Manual

Extended Web Services Use Cases

Version 2.8

CONTENTS

Extended Web Services Use Cases 4
Terminology 4
Consumer Company 4
Facility 4
Service / Facility Account 5
User 5
Request Initiator 5
Account / Contract / Customer / Member / Client 5
Payment History 6
Audit Trail 6
Call History 6
Acquisition 7
USE CASE: Single Payment – Account Remains Open 8
USE CASE: Single Payment – Account Closes 9
USE CASE: Ongoing Service 9
USE CASE: Trial Period 9
USE CASE: Hire Purchase – Equal Deposit and Installments 10
USE CASE: Hire Purchase – Non Equal Deposit and Installments 10
USE CASE: Club Membership 11
USE CASE: Handling Direct debit Transaction reversal 12
USE CASE: Renewal of Account 14
USE CASE: Customer has purchase additional services or product 15
Fulfilment 16
USE CASE: Cancel Account 16
USE CASE: Change Customer Individual Details 16
USE CASE: Change Customer Address 17
USE CASE: Change Customer Email 17
USE CASE: Change Customer Phone Number 18

USE CASE: Change Account Total Value 18
USE CASE: Change Account: Change Payment Method 19
USE CASE: Change Account: Change to Fixed Term 19
USE CASE: Change Account: Increase Payment and Shorten Term 20
USE CASE: Change Account: Stop Payment Collection 20
USE CASE: Change Account: Resume Payment Collection 21
USE CASE: Change Account: Suspend Payment Collection 21
USE CASE: Change Suspension end date 22
USE CASE: Refund an Account 22
USE CASE: Change Account: Change Payment Schedule Frequency 24
USE CASE: Change Account: CHANGE Payment Schedule Instalment 25
USE CASE: Change Account: CHANGE Payment Schedule In The Future 26
USE CASE: Change Account: Bring Payment Schedule Payment Day Forward 27
USE CASE: Change Account: Move Payment Schedule Payment Day Backward 28
USE CASE: Change Account: External account reference number 29
USE CASE: Change Account: New One-off Payment Schedule 30
USE CASE: Change Payment Schedules: Date of Payment 31
USE CASE: Change Account: Next Payment Collection Date 31
USE CASE: Change Account: Delete Pay Schedule 32
USE CASE: Change Payment Collection: Adjust Catch Up Amount 32
USE CASE: Record a Payment Received for an Account outside the Debitsuccess billing system 32
USE CASE: Initiate a Real Time Credit Card Payment For an Account 34
USE CASE: Get Card Payment Status For Customer Account 35
Assurance 38
USE CASE: Retrieve Account Records 38
USE CASE: Retrieve Payment History 40
USE CASE: Retrieve Audit Trail 41
USE CASE: Retrieve Call History 42
USE CASE: Retrieve Facility Account Configuration 43

USE CASE: Add Commission Value To Payment Amount 44

EXTENDED WEB SERVICES USE CASES

The Debitsuccess Web Service (DWS) use cases can be categorised into three general categories:

Acquisition use cases result in the creation of a new account through which a customer is billed.

Fulfilment use cases result in changes in information held about an account or customer.

Assurance use cases enable the retrieval of historical information about changes and activity related to accounts.

Aspects of the Debitsuccess service which are not managed through DWS are managed through the Debitsuccess Business Development Managers (BDMs).

TERMINOLOGY

There are a number of terms used within Debitsuccess to refer to entities or concepts managed by the business. Sometimes there are a number of different terms commonly used to describe the same entity.

The reason for this can be partially explained in terms of Debitsuccess's history and the fact that Debitsuccess provides services to a number of commercial sectors, each of which have their own sets of terminology.

Where more than one term is in common use, they are listed in the headings of the following sections. The first term listed is used consistently in this document.

The DWS consists of functions which take a request object and return a response object. The response object contains success or failure of the operation and in the case of failure, some explanation for the failure. Care should be taken in external systems to manage communications exceptions such as timeouts and unreachable services.

The following entities and concepts are, where appropriate, illustrated using code snippets which are simplified C#.

CONSUMER COMPANY

An entity, typically a company, that uses Debitsuccess services and provides products and services to their customers. Although Consumer Companies could equally be described as Debitsuccess's clients or customers, these terms are avoided because they can cause confusion with the Consumer Companies' own clients and customers.

FACILITY

This is a channel through which a Consumer Company provides its services. A consumer company has one or many Facilities.

The Debitsuccess BDM will be able to provide advice on this.

SERVICE / FACILITY ACCOUNT

A single Facility may offer one or more Services. Services are not managed through the DWS.

The Debitsuccess BDM can arrange the Services required.

There are no templates for Services managed through DWS. It is the responsibility of the external DWS user to enforce consistency on the Accounts they manage through DWS.

Debitsuccess will provide a unique Contract Prefix for each service.

DWS acquisition requests represent a service as a (string) Contract Prefix field.

```
partial class Request { string ContractPrefix; }
```

USER

A User is the basic access credentials to DWS. A single Consumer Company has one or more Users. Users may be restricted to subsets of Facilities and Services.

The Debitsuccess BDM can arrange the Services required.

Every DWS request contains a user object:

```
class User { string Username; string Password; }
```

REQUEST INITIATOR

A property called Request initiator is included with every request. This is an opportunity for an external system to include information on who initiated the request. For example, it may be used to identify the Consumer Company's account manager who authorised the operation.

Every DWS request provides the opportunity to include Request Initiator identification:

```
partial class Request { string RequestInitiator; // Optional }
```

ACCOUNT / CONTRACT / CUSTOMER / MEMBER / CLIENT

Debitsuccess's services are centred on billing customers through Accounts. Accounts are created and managed through DWS.

