXMLElement
     */
    protected $_namespace = null;

    /**
     * Class constructor; initialise the object with a Service node from the
     * XRDS document, and the current Zend_Services_Yadis_Xrds_Namespace
     * object to provide the current namespaces for using XPath queries.
     *
     * @param   SimpleXMLElement $serviceNode
     * @param   Zend_Yadis_Xrds_Namespace $namespace
     */
    public function __construct(SimpleXMLElement $serviceNode, Zend_Yadis_Xrds_Namespace $namespace)
    {}

    /**
     * Return an array of Service Type URI strings. This method will NOT
     * validate the resulting URIs.
     *
     * @return  array
     */
    public function getTypes()
    {}
```

```
    /**
     * Return an array of Service Type URI strings. This method will NOT
     * validate the resulting URIs. URI values in the array will have key
     * values matching their priority, and be ordered based on their
     * priority values - highest (i.e. lowest integer) priority first.
     *
     * @return  array
     */
    public function getUris()
    {}

    /**
     * Returns the Priority integer of this Service node.
     *
     * @return  integer
     */
    public function getPriority()
    {}

    /**
     * Return the current XRDS Service node as a SimpleXMLElement object.
     *
     * @return  SimpleXMLElement
     */
    public function getXmlObject()
    {}

    /**
     * Return the current Zend_Yadis_Xrds_Namespace object.
     *
     * @return  Zend_Yadis_Xrds_Namespace
     */
    public function getNamespaceObject()
    {}

    /**
     * Return an array of the current XRDS namespaces for working with any
     * XPath queries on the Service node.
     *
     * @return  SimpleXMLElement
     */
    public function getNamespaces()
    {}

    /**
     * Retrieve Elements of the current Service node by their name, and return
     * as a SimpleXMLElement object. The Elements should be direct children of
     * the Service node. This method basically just passes the $element string
     * as an XPath query so it's open to other uses despite the assumed use
     * case.
     *
     * @param   string $element
     */
    public function getElements($element)
    {}

    /**
     * Run an XPath query against the current Service node.
     *
     * @param   string $query
     */
    public function xpath($query)
    {}

}

/**
 * Provides methods for translating an XRI into a URI.
 *
 * @uses       Zend_Service_Abstract
```

```
 * @category   Zend
 * @package    Zend_Service
 * @subpackage Yadis
 * @author     Pádraic Brady (http://blog.astrumfutura.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
class Zend_Yadis_Xri extends Zend_Service_Abstract
{

    /**
     * Hold an instance of this object per the Singleton Pattern.
     *
     * @var Zend_Yadis_Xri
     */
    protected static $_instance = null;

    /*
     * Array of characters which if found at the 0 index of a Yadis ID string
     * may indicate the use of an XRI.
     *
     * @var array
     */
    protected $_xriIdentifiers = array(
        '=', '$', '!', '@', '+', '('
    );

    /**
     * Default proxy to append XRI identifier to when forming a valid URI.
     *
     * @var string
     */
    protected $_proxy = 'http://xri.net/';

    /**
     * Instance of Zend_Yadis_Xrds_Namespace for managing namespaces
     * associated with an XRDS document.
     *
     * @var Zend_Yadis_Xrds_Namespace
     */
    protected $_namespace = null;

    /**
     * The XRI string.
     *
     * @var string
     */
    protected $_xri = null;

    /**
     * The URI as translated from an XRI and appended to a Proxy.
     *
     * @var string
     */
    protected $_uri = null;

    /**
     * A Canonical ID if requested, and parsed from the XRDS document found
     * by requesting the URI created from a valid XRI.
     *
     * @var string
     */
    protected $_canonicalId = null;

    /**
     * Constructor; protected since this class is a singleton.
     */
    protected function __construct()
    {}

    /**
```

```
     * Return a singleton instance of this class.
     *
     * @return  Zend_Yadis_Xri
     */
    public static function getInstance()
    {}

    /**
     * Set a Namespace object as required to parse any resulting XRDS document
     * received while resolving an XRI, to a URI and finally a Canonical Id.
     */
    public function setNamespace(Zend_Yadis_Xrds_Namespace $namespace)
    {}

    /**
     * Set an XRI proxy URI. A default of "http://xri.net/" is available.
     *
     * @param   string $proxy
     * @return  Zend_Yadis_Xri
     * @throws  Zend_Yadis_Exception
     * @uses    Zend_Uri
     */
    public function setProxy($proxy)
    {}

    /**
     * Return the URI of the current proxy.
     *
     * @param   string $proxy
     * @uses    Zend_Uri
     */
    public function getProxy()
    {}

    /**
     * Set an XRI to be translated to a URI.
     *
     * @param   string $url
     * @return  Zend_Yadis_Xri
     * @throws  Zend_Yadis_Exception
     */
    public function setXri($xri)
    {}

    /**
     * Return the original XRI string.
     *
     * @return  string
     */
    public function getXri()
    {}

    /**
     * Attempts to convert an XRI into a URI. In simple terms this involves
     * removing the "xri://" prefix and appending the remainder to the URI of
     * an XRI proxy such as "http://xri.net/".
     *
     * @param   string $xri
     * @return  string
     * @throws  Zend_Yadis_Exception
     * @uses    Zend_Uri
     */
    public function toUri($xri = null, $serviceType = null)
    {}

    /**
     * Based on an XRI, will request the XRD document located at the proxy
     * prefixed URI and parse in search of the XRI Canonical Id. This is
     * a flexible requirement. OpenID 2.0 requires the use of the Canonical
     * ID instead of the raw i-name. 2idi.com, on the other hand, does not.
```

```
     *
     * @param   string $xri
     * @return  string
     * @throws  Zend_Yadis_Exception
     * @uses    Zend_Uri
     */
    public function toCanonicalId($xri = null)
    {}

    public function getCanonicalId()
    {}

    /**
     * Required to request the root i-name (XRI) XRD which will provide an
     * error message that the i-name does not exist, or else return a valid
     * XRD document containing the i-name's Canonical ID.
     *
     * @param   string $uri
     * @return  Zend_Http_Response
     * @uses    Zend_Http_Client
     * @todo    Finish this correctly using the QXRI rules.
     */
    protected function _get($url, $serviceType = null)
    {}

}
```

```
class Zend_Yadis_Exception extends Zend_Exception
{}
```

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>