

# **Development - .NET**

Aptean Ltd  
Copyright © 2011-2025.

# Contents

<b>1 Development - .NET</b>	<b>1</b>
1.1 .NET Development Environment	1
1.2 .NET Programming Guide	1

# 1 Development - .NET

## 1.1 .NET Development Environment

### 1.1.1 Development Tools List

- Visual Studio 2010
- Visual Studio 2008 - WM 6.5 PDA only
- NotePad++
- SQL Server Managment Studio
- SQL Server Express 2008
- SQL Server Compact
- Windows Mobile 6 SDK
- Windows Mobile 6.5 SDK
- SOAPUI
- ActiveSync/Windows Mobile Device Center
- Visual Source Safe
- IIS

## 1.2 .NET Programming Guide

### 1.2.1 Database

The database is currently exclusive deployed on MS SQL Server and is support from 2005 upwards. Due to the nature of the Data Access Layers implementation, the system is to some extent database agnostic, and that simply rewriting the database table and packages to support another database should be all that is required to port over to MySQL or Oracle.

The database consists of a series of tables that are required to hold the needed data for the ePOD product. This data is manipulated through stored procedures which can be call from the data access layer.

Each table generally has four stored procedures:

- TABLE\_NAME\_INSERT
- TABLE\_NAME\_UPDATE
- TABLE\_NAME\_SELECT
- TABLE\_NAME\_DELETE

There may also be the following stored procedures:

- TABLE\_NAME\_GET\_MAXID
- TABLE\_NAME\_CANCEL
- TABLE\_NAME\_SEARCH
- TABLE\_NAME\_SELECT\_UPDATED\_DATA - Not currently used due to improvements needed in the PDA Client and the Webservices

#### 1.2.1.1 Tables

- EPOD\_DEVICE\_TYPE

Holds a list of models of devices that have logged on to the calidus\_epod.asmx webservice and configuration flags such as barcode scanning and camera capture capabilities. The intention is to use this data to restrict functionality on devices where there may be a issue using the functionality, for example no camera on the device.

- EPOD\_CONTAINER

Holds container level information this is linked through primary key to the job record. Container 0000000000 is always the loose product container.

- EPOD\_CUSTOMER

Holds customer level information.

- EPOD\_JOB\_ADDRESS



Holds address and contact information similar to a customer record but is used in the scenario of a different delivery location being provided for the customer.

- EPOD\_JOB

Holds Job information such as type, customer, and times.

- EPOD\_LOAD

Holds load level information, representing a drivers workload (a set of jobs).

- EPOD\_DEVICE

Holds device specific identification details such as Unique ID, Type, OS and the last User. This is updated with every relevant communications message from the PDA.

- EPOD\_JOB\_GROUPS

Holds the configuration details for a job group

- EPOD\_LOTS\_STATUS

This table is not currently used, but was part of the initial design for LOTS integration.

- EPOD\_PRODUCT

Holds product level information associated to the Container and Job.

- EPOD\_PHOTO

Holds photo information against job, container and products, this is held as a base64 encoded jpg file.

- EPOD\_REASON\_CODE

Holds the static reason code information.

- EPOD\_USER\_AUDIT

Holds a audit of all communications between the PDA and the server, identifying the type of communication and the GPS location.

- EPOD\_SERVICE\_ACTIVITY\_MASTER

Holds the static activity data.

- EPOD\_SERVICE

Holds the service job information, this will be linked to a job record.

- EPOD\_SERVICE\_PRODUCT

This table holds all products used as part of a service.

- EPOD\_SERVICE\_PRODUCT\_MASTER

This table holds static product information.

- EPOD\_SERVICE\_VEHICLE\_PRODUCT

This table holds a list of service products that are held in each vehicle.

- EPOD\_SITE

This table is at the top of the hierarchy, allowing multiple sites to be hosted on the same database. This holds high level configuration.



- EPOD\_USER\_ACCESS\_GROUPS

This table holds the association between users and jobs groups allow users to view jobs of a particular group within the system.

- EPOD\_USER

This table contains the users details.

- EPOD\_VEHICLE\_CHECK

This table holds the raw XML returned from vehicle checks performed on the device.

- EPOD\_VEHICLE

The vehicles within the system are held here.

- EPOD\_XF\_AUDIT\_HEADER

This table contains all the export audit information.