An Account is a relationship between an individual customer and a Service. As such it represents a contract between the Customer and the Consumer Company for the provision of and payment for goods or services. The Contract authorises Debitsuccess to collect payments from the Customer on behalf of the Consumer Company.

Although a real individual may use more than one Service provided by a Facility, the [Account / Contract / Customer / Member / Client] relationship is strictly one-to-one and de-normalised. It is not a many-to-many relationship between Customers and Services as might be modelled in a normalised relational database. Bearing this in mind, the terms Account / Contract / Customer / Member / Client all refer to the same entity in the Debitsuccess business view. They are also used to distinguish between different groups of information on Account records. This has proved to be a useful simplification in Debitsuccess's overall Account centric business.

The de-normalised nature of Debitsuccess's view of Customers and Accounts does not preclude external DWS system from treating customers as individuals and having Accounts as many-to-many relationship between customers and Services. This is entirely the responsibility of the developers of DWS external systems. There are fields provided to allow external systems to supply external identifiers for Accounts managed through DWS. These include the Request Initiator and ExternalReferenceNo. Use of these fields is entirely the responsibility of external DWS systems. No validation is performed on these optional fields.

External DWS systems must be aware that if a real individual customer has more than one Account and needs their personal details changed, the details must be changed for each of their Accounts with a separate request.

As noted before, DWS does not provide any templates for Services. This means that separate Accounts for the same Service may legitimately have completely different Contract and Payment terms. It is the responsibility of external DWS systems to enforce consistency as required by the Consumer Company.

For an explanation of the information required to create a new Account, see the Acquisition use cases.

For a detailed description of the information held for Accounts, see the Retrieve Account Records use case.

PAYMENT HISTORY

Each Account has a Payment History which details all financial transactions made against an account.

AUDIT TRAIL

Each Account has an Audit Trail which details all changes that have been made to the information held about an Account.

CALL HISTORY

Each Account has a Call History which details all communications with a Customer about their Account made by Debitsuccess customer services or credit control functions.

ACQUISITION

Acquisition results in a new Account being created.

The DWS method used for all acquisition cases is:

```
public ResponsePostCustomerAccount PostCustomerAccount(RequestPostCustomerAccount request)
```

The `PostCustomerAccount` method is called as follows:

```
var request = new RequestPostCustomerAccount() {

    // Setup Request Properties             //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // Customer / Member | Account / Contract//
    // PAYMENT METHOD                       //
    // PAYMENT METHOD                       //
    // CONTRACT                             //
    // PAYMENT / INVOICING SCHEDULE         //
};
ResponsePostCustomerAccount result = PostCustomerAccount(request);
```

The members of `RequestPostCustomerAccount`:

```
    // Customer / Member | Account / Contract//
    // INDIVIDUAL                           //
    string FirstName;                       // Required
    string MiddleName;                      // Optional
    string LastName;                        // Optional
    string Title;                           // Optional
    DateTime? DateOfBirth;                  // Optional
    Gender Gender;                          // Optional defaults to Unknown

    // ADDRESSES                            //
    string  PhysicalAddress;                // Required Street
    string  PhysicalSuburb;                 // Optional (Required for Australia)
    string  PhysicalCity;                   // Optional (Required for NZ)
    State   PhysicalState;                  // Optional (Required for Australia)
    Country PhysicalCountry;                // Required
    string  PhysicalPostcode;               // Required

    string  BillingAddress;                 // Optional Street
    string  BillingSuburb;                  // Optional (Required for Australia)
    string  BillingCity;                    // Optional (Required for NZ)
    State   BillingState;                   // Optional (Required for Australia)
    Country BillingCountry;                 // Optional
    string  BillingPostcode;                // Optional

    // CONTACTS                             //
    // * Note: At least one contact number; Home, Business or Mobile, MUST be supplied.
    string EmailAddress;                    //   Optional
    string HomeNumber;                      // * Optional
    PhoneCountryCode HomeCountryCode;       // * Optional {NotSpecified|Aus|NZ}
    string HomeSTD;                         // * Optional
    string BusinessNumber;                  // * Optional
    PhoneCountryCode BusinessCountryCode;   // * Optional {NotSpecified|Aus|NZ}
    string BusinessSTD;                     // * Optional
    string EmergencyNumber;                 //   Optional
    PhoneCountryCode EmergencyCountryCode;  //   Optional {NotSpecified|Aus|NZ}
    string EmergencySTD;                    //   Optional
    string MobileNumber;                    //   Optional
    PhoneCountryCode MobileCountryCode;     // * Optional {NotSpecified|Aus|NZ}
    string MobileSTD;                       // * Optional
    string EmergencyName;                   // * Optional

    // PAYMENT METHOD                       //
    string AccountNo;                       // Required
    DateTime? ExpiryDate;                   // Required If using Credit Card
    string AccountHolder;                   // Required Name on card or account
```

```
    AccountType AccountType;                // Required {BankAccount|CreditCard}
    CreditCardType CreditCardType;}         // Required (None|AmericanExpress|Diners|Visa)

    // CONTRACT TERMS                       // Note: See detailed use cases
    DateTime DateAccountStarted;            // Required
    int Term;                               // Required
    TermType TermType;                      // Required {Months|Payments}
    string AccountNotes;                    // Optional
    string ExternalAccountReferenceNo;      // Optional
    bool FixedTerm;                         // Required
    Country AccountCountry;                 // Required
    bool FixTotalValue;                     // Required
    decimal? TotalValue; }                  // Required if FixTotalValue

    // PAYMENT / INVOICING SCHEDULE         // Note: See detailed use cases.
    // Note: See detailed use cases.        // At least payment schedule MUST be supplied
    // One-off Initital payment             //
    DateTime InitialOneOffScheduleStartDate; // Required if InitialOneOffScheduleInstalment>0
    decimal InitialOneOffScheduleInstalment; // Optional *
    string InitialOneOffScheduleDescription; // Optional
    // Recurring Initital payment           //
    DateTime RecurringScheduleStartDate;    // Required if RecurringScheduleInstalment > 0
    decimal RecurringScheduleInstalment;    // Optional *
    Frequency RecurringScheduleFrequency;   // Required if RecurringScheduleInstalment > 0
```

The following Acquisition use case variations differ only in the Contract details and Payment / Invoicing Schedule.

