|  |
| --- |
|  |
|  |

Overview of developing

a Posting Engine Channel

Andrew Hudson

Version history

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Date** | **Author** | **Comments** |
| 0.1 | 14/07/2016 | AJH | Initial draft. |
| 0.2 | 23/08/2016 | AJH | Changes after review. |
| 0.3 | 30/08/2016 | AJH | Changes after second review. |

Contents

[1 Overview 5](#_Toc8028366)

[1.1 Scope 5](#_Toc8028367)

[1.1.1 Feature boundaries 5](#_Toc8028368)

[1.1.2 Target users 5](#_Toc8028369)

[2 Planning a New Feed 5](#_Toc8028370)

[2.1 Getting Information from the Job Board 5](#_Toc8028371)

[2.1.1 Also Consider Whitelisting 5](#_Toc8028372)

[2.2 Our Example: Jobserve 5](#_Toc8028373)

[2.2.1 Jobserve Test URL and Credentials 6](#_Toc8028374)

[2.3 How is Data Sent to the Job Board 6](#_Toc8028375)

[2.3.1 HTTP Form Post 6](#_Toc8028376)

[2.3.2 JSON API 6](#_Toc8028377)

[2.3.3 XML HTTP Post 8](#_Toc8028378)

[2.3.4 SOAP Service 12](#_Toc8028379)

[2.4 Job Board Capabilities 13](#_Toc8028380)

[2.4.1 Debug Mode 13](#_Toc8028381)

[2.4.2 Test Mode 13](#_Toc8028382)

[2.4.3 Production Mode 14](#_Toc8028383)

[2.4.4 Posting Capabilities 14](#_Toc8028384)

[2.4.5 Post Duration 14](#_Toc8028385)

[2.4.6 Account Verification 15](#_Toc8028386)

[2.4.7 Jobserve’s Job Board Capabilities 17](#_Toc8028387)

[2.5 Mapping Job Board Fields to the PE 18](#_Toc8028388)

[2.5.1 Mapping Salaries 19](#_Toc8028389)

[2.5.2 Mapping Vacancy Type 20](#_Toc8028390)

[2.5.3 Mapping Working Hours 21](#_Toc8028391)

[2.5.4 Mapping Pay Frequency 21](#_Toc8028392)

[2.6 Ignore Web-cruit Considerations 21](#_Toc8028393)

[3 Create the Basic Feed Components 22](#_Toc8028394)

[3.1 Basic Feed Classes 22](#_Toc8028395)

[3.2 Create Basic Feed Classes 23](#_Toc8028396)

[3.2.1 Create a Vacancy class 23](#_Toc8028397)

[3.2.2 Create Custom Data Classes 25](#_Toc8028398)

[3.2.3 Create the Account Data Class 27](#_Toc8028399)

[3.2.4 OutputData class 28](#_Toc8028400)

[3.2.5 Create the Settings Factory 29](#_Toc8028401)

[3.2.6 Create the Channel Class 29](#_Toc8028402)

[4 Create Test Classes 31](#_Toc8028403)

[4.1 Create a Content Class 31](#_Toc8028404)

[4.1.1 JobserveContent.cs 32](#_Toc8028405)

[4.2 Create a Debug Mode Unit Test 33](#_Toc8028406)

[4.2.1 Running our First Test 33](#_Toc8028407)

[4.2.2 Fixing our First Unit Test 34](#_Toc8028408)

[4.2.3 Parsing the Response 35](#_Toc8028409)

[4.2.4 A Working Unit Test 36](#_Toc8028410)

[4.3 Create Test Mode Unit Test Class 36](#_Toc8028411)

[4.4 Create a Production Mode Unit Test Class 37](#_Toc8028412)

[5 Building the Request 38](#_Toc8028413)

[5.1 Request Types 38](#_Toc8028414)

[5.2 Base Class 39](#_Toc8028415)

[5.2.1 RequestBase 39](#_Toc8028416)

[5.2.2 UpdatableRequest 40](#_Toc8028417)

[5.3 Concrete Classes 40](#_Toc8028418)

[5.3.1 PostRequest 40](#_Toc8028419)

[5.3.2 UpdateRequest 40](#_Toc8028420)

[5.3.3 RepostRequest 41](#_Toc8028421)

[5.3.4 DeleteRequest 41](#_Toc8028422)

[5.3.5 VerifyAccount Request 41](#_Toc8028423)

[5.4 Generating the Request 42](#_Toc8028424)

[5.4.1 Implementing the GenerateRequest method 42](#_Toc8028425)

[5.5 Building a Request Factory and Handling Mapping Logic 44](#_Toc8028426)

[6 Implementing the HTTP Poster 46](#_Toc8028427)

[6.1 Create a Response Class 47](#_Toc8028428)

[6.2 Interface for the HTTP Poster 47](#_Toc8028429)

[6.3 Implement the HTTP Poster 47](#_Toc8028430)

[7 Inspecting the Response 49](#_Toc8028431)

[7.1 Create a Response Parser 49](#_Toc8028432)

[8 Implementing the Channel Class 50](#_Toc8028433)

[8.1 Adding Properties to the Channel Class 50](#_Toc8028434)

[8.1.1 Adding an additional Constructor 50](#_Toc8028435)

[8.1.2 Amending the Original Constructor 50](#_Toc8028436)

[8.2 Logging the Request 51](#_Toc8028437)

[8.3 Logging the Response 51](#_Toc8028438)

[8.4 Sending the Request and Parsing the Response 51](#_Toc8028439)

[9 Expanding Unit Tests 52](#_Toc8028440)

[9.1 Unit Tests for Debug Mode 52](#_Toc8028441)

[9.1.1 Checking the Request for Part-Time Positions 52](#_Toc8028442)

[9.1.2 Checking the Request for Update, Repost and Delete Requests 53](#_Toc8028443)

[9.1.3 Testing for Unknown Errors (and an introduction to Moq) 54](#_Toc8028444)

[9.1.4 Testing for Web Exception 54](#_Toc8028445)

[9.2 Unit Tests for Test Mode 55](#_Toc8028446)

[9.2.1 Testing Verify Account with Invalid Credentials 55](#_Toc8028447)

[9.2.2 Testing Verify Account with Valid Credentials 55](#_Toc8028448)

[9.3 Unit Tests for Production Mode 55](#_Toc8028449)

[9.3.1 Unit Test to Delete a Non-Existent Adverts 55](#_Toc8028450)

[10 Web-cruit Integration 56](#_Toc8028451)

[10.1 Check for Null Reference Exceptions 56](#_Toc8028452)

[10.2 Debugging via Web-cruit 56](#_Toc8028453)

[10.3 Editing the Generated Code Files 57](#_Toc8028454)

[10.3.1 Modify the Data Dictionary 57](#_Toc8028455)

[10.3.2 Modify the CFC 58](#_Toc8028456)

[11 Appendix A – Jobserve Posting Schema 59](#_Toc8028457)

[12 Appendix B – Jobserve Industry List 60](#_Toc8028458)

[13 Appendix C - Jobserve Category List 60](#_Toc8028459)

# Overview

## Scope

### Feature boundaries

The Posting Engine is an application service for posting “content” to “channels”.

The current implementation of the Posting Engine supports posting vacancies (content objects) to job boards (channels) thereby creating job board adverts (channel postings).

The scope of this document also covers the integration of a channel into web-cruit.

### Target users

This document is intended to be used by software developers employed by Kaonix to aid them through the process of developing a channel from start to finish.

As an example, an integration of Jobserve will be used. Parts of the Jobserve specification have been copied into the Appendix.

# Planning a New Feed

## Getting Information from the Job Board

We ideally need the following information from any job board we wish to integrate with:

* Specification – details how we send data to the job board and in what format (HTTP form post, SOAP etc)
* Test site credentials and URL
* Live site URL

However, the information we get from job boards can vary. It is not uncommon for a job board not to provide a testing platform and we will sometimes have to make do with using their live site for testing.

As long as we have documentation, then we can start work on the feed straight away even if we don’t have working credentials yet. We can build up most of the feed and we can test authentication failures on the job feed in the meantime.

### Also Consider Whitelisting

Some job boards may need us to be whitelisted. At the time of writing we will need to provide them with our external IP address from Queen Anne House which is xxx.xxx.xxx.xxx and the external IP address for our Rockford data centre which is xxx.xxx.xxx.xxx.

For a developer working from home, we can access the job board by ticking “Use Default Gateway on remote network” on our VPN settings:

## Our Example: Jobserve

The example we will be using for this document is an integration into Jobserve.

### Jobserve Test URL and Credentials

|  |  |
| --- | --- |
| Test Post URL | <http://193.119.59.225> |
| Live | <https://xml.jobserve.com> |
| Account Username (Test only) | 999111 |
| Password | test111 |

## How is Data Sent to the Job Board

The specification will tell you how data is sent to the job board, and will help us decide which C# technology to use in our code.

### HTTP Form Post

For a simple HTTP form post, you should use a Friday Media Group as an example. The form fields are built using the NameValueCollection class and any values added are automatically HTML encoded for you.

NameValueCollection coll = new NameValueCollection();

coll.Add("Username", "someone");

coll.Add("Password", "letmein");

string response;

using (var webClient = new ServerWebClient())

{

byte[] responseArr = webClient.UploadValues(postingUrl, coll);

response = Encoding.UTF8.GetString(responseArr);

}

### JSON API

Some job board APIs use JSON as their chosen format for sending and receiving requests. If this is the case, you can use the Not Going to Uni channel as an example.

#### Example of Class Representing a Request

namespace Kaonix.PE.Channels.NotGoingToUni

{

[DataContract(Name = "data")]

public class NotGoingToUniRequest

{

private static string notGoingToUniDateFormat = "yyyy-MM-dd";

[DataMember(Name = "heading", IsRequired = true, Order = 1)]

public string Heading { get; set; }

[DataMember(Name = "body", IsRequired = true, Order = 2)]

public string Body { get; set; }

[DataMember(Name = "ongoing", IsRequired = true, Order = 5)]

public bool Ongoing { get; set; }

// removed for brevity

public string[] Locations { get; set; }

[DataMember(Name = "categories", IsRequired = true, Order = 12)]

public string[] Categories { get; set; }

}

}

#### Example of Serialising a Request

private string SerializeRequest(NotGoingToUniRequest request)

{

string json = "";

using (MemoryStream ms = new MemoryStream())

{

DataContractJsonSerializer serializer = new DataContractJsonSerializer(typeof(NotGoingToUniRequest));

serializer.WriteObject(ms, request);

ms.Position = 0;

using (StreamReader sr = new StreamReader(ms))

{

json = sr.ReadToEnd();

}

}

if (this.Host.ExecutionMode == ChannelExecutionMode.Debug)

this.Host.Trace("Running in debug mode.");

this.Host.Trace(json + Environment.NewLine);

return json;

}

#### Example of Class Representing a Response

namespace Kaonix.PE.Channels.NotGoingToUni

{

[DataContract]

public class NotGoingToUniResponse

{

[DataMember(Name = "status", IsRequired = true)]

public string Status { get; set; }

[DataMember(Name = "message", IsRequired = true)]

public string Message { get; set; }

[DataMember(Name = "errors", IsRequired = false)]

public string[] Errors { get; set; }

[DataMember(Name = "ID", IsRequired = false)]

public int ID { get; set; }

}

}

#### Example of Deserialising a Response

private NotGoingToUniResponse DeserializeResponse(string jsonResponse)

{

NotGoingToUniResponse response = new NotGoingToUniResponse();

using (MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(jsonResponse)))

{

try

{

DataContractJsonSerializer serializer = new DataContractJsonSerializer(typeof(NotGoingToUniResponse));

response = (NotGoingToUniResponse)serializer.ReadObject(ms);

}

catch (Exception e)

{

throw new RemoteChannelException(string.Format("Could not deserialise response. {0}", e.Message), e);

}

}

return response;

}

Notice that the deserialization is wrapped within a try…catch block in case the job board returns an unexpected response, and if there is an exception, we should rethrow a RemoteChannelException.

More information on serialising JSON can be found here: <https://msdn.microsoft.com/en-us/library/bb412179(v=vs.110).aspx>

### XML HTTP Post

There are two methods how you might want to implement a job feed which uses XML over a HTTP POST request.

#### XML Serialisation

If the structure of the XML is relatively flat then XML serialisation might be the most convenient way to generate the request. However, if the XML is complex with many nested elements, you may consider using XML Writer instead as you would have to create a class for each complex element in the XML.

The Empty Lemon channel has a good example of how to serialise and deserialise XML

##### Example of Class Representing a Request

namespace Kaonix.PE.Channels.EmptyLemon.Request

{

[Serializable]

[XmlRoot("Job")]

public class EmptyLemonJob

{

[XmlAttribute]

public EmptyLemonJobAction Action { get; set; }

[XmlElement(Order = 1)]

public string LoginEmail { get; set; }

[XmlElement(Order = 2)]

public string LoginPassword { get; set; }

[XmlElement(Order = 3)]

public string JobReference { get; set; }

public bool ShouldSerializeJobReference()

{

return Action != EmptyLemonJobAction.VerifyAccount;

}

// removed for brevity

[XmlElement(Order = 21)]

public int DaysToAdvertise { get; set; }

public bool ShouldSerializeDaysToAdvertise()

{

return IsActionPostOrAmend();

}

/// <summary>

/// Method returns true if the post is not a delete

/// </summary>

/// <returns></returns>

private bool IsActionPostOrAmend()

{

return Action != EmptyLemonJobAction.Delete;

}

}

}

##### Example of Serialising a Request

private string SerialiseVacancyToXml(EmptyLemonJobAction action)

{

EmptyLemonRequest emptyLemonRequest = CreateRequest(action);

XmlSerializerNamespaces ns = new XmlSerializerNamespaces();

ns.Add("", "");

StringWriterUTF8 sw = new StringWriterUTF8();

XmlSerializer serializer = new XmlSerializer(typeof (EmptyLemonRequest));

serializer.Serialize(sw, emptyLemonRequest, ns);

if (this.Host.ExecutionMode == ChannelExecutionMode.Debug)

this.Host.Trace("PLEASE NOTE: this running in Debug Mode");

this.Host.Trace(sw + Environment.NewLine); // store the request as part of the trace

return sw.ToString();

}

##### Example of a Class Representing a Response

namespace Kaonix.PE.Channels.EmptyLemon.Response

{

[Serializable]

[XmlRoot("Job")]

public class EmptyLemonResponse

{

public string Message { get; set; }

public bool Successful { get; set; }

public string SenderReference { get; set; }

}

}

##### Example of Deserialising a Response

private EmptyLemonResponse ParseResponse(string xmlResponse)

{

XmlSerializer serializer = new XmlSerializer(typeof(EmptyLemonResponse));

StringReader stringReader = new StringReader(xmlResponse);

EmptyLemonResponse response = (EmptyLemonResponse)serializer.Deserialize(stringReader);

return response;

}

For more information see:

* <https://msdn.microsoft.com/en-us/library/58a18dwa(v=vs.110).aspx>
* <http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part>
* <http://www.codeproject.com/Articles/487571/XML-Serialization-and-Deserialization-Part-2>

##### XML Attribute Overrides

If we decide to use XML serialisation to generate our requests, we can also manipulate what is generated if there are small differences between the post job and update job requests, and save us the effort of having to replicate our code.