- EPOD\_XF\_AUDIT\_DETAIL

This table has not been implemented within the system, and will be used if the Audits need expanding.

- EPOD\_XF\_CONFIG

This holds the configuration for the system exports.

## 1.2.2 Data Access Layer

The data access layer (EPOD Server project) provides a series of classes for interacting with the database. These classes are mapped out as the following #regions:

- Class
  - ◆ Private Members
  - ◆ Public Members
  - ◆ Public Constructor
  - ◆ Private Constructor
  - ◆ Public Methods
  - ◆ Private Methods
  - ◆ Public Static Methods

### 1.2.2.1 Private and Public Members

The private members represent columns within the data rows. These are all C# data types. This should not be manipulated manually and should be accessed via the public members which provide Get and Set methods to access them, implying various rules, for example maximum length.

### 1.2.2.2 Public Constructor

This is used to create a instance of the object. Calling this with the relevant parameters (those making up the primary key) will cause the object to call the TABLE\_NAME\_SELECT method and load the first record returned into the public members. If the object has been found then the DBExists property will be set to true.

### 1.2.2.3 Private Constructor

This is used to construct the object internally within the object without calling the database. This will simply take all of the properties as parameters and will set-up the object as with the public constructor. This is used when returning a <TABLE\_NAME>List of objects, and will be called multiple times rather than repeating calls to the database. This can only be and should only every called in the scenario where your query will return multiple records.



#### 1.2.2.4 Public Methods

These methods typically consist of the following:

- Update() - Commits the current private members to the database using TABLE\_NAME\_INSERT or TABLE\_NAME\_UPDATE dependant on the DBExists property.
- Cancel() - Sets the object to cancelled status, either through setting the properties manually and calling update or calling a TABLE\_NAME\_CANCEL stored procedure.
- Delete() - Calls the TABLE\_NAME\_DELETE stored procedure deleting the object from the database.
- Complete() - Sets the object to complete
- ToXElement() - Returns a XElement object containing all private members in XML form.
- ToXElementMin() - Returns a reduced set of the above, not including any hierarchy.

#### 1.2.2.5 Private Methods

This region contains methods internal to the objects. This will typically contain:

- GetParameters() - Adds all the private members to the parameters in the DbCommand object to be passed to the store procedure.
- GetParametersForSelect() - Adds all the database fields as null parameters to the DbCommand object.
- SetNewId() - Used in the constructor methods to set a new id for the object if one is not provided.

#### 1.2.2.6 Public Static Methods

These methods provide access to the search functionality of the system. Typically you will provide a series of parameters, which will be passed to the TABLE\_NAME\_SELECT or TABLE\_NAME\_SEARCH. The method will then call the private constructor of the object for each row found and store the objects in a <TABLE\_NAME>List which is returned. This are structured in the same way as the constructor but will use a loop to populate a list rather than simply using the first row found.

### 1.2.3 WebServices

All communications with ePOD are currently done via WebServices supporting both SOAP and Restful. ePOD uses XML as it's communications medium.

#### 1.2.3.1 Message Process

The message process C# class within the ePOD\_Server project handles all communications from the PDA. This is called from the Calidus\_epod.asmx code behind when a request is received. All messages are passed in as a XElement object. As all messages from the PDA are passed to a single end-point, the MessageProcess must evaluate how to process this based on the root tags of the XElement object. All responses from the Message Process are given a RESULT attribute, container ACK for success and NAK for failure. All messages processed have a record written to the EPOD\_USER\_AUDIT table.

##### 1.2.3.1.1 ProcessEPODMessage

The ProcessEPODMessage method is the method that is called by the Calidus\_epod.asmx code behind. The contents of the Soap request from the PDA are passed to this method as a XElement object. This method initially validates the EPOD\_SITE and EPOD\_USER exist and the password sent with the message are correct, if not the error is returned as a NAK XML to the Calidus\_epod.asmx. If validation is successful this will be passed to the relevant method for processing and the results returned to Calidus\_epod.asmx.

If the device type is android the EPOD\_DEVICE record is inserted or updated, if inserted and a EPOD\_DEVICE\_TYPE record does not exists this is inserted as well.