Only properties which could or should be set to anything other than their default are documented per use case.

Payments will be collected by the Payment Method specified.

USE CASE: SINGLE PAYMENT – ACCOUNT REMAINS OPEN

A Customer purchases goods/services from the user with a one off payment.

This is a quite an unusual account creation. It will create an account and bill the customer once. The account will remain open allowing the account to be billed in the future but no further automated payments will be collected.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);                    // Required, future date
AccountNotes = "some notes ...";                                 // Optional
ExternalAccountReferenceNo = "external system reference";        // Optional
AccountCountry = Country.Australia;           // OR Country.NewZealand Required
```

PAYMENT / INVOICING SCHEDULE

```
InitialOneOffScheduleStartDate = new DateTime(yyyy,MM,dd);  // future date payment collected
InitialOneOffScheduleInstalment = 99.99M;                   // amount to collect
InitialOneOffScheduleDescription = "a note";  // Optional note about the payment set-up
```

USE CASE: SINGLE PAYMENT – ACCOUNT CLOSES

A Customer purchases goods/services from the user with a one off payment.

This will create an account and bill the customer once. The account will close after the first single collection.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);                    // Required, future date
AccountNotes = "some notes ...";                                 // Optional
ExternalAccountReferenceNo = "external system reference";        // Optional
AccountCountry = Country.Australia;           // OR Country.NewZealand Required
```

PAYMENT / INVOICING SCHEDULE

```
RecurringScheduleStartDate = new DateTime(yyyy,MM,dd);// date first payment will be collected
RecurringScheduleInstalment = 99.99M;                 // amount collected on each payment
TermType = TermType.Payments;                            // term is counted in #of payments
Term = 1,                                                // runs for one payment only
FixedTerm = true;                                        // payments will end at end of term
```

USE CASE: ONGOING SERVICE

A customer joins a service and makes payments weekly until further notice.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);           // Required, future date
AccountNotes = "some notes ...";                         // Optional
ExternalAccountReferenceNo = "external system reference";// Optional
AccountCountry = Country.Australia;                      // OR Country.NewZealand Required
```

PAYMENT / INVOICING SCHEDULE

```
RecurringScheduleStartDate = new DateTime(yyyy,MM,dd);// date first payment will be collected
RecurringScheduleInstalment = 99.99M;                 // amount collected on each payment
RecurringScheduleFrequency = Frequency.Weekly;        // frequency of collections
FixedTerm = false;                                    // payments will continue
```

USE CASE: TRIAL PERIOD

A customer signs up for a service on a two month trial period and makes payments weekly. At the end of the trial period, the payment collections will stop unless other action is taken.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);           // Required, future date
AccountNotes = "some notes ...";                         // Optional
ExternalAccountReferenceNo = "external system reference";// Optional
AccountCountry = Country.Australia;                      // OR Country.NewZealand Required
TermType = TermType.Months;                              // term is counted in months
Term = 2;                                                // runs for two months
FixedTerm = true;                                        // payments will end at end of term
```

PAYMENT / INVOICING SCHEDULE

```
RecurringScheduleStartDate = new DateTime(yyyy,MM,dd);// date first regular payment will be
                                                      // collected
RecurringScheduleInstalment = 99.99M;                 // amount collected on each payment
RecurringScheduleFrequency = Frequency.Weekly;        // frequency of collections
```

USE CASE: HIRE PURCHASE – EQUAL DEPOSIT AND INSTALLMENTS

A customer buys a product. There is an initial deposit and nine payments, one a month, until the product is paid for. Each payment is 10% of the total.

Note: Because the payments are all equal the Initial One Off payment has not been used.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);            // Required, future date
AccountNotes = "Some notes ...";                         // Optional
ExternalAccountReferenceNo = "External system reference";// Optional
AccountCountry = Country.Australia;                      // OR Country.NewZealand Required
TermType = TermType.Payments;                            // term is counted in #of payments
Term = 10,                                               // runs for ten payments
FixedTerm = true;                                        // payments will end at end of term
FixTotalValue = true;                                    // there is a fixed total amount
TotalValue = 999.90M;                          // the fixed total amount, not to be exceeded
```

PAYMENT / INVOICING SCHEDULE

```
RecurringScheduleStartDate = new DateTime(yyyy, MM, dd );// date of first recurring payment
RecurringScheduleInstalment = 99.99M,                    // amount collected on each payment
RecurringScheduleFrequency = Frequency.Monthly,          // frequency of collections
```

USE CASE: HIRE PURCHASE – NON EQUAL DEPOSIT AND INSTALLMENTS

A customer buys a product. There is an initial deposit of 20% and eight payments, one a month, until the product is paid for. Each payment is 10% of the total.

Note: Because the deposit is not the same as the instalments, the Initial One Off payment has been used.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);            // Required, future date
AccountNotes = "Some notes ...";                         // Optional
ExternalAccountReferenceNo= "External system reference";// Optional
AccountCountry = Country.Australia;                      // OR Country.NewZealand Required
TermType = TermType.Payments;                            // term is counted in payments
Term = 8,                                                // runs for eight payments
FixedTerm = true;                                        // payments will end at end of term
FixTotalValue = true;                                    // there is a fixed total amount
TotalValue = 999.90M;                          // the fixed total amount, not to be exceeded
```