Consider the following class which would generate the XML as follows:

<SourcedJob UserName=”someone” Password=”letmein” />

namespace Kaonix.PE.Channels.Welfare2Work.SourcedJobRequest

{

[Serializable]

[XmlRoot("SourcedJob")]

public class SourcedJob

{

/// <param name="requestType"></param>

public SourcedJob()

{

this.Vacancy = new SourceJobVacancy();

}

private Welfare2WorkRequestType requestType;

[XmlIgnore]

public Welfare2WorkRequestType RequestType

{

get { return this.requestType; }

set

{

this.Vacancy.RequestType = value;

this.requestType = value;

}

}

[XmlAttribute]

public string UserName { get; set; }

[XmlAttribute]

public string Password { get; set; }

}

}

If the only difference between a post and an update request is that the <SourcedJob> element is changed to either <CreateSourcedJob> or <UpdateSourcedJob> then you do not have to create another class and decorate with another attribute. The XmlAttributeOverrides class can help us do this.

// if the request type is 'verify account' then default to 'Create' for Xml attribute overrides

string attribOverrideValue = “Create”;

// A new XmlAttributes class would generate <CreateSourcedJob>

XmlAttributes customRootAttrib = new XmlAttributes

{

XmlRoot = new XmlRootAttribute(attribOverrideValue + "SourcedJob")

};

XmlAttributeOverrides overrides = new XmlAttributeOverrides();

overrides.Add(typeof(SourcedJob), customRootAttrib);

XmlSerializer serializer = new XmlSerializer(typeof(SourcedJob), overrides);

serializer.Serialize(writer, CreateSourcedJob(requestType), ns);

More information on XML Overrides: <http://www.ikriv.com/dev/dotnet/OverrideXml.shtml>

#### XML Writer

The problem with XML serialisation is if XML you are creating consists of a large hierarchy, you then need to create a class for each element which will be serialised. The XML Writer could be considered as a simpler solution in this case.

Take the following XML:

<?xml version="1.0" encoding="utf-8"?>

<Vacancy action = "add" reference = "VAC-1">

<JobTitle>Manager</JobTitle>

<Description><![CDATA[This is a <b>HTML</b> description]]></Description>

<Salary currency = "GBP">

<MinSalary>22000.00</MinSalary>

<MaxSalary>27500.00</MaxSalary>

</Salary>

</Vacancy>

Generating the XML is done easily by writing to a MemoryStream and then reading the stream to a string.

string generatedXml = string.Empty;

XmlWriterSettings settings = new XmlWriterSettings { Indent = true };

using (MemoryStream ms = new MemoryStream())

{

using (XmlWriter writer = XmlWriter.Create(ms, settings))

{

writer.WriteStartDocument();

writer.WriteStartElement("Vacancy");

writer.WriteAttributeString("action", "add");

writer.WriteAttributeString("reference", "VAC-1");

writer.WriteElementString("JobTitle", "Manager");

writer.WriteStartElement("Description");

writer.WriteCData("This is a <b>HTML</b> description");

writer.WriteEndElement();

writer.WriteStartElement("Salary");

writer.WriteAttributeString("currency", "GBP");

writer.WriteElementString("MinSalary", string.Format("{0:f2}", 22000));

writer.WriteElementString("MaxSalary", string.Format("{0:f2}", 27500));

writer.WriteEndElement();

writer.WriteEndElement();

writer.WriteEndDocument();

}

ms.Flush();

ms.Position = 0;

using (StreamReader sr = new StreamReader(ms, Encoding.UTF8))

{

generatedXml = sr.ReadToEnd();

}

}

Job feeds which use XML Writer include:

* LinkedIn
* MyJobGroup
* Pertemps
* Welfare2Work

More information on XML Writer:

<http://www.dotnetperls.com/xmlwriter>

<https://msdn.microsoft.com/en-us/library/system.xml.xmlwriter(v=vs.110).aspx>

### SOAP Service

#### Generating a Proxy Class

For complex web services a proxy class can be generated from a WSDL which takes care of the XML generation and serialisation for us.

The Strike Jobs feed (under Zod) is such a feed which is implemented like this

More information can be read here: <https://msdn.microsoft.com/en-us/library/bb628652.aspx>

However, using WCF means that if an error occurs then the .NET framework may disguise the error as there is a known issue with disposing of the client, so we have to implement a solution ourselves using a partial class. An example of this is in the Strike Jobs feed.

More details here: <https://www.sslvpn.online/does-the-c-wcf-proxy-clientbaset-disposal-issue-still-exist-in-net-4-5/>

#### Manually Implementing a SOAP Service

The Welfare 2 Work job feed uses a SOAP service to send and receive requests. Because the SOAP service only contains one operation called “Process” where we send some generated XML, it wouldn’t be a big benefit to generate a proxy class for this.

POST /CME/Service.asmx HTTP/1.1

Host: staging.welfare2work.co.uk

Content-Type: text/xml; charset=utf-8

Content-Length: length

SOAPAction: "http://www.kaonix.com/CandidateMatchingEngine/Process"

<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Process xmlns="http://www.kaonix.com/CandidateMatchingEngine">

<xmlRequest>string</xmlRequest>

</Process>

</soap:Body>

</soap:Envelope>

In this instance, we serialised a request to XML using the methods described in section 2.3.3.1 and sent the XML in a manually constructed SOAP envelope which contains placeholder for the generated request.

private string CreateSoapEnvelopeForRequest()

{

StringBuilder soapEnvelope = new StringBuilder();

soapEnvelope.Append(@"

<?xml version=""1.0"" encoding=""utf-8""?>

<soap:Envelope xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance"" xmlns:xsd=""http://www.w3.org/2001/XMLSchema"" xmlns:soap=""http://schemas.xmlsoap.org/soap/envelope/"">

<soap:Body>

<Process xmlns=""http://www.kaonix.com/CandidateMatchingEngine"">

<xmlRequest><![CDATA[{0}]]></xmlRequest>

</Process>

</soap:Body>

</soap:Envelope>

");

return soapEnvelope.ToString();

}

## Job Board Capabilities

There are many things to consider when analysing a job boards capabilities, such as:

* The PE expects a job feed to have up to three execution modes (Debug, Test and Production), and the main factor in this is if the job board has a separate test environment from the live environment.
* The PE needs to know whether a job board allows us to post, update, delete or repost a job. The update option is usually the one which needs more consideration.
* We need to decide if the PE feed for the job board should support account verification. This needs consideration about how this should be implemented, and if it should be supported at all.
* The PE needs to know how a job board defines a job adverts duration on a board. Some job boards required an explicit expiry date, whereas other job boards will ask for a duration (such as “1 week” or “14 days”). Some job boards won’t allow flexibility in how long an advert is live for, and will have fixed rules in place. For example, some job boards will force all job postings to be live for 28 days only (and in which case, if we want the job advert to be live for longer, we’d have to delete the original advert, and post a new one as a replacement.
* The PE needs to know how to handle job expiration. Should the PE remove the advert from the board itself automatically when the vacancy is due to be expired, or should we let the job board handle this?

The PE has the ability to post vacancies in three execution modes.

We need a better introduction to this section. It’s not just about execution mode but also about whether post, update, delete and repost are supported, whether account verification is supported and how we map post duration.

### Debug Mode

Every PE feed will be able to support debug mode. In this scenario, the job feed doesn’t send a real request to the job board, it simply returns a string faking a response which is in the format of how a response would be returned from the job board.

The purpose of this mode is to enable the developer to verify that the channel correctly implements the contract required by the Posting Engine.

### Test Mode

If the job board has a test environment with a separate posting URL from the live site then we can say that the PE feed supports test mode.

This is not the same as having a test account which is posting to the live site, in this scenario we have to say the feed does not support test mode.

#### Our Example: Jobserve Test Mode

The specification will usually tell us if the job board supports a test environment. This is the case for Jobserve, which in their specification provides us with test details.



### Production Mode

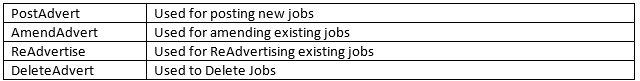
All feeds will need to support production mode if they are going to be used by customers. Scenarios where production mode might be turned off is if a feed is deployed but not yet ready for customer use, or if the feed is to be decommissioned.

### Posting Capabilities

We need to determine if the job board can Post, Update, Repost and Delete.

#### Our Example: Jobserve Posting Capabilities

Our specification clearly tells us we can utilise every capability the PE can offer us:

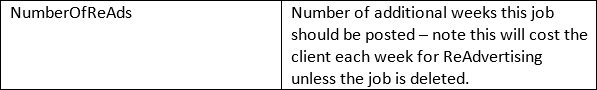


### Post Duration

Most job boards will allow us to determine when a job expires. This is typically done by either specifying an explicit date (e.g. the vacancy will expire on 31st August 2016) or we specify a time span period (e.g. the vacancy will expire in 4 weeks’ time).

#### Our Example: Jobserve Posting Duration

How Jobserve manages it’s posting durations for a vacancy is actually rather unusual compared to most other job boards. In the specification it says this:



What this is saying is that we can only specify a fixed number of weeks an advert can be posted for. This is not too much of a problem for us, as we can give the Jobserve feed a fixed date, and use the PostProperties class has a GetDurationInWeeks method which will calculate this for us. Depending on the number of re-advertisements used, the advert can be extended for a period of time.

### Account Verification

One of the most common causes of job board errors are incorrect credentials. For this, the PE provides a VerifyAccount method to test the credentials.

Two are two primary methods in testing account verification.

#### Logging in via a User Interface

A small robot function can be written which will log into the job board account via a user interface and if successful, will log back out. If any problems occur then an exception is thrown to signal that the account details are incorrect. However, if the website changes, then this functionality can be brittle and break easily.

The BarZone channel has an example of this functionality.

#### Partial Job Post with Deliberately Omitted Mandatory Fields

The most common way to test account authenticity is to send a post request but with deliberate errors. Most job board API’s will have different account credentials to that of the user interface, so this method far more widely used than that of logging in via a user interface.

Most job boards will check the credentials first and verify them before checking the validity of the other parts of the request.

Caution: Sometimes a job feed may only validate the credentials once the whole XML request is valid and all mandatory fields are populated with valid values, meaning the only way to verify account details is to actually make a job post. In this scenario, you will have to conclude that account verification is not supported. We cannot verify an account if it means using up a client’s credits.

As an example, most, if not all job feeds are likely to have a job title or a description as a required field. So we could send a request such as:

<?xml version="1.0"?>

<JobFeedApiRequest>

<Credentials>

<Username>someone</Username>

<Password>letmein</Password>

</Credentials>

<PostJob>

<Title />

<Description />

<Ref />

</PostJob>

</JobFeedApiRequest>

And a response may be:

<?xml version="1.0"?>

<JobFeedApiResponse>

<Credentials>

<Username>someone</Username>

<Password>letmein</Password>

</Credentials>

<PostJobResult Success="False">

<ErrorMessages>

<Error>Title is required</Error>

<Error>Description is required</Error>

<Error>Ref is required</Error>

</ErrorMessages>

</PostJobResult>

</JobFeedApiResponse>

In this instance, receiving a “Title is required” error would mean the account authentication has been a success. However, if authentication has failed, you may end up with a response such as:

<?xml version="1.0"?>

<JobFeedApiResponse>

<Credentials>

<Username>someone</Username>

<Password>letmein</Password>

</Credentials>

<PostJobResult Success="False">

<ErrorMessages>

<Error>Username or Password is incorrect</Error>

</ErrorMessages>

</PostJobResult>

</JobFeedApiResponse>

With an error message like this, you know the credentials are incorrect.

#### Our Example: Jobserve’s Account Verification

As Jobserve is an XML based API, we can use the method as described above to implement account verification. We can populate the elements which require account information, but omit everything else.

<?xml version="1.0" encoding="utf-8"?>

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Header>

<AccountNumber>999111</AccountNumber>

<AuthorisationCode>test111</AuthorisationCode>

<Source>xs</Source>

</SOAP:Header>

<SOAP:Body>

<PostAdvert>

<Adverts>

<Advert>

<MarketID />

<AccountNumber>999111</AccountNumber>

<!-- remove for brevity 🡪

</Advert>

</Adverts>

</PostAdvert>

</SOAP:Body>

</SOAP:Envelope>

When we send the above request to the Jobserve, we get the below returned. There are no issues with authentication, but there is with missing mandatory fields, which is exactly what we expected and what we wanted.