##### 1.2.3.1.2 ProcessLogonRequest

This method will process all logon requests. The devices last data update date and time are send with the login request, this is used to retrieve standing data from, EPOD\_SITE, EPOD\_JOB\_GROUPS, EPOD\_REASON\_CODE, EPOD\_SERVICE\_ACTIVITY\_MASTER, EPOD\_SERVICE\_PRODUCT\_MASTER and EPOD\_USER, where the data has been updated since this time. Version numbers are also sent across with login request from the PDA, these are passed to ProcessUpdateRequest().



#### 1.2.3.1.3 ProcessUpdateRequest

These version numbers are compared to the version numbers stored within the PDA\_Updates folder. If the version on the server are greater then the updated version numbers are returned with the URL to access the updates as part of the response message.

#### 1.2.3.1.4 ProcessLoadRequest

It will retrieve the first available load for the user of the user id passed to it and will return this as XML. It will filter out any jobs at completed or cancelled status. During the retrieval process the load will be set to inprogress status.

#### 1.2.3.1.5 ProcessJobLockRequest

This message is processed by validating the desired job is still available to the current user, if not a delete message is returned. If the job is available the job is checked for updates and any are then sent to the device. The job is set to In Progress on the server at this point, and all other In Progress jobs on the drivers load are set back to pending.

#### 1.2.3.1.6 ProcessAutoUpdateRequest

This message is responsible for checking that a users current load is up to date. The message sent from the device will contain the current load the user is working on and a last changed date for this load. When received there are the following scenarios:

- The load has been removed or all jobs completed and cancelled

The method will return a message to the device advising that the load has been removed.

- The load is in the same state as when found.

The unique keys of each record at load, job, container and product are returned as XML.

- The load has been updated

The full XML for the load is returned.

- A Job, Container and Product has been updated

The updated record is return in full as XML.

#### 1.2.3.1.7 ProcessVehicleCheck

This function processes completed vehicle check data from the PDA. This method will write the vehicle check record to the database and return a ACK if successful.

#### 1.2.3.1.8 ProcessJobUpdate

This function processes any complete or cancellation messages sent at the point of job completion or cancellation. This will process the generic job data followed by the container and product or service data. Finally if this is the last job still at pending or in progress status it will complete the load record, it is worth noting that a separate message 'Load Update' is sent to update this from the PDA as well.

#### 1.2.3.1.9 ProcessPhotoRequest

This method will process photo uploads from the PDA's.

This photo upload allow for message chunks. Multiple messages are sent each containing a chunk of the image. For the first chunk received, a photo record is written and associated to the corresponding record as with the previous message except for each missing chunk of the picture a XML <CHUNKX> tag is written into the EPL\_PHOTO field in sequence. As each other chunk is received, the photo record is recalled via its corresponding job, container or product record and the <CHUNKX> tag replaced with the image chunk.

Both messages return a ACK or NAK based on success or fail.



#### 1.2.3.1.10 ProcessLoadUpdateRequest

This message is processed updating the load record with the final metrics of a drivers workload. This typically will not complete the load only update the record.

#### 1.2.3.1.11 ProcessGPSStatusRequest

This message is currently only Android. This message writes data to the user audit table with a GPS stamp of the current location.

### 1.2.3.2 EPOD\_DATA\_SERVICE

The EPOD\_DATA\_SERVICE C# class handles importing of data.

The import as within the MS Excel imports are a one error fail all system.

This class has one publicly exposed function, ProcessXMLImport(EPOD\_DBCONNECTION,XDocument). This function will process the XDocument received by:

- Validating the XML against the XMLUpload.xsd file.
- Importing all loads and their hierarchy.
- Importing all jobs and their hierarchy.
- Importing all customers.
- Importing all service product master records.
- Importing all vehicles.
- Importing all users.
- Importing all reason codes.

This is done through recursively calling the internal private functions.

There are two main variables that can be accessed externally:


- DataSet ErrorList
- DataSet CompleteList

These variables are populated with any success or error messages. If the ErrorList has any data in it the import will not have been committed.