PAYMENT / INVOICING SCHEDULE

```
InitialOneOffScheduleStartDate = new DateTime(yyyy, MM, dd);// date first payment collected
InitialOneOffScheduleInstalment = 199.98M;                  // amount to collect as deposit (20%)
InitialOneOffScheduleDescription = "a note";              //Optional note about the payment set-up
RecurringScheduleStartDate = new DateTime(yyyy, MM, dd );// date first regular payment will
                                                         // be collected
RecurringScheduleInstalment = 99.99M,                    // amount collected on each payment
RecurringScheduleFrequency = Frequency.Monthly,          // frequency of collections
```

USE CASE: CLUB MEMBERSHIP

A customer joins a club which has a $10 joining fee. The minimum term of membership is 12 months with monthly billing of $50. At the end of the term the payment collection will continue until further notice.

Please note that Services may have Establishment fees which are usually paid out of the Initial One Off. Consult your BDM for the nature of Establishment fees.

CONTRACT

```
DateAccountStarted = DateTime.Now.AddDays(1);            // Required, future date
AccountNotes = "Some notes ...";                         // Optional
ExternalAccountReferenceNo = "External system reference";// Optional
AccountCountry = Country.Australia;                      // OR Country.NewZealand Required
TermType = TermType.Payments;                           // term is counted in payments
Term = 9,                                               // runs for two months
FixedTerm = false;                                      // payments will end at end of term
```

PAYMENT / INVOICING SCHEDULE

```
InitialOneOffScheduleStartDate = new DateTime(yyyy, MM, dd);// date first payment collected
InitialOneOffScheduleInstalment = 10.00M;                   // amount to collect as deposit
InitialOneOffScheduleDescription = "a note";      // Optional note about the payment set-up
RecurringScheduleStartDate = new DateTime(yyyy, MM, dd );   // date first regular payment will
                                                            // be collected
RecurringScheduleInstalment = 50.00 M,                  // amount collected on each payment
RecurringScheduleFrequency = Frequency.Monthly,         // frequency of collections
```

USE CASE: HANDLING DIRECT DEBIT TRANSACTION REVERSAL

With direct debit banking the banks can reversed a transaction 72hrs later.

When an account is being charged with a scheduled payment. The amount will be deducted from a customer's bank account on the scheduled day. Debitsuccess will display that payment in a form of a payment either by calling from the api GetPaymentHistoryByAccountId or by daterange and set the account as balanced. 72 hours later, the bank can reversed this payment transaction for various reason, resulting in the client's customer's account to be in an overdue state and a reversal payment being lodged for the account.

1. On the day of the first payment, use GetPaymentHistoryByAccountId and validate that there is a docket with the correct amount added at the correct date and the ReversedPaymentId is set to 0.

```
<a:Payment>
    <a:PaymentAmount>15.00</a:PaymentAmount>
    <a:PaymentErrorCode>NoError</a:PaymentErrorCode>
    <a:PaymentId>1251846248</a:PaymentId>
    <a:PaymentType>DirectDebit</a:PaymentType>
    <a:ReversedPaymentId>0</a:ReversedPaymentId>
</a:Payment>
```

2. Wait for 72 hours and try GetPaymentHistoryByAccountId again, you will notice that this time the account have another docket with a negative amount and is set as reversed.

```
<a:Payment>
    <a:PaymentAmount>15.00</a:PaymentAmount>
    <a:PaymentErrorCode>NoError</a:PaymentErrorCode>
    <a:PaymentId>1251846248</a:PaymentId>
    <a:PaymentType>DirectDebit</a:PaymentType>
    <a:ReversedPaymentId>0</a:ReversedPaymentId>
</a:Payment>
<a:Payment> >
    <a:PaymentAmount>-15.00</a:PaymentAmount>
    <a:PaymentCode>Payment</a:PaymentCode>
    <a:PaymentErrorCode>InsufficientFunds</a:PaymentErrorCode>
    <a:PaymentId>1251846253</a:PaymentId>
    <a:PaymentType>DirectDebit</a:PaymentType>
    <a:ReversedPaymentId>1251846248</a:ReversedPaymentId>
</a:Payment>
```

Scheduled Payment Reversals

The following bank account and credit card numbers can be used to create reversals for scheduled payments in the TEST ENVIROMENT. These Bank Account/Credit Card numbers will result in a reversal of the most recent payment in the last three days. Please use the below credit card number for scheduled payments only, using them for real-time payments will not create reversals payments.

Account Number	Transaction Error
031556012476700	AccountClosed
031556043112600	AutorityStopped
010843005219130	Declined
118003090392230	InvalidAccount
030547063352200	InsufficientFunds

123233059863300	NoAutority

Credit Card Number	**Transaction Error**
378282246310005	Declined
371449635398431	InsufficientFunds
4012001038443335	LostOrStolenCard

USE CASE: RENEWAL OF ACCOUNT

The consumer wants to renewal the customer's subscription with the consumer company for continuation of services or product.

This use case applies to a fixed term account, the account has reached termed or about to reach term and the customer's balance with the consumer company is nearing zero.

The consumer company can renew an account by duplicating the account details such as the account's customer, address, contacts and pay method; these will be copied into a new account. New payment and invoicing schedule will be required to be supplied for the new account.