<SOAP:Envelope xmlns:SOAP = "urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Body>

<Summary>Number of Adverts Received : 1 Number successfully posted onto the website : 0</Summary>

<Results/>

<ValidationMessages>

<ValidationMessage>

<AdvertNumber>1</AdvertNumber>

<Message>Consultant Email was not specified and is mandatory with the schema being used.</Message>

</ValidationMessage>

</ValidationMessages>

</SOAP:Body>

</SOAP:Envelope>

### Jobserve’s Job Board Capabilities

This section explains how we conclude what the capabilities of Jobserve are, and how we populate the Channel attribute of the Channel class. The Channel attribute is discussed further in section 3.2.6.

|  |  |  |
| --- | --- | --- |
| Channel Attribute Property | Value | Explanation |
| PostDurationType | FixedDuration | Jobserve only allows us to specify how many whole weeks an advert will be visible for, so we cannot use an explicit date, therefore a fixed duration is the only option available for us. |
| UpdateDurationBehaviour | Extend | When we update a Jobserve posting we have to tell the job board how many more weeks we want the advert to be live for. We are therefore extending the life of the advert.  There are four different options for UpdateDurationBehaviour, which are as follows:   * NotUpdatable: Channel supports updates, but the expiry date itself cannot be updated. * NotApplicable: Channel does not support updates. * Extend: The advert which was posted is updated and the life of the advert on the job board is extended.   Replace: Generally for feeds whose post duration type is of Explicit Expiry Date. The advert which was posted originally is replaced with a new advert. |
| ExpiryDeleteBehaviour | Delete | We want the PE to delete the advert for us automatically when the posting is due to expiry.  There are two options for ExpiryDeleteBehaviour which are as follows:   * Delete: The PE should delete the job from the job board automatically before deleting the vacancy from its internal database. * NoAction: The PE should delete the vacancy from its internal database without any interaction with the job board itself. This option is preferred if the job board does not support deletes (most likely social media such as Twitter) or if the job board deletes the job posting themselves, and the Delete operation would result in a “job not found” type error.   In respect to Jobserve, a request to remove a job with a non-existent id does return an “Advert not valid” error, so we have to set this to “No Action”.  The best way to determine this at development time is to write a unit test in either test or production mode and try to delete an advert with an id of “xxx” or another nonsense identifier and see what happens. An example of a unit test can be seen in [section 9.3.1](#_Unit_Test_to). |
| RequiresAccount | True | The Jobserve feed requires credentials before we can post to it. |
| SupportsAccountVerification | True | We have already determined if Jobserve can support account verification. |
| SupportsGetUrl | False | The Jobserve specification gives us an example of a response here:    The response doesn’t supply us with a direct URL of the advert. However, this doesn’t necessarily mean we can’t support GetUrl.  Sometimes the Job ID returned can be appended to the end of the URL like so:  <http://www.jobserve.com/jobs/x0146000e85c81>  If we can access an advert by simply appending the Job ID to the end of the job board URL, then we should support GetUrl.  However, Jobserve URL’s are actually like this:  <http://www.jobserve.com/EpgOs>  We have no way of determining this from the job board response, so we have to conclude we cannot support GetUrl in this instance. |

## Mapping Job Board Fields to the PE

The PE will hold basic information about a vacancy such as (but not limited to):

* Vacancy Type (Permanent, Contract, or Temporary)
* Working Hours (Full time or part time)
* Minimum and maximum salary expressed as a double value.
* Job title
* Description
* Vacancy reference
* Application URL

This is an example of “common” vacancy data, because all vacancies which are advertised on the internet will always require this data.

But most job boards will usually have a requirement to provide us with a fixed list of options to choose from for particular fields that the PE isn’t aware of. This could be a list of industry sectors or job categories, or a list qualifications.

This type of data is known as “custom” vacancy data, and is data which is specific to the job board only.

Getting this information in advance will help us decide which bits of the job feed we can classify as common vacancy data, and which bits of the feed we can consider custom data.

### Mapping Salaries

Some job boards require us to send a unique identifier to represent a salary range. Let’s suppose the range of salaries is something like this:

|  |  |  |
| --- | --- | --- |
| **Salary Id** | **Min (per year)** | **Max (per year)** |
| 1 | 0k | 10k |
| 2 | 10k | 20k |
| 3 | 20k | 30k |
| 4 | 30k | 40k |
| 5 | 40k | 50k |
| 6 | 50k | 999,999 |

It’s reasonable to hard code these values as salaries do not change very often. We could represent one of these salary ranges in a class like so:

public class JobBoardSalaryValue

{

public int SalaryId { get; set; }

public double LowerRange { get; set; }

public double HigherRange { get; set; }

}

Then we could populate a list of salary ranges like so:

private List<JobBoardSalaryValue> salaryRanges;

salaryRanges = new List<JobBoardSalaryValue>();

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 1, LowerRange = 0, HigherRange = 10000 });

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 2, LowerRange = 10000, HigherRange = 20000 });

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 3, LowerRange = 20000, HigherRange = 30000 });

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 4, LowerRange = 30000, HigherRange = 40000 });

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 5, LowerRange = 40000, HigherRange = 50000 });

salaryRanges.Add(new JobBoardSalaryValue { SalaryId = 6, LowerRange = 50000, HigherRange = 999999 });

A simple LINQ query can pull out the relevant salary Id we need to send to the job board:

JobserveContent content = new JobserveContent();

content.Vacancy.MinSalary = 28000;

content.Vacancy.MaxSalary = 32000;

float minSalary = (float)content.Vacancy.MinSalary;

int selectedSalaryId = salaryRanges.Where(x => x.LowerRange <= minSalary && x.HigherRange >= minSalary).Select(r => r.SalaryId).FirstOrDefault();

Assert.AreEqual(3, selectedSalaryId);

If a boundary salary is chosen (such as £10,000) then the first lower range of the list will be returned.

### Mapping Vacancy Type

The PE supports three types of vacancies as defined by the VacancyType enumeration: Permanent, Temporary and Contract. What about job boards that support options in addition to this? It’s not uncommon for a job board to offer a vacancy type such as:

* Internment
* Apprenticeship

The PE does not natively support these vacancy types. In this case, we should treat each of these as custom vacancy options which would override any selection of vacancy types.

#### Our Example: Jobserve Vacancy Types



According to the Jobserve specification, we can either have Permanent, or Contract, or both. The PE understands that vacancies can natively be any of the following as defined by the Kaonix.PE.API.Channels.Vacancies.VacancyType enumeration:

* Perm (Displayed as “Permanent”)
* Temp (Displayed as “Temporary”)
* Contract (Displayed as “Contract”)

But it doesn’t understand how a vacancy can be both permanent and contract, and Jobserve doesn’t understand the notion of a temporary vacancy. In this case, we will write a method to control the mapping like so:

private char GetJobType()

{

if (customData.PermanentAndContract)

{

return JobserveContractType.Both;

}

switch (vacancyData.VacancyType)

{

case VacancyType.Perm:

return JobserveContractType.Permanent;

case VacancyType.Contract:

return JobserveContractType.Contract;

case VacancyType.Temp:

return JobserveContractType.Contract;

default:

return JobserveContractType.Both; // required to satisfy compiler so method has return statement for all possibilities

}

}

This example is repeated in section 5.5, but to explain, the JobserveContractType is a static class with methods returning a char representing the different types of contracts which Jobserve offer. If a PE vacancy is a temporary vacancy, we have no choice but to force it become a Jobserve contract vacancy as Jobserve doesn’t support temporary vacancies. The default in the switch statement is a fall back, it is unlikely to ever fall through to that point.

### Mapping Working Hours

The PE supports three types of working hours as defined by the WorkingHours enumeration: Full-Time, Part-Time or N/A.

#### Our Example: Jobserve Working Hours

Jobserve does have temporary and part-time option in their category extension list (see Appendix C). For a part-time vacancy, we’d have to generate another element in the XML which looks like this:

<Categories>Part Time</Categories>

We can write a simple method returning a string, if the string is empty then we know we don’t want to generate a <Categories> element:

private string GetCategory()

{

const string PartTime = "Part Time";

return vacancyData.WorkingHours == WorkingHours.PartTime ? PartTime : string.Empty;

}

When we need to generate the XML, all we need to in this case is test if the string returned from GetCategory is empty or not:

// Result of GetCategory() is assigned to the Category property

if (!string.IsNullOrEmpty(Category) && RequestType != JobserveRequestType.AmendAdvert)

{

xw.WriteElementString("Categories", Category);

}

## Ignore Web-cruit Considerations

Web-cruit is a third party system which will integrate with the PE feed once the feed has been written. However, we should not think about web-cruit whilst developing the feed. The PE is designed to be a stand-alone service.

For example, although the PE has three different types of vacancy, web-cruit in fact supports more types of vacancies (e.g. Maternity Leave, Holiday Cover, Apprenticeships). However, most job boards do not support these options, so we do not consider them for the PE. These extra vacancy types will be displayed on the web-cruit career site but we will have to map them differently to job boards. In most cases, Maternity Leave and Holiday Cover can be mapped as “Temporary” in the PE because that is essentially what they are, temporary vacancies.

So the rule of thumb is to forget about web-cruit until the PE feed itself is complete.

# Create the Basic Feed Components

This section will describe how to set up the bare minimum requirements of a feed so we can get to a stage where we can compile and test the feed.

## Basic Feed Classes

This section will outline the minimum classes we would expect in a job feed. The naming convention would typically be the job board name followed by the class name, so for example, a CustomData class for Jobserve would be called “JobserveCustomData”.

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| Vacancy | This class contains common fields which are expected to be found in most job boards, for example, we expect every job board to have a job title and a job description. Properties are overridden to tell the PE we expect them to be used within the feed, and we also set validation rules on these properties. The Vacancy class inherits from VacancyData. |
| AccountData | This class allows us to tell the PE how the job board credentials are stored in the feed. There is a default class of AccountData which can be used if the job board has a simple Username and Password setup. We can either inherit from this if we wish to extend beyond the Username and Password, or we can create our own Account class inherited from ChannelData. |
| CustomData | If a job board contains fields in the feed that the PE cannot provide in the Vacancy class, then we need to create a CustomData class and add properties to it. The CustomData class inherits from ChannelData. If a property of a class requires a specific set of values then we will need to create an additional class inherited from a RestrictionValueFactory or create an enum. |
| SettingsFactory | The settings factory is used to store configurable settings which may vary depending on the environment the feed is running in. There are three modes that a feed can be executed in (Debug, Testing, or Production). If the URL between the test site of the job board is different to that of the live site, then this is the class where the job board URL will be stored. This class is inherited from VacancySettingsChannelFactory or ChannelSettingsFactory. |
| Channel | Whilst the other classes previously mentioned are responsible for holding data that we will use to build a request, this class actually does something with the data and is responsible for sending and receiving the requests. This class requires all the previous classes to already be written before we can create a channel class.  A channel class inherits from VacancyChannel<TVacancy, TAccount, TCustomData, TOutputData>. It has up to four methods which will be overridden if the job board feed supports it:   * Post (all job feeds should at least support this) * Update * Repost (if a job board supports delete, then this should also be supported) * Delete (most job boards apart will support this apart from social media API’s)   Optionally, we can also override:   * VerifyAccount * GetUrl   Each channel class must be decorated with the [Channel] attribute as this tells the PE (and anyone consuming the PE web service) important information about the feed and what its capabilities are. |

There are a couple of other notable classes that we need to be aware of, although we don’t necessarily create them:

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| OutputData | If a job board returns a unique job id, then we should store it in OutputData.JobId. The PE will then use this as a reference when it comes to updating the same job later on. |
| PostProperties | This class stores the expiry date or the duration of the vacancy about to be posted to the job board, as well as OAuth tokens if the job feed requires us to use this method of authentication (currently only Twitter at the time of writing).  The PostProperties class contains a number of useful properties and methods to get the expiry date or duration:   * GetDurationInDays() * GetDurationInWeeks() – rounds up to the nearest number of whole weeks. * Duration (returns a PostDuration which includes Period and Units properties) * ExpiryDate (returns a nullable DateTime) |

## Create Basic Feed Classes

We will create the relevant classes in the Kaonix.PE.Channels project. You’ll need to create a folder for the now job feed.



### Create a Vacancy class

A vacancy class represents any common data that the job board requires. For example, every job board on the internet expects a job title and a description at the very least.

We will need to create a class to inherit from VacancyData and override some properties to let the PE know we intend to populate them so we can send to the job board. We will then apply attributes to the properties which will act as validation rules.

#### Further Reading on Common Vacancy Data

Please refer to sections 2.2 and 3 of PEChannelDeveloperGuide.aspx and slide 16 of the PE presentation slides.

#### JobserveVacancy class

We create JobserveVacancy which inherits from VacancyData and then override the virtual properties that we need by consulting the specification.

##### Override Vacancy Data Properties

Appendix A is an excerpt from the Jobserve specification, we can see that we need the following fields

|  |  |
| --- | --- |
| **Jobserve Field** | **VacancyData Property to override** |
| Position | Title |
| HtmlSkills | Description |
| PostZipCode | PostCode |
| Reference | VacancyRef |
| StartDate | StartDateText |
| Rate | SalaryDescription |
| Type | VacancyType |
| URL | ResponseUrl |
| ConsultantEmail | ResponseEmail |
| ConsultantFirstname | ContactFirstname |
| ConsultantSurname | ContactSurname |
| Telephone | ContactPhoneNumber |

##### Apply Attributes to VacancyData Properties

Next we need to apply attributes to each of the properties which will help the PE determine validation rules. If any data is sent to the PE which breaks these validation rules, then the PE will not bother attempting to send the request to the job board, but will instead return an error and the cause of it.

More information on what attributes to apply to these properties can be seen in the [PEChannelDeveloperGuide.docx](https://crsdev1.crs-warr.co.uk/svn/TNG.NET/trunk/PE/Docs/PEChannelDeveloperGuide.docx) in section 2.2.5.

##### JobserveVacancy.cs

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

public class JobserveVacancy : VacancyData

{

[Availability(Availability.Required)]

[DisplayAs("Reference")]

public override string VacancyRef { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Position")]

public override string Title { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("HTML Skills")]

[EditorHint(EditorType.HTMLEditor)]

[HTMLSupport("p,b,u,i,br")]

public override string Description { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Consultant Firstname")]

public override string ContactFirstName { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Consultant Surname")]

public override string ContactSurname { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Telephone")]

public override string ContactPhoneNumber { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Consultant Email")]

[RegExValidator(CommonFieldFormats.EmailAddress, "Consultant Email must be a valid e-mail address")]

public override string ResponseEmail { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Application URL")]

[RegExValidator(CommonFieldFormats.Url, "Application URL must be a valid URL")]

public override string ResponseUrl { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Type")]

public override VacancyType VacancyType { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Rate")]

public override string SalaryDescription { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Post Zip Code")]

[RegExValidator(CommonFieldFormats.UKPostCode, "Post Zip Code can only be a valid UK Post Code")]

public override string PostCode { // removed for brevity }

[Availability(Availability.Required)]

[DisplayAs("Start Date Description")]

public override string StartDateText { // removed for brevity }

}

}

### Create Custom Data Classes

Any field the job board expects that is specific to the feed will need to be contained in a custom data class. If any field expects a predefined list of values, such as selection of industry sectors to choose from, then this will involve at least one RestrictionValueFactory class. In terms of web-cruit these are similar to job media extensions.