## 1.2.4 Web Application (Admin)

 **Warning:** Section Incomplete

### 1.2.4.1 POD Report Development

- All new reports should be created by using existing pagination code - this should be centralised and reusable where possible.
- To simplify development, existing reports *that already use the pagination code* should be copied.
- Pagination allows for a string definition of just the following 3 sections: Header, Details and Footer.  **Note:** This may be extended in time to Initial and Final headers, plus Page Header and Footers.
- The call of the Pagination object (ReportPager) allows for the configuration of each section (for example, whether a particular section should be displayed, whether to reserve space for it, etc) - see this code and extensive examples for details of how this works and how to extend this for your report, if necessary.
- A prototype is always created in plain text - use this to copy in the header, detail and footer sections, escaping special characters as required. Also use this to source the correct CSS.
- Each section is defined in the C#.NET file (i.e. the "aspx.cs" file) rather than defined as objects in the asp.NET file (i.e. the ".aspx" file), modifying the copied populateReport method and any called methods, such as generateContent, generateHeader, generateFooter. These will be built by copying the provided prototype HTML code replacing data from the records found, escaping special characters as required. Note that data may be retrieved from globally- or locally-declared DAL objects within the report, if this is required.
- All other methods in the C#.NET file should be common to all POD reports, as follows:
  - ◆ private int ValidateUser()
  - ◆ public void Page\_Error(object sender, EventArgs e)
  - ◆ protected void Page\_Load(object sender, EventArgs e)





- ◆ protected override void Render(HtmlTextWriter writer)
- ◆ protected override void OnUnload(EventArgs e)
- ◆ protected void emailBT\_clicked(object sender, EventArgs e)
- ◆ private string convertJobType(String EPL\_JOB\_TYPE)
- ◆ public static int roundup(int i)
- The asp.NET file contains only:
  - ◆ The required CSS code for this format, sourced mainly from the prototype rather than the copied source code.
  - ◆ The "Email" div
  - ◆ A general "POD" div.
- When complete:
  - ◆ Check against the prototype - pay particular attention to line breaks (both forced by the HTML BR tag and those automatically placed into data and labels by size restrictions).
  - ◆ Check the final PDF result by converting the produced page to PDF using the conversion tool.
  - ◆ It should not then be necessary to check on multiple browsers, as the prototype has already been tested against all major browsers and likely versions, such as:
    - ◇ Firefox (latest)
    - ◇ Chrome (latest)
    - ◇ Internet Explorer 11
    - ◇ Internet Explorer 6-10 (through emulation)
    - ◇ Safari (through PDF conversion)

When creating a report following these instructions, the produced report should match the prototype completely, and therefore there is less chance of a customer being unhappy upon delivery, having received a report that does not match the specification and prototype that they agreed before development commenced.

#### 1.2.4.2 Multi-Lingual

 **Warning:** Section Incomplete

### 1.2.5 ePOD Manager

The ePOD manager application is still in alpha phases. The application provides the following administrative functions through a GUI or through command prompt:

- DeleteData

This will delete all dynamic (load, job, container, product and photo) data from the database within a particular number of days previous to today.

- DeleteImageData

This will delete all photo data from the database within a particular number of days previous to today.

- DeleteVehicleCheckData

This will delete all vehicle check data from the database within a particular number of days previous to today.

- DeleteXFAuditLogs

This will delete all Export Audit data from the database within a particular number of days previous to today.

- TrimDataBase

This will trim all data within the database.

- ShrinkDataBase

This will shrink and reclaim any space unused in the database.

- RebuildIndexDataBase

This will rebuild all indexes on the database.



- DeleteManagerLogs

This will delete all logs generated by the ePOD manager application.

- DeleteePODLogs

This will delete all logs generated by the ePOD system.

## 1.2.6 ePOD AutoExport

The ePOD AutoExport program is set up as a scheduled service to run every X minutes on the host machine. This process will perform two tasks:

### 1.2.6.1 Export

This process will retrieve all records (Job and Load) in the database with the EPL\_XFER\_FLAG set to N. Once retrieved the process will select their associated Job Group configuration for export and if it exists it will export them through the given method, either SOAP, POST or email. As with all other exports and imports in ePOD XML is the medium used and this is generated through the standard toXElement() methods in each DAL object.

### 1.2.6.2 POD Email

The POD Email functionality in the ePOD AutoExport procedure works similarly to the Export functionality. A stored procedure is ran to collate all jobs that have the EPL\_EMAIL\_FLAG set to Y. Each job is then evaluated (not this evaluation should be migrated to the Store Procedure as will likely be faster), is the customer has a email address and the job group allows for auto emails then the POD will be generated and sent by email. To generate the POD, first the configured report is called locally from the server through a http request using a local cookie for authentication. The HTML is retrieved this is either sent via email to the customer or is stored on disk and requested by WkHtmltoPdf to generate a PDF which can then be sent via email.