For this use case the consuming company must also consider the following business process, the consuming company must determine when the account has reached term as the api is agnostic to whether it used as a renewal use case or not

- Consumer's customer has paid in full.
- Consumer's customer does not wish to renewal after term.

```
public ResponsePostCustomerAccount
PostCustomerAccountForExistingCustomer(PostCustomerAccountForExistingCustome request)

 // Setup Request Properties              //
 User = new User() { Username="yourUsername", Password="yourPassword" },
 ContractPrefix = "yourServicePrefix",
 RequestInitiator = "an external identifier",

 // CONTRACT TERMS                        // Note: See detailed use cases
 DateTime DateAccountStarted;             // Required
 int Term;                                // Required
 TermType TermType;                       // Required {Months|Payments}
 string AccountNotes;                     // Optional
 string ExternalAccountReferenceNo;       // Optional
 bool FixedTerm;                          // Required
 Country AccountCountry;                  // Required
 bool FixTotalValue;                      // Required
 decimal? TotalValue; }                   // Required if FixTotalValue
 bool? WaiveEstFee;                       // Optional, if true then the establishment fee
for the new account will be waived in case the facility charges the customer by default.


 // PAYMENT / INVOICING SCHEDULE          // Note: See detailed use cases.
 // Note: See detailed use cases.         // At least payment schedule MUST be supplied
 // One-off Initital payment              //
 DateTime InitialOneOffScheduleStartDate; // Required if InitialOneOffScheduleInstalment>0
 decimal InitialOneOffScheduleInstalment; // Optional *
 string InitialOneOffScheduleDescription; // Optional
 // Recurring Initital payment            //
 DateTime RecurringScheduleStartDate;     // Required if RecurringScheduleInstalment > 0
 decimal RecurringScheduleInstalment;     // Optional *
 Frequency RecurringScheduleFrequency;    // Required if RecurringScheduleInstalment > 0
```

USE CASE: CUSTOMER HAS PURCHASE ADDITIONAL SERVICES OR PRODUCT

A consumer company has sold an existing customer additional services or product. If a customer has an account previously loaded for a service of product, the consumer company can duplicate the account, details such as the account's customer, address, contacts and pay method these will be copied into a new account. New payment and invoicing schedule will be required to be supplied for the new account.

```
   public ResponsePostCustomerAccount
   PostCustomerAccountForExistingCustomer(PostCustomerAccountForExistingCustome request)

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // CONTRACT TERMS                        // Note: See detailed use cases
    DateTime DateAccountStarted;             // Required
    int Term;                                // Required
    TermType TermType;                       // Required {Months|Payments}
    string AccountNotes;                     // Optional
    string ExternalAccountReferenceNo;       // Optional
    bool FixedTerm;                          // Required
    Country AccountCountry;                  // Required
    bool FixTotalValue;                      // Required
    decimal? TotalValue; }                   // Required if FixTotalValue
    bool? WaiveEstFee;                       // Optional, if true then the establishment fee
for the new account will be waived in case the facility charges the customer by default.

    // PAYMENT / INVOICING SCHEDULE          // Note: See detailed use cases.
    // Note: See detailed use cases.         // At least payment schedule MUST be supplied
    // One-off Initital payment              //
    DateTime InitialOneOffScheduleStartDate; // Required if InitialOneOffScheduleInstalment>0
    decimal InitialOneOffScheduleInstalment; // Optional *
    string InitialOneOffScheduleDescription; // Optional
    // Recurring Initital payment            //
    DateTime RecurringScheduleStartDate;     // Required if RecurringScheduleInstalment > 0
    decimal RecurringScheduleInstalment;     // Optional *
    Frequency RecurringScheduleFrequency;    // Required if RecurringScheduleInstalment > 0
```

FULFILMENT

Fulfilment use case result in the information held on, or the state of a Customer Account.

The Account will be closes immediately and not further payments will be collected.

USE CASE: CANCEL ACCOUNT

```
public ResponseAccountCancellation CancelAccount(RequestAccountCancellation request)

var request = new RequestAccountCancellation() {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    CancellationNote = "a note"                  // a note about the cancellation
};
ResponseAccountCancellation result = CancelAccount(request);
```

USE CASE: CHANGE CUSTOMER INDIVIDUAL DETAILS

Only the attributes that are required to change need to be set in the request. As long as the other attributes are not set and are passed effectively as null, the currently recorded values will remain.

```
public ResponseAdjustAccountClientDetails
    UpdateClientDetails(RequestAdjustAccountClientDetails request)

var request = new RequestAdjustAccountClientDetails() {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // INDIVIDUAL                            // All fields are optional
    FirstName = "newfirstname",              // Optional
    MiddleName = "newmiddlename",            // Optional
    LastName = "newlasttname",               // Optional
    Title = "newtitle",                      // Optional
    DateOfBirth = new DateTime(),            // Optional
    Gender Gender = Gender.Male              // Optional
};
ResponseAdjustAccountClientDetails result = UpdateClientDetails(request);
```

USE CASE: CHANGE CUSTOMER ADDRESS

The whole address record must be entered. The original address of the type entered will be marked as previous and held in the account history.

```
public ResponseAdjustAccountClientAddress
    UpdateClientAddress (RequestAdjustAccountClientAddress request)

var request = new RequestAdjustAccountClientAddress () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // ADDRESS                               // All fields must be set
    Address = "newstreet",                   // Required Street
    Suburb = "newsuburb",                    // Optional (Required for Australia)
    City = "newcity",                        // Optional (Required for NZ)
    State = State.Queensland,                // Optional (Required for Australia)
    Country = "newcountry",                  // Required
    Postcode = "newpostcode"                 // Required
    AddressType = AddressType.Home           // Required
};
ResponseAdjustAccountClientAddress result = UpdateClientAddress(request);
```