The [PEChannelDeveloperGuide.docx](https://crsdev1.crs-warr.co.uk/svn/TNG.NET/trunk/PE/Docs/PEChannelDeveloperGuide.docx) has more information about this in section 3.5.

#### Custom Data Fields in the Jobserve Feed

In Appendix A, we can see the list of fields that the Jobserve field expects. When we look at the mandatory fields, we need to see if any of them are relevant to what we already have in the Vacancy class. There are a couple fields however, that have no relevance to what the Vacancy class already provides and these are candidates for fields to be included into the CustomData class. These are:

* MarketID (list of industry codes) – we’ll need to create a restriction value factory for this.
* ISO Country Code – we’ll need to create a string field so we can pass this value in when we send a post request.

#### Types of Restriction Value Factories

There are two types of factories we could consider to use.

##### RestrictionValueFactory

This is a class which inherits from RestrictionValueFactory which holds an array of values. This is the most commonly used type of restriction value factory. The downside is any changes made here, no matter how small, requires redeployment of the PE. If the values are unlikely change, use this method.

Please consult section 3.5.6 in [PEChannelDeveloperGuide.docx](https://crsdev1.crs-warr.co.uk/svn/TNG.NET/trunk/PE/Docs/PEChannelDeveloperGuide.docx)

##### ExternallyStoredRestrictionValueFactory

This is a class which inherits from ExternallyStoredXmlRestrictionValueFactory and instead of storing the values in an array, they are stored in an XML file. The disadvantage is that the PE will have to read from a file which adds a slight overhead to performance, but the advantage is that if any changes are made to values, you only need to deploy the XML file, and there is no need to deploy the entire PE. If the values may change frequently, use this method.

Please consult section 3.5.7 in [PEChannelDeveloperGuide.docx](https://crsdev1.crs-warr.co.uk/svn/TNG.NET/trunk/PE/Docs/PEChannelDeveloperGuide.docx)

Although there is no strict rule on which type of factory to favour, it would make sense to start off with a RestrictionValueFactory, and if we realise we have to make frequent changes to the factory, we can then migrate this to an externally stored factory instead to make adding and updating changes easier in the future. This was the case for feeds required by Pertemps and ESOS.

##### Jobserve’s Restriction Value Factories

Appendixes B and C shows us that Jobserve has a list of values for industries and a list for categories.

##### JobserveIndustryRestrictionValueFactory.cs

Firstly we are going to going to create a class to hold industry values. We need to create a class called JobserveIndustryRestrictionValueFactory which inherits from RestrictionValueFactory. Below is an example:

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

public class JobserveIndustryRestrictionValueFactory : RestrictionValueFactory

{

private static readonly RestrictionValue[] values;

static JobserveIndustryRestrictionValueFactory()

{

values = new[]

{

new RestrictionValue("Advertising / Media and Entertainment", "13"),

new RestrictionValue("Call Centre and Customer Services", "9"),

new RestrictionValue("Construction and Civil Engineering", "11"),

new RestrictionValue("Defence / Military and Armed Forces", "18"),

new RestrictionValue("Education", "20"),

new RestrictionValue("Engineering", "3"),

new RestrictionValue("Finance / Accounting and Banking", "2"),

new RestrictionValue("Food and Hospitality", "14"),

new RestrictionValue("Healthcare and Medical", "8"),

new RestrictionValue("HR and Recruitment", "10"),

new RestrictionValue("IT and Telecommunications", "1"),

new RestrictionValue("Legal", "7"),

new RestrictionValue("Manufacturing", "21"),

new RestrictionValue("Mining / Energy / Oil and Gas", "15"),

new RestrictionValue("Office and Administration", "6"),

new RestrictionValue("Retail and Consumer Products", "17"),

new RestrictionValue("Sales and Marketing", "4"),

new RestrictionValue("Science / Biotech and Pharmaceuticals", "16"),

new RestrictionValue("Trades and Services", "22"),

new RestrictionValue("Transport and Logistics", "12"),

new RestrictionValue("Travel and Tourism", "19")

};

}

public override RestrictionValue[] GetValues()

{

return values;

}

}

}

#### Custom Data Class

Once we have our RestrictionValueFactory classes, they need to wrapped within a CustomData class. This class encapsulates any data that is considered customised data to the job board.

To do this, we:

* Create a class which inherits from ChannelData
* Add publically accessible properties to the class
* Append attributes to each property

##### JobserveCustomData.cs

For Jobserve we have already determined that we have a RestrictionValueFactory for industry, and another for category.

Another check at the specification in Appendix A shows us that there is a mandatory field for:



ISO codes are publically available here <http://www.nationsonline.org/oneworld/country_code_list.htm> so creating a RestrictionValueFactory class is probably unnecessary. We will need to add a property to cater for this.

In total we have four properties in out JobserveCustomData class:

public class JobserveCustomData : ChannelData

{

[Availability(Availability.Required)]

[MaxLength(3, "The ISO Country Code must be 3 characters in length")]

[RegExValidator(CommonFieldFormats.IsoCountry3LetterCode, "The ISO Country Code is not valid and must be 3 characters long. Please see http://www.nationsonline.org/oneworld/country\_code\_list.htm for valid 3 letter codes.")]

[DisplayAs("ISO Country Code")]

[DisplayOrderId(1)]

public string IsoCountryCode { get; set; }

[Availability(Availability.Required)]

[Restictions(typeof(JobserveIndustryRestrictionValueFactory))]

[DisplayOrderId(2)]

[RegExValidator(@"\d{2}", "The Industry is not valid")]

public string Industry { get; set; }

[Availability(Availability.Optional)]

[DisplayOrderId(3)]

public bool PermanentAndContract { get; set; }

}

### Create the Account Data Class

The default account data class is AccountData which exposes a Username and a Password string properties. If the job board only requires a simple username and password combination, then you probably won’t need to create a new class to represent the job board credentials.

Some job boards require extra information such as an additional username for the job posting tool. In this case we may need to add more properties to the AccountData class to represent a third party integrator username, and possibly the password too. To do this, you could create a new account class to inherit from AccountData and then add additional properties.

If you do not wish to use a Username and Password structure for credentials then you can create your own account data class by creating a new class and inheriting from ChannelData.

#### JobserveAccountData.cs

The Jobserve specification tells us we need the following values for credentials:



Whilst this is pretty much the same as a username and a password, they have different names, so this would warrant a new class to be created and to inherit from ChannelData.

Note: In the Channel attribute, there are values for UsernameText and PasswordText. These are purely used to populate the correct fields in web-cruit’s JobMedia table, as web-cruit typically expects authentication to only consist of a username and password combination. Some job boards however, can have more complex

The simple rule is to remember that the DisplayAs attribute for an Account property is only going to be used for users who use the PE as an API, whereas the Channel attribute properties for UsernameText and PasswordText are used for web-cruit only.

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

public class JobserveAccountData : ChannelData

{

[Availability(Availability.Required)]

[DisplayAs("Account Number")]

[UsernameRef(true)]

public string AccountNumber { get; set; }

[Availability(Availability.Required)]

[DisplayAs("Authorisation Code")]

[PasswordRef(true)]

public string AuthorisationCode { get; set; }

}

}

### OutputData class

The OutputData class represents any data that is returned from the job board after a successful post. The PE has a ready-made class called JobIdOutputData which is ideal if the job returns a reference identifier. The PE will use this reference if the vacancy is updated in the future.

If the job board doesn’t return any meaningful data then there is a ready-made class of EmptyChannelData.

However, if the job board returns a lot of information that we might want to record, then we can create our own OutputData class and inherit it from ChannelData.

#### Jobserve’s Output Data

The Jobserve specification unhelpfully doesn’t give us an example of a response but looking at the web-cruit vacancy outbox can help. Here is an example of a response:

<SOAP:Envelope xmlns:SOAP = "urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Body>

<Summary>Number of Adverts Received : 1 Number successfully posted onto the website : 1</Summary>

<Results>

<Result>

<AdvertNumber>1</AdvertNumber>

<JobID>x015602e7200f</JobID>

</Result>

</Results>

</SOAP:Body>

</SOAP:Envelope>

The only useful piece of information here is the <JobId> element, which fits nicely into the JobIdOutputData class, so we do not need to create our own OutputData class here.

### Create the Settings Factory

The purpose of a settings factory is to store configurable settings which may be different when we are posting jobs in debug mode, testing mode or on a production server. The most common use is store different posting URL’s for a testing environment and a posting environment. We don’t need a posting URL for debug mode, but we’ll usually just bundle it in with the testing environment URL, even though in debug mode, we don’t actually post anything, we just return a string containing a mocked response from the job board.

#### JobserveSettingsFactory.cs

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

public class JobserveSettingsFactory : VacancyChannelSettingsFactory

{

private const string WebPostUrl = "WebPostUrl";

public JobserveSettingsFactory(ChannelExecutionMode executionMode)

: base(executionMode)

{

switch (executionMode)

{

case ChannelExecutionMode.Debug:

case ChannelExecutionMode.Test:

settings.Add(WebPostUrl, "http://193.119.59.225");

break;

case ChannelExecutionMode.Production:

settings.Add(WebPostUrl, "https://xml.jobserve.com");

break;

}

}

}

}

The settings factory class can then be accessed in the channel class via the host property as follows:

public override void Post()

{

string postUrl = Host.SettingsFactory[JobserveSettingsFactory.WebPostUrl];

…

}

### Create the Channel Class

The channel class inherits from VacancyChannel and has four generic type parameters for the vacancy data (common data), account data, custom data and output.

Because of the required generic type parameters, we were unable to create this class until now.

The class requires:

* A constructor to take an IChannelHost and PostProperties as parameters
* A channel attribute
* An override of the Post method, Repost method and Delete method.
* An override of the Update method may be required if the job board supports updates.

#### JobserveChannel.cs Declaration

An example of the JobserveChannel class is below, with the required constructor:

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

public class JobserveChannel : VacancyChannel<JobserveVacancy, JobserveAccountData, JobserveCustomData, JobIdOutputData>

{

public JobserveChannel(IChannelHost host, PostProperties postProperties)

: base(host, postProperties)

{

}

}

}

#### Applying the Channel Attribute

The channel attribute must be applied to the class and its purpose is to provide meta-data to the PE which will describe what the job board capabilities are. In terms of web-cruit, we must carefully choose what to use for the “ExternalSystemRef” value, typically choose one that hasn’t already been taken in web-cruit’s JobMedia database table. As a convention, PE job feeds in web-cruit begin at 1000 so they are easily distinguishable from ColdFusion based web feeds.

##### Channel Attribute for JobServe

[Channel("{9E4BF16E-40CB-45DB-973A-AEC50D178DE6}",

ChannelVisibility.Public,

"www.jobserve.com",

"Jobserve",

PostDurationType.FixedDuration,

UpdateDurationBehaviour.Extend,

ExpiryDeleteBehaviour.NoAction,

PostMechanism.XmlHttp,

ContactEmail = "customer.care@jobserve.com",

ContactName = "Jobserve Support",

ContactPhone = "+44 (0)1621 817335",

Description = "In 2010, JobServe published 200,000 jobs and etc",

ExternalSystemRef = "1072",

RequiresAccount = true,

UsernameText = "Account Number",

PasswordText = "Authorisation Code",

SupportsAccountVerification = true,

SupportsGetUrl = false,

SupportsDebugMode = true,

SupportsTestMode = true,

SupportsProductionMode = true,

SupportsPosting = true,

SupportsUpdating = true,

SupportsReposting = true,

SupportsDeleting = true,

SettingsFactoryType = typeof(JobserveSettingsFactory))]

#### Override Posting Methods

Now we need to override any methods that the job board will implement. These methods are indicative of what the job board capabilities are and can include any of the following:

* Post
* Update
* Report
* Delete

##### Our Example:Jobserve Posting Methods

For now, we’ll just put NotImplementedException placeholders in here for now, as we want to create some unit tests next so we can utilise [test driven development](http://agiledata.org/essays/tdd.html).

public override void Post()

{

throw new **NotImplementedException**();

}

public override void Update()

{

throw new **NotImplementedException**();

}

public override void Repost()

{

throw new **NotImplementedException**();

}

public override void Delete()

{

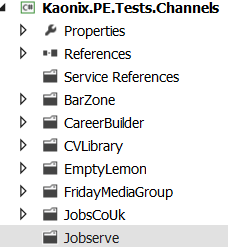
throw new **NotImplementedException**();

}

# Create Test Classes

We’ve now reached a stage where we have the basic structure of our feed. We have enough to create our unit test classes.

Firstly we need to create a folder in the Kaonix.PE.Tests.Channels project like so:



## Create a Content Class

The purpose of a content class is provide test data to our unit tests. We need to create a class to inherit from ChannelContent<TVacancy, TAccountData, TCustomData, TOutputData> and then we’ll have to populate the vacancy, account data, custom data and output data classes in an overridden method called InitializeTestData.

### JobserveContent.cs

using System;

using System.Text;

namespace Kaonix.PE.Tests.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

using Kaonix.PE.Channels.Jobserve;

public class JobserveContent : ChannelContent<JobserveVacancy, JobserveAccountData, JobserveCustomData, JobIdOutputData>

{

public override void InitializeTestData()

{

/\*

\* using StringBuilder is a great way to see how the job description is formatted on the job board

\* we can see if line breaks work and we can see if special characters such as pound signs, ampersands are formatted correctly

\* if they are not, we may need to encode them

\*/

StringBuilder jobDesc = new StringBuilder();

jobDesc.AppendLine("This is a test vacancy only.");

jobDesc.AppendLine("Please do not apply.");

jobDesc.AppendLine("Some special characters.");

jobDesc.AppendLine("£ ! $ % ^ & \* ( ) { } @ ' : ; ~ # < > , . / ? \"");

jobDesc.AppendFormat("Posted at: {0}", string.Format("{0:d MMM yyyy HH:mm:ss}", DateTime.Now));

/\* Even though we have a CustomData property for handling the special case of post durations for Jobserve,

\* the PE requires the PostProperties to be populated also.