## 1.2.7 PDA Client

ePOD system current has two PDA clients Windows Mobile and Android. These both have local databases and communicate to the centralised server using the pre-specified web services.

### 1.2.7.1 Process Flow

#### 1.2.7.1.1 Login

```

Application Start
  PreStart Checks
    IF first login
      IF Config XML file exists
        Send Grace Login Request
      ELSE
        Open Config Form
    ELSE
      Load User Data (DBConnection.cs and Program.cs)
      Open Login Form

```

#### 1.2.7.1.2 Job List

```

Open Job List
  IF Check for load with uncompleted (Status I & P) jobs on local DB == TRUE
    Prompt user to refresh data
    IF Refresh data
      Send Load Request
  BIND GET * JOB DATA to Job List
  START Pending Timer
  START Auto-Update Timer
  AUTO-UPDATE timer fires
    Send Load update request
  If Updates
    BIND Updates to Job List
  IF User Selects Job
    Open Job Detail
    On Close

```



```

IF Check for load with uncompleted (Status I & P) jobs on local DB == FALSE
  PROMPT for more work
  SEND Load Request or Exit
ELSE
  BIND GET * JOB DATA to Job List

```

 **Warning:** Section incomplete

### 1.2.7.2 GUI

Both device work on the principles of a card stack. Each form or window will open another and on exiting of a window will refresh the data it is currently displaying showing any modifications. (You need to beware that this means you are holding the previous window open in memory and on some\*\* devices this is limited). Each Form or Window will run some set-up either both loading or at the point of loading which will set up any user or background controls, as well as getting any data from resources it may need. Once at this point all interaction is user driven (other than separate timed processes).

### 1.2.7.3 Communications

All communications with the central server are performed via SOAP requests using XML. There are two types of request that occur from the devices:

- Send and Wait

These are messages that must be recieved and processed before a action can occur.

```

For example the logon process:
User enters user and password
User clicked Logon
Send Logon Request
If Response
  Process Updates
  Process Data
  Local Logon
Else
  Local Logon

```

In reality the user can always work without sending these messages due to the local database, but they may not be working with the most recent data if they do/cannot.

- Send and Forget (Send Pending Data)

This uses the same mechanism as Send and Wait. A timed process is started on the device that reads data from the Pending data table in the local database (this will contain XML written to it for job updates ect), if a records is found then it will send the data to the server and await a response. Once receieved the data will then be removed from the local database and the next record sent. To conserve battery (timers are a expensive resource) the timers are only started when data is written to or existing in the table, once all data is sent they are shutdown until the next record is written.

#### 1.2.7.3.1 Android

Android uses a modified suds.js (Suds2.js) client (<https://github.com/kwhinnery/Suds>). This is a javascript implementation of SOAP. This has been modified in the invoke method to handle our webservices.

#### 1.2.7.3.2 WM6.5

The WM6.5 client uses the standard build in soap calls to VS2008. This allows you to add the service as a webmethod within the project and reference it as a standard method.

### 1.2.7.4 Photos

The photos function on both phone utilising the native camera application, the image from this is then compressed to a .JPEG image, converted to base64 and stored within the database in this format. From the database they can then be transmitted to the server.

The client post version 1.31.1.0 will now attempt to split images over 50kb into chunks to send back to the server. Upon job completion and cancellation, the job object is passed to the process photos function which will recursively check the job hierarchy for photos. If any photos are present they are loaded into a photo object and the data of the photo is split into chunks and each chunk written to a photo request message.



#### 1.2.7.5 Multi-Lingual

⚠ Warning: Section Incomplete

#### 1.2.8 Variable Notation

⚠ Warning: Section Incomplete

#### 1.2.9 Source Control

⚠ Warning: Section Incomplete

#### 1.2.10 Commenting Code

⚠ Warning: Section Incomplete

#### 1.2.11 Developer Testing and Debugging

⚠ Warning: Section Incomplete

#### 1.2.12 Other

⚠ Warning: Section Incomplete