USE CASE: CHANGE CUSTOMER EMAIL

```
public ResponseAdjustAccountClientEmail
   UpdateClientEmailAddress(RequestAdjustAccountClientEmail request)

var request = new RequestAdjustAccountClientEmail() {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    EmailAddress = "newemail@address.not”     // Required
};
ResponseAdjustAccountClientEmail result = UpdateClientEmailAddress(request);
```

USE CASE: CHANGE CUSTOMER PHONE NUMBER

The whole phone number record must be set, except for the Name field where it is optional. (Name is required for Emergency phone type). The original phone number of the type entered will be marked as previous and held in the account history.

```
public ResponseAdjustAccountClientPhone
   UpdateClientPhoneNumber(RequestAdjustAccountClientPhone request)

var request = new RequestAdjustAccountClientPhone () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // PHONE NUMBER                          // All fields must be set
    Number = "anewnumber",                   // Required
    CountryCode = PhoneCountryCode.NZ,       // Required
    StdCode = "anewstd",                     // Required
    PhoneNumberType = PhoneType.Home,        // Required
    Name = "anewname"                        // Required for Emergency phone type

 };
ResponseAdjustAccountClientPhone result = UpdateClientPhoneNumber(request);
```

USE CASE: CHANGE ACCOUNT TOTAL VALUE

The Total Value of an Account is the minimum amount the Customer must pay before the Account can be closed without further penalties. All Accounts, whatever their Contract or Payment Schedule have this value stored in the MinTermTotalValue field.

When the Account Total Value is changed, the Account is automatically made a Fixed Term account. Collections will end once the Total Value has been paid. The Account will remain open, but automatically close when it reaches the end of the Contract period.

```
public ResponseAdjustAccountTotalValue
   AdjustAccountTotalValue(RequestAdjustAccountTotalvalue request)

var request = new RequestAdjustAccountTotal () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Account Total Value                   //
    CurrentTotalValue = 999.99M,             // Required
    NewTotalValue = 888.88M                  // Required
 };
ResponseAdjustAccountTotalValue result = AdjustAccountTotalValue(request);
```

USE CASE: CHANGE ACCOUNT: CHANGE PAYMENT METHOD

Change the Payment method used to collect payments for an Account.

```
public ResponseAdjustAccountPaymentMethod
   ChangePaymentMethod(RequestAdjustAccountPaymentMethod request)

var request = new RequestAdjustAccountPaymentMethod () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Method                            //
    AccountNo = "bank or credit card no",        // Required
    AccountHolder = "nameonbankaccorcreditcard",// Required
    AccountType = AccountType.CreditCard,        // Required
    CreditCardType = CreditCardType.Visa,        // Required for credit card
    ExpiryDate = new DateTime()                  // Required for credit card
};
ResponseAdjustAccountPaymentMethod result = RequestAdjustAccountPaymentMethod(request);
```

USE CASE: CHANGE ACCOUNT: CHANGE TO FIXED TERM

** Not Currently Implemented **

Change a customer's account to fixed term from on-going. Once this change is made the account will be closed when the total balance is collected.

```
public ResponseAdjustAccountFixedTerm
   AdjustAccountFixedTerm(RequestAdjustAccountFixedTermByAccountId request)

var request = new RequestAdjustAccountFixedTermByAccountId () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
};
ResponseAdjustAccountFixedTerm result = AdjustAccountFixedTerm(request);
```

USE CASE: CHANGE ACCOUNT: INCREASE PAYMENT AND SHORTEN TERM

** Not Currently Implemented **

Increase the amount a customer is paying and set the account to term so the amount owed is paid off sooner.

```
public ResponseAdjustPaymentInAdvance
   AdjustPaymentInAdvance(RequestAdjustPaymentInAdvance request)

var request = new RequestAdjustAccountTotal () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Increase Payment                          //
    IncreasedPaymentAmount = 9.99M,              // Required
    EndDate = new DateTime()                     // Optional
};
ResponseAdjustPaymentInAdvance result = AdjustPaymentInAdvance(request);
```

USE CASE: CHANGE ACCOUNT: STOP PAYMENT COLLECTION

Stop collection of payments on an Account for a period of time.

Payments owing will continue to accrue based on payment schedules irrespective of a stop in customer's payments. To stop the accumulation of payments owing, an Account suspension should be used instead.

The `StopCreditControlLetters` property controls whether overdue letters will be sent to the customer for the period that payments are stopped.

```
public ResponseStopPayment
   StopPayment(RequestStopPayment request)

var request = new RequestStopPayment () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Stop Payment                          //
    StopPaymentUntil = new DateTime(),       // Required
    StopCreditControlLetters = 0             // Required (0=Letters Sent|1=No Letters Sent)
};
ResponseStopPayment result = StopPayment(request);
```

USE CASE: CHANGE ACCOUNT: RESUME PAYMENT COLLECTION

Resume collections on an Account after a Stop Payment.

```
public ResponseResumePayment
   ResumePayment(RequestResumePayment request)

var request = new RequestResumePayment () {

    // Setup Request Properties             //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
    ////////////////////////////////////////
};
ResponseResumePayment result = ResumePayment(request);
```

USE CASE: CHANGE ACCOUNT: SUSPEND PAYMENT COLLECTION

Suspending an Account stops payment collection and stops accrual of charges as per the Payment Schedule.

A Suspension fee may be levied in lieu of the standard Account fees. The Suspension fee is in fact a new category of Regular Payment Schedule called a Suspension Schedule and as such may be charged periodically during the Suspension period.

A Suspension may be ended using the Resume Payment Collection case.

There are two methods of putting an account into Suspension.