\*

\*/

PostProperties.Duration = new PostDuration { Period = 2, Units = PostDurationUnits.Weeks };

AccountData.AccountNumber = "999111";

AccountData.AuthorisationCode = "test111";

Vacancy.VacancyType = VacancyType.Perm;

Vacancy.WorkingHours = WorkingHours.FullTime;

Vacancy.PayFrequency = PayFrequency.Yearly;

Vacancy.CompanyName = "Kaonix Test";

Vacancy.ContactFirstName = "Andy";

Vacancy.ContactSurname = "Hudson";

Vacancy.ContactPhoneNumber = "01257 123456";

Vacancy.Description = jobDesc.ToString();

Vacancy.StartDateText = "ASAP";

Vacancy.Title = "C# Developer";

Vacancy.PostCode = "PR7 4NA";

Vacancy.SalaryDescription = "Excellent";

Vacancy.VacancyRef = "KAONIX-1";

Vacancy.ResponseEmail = "jobs@kaonix.com";

Vacancy.ResponseUrl = "http://www.kaonix.com/jobs/123";

Vacancy.Id = StringUtils.NewID(); // returns a GUID

CustomData.Category = "Graduate";

CustomData.Industry = 1; // IT

CustomData.IsoCountryCode = "GBR";

CustomData.NumberOfWeeksDuration = JobservePostingDuration.TwoWeeks;

}

}

}

## Create a Debug Mode Unit Test

All unit test classes inherit from ChannelTestBase with six generic type parameters consisting of <TChannel, TContent, TVacancyData, TAccountData, TCustomData, TOutputData>.

ChannelTestBase includes an abstract property of ExecutionMode which we will be asked to override, and as this is a debug unit test class, we will just have to apply the appropriate ChannelExecutionMode.Debug enumeration.

Adding our first unit test is straight forward. We use the utility method CreateChannel and pass in the DefaultContent property, which is a reference to our content class we create just before.

namespace Kaonix.PE.Tests.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

using Kaonix.PE.Channels.Jobserve;

using NUnit.Framework;

[TestFixture]

public class TestDebugJobserveChannel : ChannelTestBase<JobserveChannel, JobserveContent, JobserveVacancy, JobserveAccountData, JobserveCustomData, JobIdOutputData>

{

[Test]

public void TestPost()

{

IChannel channel = this.CreateChannel(DefaultContent);

channel.Post();

}

protected override ChannelExecutionMode ExecutionMode

{

get { return ChannelExecutionMode.Debug; }

}

}

Note the [TestFixture] and [Test] attributes, these are part of the Nunit testing framework and we’ll have to add the namespace for that.

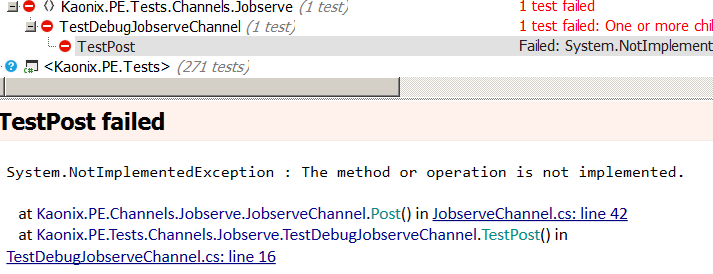
### Running our First Test

Let’s run the TestPost method, and it will fail because it currently returns nothing but a NotImplementedException.

It is doesn’t get that far, it will alert us to other issues with the feed. Such errors might appear as:

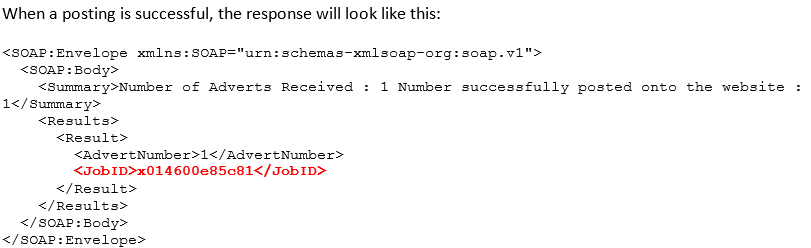
Error 'ResponseEmail is required but was null.' occurred validating property ResponseEmail. Error 'ResponseUrl is required but was null.' occurred validating property ResponseUrl. Error 'Id is required but was null.' occurred validating property Id.

However, at this stage we expect to get this in the unit test output:



### Fixing our First Unit Test

We only want out first unit test to return a string faking a response from the job board. In terms of Jobserve, their specification helpfully gives us an example:



Therefore, we can modify our JobserveChannel.cs file to return this if the channel is running in debug mode:

public override void Post()

{

if (Host.ExecutionMode == ChannelExecutionMode.Debug)

{

string response = this.GetDebugResponse();

Host.Trace(response);

}

else

{

throw new **NotImplementedException**();

}

}

/// <summary>

/// Create a faked response from jobserve

/// </summary>

/// <returns></returns>

private string GetDebugResponse()

{

StringBuilder resp = new StringBuilder();

resp.AppendLine("<SOAP:Envelope xmlns:SOAP=\"urn:schemas-xmlsoap-org:soap.v1\">");

resp.AppendLine("<SOAP:Body>");

resp.AppendLine("<Summary>Number of Adverts Received : 1 Number successfully posted onto the website : 1</Summary>");

resp.AppendLine("<Results>");

resp.AppendLine("<Result>");

resp.AppendLine("<AdvertNumber>1</AdvertNumber>");

resp.AppendLine("<JobID>x014600e85c81</JobID>");

resp.AppendLine("</Result>");

resp.AppendLine("</Results>");

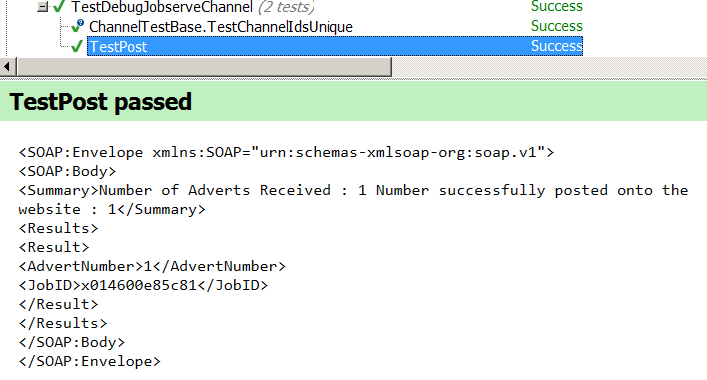
resp.AppendLine("</SOAP:Body>");

resp.AppendLine("</SOAP:Envelope>");

return resp.ToString();

}

Now because we are returning a string and not throwing an exception, the unit test will work.



### Parsing the Response

However, simple getting the response back from Jobserve isn’t very useful. We will need to parse the XML here and pass the JobId to the OutputData class.

We can add a method to the JobserveChannel.cs file which will be responsible for parsing the job Id by using Linq to XML:

private void ExtractJobIdFromResponse(string response)

{

XDocument doc = XDocument.Parse(response);

var resultNode = doc.Descendants().FirstOrDefault(e => e.Name.LocalName == "Result");

if (resultNode != null)

{

string advertNumberValue =

resultNode.Descendants()

.Where(e => e.Name.LocalName == "AdvertNumber")

.Select(e => e.Value)

.FirstOrDefault();

string jobId =

resultNode.Descendants()

.Where(e => e.Name.LocalName == "JobID")

.Select(e => e.Value)

.FirstOrDefault();

int numberAdvertsPosted;

bool parsedNumberAdvertsPosted = int.TryParse(advertNumberValue, out numberAdvertsPosted);

if (!parsedNumberAdvertsPosted || numberAdvertsPosted == 0)

{

throw new RemoteChannelException(string.Format("Unable to parse JobID from response: {0}", response));

}

OutputData.JobId = jobId;

}

}

If we cannot find the JobId, then this means the response is not in the format we expected. We have to return an exception. The RemoteChannelException is the most general-type of exception a PE feed should throw.

And then we change our Post method to ensure we use this method:

public override void Post()

{

if (Host.ExecutionMode == ChannelExecutionMode.Debug)

{

string response = this.GetDebugResponse();

Host.Trace(response); // ensure this is placed before we attempt to extract the job id, so we have record of the response in event of an exception being thrown

this.ExtractJobIdFromResponse(response);

}

else

{

throw new **NotImplementedException**();

}

}

When we run our test, it should be successful, and we should be able to parse a response successfully. This is a great start.

### A Working Unit Test

When we run our test, it should be successful, and we should be able to parse a response successfully. This is a great start.

## Create Test Mode Unit Test Class

We now need to create a unit test class for sending posts in test mode. Instead of returning a faked response, this will help us test sending posts to the Jobserve staging servers. We haven’t implemented any code yet to send anything to their server, so this will have to be written soon.

Creating a unit test class is the same as specified in this document in section 5.2 except in the execution mode property will return ChannelExecutionMode.Test;

namespace Kaonix.PE.Tests.Channels.Jobserve

{

using System;

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

using Kaonix.PE.Channels.Jobserve;

using NUnit.Framework;

[TestFixture]

public class TestJobserveChannel : ChannelTestBase<JobserveChannel, JobserveContent, JobserveVacancy, JobserveAccountData, JobserveCustomData, JobIdOutputData>

{

[Test]

public void TestPost()

{

throw new **NotImplementedException**();

}

protected override API.Channels.ChannelExecutionMode ExecutionMode

{

get { return ChannelExecutionMode.Test; }

}

}

}

## Create a Production Mode Unit Test Class

We might as well create one final class for production mode, which will test the feed against the live Jobserve feed.

Creating a unit test class is the same as specified in this document in section 5.2 except in the execution mode property will return ChannelExecutionMode.Production.

This should be used with caution as we could well be using live credentials. It is quite common to mark most, if not all, of the methods with NUnit’s [Ignore] attribute, so that running the unit test against the [TestFixture] attribute does not accidentally post jobs to the Jobserve site unintentionally and thus using credits.

using System;

namespace Kaonix.PE.Tests.Channels.Jobserve

{

using Kaonix.PE.API.Channels;

using Kaonix.PE.API.Channels.Vacancies;

using Kaonix.PE.Channels.Jobserve;

using NUnit.Framework;

[TestFixture]

public class TestProductionJobserveChannel : ChannelTestBase<JobserveChannel, JobserveContent, JobserveVacancy, JobserveAccountData, JobserveCustomData, JobIdOutputData>

{

[Test]

[Ignore]

public void TestPost()

{

throw new **NotImplementedException**();

}

protected override ChannelExecutionMode ExecutionMode

{

get { return ChannelExecutionMode.Production; }

}

}

}

# Building the Request

The next stage is to build some classes which will closely represent the fields or elements of a request we will send to the job board. The purpose of these classes are to provide a façade between the PE objects of Vacancy, CustomData and AccountData and to create one object we can use to generate the request.

Below is a rough approximation of what the class hierarchy will look like in order to build the request for the any given channel.

abstract RequestBase

+ RequestType

+ Username

+ Password

+ JobTitle

+ VacancyRef

+ Description

+ ApplyUrl

+ JobId

+ string GenerateXml()

abstract UpdatableRequest

PostRequest

Override RequestType Post

UpdateJobRequest

Override RequestType Update

DeleteJobRequest

Override RequestType Delete

RepostJobRequest

Override RequestType Repost

To start off, we’d create a new sub-folder in the job feed folder so these classes are kept separate from the main code base.



## Request Types

Now we will create an enumeration to represent the different types of requests there are:

RequestType.cs

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public enum RequestType

{

PostAdvert,

ReAdvertise,

AmendAdvert,

DeleteAdvert,

VerifyAccount

}

}

## Base Class

### RequestBase

Most of the properties of the job feed will be here. The class is abstract because we don’t want to create an instance of this, and there is a RequestType property which will both be overridden in subclasses, and a GenerateRequest() which we will implement later. We have a couple of constructors which be used in base classes.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

using System;

public abstract class RequestBase

{

public string Username { get; set; }

public string Password { get; set; }

public string JobTitle { get; set; }

public string VacancyRef { get; set; }

public string Description { get; set; }

public string ApplyUrl { get; set; }

public string JobId { get; set; }

public string GenerateRequest()

{

throw new **NotImplementedException**();

}

public abstract RequestType RequestType { get; }

/// <summary>

/// Constructor for Posting a new vacancy

/// </summary>

protected RequestBase(JobserveVacancy vacancyData, AccountData accountData, JobserveCustomData customData)

{

this.Username = accountData.Username;

this.Password = accountData.Password;

this.JobTitle = vacancyData.Title;

this.VacancyRef = vacancyData.Reference;

this.Description = vacancyData.Description;

this.ApplyUrl = vacancyData.ResponseUrl;

}

/// <summary>

/// Constructor for Updating an existing vacancy

/// </summary>

/// <param name="jobId"></param>

protected RequestBase(string jobId)

{

this.JobId = jobId;

}

}

### UpdatableRequest

The updatable request is for updating or reposting a vacancy where we need to resend all of the vacancy details. This request class will allow us to replicate the base constructor with the addition of a job Id.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public abstract class UpdatableRequest : RequestBase

{

/// <summary>

/// Constructor for updating or reposting

/// </summary>

protected UpdatableRequest(string jobId, VacancyData vacancyData, AccountData accountData, JobserveCustomData customData)

: base(vacancyData, accountData, customData)

{

this.JobId = jobId;

}

}

}

## Concrete Classes

Now we need to create concrete classes for each type of request. They will all consist of a constructor and an overridden RequestType property.

### PostRequest

A post request will have a constructor and override the RequestType property.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public class PostRequest : RequestBase

{

public override RequestType RequestType

{

get { return RequestType.PostAdvert; }

}

public PostRequest(VacancyData vacancyData, AccountData accountData, JobserveCustomData customData)

: base(vacancyData, accountData, customData)

{

}

}

}

### UpdateRequest

This will inherit from UpdatableRequest so we have access to the constructor with a job Id parameter.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public class UpdateRequest : UpdatableRequest

{

public override RequestType RequestType

{

get { return RequestType.AmendAdvert; }

}

public UpdateRequest(string jobId, VacancyData vacancyData, AccountData accountData, JobserveCustomData customData)

: base(jobId, vacancyData, accountData, customData)

{

}

}

}

### RepostRequest

This will inherit from UpdatableRequest so we have access to the constructor with a job Id parameter.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public class RepostRequest : UpdatableRequest

{

public override RequestType RequestType

{

get { return RequestType.ReAdvertise; }

}

public RepostRequest(string jobId, VacancyData vacancyData, AccountData accountData, JobserveCustomData customData)

: base(jobId, vacancyData, accountData, customData)

{

}

}

}

### DeleteRequest

A delete request usually needs the job Id, so we won’t inherit from the UpdatableRequest class as we are not interested in populating the other properties apart from JobId here. We’ll inherit from RequestBase which has the constructor we need.

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public class DeleteRequest : RequestBase

{

public override RequestType RequestType

{

get { return RequestType.DeleteAdvert; }

}

public DeleteRequest(string jobId)

: base(jobId)

{

}

}

}

### VerifyAccount Request

If we have determined that the feed can support account verification, then we will also need to create a class to represent this request. The difference here is that we’ll only accept the credentials in the constructor, and in the chained constructor to the base class, any required parameters will be defaulted to an empty string (or a ‘0’ if the type is an int or float) and thus invalidating any mandatory fields.

A verify account request could look something like this:

namespace Kaonix.PE.Channels.Jobserve.RequestBuild

{

public class VerifyAccountRequest : RequestBase

{

public VerifyAccountRequest(string username, string password)

: base(username, password, string.Empty, string.Empty, string.Empty, string.Empty)

{

}

public override RequestType RequestType

{

get { return RequestType.VerifyAccount; }

}

}

}

## Generating the Request

Now we need to implement the request generation method. What goes in here really depends on what format the job feed is expecting the data. Refer to section of 2.3 of this document to decide.

### Implementing the GenerateRequest method

Let’s suppose our job feed wants us to produce some XML which looks like this for a post request:

<?xml version="1.0" encoding="utf-8"?>

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Header>

<UserName>myusername</UserName>

<Password>letmein</Password>

</SOAP:Header>

<SOAP:Body>

<PostAdvert>

<JobRef>ABC123</JobRef>

<Title>Manager</Title>

<Description><![CDATA[This is a description!]]></Description>

<ApplicationUrl><![CDATA[http://www.kaonix.com/jobs/123]]></ApplicationUrl>

</PostAdvert>

</SOAP:Body>

</SOAP:Envelope>

Or this for a repost request (note the <PostAdvert> element has been renamed to <ReAdvertise>)

<?xml version="1.0" encoding="utf-8"?>

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Header>

<UserName>myusername</UserName>

<Password>letmein</Password>

</SOAP:Header>

<SOAP:Body>

<ReAdvertise>

<JobId>AA1</JobId>

<JobRef>ABC123</JobRef>

<Title>Manager</Title>

<Description><![CDATA[This is an updated description!]]></Description>

<ApplicationUrl><![CDATA[http://www.kaonix.com/jobs/123]]></ApplicationUrl>

</ReAdvertise>

</SOAP:Body>

</SOAP:Envelope>

Or this for a delete request

<?xml version="1.0" encoding="utf-8"?>

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">

<SOAP:Header>

<UserName />

<Password />

</SOAP:Header>

<SOAP:Body>

<DeleteAdvert>

<JobId>AA1</JobId>

</DeleteAdvert>

</SOAP:Body>

</SOAP:Envelope>

In each of these requests, you should at least be able to see some properties which are in the RequestBase class. These are Title, Description and ApplicationUrl for example. Some elements also have CDATA sections to allow us to place special characters within the content.

This document is not a tutorial on the C# language, and doesn’t recommend one technique over another, but for this example, the XmlWriter class from the System.Xml assembly was used to generate the XML relatively easily.

We use the RequestType property to determine whether we want to display certain elements or not in the XML.

Here is how the GenerateRequest method now looks in our RequestBase class:

public string GenerateRequest()

{

string request;

const string SchemaUrn = "urn:schemas-xmlsoap-org:soap.v1";

string xmlStr;

XmlWriterSettings settings = new XmlWriterSettings { Indent = true };

using (MemoryStream ms = new MemoryStream())

{

using (XmlWriter xw = XmlWriter.Create(ms, settings))

{

xw.WriteStartDocument();

xw.WriteStartElement("SOAP", "Envelope", SchemaUrn); // start envelope

xw.WriteStartElement("SOAP", "Header", SchemaUrn); // start header

this.GenerateCredentialsSection(xw);

xw.WriteEndElement(); // end header

xw.WriteStartElement("SOAP", "Body", SchemaUrn); // start body

xw.WriteStartElement(RequestType.ToString()); // start [action]advert

// JobId is included in every request except post requests

if (RequestType != RequestType.PostAdvert)

{

xw.WriteElementString("JobId", JobId);

}

if (RequestType != RequestType.DeleteAdvert)

{

xw.WriteElementString("JobRef", VacancyRef);

xw.WriteElementString("Title", JobTitle);

xw.WriteStartElement("Description");

xw.WriteCData(Description);

xw.WriteEndElement();

xw.WriteStartElement("ApplicationUrl");

xw.WriteCData(ApplyUrl);

xw.WriteEndElement();

}

xw.WriteEndElement(); // end [action]advert

xw.WriteEndElement(); // end body

xw.WriteEndElement(); // end envelope

xw.WriteEndDocument();

}

ms.Flush();

ms.Position = 0;

// XML files are generally UTF-8 encoding whereas streams are UTF-16 by default so we need to tell the stream explicitly which encoding we want.

using (StreamReader sr = new StreamReader(ms, Encoding.UTF8))

{

request = sr.ReadToEnd();

}

}

return request;

}

/// <summary>

/// Generate an Xml section with credentials

/// </summary>

/// <param name="xw"></param>

private void GenerateCredentialsSection(XmlWriter xw)

{

xw.WriteElementString("UserName", Username);

xw.WriteElementString("Password", Password);

}

## Building a Request Factory and Handling Mapping Logic

Now we have all our request objects, but they don’t have anything to do with the PE classes that actually contain the information we need to send to the job board.

This is where the request factory comes in. We’re going to create a class which accepts instances of VacancyData, CustomData, and an AccountData. There will be a method for each type of request we want to make.

This class will also contain any mapping logic between the PE and the job board. For example, let’s suppose we decided to create a static class to hold the job types Jobserve require:

namespace Kaonix.PE.Channels.Jobserve.RequestBuilder

{

/// <summary>

/// P = Permanent, C = Contract, B = Both

/// </summary>

public static class JobserveContractType

{

public static char Both

{

get

{

return ‘B’;

}

}

public static char Contract

{

get

{

return ‘C’

}

}

public static char Permanent

{

get

{

return ‘P’;

}

}

}

}

We then need a method to map the PE vacancy types, and this can be done like so:

/// <summary>

/// Determine what the job type should be

/// </summary>

/// <returns></returns>

private char GetJobType()

{

if (customData.PermanentAndContract)

{

return JobserveContractType.Both;

}

switch (vacancyData.VacancyType)

{

case VacancyType.Perm:

return JobserveContractType.Permanent;

case VacancyType.Contract:

return JobserveContractType.Contract;

case VacancyType.Temp:

return JobserveContractType.Contract;

default:

return JobserveContractType.Both; // required to satisfy compiler so method has return statement for all possibilities

}

}

Also, Jobserve requires us to create an extra XML element if the job is for a part time position. We can handle this with a method like so:

private string GetCategory()

{

const string PartTime = "Part Time";

return vacancyData.WorkingHours == WorkingHours.PartTime ? PartTime : string.Empty;

}

The request factory should hold four private properties for VacancyData, CustomData and AccountData, and also another to hold the job id if we are to update, repost or delete a job:

private readonly JobserveVacancy vacancyData;

private readonly JobserveAccountData accountData;

private readonly JobserveCustomData customData;

private readonly string jobId;

Now we need the necessary constructors:

// Constructor for new jobs

public JobserveRequestFactory(

JobserveVacancy vacancyData, JobserveAccountData accountData, JobserveCustomData customData)

{

this.vacancyData = vacancyData;

this.accountData = accountData;

this.customData = customData;

}

// Constructor for update / repost / delete

public JobserveRequestFactory(

JobserveVacancy vacancyData, JobserveAccountData accountData, JobserveCustomData customData, string jobId) : this(vacancyData, accountData, customData)

{

this.jobId = jobId;

}

Now for the methods which will create a request object and return an instance of it. Note the use of the GetJobType() and GetCategory() methods (the latter used to generate a “part-time” XML element) discussed earlier.

/// <summary>

/// Create a request of <see cref="JobservePostRequest"/>

/// </summary>

/// <returns></returns>

public JobservePostRequest CreatePostRequest()

{

var postRequest = new JobservePostRequest(

vacancyData,

customData,

accountData,

this.GetCategory(),

this.GetJobType(),

DefaultSource,

this.GetDuration(),

customData.NumberOfWeeksDuration);

return postRequest;

}

Creating a method to return an instance of a verify account request could look something like this:

public JobserveVerifyAccountRequest CreateVerifyAccountRequest()

{

var verifyAccountRequest = new JobserveVerifyAccountRequest(accountData, DefaultSource);

return verifyAccountRequest;

}

# Implementing the HTTP Poster

This section is applicable to most job feeds where we wish to control how the HTTP requests are handled. There are exceptions to the rule however, if a particular job feed uses a service reference. Madgex and Strike Jobs are the only feeds which use service references (this is where we let Visual Studio generate proxy classes for us based on a provided WSDL).

Information on consuming services:

<http://www.c-sharpcorner.com/uploadfile/0c1bb2/consuming-web-service-in-asp-net-web-application/>

<https://msdn.microsoft.com/en-us/library/bb628649.aspx>

Now we have some XML being generated, we can think about how to send this to the job board. In C# there are many ways to construct a HTTP request, but WebClient is the simplest class for us to use. If you want to know why, have a look at <http://www.diogonunes.com/blog/webclient-vs-httpclient-vs-httpwebrequest>.

However, the default implementation of WebClient only allows 2 connections at once. But, in the Kaonix.PE.API project, there is an overridden version called ServerWebClient which has overridden the connection limit.

For any HTTP operations, using the ServerWebClient class should serve all our needs.

## Create a Response Class

Firstly we want somewhere to store our response from the job board. The response could be XML or JSON but for our immediate purposes, holding the response in a string will suffice. Also, a Boolean flag to determine if a request was successful or not will be helpful.

namespace Kaonix.PE.Channels.Jobserve.RequestBuilder

{

public class JobserveResponse

{

public bool IsSuccess { get; private set; }

public string ResponseMessage { get; private set; }

public JobserveResponse(bool isSuccess, string responseMessage)

{

this.IsSuccess = isSuccess;

this.ResponseMessage = responseMessage;

}

}

}

## Interface for the HTTP Poster

We want to create an interface that will abstract away the implementation of sending a request and receiving a response. This will help us is mocking up tests later. We will want to pass through the URL of the job feed, the request type enumeration and the request as a string.

namespace Kaonix.PE.Channels.Jobserve

{

using Kaonix.PE.Channels.Jobserve.RequestBuilder;

public interface IJobserveWebClient

{

JobserveResponse SendRequest(string url, JobserveRequestType requestType, string request);

}

}

## Implement the HTTP Poster

Now we want a concrete representation of the HTTP poster which will implement the interface in the earlier section. This class only deals with posting. Checking the response for errors is done elsewhere for easy testability. The example below uses ServerWebClient and shows how to add a HTTP header and to post a response.

When we get the response, we’ll use a simple string comparison to check the tell-tale signs of an error, we then know immediately there is an error. If there is, we can set our failed flag to true.

If there was a problem, we can check the HTTP status code by looking at the web exception status code to see if there was a problem connecting to the job board and return a ChannelDownException.

We’ll also capture the response, and if an exception was thrown, we’ll have to extract the response via a stream.

At the end we’ll create a new instance of the response class we created in the last section.

namespace Kaonix.PE.Channels.Jobserve

{

using System.IO;

using System.Net;

using Kaonix.PE.API;

using Kaonix.PE.Channels.Jobserve.RequestBuilder;

public class JobserveWebClient : IJobserveWebClient

{

#region IJobserveWebClient Members

public JobserveResponse SendRequest(string url, JobserveRequestType requestType, string request)

{

const string SoapMethodNameHeader = "SOAPMethodName";

string resp = string.Empty;

bool failed = false; // flag to tell us if an error was found or not

using (ServerWebClient client = new ServerWebClient())

{

client.Proxy = null;

client.Headers.Add(SoapMethodNameHeader, requestType.ToString());

try

{

resp = client.UploadString(url, request);

// look for key words in the response. some job boards may have a <Success>true</Success> type field in which case we'd use Linq-to-XML

if (resp.Contains("Error") || resp.Contains("ValidationMessages"))

{

failed = true;

}

}

catch (WebException webException)

{

failed = true;

resp = webException.Message; // fall back in case webResponse is null

// check if connection failed then return ChannelDownException

if (webException.Status == WebExceptionStatus.ConnectFailure || webException.Status == WebExceptionStatus.NameResolutionFailure)

{

throw new ChannelDownException("Jobserve is not available.", webException);

}

HttpWebResponse webResponse = (HttpWebResponse)webException.Response;

if (webResponse != null)

{

using (StreamReader sr = new StreamReader(webResponse.GetResponseStream()))

{

resp = sr.ReadToEnd();

}

}

}

}

return new JobserveResponse(!failed, resp);

}

#endregion

}

}

# Inspecting the Response

If there are any errors returned by the job board, or if there is no response from job board then the convention in the PE is to throw an exception.