- Suspend an Account for a fixed number of payment cycles.

```
public ResponseSuspendAccount
   SuspendAccountForNumberOfPaymentCycles (RequestSuspendAccount request)

var request = new RequestSuspendAccount () {

    // Setup Request Properties             //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
    ////////////////////////////////////////

    // Suspend Account Collection for Number of Payment Cycles //
    MinimumEffectiveDate = new DateTime(),   // Required: when suspension starts
    NumberOfPaymentCycles = 4,               // Required
    SuspensionFee = 9.99M,                   // Required: (may be zero)
    SuspensionFeeFrequency = Frequency.Weekly// Required: if SuspensionFee > 0.00M
};
ResponseSuspendAccount result = ForNumberOfPaymentCycles(request);
```

- Suspend an Account between two dates.

```
public ResponseSuspendAccount
   SuspendAccountBetweenDates(RequestSuspendAccountBetweenDates request)

var request = new RequestSuspendAccountBetweenDates () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
    ////////////////////////////////////////

    // Suspend Account Collection for Number of Payment Cycles //
    SuspensionStartDate = new DateTime(),    // Required: when suspension starts
    SuspensionEndDate = new DateTime(),      // Optional: when suspension ends, if null then
open ended
    SuspensionFee = 9.99M,                   // Required: (may be zero)
    SuspensionFeeFrequency = Frequency.Weekly// Required: if SuspensionFee > 0.00M
};
ResponseSuspendAccount result = SuspendAccountBetweenDates (request);
```

USE CASE: CHANGE SUSPENSION END DATE

Reset the end date of a given suspension. If the end date is null the suspension will become an open ended suspension.

```
public ResponseAdjustSuspensionEndDate
   AdjustSuspensionEndDate (RequestAdjustSuspensionEndDate request)

var request = new RequestAdjustSuspensionEndDate() {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
    ////////////////////////////////////////////
    PayScheduleId = "Schedule id of suspension",
    NewEndDate = "optional new end date"


};
ResponseAdjustSuspensionEndDate result = AdjustSuspensionEndDate(request);
```

USE CASE: REFUND AN ACCOUNT

Request a refund to an Account.

The amount to refund can be any value up to the total amount collected through the Account to date.

```
public ResponseInitiateRefund
   InitiateRefund(RequestInitiateRefund request)

var request = new RequestInitiateRefund () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",
```

```
    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Account Total Value                  //
    Amount = 999.99M,                       // Required
    Description = "a reason for the refumd"  // Required
};
ResponseInitiateRefund result = InitiateRefund(request);
```

USE CASE: CHANGE ACCOUNT: CHANGE PAYMENT SCHEDULE FREQUENCY

To change the current payment frequency of an account, create a new payment schedule with the specified frequency. For example, to change the payment frequency of an account from weekly to fortnightly create a new schedule with the `NewPaySchedule.PaymentFrequency` property set to `Frequency.Fortnightly`. The `NewPaySchedule.MinimumEffectiveDate` property specifies the earliest date that the schedule should start but the actual start date will be the first date after this that an existing schedule is set to bill.

The `DeleteFutureSchedules` property will cause any schedules that have been set-up to come into effect after this schedule to be discarded and replaced with this schedule. Currently, true is the only valid value for this property. Future functionality will allow a false setting which will have the new schedule become active and run until another, already defined schedule, replaces it in the future.

The `AdjustFutureSchedules` property will ensure the frequency of future, already defined schedules, are set to the frequency defined in this new schedule. This functionality is not implemented yet so this property should currently always be set to false.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 99.99M,                           // Required amount to collect
        PaymentFrequency = Frequency.Fortnightly,       // Required
        MinimumEffectiveDate = new DateTime(),          // Required when the schedule starts
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = true,                       // Required only true is valid
    AdjustFutureSchedules = false                       // Required only false is valid
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE ACCOUNT: CHANGE PAYMENT SCHEDULE INSTALMENT

To change the current payment instalment of an account, create a new payment schedule with the specified instalment. For example, to change the payment instalment of an account from $5 to $10, create a new schedule with the `NewPaySchedule.Installment` property set to `10`. The `NewPaySchedule.MinimumEffectiveDate` property specifies the earliest date that the schedule should start but the actual start date will be the first date after this that an existing schedule is set to bill.

The `DeleteFutureSchedules` property will cause any schedules that have been set-up to come into effect after this schedule to be discarded and replaced with this schedule. Currently, true is the only valid value for this property. Future functionality will allow a false setting which will have the new schedule become active and run until another, already defined schedule, replaces it in the future.

The `AdjustFutureSchedules` property will ensure the frequency of future, already defined schedules, are set to the frequency defined in this new schedule. This functionality is not implemented yet so this property should currently always be set to false.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 10.00M,                          // Required amount to collect
        PaymentFrequency = Frequency.Fortnightly,      // Required
        MinimumEffectiveDate = new DateTime(),         // Required when the schedule starts
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = true,                      // Required only true is valid
    AdjustFutureSchedules = false                      // Required only false is valid
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE ACCOUNT: CHANGE PAYMENT SCHEDULE IN THE FUTURE

To change an account's payment schedule at a future date, create a new payment schedule with the specified minimum effective date set to the earliest date that the schedule should come into effect. For example, to change an account that bills weekly to bill monthly after 4 weeks have passed, create a new schedule with the `NewPaySchedule.MinimumEffectiveDate` property set to 4 weeks in the future and `NewPaySchedule.PaymentFrequency` property set to `Frequency.Monthly`. The new schedule will come into effect on the first payment date after the minimum effective date that an existing schedule is set to bill.