There are 6 different exceptions, the most general purpose exception being RemoteChannelException. A full list can be seen in section 3.8 of [PEChannelDeveloperGuide.docx](https://crsdev1.crs-warr.co.uk/svn/TNG.NET/trunk/PE/Docs/PEChannelDeveloperGuide.docx).

However, we should throw the appropriate exception where possible. Not all job specification provide a list of errors they will return, so we will have to try and determine this ourselves in testing.

The most important thing to do is to throw an exception if we detect an error, even if we don’t know what the error is. In section 6.3 we implemented a concrete HTTP poster which would flag an error if “Error” or “Validation Message” appeared anywhere in the response. If an exception was caught from ServerWebClient, then we also flagged an error in this event too.

Messages such as “Invalid username” should throw the LogonErrorException, whereas “Invalid job type” should throw InvalidContentException.

## Create a Response Parser

Create a new class and ensure it inherits from PostingEngineResponseParser. It should contain a private property of a response class we created in section 7.1 and an appropriate constructor.

namespace Kaonix.PE.Channels.Jobserve.RequestBuilder

{

public class JobserveResponseParser : PostingEngineResponseParser

{

private readonly JobserveResponse response;

public JobserveResponseParser(JobserveResponse response)

{

this.response = response;

}

}

}

The PostingEngineResponseParser class already has a dictionary so all we need to do is add what the job board might return as an error as the key, and an enumeration representing the PE exception type which will be thrown in such an event. This will be done in an overridden InitialisePossibleErrors method.

protected override void InitialisePossibleErrors()

{

possibleErrors.Add("Please check your AuthorisationCode and AccountNumber.", PostingEngineExceptionTypes.LogonErrorException);

possibleErrors.Add("At least one of Email, Apply Online Email or Apply Online URL (product required) must be filled in", PostingEngineExceptionTypes.InvalidContentException);

possibleErrors.Add("is a mandatory field", PostingEngineExceptionTypes.InvalidContentException);

// removed for berevity

possibleErrors.Add("not have any credits to purchase this advert", PostingEngineExceptionTypes.RemoteChannelException);

}

We then need to override another method which will be responsible for searching and the response. This method is called FindAnyErrorsAndThrow and we need to pass in a flag to state whether a response has an error inside it and what the response from the job feed was as a string. The job board response class has these properties which is now a property of the JobserveResponseParser class.

public override void FindAnyErrorsAndThrow()

{

this.InspectResponseForErrors(response.IsSuccess, response.ResponseMessage);

}

# Implementing the Channel Class

We’ve implemented all our code now, now we just need it to be plumbed together. This will be done in the channel class, and because all our implementation has been done elsewhere, the channel class is essentially just a wrapper with not a lot of code inside it which looks clean and tidy.

We already have the ExtractJobIdFromResponse and GetDebugResponse methods in there.

## Adding Properties to the Channel Class

We are going to add two public properties to hold the generated request and the response we get back from the job board. This is purely for debugging and diagnostics. Then we’ll add a private property to hold an instance of the interface abstraction of the HTTP poster.

public string JobserveRequest { get; private set; }

public string JobserveResponse { get; private set; }

private readonly IJobserveWebClient webClient;

### Adding an additional Constructor

When the PE is running, it requires an IChannelHost and PostProperties to be passed into the constructor. We want to add a third parameter to represent our HTTP poster so we can pass another one in for testing. However, when the PE is running as a service, we want by default an instance of the concrete HTTP poster. Therefore, we’re going to a new constructor like so:

public JobserveChannel(IChannelHost host, PostProperties postProperties, IJobserveWebClient webClient) : base(host, postProperties)

{

this.webClient = webClient;

}

This has to be chained to a constructor in the base class.

### Amending the Original Constructor

A new constructor is required for testing. It will take a IChannelHost, PostProperties and also a third parameter will create a new instance of the HTTP poster which is then chained to the constructor we amended.

public JobserveChannel(IChannelHost host, PostProperties postProperties)

: this(host, postProperties, new JobserveWebClient())

{

}

## Logging the Request

We’ll need a convenience method to log the request and to also set the request to the public property we created in the last section.

private void LogRequest(string request)

{

this.JobserveRequest = request;

this.Host.Trace(request);

this.Host.Trace(Environment.NewLine);

}

## Logging the Response

We’ll need a convenience method to log the response and to also set the response to the public property we created in the last section.

private void LogResponse(string response)

{

this.JobserveResponse = response;

this.Host.Trace(response);

this.Host.Trace(Environment.NewLine);

}

## Sending the Request and Parsing the Response

We’ll need a convenience method to be responsible for sending the request, extracting the job id (for which we already have a method for called ExtractJobIdFromResponse) and to run the response parser, which will find any errors for us.

If the channel is running in debug mode, then we want to extract the fake response from the GetDebugResponse method.

We also want to put the call to SendRequest inside a try..catch block to ensure we get to know about any issues.

Note that the post URL is taken from the settings factory.

private void SendRequestAndProcessResponse(JobserveRequestType requestType, string request)

{

JobserveResponse response;

string postUrl = Host.SettingsFactory[JobserveSettingsFactory.WebPostUrl];

this.Host.Trace(string.Format("Posting to: {0}", postUrl));

this.Host.Trace(Environment.NewLine);

if (Host.ExecutionMode == ChannelExecutionMode.Debug)

{

string responseStr = this.GetDebugResponse();

response = new JobserveResponse(true, responseStr);

}

else

{

try

{

response = webClient.SendRequest(Host.SettingsFactory[JobserveSettingsFactory.WebPostUrl], requestType, request);

}

catch (WebException webException)

{

if (webException.Status == WebExceptionStatus.ConnectFailure

|| webException.Status == WebExceptionStatus.NameResolutionFailure)

{

throw new ChannelDownException("Jobserve channel is not available", webException);

}

throw new WebException("An error occurred whilst connection to Jobserve", webException);

}

}

this.LogResponse(response.ResponseMessage);

if (requestType != JobserveRequestType.DeleteAdvert && requestType != JobserveRequestType.VerifyAccount)

{

this.ExtractJobIdFromResponse(response.ResponseMessage);

}

JobserveResponseParser responseParser = new JobserveResponseParser(response);

responseParser.FindAnyErrorsAndThrow();

}

# Expanding Unit Tests

Now we have all our code in place, we must now turn our attention to expanding upon our unit tests to ensure everything is working as we expect. Each unit test class should at least Post, Update (if applicable), Repost and Delete a vacancy.

It may be a good idea to include a convenience unit test to go through each of the operations in one go, so we can see the job Id being returned by the job board, and to see the subsequent update using that same job Id.

[Test]

[Ignore]

public void TestVacancyLifeCycle()

{

IChannel channel = this.CreateChannel(DefaultContent);

channel.Post();

channel.Update();

channel.Repost();

channel.Delete();

}

## Unit Tests for Debug Mode

Debug mode doesn’t physically send any requests to the job board, but we can test the structure of our requests and ensure it is valid. In section 8.1 we added public properties so hold the generated request.

### Checking the Request for Part-Time Positions

In terms of Jobserve, we can specify a job is part time with some XML such as:

…

<Categories>

<Category>Part-Time</Category>

</Categories>

…

We would probably want to make sure this element definitely exists if a vacancy is for a part-time position. Firstly, we’d write a convenience method to inspect the generated request using Linq-to-XML.

private void AssertXmlElementExists(string xmlStr, string elementName)

{

XDocument doc = XDocument.Parse(xmlStr);

bool elementExists = doc.Descendants().Any(e => e.Name.LocalName == elementName);

Assert.IsTrue(elementExists, string.Format("Could not find any element of <{0}> in the XML.", elementName));

}

Our unit test would make a slight change to the default vacancy as defined in JobserveContent.cs and ensure a part-time position is specified.

[Test]

public void TestPostWithPartTime()

{

JobserveContent content = DefaultContent;

content.Vacancy.WorkingHours = WorkingHours.PartTime;

IChannel channel = this.CreateChannel(content);

channel.Post();

// cast the interface to JobserveChannel so we can access the JobserveRequest public property

JobserveChannel jobserveChannel = (JobserveChannel)channel;

this.AssertXmlElementExists(jobserveChannel.JobserveRequest, "Categories");

XDocument doc = XDocument.Parse(jobserveChannel.JobserveRequest);

string elementValue = doc.Descendants().Where(e => e.Name.LocalName == "Categories").Select(e => e.Value).FirstOrDefault();

Assert.AreEqual("Part Time", elementValue);

}

### Checking the Request for Update, Repost and Delete Requests

Jobserve has different elements for updating, reposting and deleting. We can ensure our unit tests check the existence of such elements to ensure the XML generated is what we expect.

A post request can check to ensure the <PostAdvert> exists

[Test]

public void TestPost()

{

IChannel channel = this.CreateChannel(DefaultContent);

channel.Post();

JobserveChannel jobserveChannel = (JobserveChannel)channel;

this.AssertXmlElementExists(jobserveChannel.JobserveRequest, "PostAdvert");

}

### Testing for Unknown Errors (and an introduction to Moq)

In section 7.1 we created a response parser which would inspect the response received from the job board and throw an appropriate exception depending on the content. However, we can’t always be sure that we know what the error is in advance. The error messages returned may change, or new ones could be added. We need to know that our feed will fail in the event of an error message being returned that we don’t recognise. Any error that occurs in the feed should at least return a RemoteChannelException if a suitable other exception wasn’t determined.

To help us do this, we are going to use Moq. This is a framework which lets us pretend we are sending requests to a job board. For example, we don’t really want to send anything, or even get the debug response back. Moq allows us to specify in a unit test that when we invoke the SendRequest method of our HTTP poster, we can predetermine what is sent back.

Firstly, we want to create an instance of a response class with a generic error message like so:

JobserveResponse response = new JobserveResponse(false, "<FakeResponse>Something bad happened</FakeResponse>");

Then we create a skeleton instance of our web client HTTP poster using Moq:

Mock<IJobserveWebClient> webClient = new Mock<IJobserveWebClient>();

Now we want to tell Moq, that when we invoke SendRequest in the HTTP poster, we want it to return an instance of the response class we created just above. The parameters sent to the SendRequest method are placeholders, as we don’t care what we’re passing through to the method as we are not really calling it. We just want it to return a response that we have defined.

webClient.Setup(x => x.SendRequest(It.IsAny<string>(), It.IsAny<JobserveRequestType>(), It.IsAny<string>())).Returns(response);

Now we pass our “mocked up” instance of the HTTP poster to the channel class via construction dependency:

JobserveChannel channel = new JobserveChannel(this.GetHost(), DefaultContent.PostProperties, webClient.Object);

We then have to assign the VacancyData, AccountData, CustomData and OutputData because we are not using the CreateChannel convenience method.

AssignContentToChannel(channel, DefaultContent);

Then finally, we can use NUnit’s Throws extension method to check that the RemoteChannelException was indeed thrown.

Assert.Throws<RemoteChannelException>(channel.Post);

### Testing for Web Exception

As we’ve used .NET’s WebClient to send and receive requests, if anything severe happens, such as the job feed is unreachable then we want to ensure we throw the appropriate exception.

Here, instead of returning a response from SendRequest, we are telling Moq we want an exception to be thrown mimicking an issue with the job board. We then test to make sure the correct exception was thrown.

[Test]

public void TestPostWhenJobBoardDown()

{

Mock<IJobserveWebClient> webClient = new Mock<IJobserveWebClient>();

webClient.Setup(x => x.SendRequest(It.IsAny<string>(), It.IsAny<JobserveRequestType>(), It.IsAny<string>()))

.Throws(new WebException("Not available", WebExceptionStatus.ConnectFailure));

JobserveChannel channel = new JobserveChannel(this.GetHost(), DefaultContent.PostProperties, webClient.Object);

AssignContentToChannel(channel, DefaultContent);

Assert.Throws<ChannelDownException>(channel.Post);

}

## Unit Tests for Test Mode

If the job feed has a test environment, then we will create unit tests which will physically send a request the job board.

### Testing Verify Account with Invalid Credentials

When testing for account validity, we want to ensure we get a LogonErrorException is thrown. This is easily done with a try...catch block.

[Test]

public void TestVerifyAccountWithInvalidCredentials()

{

bool logonErrorCaught = false;

JobserveContent content = DefaultContent;

content.AccountData.AccountNumber = "123456";

content.AccountData.AccountNumber = "abc987";

IChannel channel = this.CreateAccountVerificationChannel(content.AccountData);

Assert.Throws<LogonErrorException>(channel.VerifyAccount);

}

### Testing Verify Account with Valid Credentials

When testing for valid credentials, we don’t expect a LogonErrorException so we test to make sure it was never thrown.

[Test]

public void TestVerifyValidAccountWithCorrectCredentials()

{

bool logonFailed = false;

IChannel channel = this.CreateAccountVerificationChannel(DefaultContent.AccountData);

channel.VerifyAccount();

}

If an exception is thrown during the validation process, then the unit test will fail.

## Unit Tests for Production Mode

If a test environment was provided by the job feed, most of the tests in the test mode unit class would probably be enough to give us confidence that the feed is working as expected. The test environment should mirror the live environment after all.

You may still want to include account verification unit tests for production mode if you have an account for the live feed.

Remember to include the [Ignore] attribute on tests that perform job posting logic in production mode!

### Unit Test to Delete a Non-Existent Adverts

In addition, section 2.4.7 discussed about implementing a unit test to determine how the value of the DeleteExpiryAction enumeration in the channel attribute. An example of such a unit test is as follows:

[Test]

public void TestDeleteNonExistentAdvert()

{

JobserveContent content = DefaultContent;

content.OutputData.JobId = "xxx";

IChannel channel = this.CreateChannel(content);

Assert.Throws<InvalidContentException>(channel.Delete);

}

# Web-cruit Integration

Once we have got to this stage, we then have to perform more testing by ensuring web-cruit can correctly post jobs and receive responses via the PE.

There is separate document detailing Web-cruit integration available at:

* <https://crsdev1.crs-warr.co.uk/svn/web-cruit/trunk/docs/Guides/JobPostingFeedGuide/WebCruitIntegrateWithPEJobFeed.docx>

## Check for Null Reference Exceptions

A common issue to appear when posting from web-cruit to a PE channel is that you may encounter null reference exceptions. This is mainly caused by web-cruit not populating a value that is carried over to the VacancyData, CustomData or AccountData classes.

The easiest way to try and avoid this is to ensure any mandatory string values use the coalesce operator so they are defaulted to an empty string like so:

vacancyData.ContactPhoneNumber ?? string.Empty;

## Debugging via Web-cruit

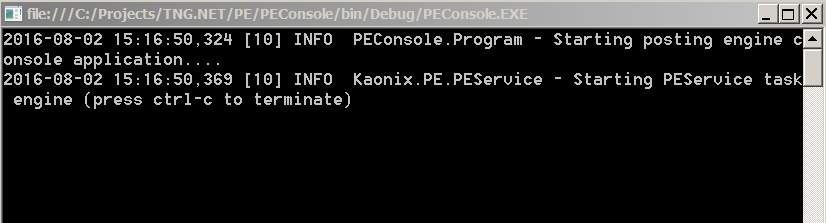
If finding the exact cause of any bugs found whilst testing a job feed from web-cruit, it is possible to debug the PE itself and step through until you find the issue.

Instead of running PEConsole from the command prompt, if you can run the console application from Visual Studio.

First, ensure you have set PEConsole as the start-up project in Visual Studio:

****

If you place a breakpoint somewhere in your code, and then run the solution, you will find this is the same as running PEConsole except as it’s in debug mode, you can step through your code.



When you run the process\_vacancyoutbox.cfm script from web-cruit, the request is forwarded to the PE and then if you have a breakpoint set up in your code, you can step through as if you were running from a unit test.

This means debugging from web-cruit is quite easy and painless.

## Editing the Generated Code Files

Sometimes we have to modify the data dictionary files or the CFCs for integration to work seamlessly between web-cruit and the PE.

We need to do this for the Jobserve feed. In the CustomData class, we specified a property to accept a 3 letter ISO country code. Someone using the Jobserve channel will have to provide this themselves, which shouldn’t be a problem as ISO codes are an internationally recognised standard.

However, we don’t to prompt the user in web-cruit for this information as a job media extension. Web-cruit already has this information in the Gazetteer table in the ISOCountryLongCode. There are two steps to get web-cruit to pass this information to Jobserve seamlessly.

Deciding whether we need to modify a generated CFC so we can pass the information seamlessly between web-cruit and the PE, or whether we just create a standard job media extension and ask the user for the information means some knowledge of web-cruit and what information the e-cruitnow database already holds that we can re-use and send to the PE.

Other than ISO codes, the most common scenario is when a job feed requires us to send the latitude and longitude of a vacancy. Empty Lemon is such a feed which requires this, and web-cruit stores the latitude and longitude for vacancies in the database, so we can easily extract this out of the database and set it to relevant property in CustomData before we send it to the PE. Prepopulating this information for the user is highly beneficial, as the user is unlikely to know the latitude and longitude of a postal code, and even if they did, they don’t need to enter the information in as this is now automated.

### Modify the Data Dictionary

Firstly we have to open up the Jobserve data dictionary in the directory it was generated in at C:\Projects\TNG.NET\PE\Web-cruit Integration\DataDictionary\PE\JobMediaRequiredData\_1072.xml.

Then we need to look for the ISCCountry entry:

<data name="JobMediaExtension\_1072\_Jobserve\_IsoCountryCode" revision="$Revision$">

<exec proc="sp\_AddOrUpdateJobMediaExtension">

<param name="JobMediaID" value="1072" />

<param name="FieldName" value="IsoCountryCode" />

<param name="FieldType" value="Text" />

<param name="FieldValues" value="" />

<param name="FieldValueText" value="" />

<param name="DisplayText" value="ISO Country Code" />

<param name="QueryAtPost" value="1" />

<param name="RequiredValidation" value="1" />

<param name="CategoryId" value="0" />

<param name="JobLocation" value="0" />

<param name="RegExpValidator" value="((?:M(?:A[CFR]|D[AGV]|N[EGP]|L[IT]|Y[ST]|[MS]R|CO|EX|HL|KD|OZ|RT|TQ|US|WI)|S(?:L[BEV]|[DEH]N|[JOP]M|G[PS]|V[KN]|W[EZ]|Y[CR]|[MU]R|AU|RB|SD|TP)|B(?:L[MRZ]|R[ABN]|E[LN]|G[DR]|H[RS]|[FW]A|DI|IH|MU|OL|TN|VT)|C(?:O[DGKLM]|H[ELN]|A[FN]|Y[MP]|[IP]V|[MX]R|CK|RI|UB|ZE)|G(?:R[CDL]|U[FMY]|I[BN]|N[BQ]|[AM]B|BR|EO|GY|HA|LP|TM)|A(?:R[EGM]|T[AFG]|L[AB]|N[DT]|U[ST]|BW|GO|IA|SM|ZE)|T(?:U[NRV]|C[AD]|K[LM]|[GT]O|[HZ]A|[OW]N|JK|LS)|P(?:R[IKTY]|A[KN]|[HO]L|CN|ER|LW|NG|SE|YF)|N(?:[CPZ]L|I[CU]|[EO]R|AM|FK|GA|LD|RU)|L(?:B[NRY]|[CKV]A|[AS]O|IE|TU|UX)|I(?:R[LNQ]|S[LR]|[DM]N|ND|OT|TA)|K(?:[AG]Z|[IO]R|EN|HM|NA|WT)|E(?:S[HPT]|CU|GY|RI|TH)|V(?:[ACU]T|EN|GB|IR|NM)|D(?:[MZ]A|EU|JI|NK|OM)|F(?:R[AO]|IN|JI|LK|SM)|H(?:[MN]D|KG|RV|TI|UN)|U(?:[GS]A|KR|MI|RY|ZB)|J(?:AM|EY|OR|PN)|R(?:[EO]U|US|WA)|Z(?:AF|MB|WE)|W(?:LF|SM)|OMN|QAT|YEM))" />

<param name="ValidationErrorMessage" value="The ISO Country Code is not valid and must be 3 characters long. Please see http://www.nationsonline.org/oneworld/country\_code\_list.htm for valid 3 letter codes." />

<param name="AllowFiltering" value="0" />

<param name="ClientSpecific" value="0" />

<param name="AllowClientSpecificSetup" value="0" />

<param name="ClientSpecificVisibleToUser" value="0" />

<param name="OrderId" value="1" />

<param name="IsPEAccountProperty" value="0" />

<param name="QueryAtRepostOnly" value="0" />

</exec>

</data>

To prevent the user being asked for this as a job media extension, we have to copy this into an editable section for this field, and edit “FieldType” so that it is blank and set “QueryAtPost” to 0.

<!-- BEGIN\_EDITABLE\_SECTION\_JOBMEDIAEXTENSION\_ISOCOUNTRYCODE -->

<data name="JobMediaExtension\_1072\_Jobserve\_IsoCountryCode" revision="$Revision$">

<exec proc="sp\_AddOrUpdateJobMediaExtension">

<param name="JobMediaID" value="1072" />

<param name="FieldName" value="IsoCountryCode" />

<param name="FieldType" value="" />

<param name="FieldValues" value="" />

<param name="FieldValueText" value="" />

<param name="DisplayText" value="ISO Country Code" />

<param name="QueryAtPost" value="0" />

<param name="RequiredValidation" value="0" />

<param name="CategoryId" value="0" />

<param name="JobLocation" value="0" />

<param name="RegExpValidator" value="((?:M[ACDEFGHKLMNOPQRSTUVWXYZ]|B[ABDEFGHIJLMNORSTVWYZ]|S[ABCDEGHIJKLMNORSTVYZ]|G[ABDEFGHILMNPQRSTUWY]|C[ACDFGHIKLMNORUVXYZ]|A[DEFGILMNOQRSTUWXZ]|T[CDFGHJKLMNORTVWZ]|P[AEFGHKLMNRSTWY]|N[ACEFGILOPRUZ]|K[EGHIMNPRWYZ]|L[ABCIKRSTUVY]|I[DELMNOQRST]|E[CEGHRST]|V[ACEGINU]|D[EJKMOZ]|F[IJKMOR]|H[KMNRTU]|U[AGMSYZ]|R[EOSUW]|J[EMOP]|Z[AMW]|W[FS]|Y[ET]|OM|QA))" />

<param name="ValidationErrorMessage" value="The ISO Country Code is not valid and must be 2 characters long. Please see http://www.nationsonline.org/oneworld/country\_code\_list.htm for valid 2 letter codes." />

<param name="AllowFiltering" value="0" />

<param name="ClientSpecific" value="0" />

<param name="AllowClientSpecificSetup" value="0" />

<param name="ClientSpecificVisibleToUser" value="0" />

<param name="OrderId" value="1" />

<param name="IsPEAccountProperty" value="0" />

<param name="QueryAtRepostOnly" value="0" />

</exec>

</data>

<!-- END\_EDITABLE\_SECTION\_JOBMEDIAEXTENSION\_ISOCOUNTRYCODE -->

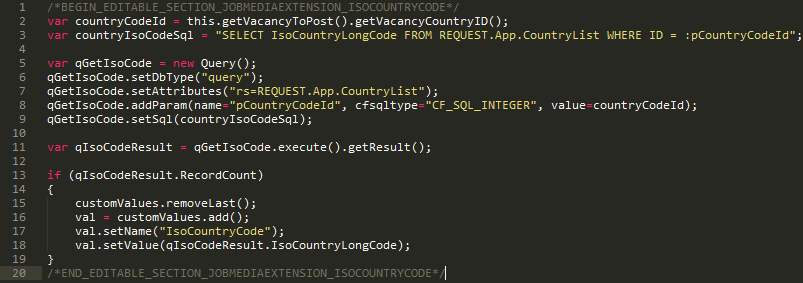
### Modify the CFC

The next step is to find the generated CFC and to open it up. The generated CFC for Jobserve is at: C:\Projects\TNG.NET\PE\Web-cruit Integration\CFCs\PE\JobserveChannel.cfc.

Again, we look for the “ISOCountryCode” entry.



And in the editable section, we write some extra ColdFusion script to get the information we need from the Gazetteer table before calling val.setValue().



# Appendix A – Jobserve Posting Schema

Excerpt from the Jobserve specification – the fields required to send to the feed.

| **Xml Node** | **Description** | **Example value** |
| --- | --- | --- |
| **AccountNumber** |  |  |
| ApplyOnline | Not for external use |  |
| ApplyOnlineEmail | Email address to be used for sending applications for this job. |  |
| ApplyOnlineTemplate | Not for external use |  |
| ApplyOnlineCustomTemplateName | Not for external use |  |
| ApplyOnlineCustomTemplateXml | Not for external use |  |
| **Contact** | Contact for the job, appears against job on the website |  |
| Telephone |  |  |
| Fax |  |  |
| Email | Email address to be used for job – this will be made available via a mailto link on the website for the job |  |
| EmailComments | Additional text to appear next to the email | Please quote reference JS123 |
| **Position** | The Job Title of the job |  |
| Skills | ***Obsolete:*** The job description – please use HTMLSkills. |  |
| SkillsExtra | ***Obsolete:*** Additional job description – please use HTMLSkills. |  |
| **HtmlSkills** | Job Description, can contain HTML |  |
| **Location** | Location of the job |  |
| **CountryCode** | ISO Country Code of the job |  |
| PostZipCode | Post Code/ ZIP code for job – is not displayed on the job. |  |
| **Reference** | Your reference for the job |  |
| StartDate | Information on when the job starts |  |
| Duration | Information on duration of job (typically for Contract positions) |  |
| Rate | Details of the salary |  |
| Type | The job type : Permanent, Contract or Permanent/Contract | P, C or B |
| URL | Optional ATS Url |  |
| AlternativeName | The company name to be associated with the job. Special permission is needed to use this feature. |  |
| VISARequired |  | 1, 2 or 3 |
| JobAdditions | Not for external use |  |
| ReferFriend | Not for external use |  |
| AnonymousAdvert | Hide company details on the job. Special permission is needed to use this feature. |  |
| SecurityClearance | Enable Security Clearance question to be asked on the online application on JobServe | 1 |
| CustomJobID | A unique ID that a posting client can use to manipulate their job. |  |
| NumberOfReAds | Number of additional weeks this job should be posted – note this will cost the client each week for ReAdvertising unless the job is deleted. | 2 |
| AdvertiserID | Not for external use |  |
| KillerVisa | Not for external use |  |
| EndDate | Not for external use |  |
| MediaURL | Not for external use |  |
| Categories | List of Job Categories | See list below |
| NetworkBoost | Send the job on to the JobServe Network – **Cost may apply**. | 1 |
| **MarketID** | JobServe Industry Code | See list below |
| SecondaryMarketID | Secondary Industry Code (if applicable) | See list below |
| TertiaryMarketID | Third Industry Code | See list below |
| PrimaryLanguage | Not for external use |  |
| **ConsultantEmail** | This should correspond to the consultant/user who owns the job. ***This does not have to be the same as the Email or Application Email on the job.*** |  |
| ConsultantFirstName | Optional first name |  |
| ConsultantLastName | Optional last name |  |

# Appendix B – Jobserve Industry List

| Code | Industry |
| --- | --- |
| 01 | IT & Telecommunications |
| 02 | Finance, Accounting & Banking |
| 03 | Engineering |
| 04 | Sales & Marketing |
| 06 | Office & Administration |
| 07 | Legal |
| 08 | Healthcare & Medical |
| 09 | Call Centre & Customer Services |
| 10 | HR & Recruitment |
| 11 | Construction & Civil Engineering |
| 12 | Transport & Logistics |
| 13 | Advertising, Media & Entertainment |
| 14 | Food & Hospitality |
| 15 | Mining, Energy, Oil & Gas |
| 16 | Science, Biotech & Pharmaceuticals |
| 17 | Retail & Consumer Products |
| 18 | Defence, Military & Armed Forces |
| 19 | Travel & Tourism |
| 20 | Education |
| 21 | Manufacturing |
| 22 | Trades & Services |

# Appendix C - Jobserve Category List

|  |
| --- |
| Category |
| Part Time |
| Temp |
| Bilingual |
| Executive |
| Graduate |
| Home Working |
| Public Sector |