The `DeleteFutureSchedules` property will cause any schedules that have been set-up to come into effect after this schedule to be discarded and replaced with this schedule. Currently, true is the only valid value for this property. Future functionality will allow a false setting which will have the new schedule become active and run until another, already defined schedule, replaces it in the future.

The `AdjustFutureSchedules` property will ensure the frequency of future, already defined schedules, are set to the frequency defined in this new schedule. This functionality is not implemented yet so this property should currently always be set to false.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 10.00M,                           // Required amount to collect
        PaymentFrequency = Frequency.Monthly,           // Required
        MinimumEffectiveDate = DateTime.Today           // Required when the schedule starts
                                        .AddDays(28),
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = true,                       // Required only true is valid
    AdjustFutureSchedules = false                       // Required only false is valid
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE ACCOUNT: BRING PAYMENT SCHEDULE PAYMENT DAY FORWARD

To change the current payment day to a date prior to the next payment date, create a new payment schedule and set the `OverrideBillingCycleAlignment` property to true. The newly created schedule will start on the `NewPaySchedule.MinimumEffectiveDate` provided and will not automatically be aligned with an existing schedule. When setting the `OverrideBillingCycleAlignment` property to true the `PreviousScheduleEndDate` property must also be set to close the payment schedule immediately prior appropriately.

The `DeleteFutureSchedules` property will cause any schedules that have been set-up to come into effect after this schedule to be discarded and replaced with this schedule. Currently, true is the only valid value for this property. Future functionality will allow a false setting which will have the new schedule become active and run until another, already defined schedule, replaces it in the future.

The `AdjustFutureSchedules` property will ensure the frequency of future, already defined schedules, are set to the frequency defined in this new schedule. This functionality is not implemented yet so this property should currently always be set to false.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 99.99M,                           // Required amount to collect
        PaymentFrequency = Frequency.Fortnightly,       // Required
        MinimumEffectiveDate = DateTime.Today           // Required when the schedule starts
                                        .AddDays(5),
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = true,                       // Required only true is valid
    AdjustFutureSchedules = false,                      // Required only false is valid
    OverrideBillingCycleAlignment = true,               // Optional
    PreviousScheduleEndDate = DateTime.Today            // Required for ...
                                      .AddDays(4)       // ...OverrideBillingCycleAlignment
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE ACCOUNT: MOVE PAYMENT SCHEDULE PAYMENT DAY BACKWARD

To change the current payment day to a date after to the next payment date, create a new payment schedule and set the `OverrideBillingCycleAlignment` property to true. The newly created schedule will start on the `NewPaySchedule.MinimumEffectiveDate` provided and will not automatically be aligned with an existing schedule. When setting the `OverrideBillingCycleAlignment` property to true the `PreviousScheduleEndDate` property must also be set to close the payment schedule immediately prior appropriately. In the scenario where the new schedule is to start soon after the next payment date it will likely be appropriate to set the `PreviousScheduleEndDate` property prior to the next payment date. This will prevent the account being billed twice in a short time frame. For example, when changing the payment schedule from weekly on Wednesdays to weekly on Thursdays the `PreviousScheduleEndDate` might be set to the forthcoming Tuesday.

The `DeleteFutureSchedules` property will cause any schedules that have been set-up to come into effect after this schedule to be discarded and replaced with this schedule. Currently, true is the only valid value for this property. Future functionality will allow a false setting which will have the new schedule become active and run until another, already defined schedule, replaces it in the future.

The `AdjustFutureSchedules` property will ensure the frequency of future, already defined schedules, are set to the frequency defined in this new schedule. This functionality is not implemented yet so this property should currently always be set to false.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 99.99M,                           // Required amount to collect
        PaymentFrequency = Frequency.Fortnightly,       // Required
        MinimumEffectiveDate = DateTime.Today           // Required when the schedule starts
                                        .AddDays(10),
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = true,                       // Required only true is valid
    AdjustFutureSchedules = false,                      // Required only false is valid
    OverrideBillingCycleAlignment = true,               // Optional
    PreviousScheduleEndDate = DateTime.Today            // Required for ...
                                                        // ...OverrideBillingCycleAlignment
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE ACCOUNT: EXTERNAL ACCOUNT REFERENCE NUMBER

Every account can have an external account reference number which can be changed with the following message.

```
public ResponseAdjustAccountExternalAccountReferenceNo
   AdjustAccountExternalAccountReferenceNo (RequestAdjustAccountExternalAccountReferenceNo
request)

var request = new RequestAdjustAccountExternalAccountReferenceNo () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////
    NewExternalAccountReferenceNo = "the new External Account Reference",
};
ResponseAdjustAccountExternalAccountReferenceNo result =
AdjustAccountExternalAccountReferenceNo(request);
```

USE CASE: CHANGE ACCOUNT: NEW ONE-OFF PAYMENT SCHEDULE

Create a new one-off payment schedule for an Account, to be billed at some point in the future. The `NewPaySchedule.Frequency` property must be set to `Frequency.OneOff` and `NewPaySchedule.MinimumEffectiveDate` property to the precise date that the one-off payment will be taken.

The `DeleteFutureSchedules`, `AdjustFutureSchedules` and `NewPaySchedule.EndDate` properties are invalid when adding a one-off schedule.

```
public ResponseCreateSchedule
   CreateSchedule(RequestCreateSchedule request)

var request = new RequestCreateSchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Schedule                                  //
    NewPaySchedule = new RequestPaySchedule() {
        Installment = 99.99M,                          // Required amount to collect
        PaymentFrequency = Frequency.OneOff,           // Required
        MinimumEffectiveDate = new DateTime(),         // Required when payment will collect
        Description = "a note about the new schedule" // Optional
    },
    DeleteFutureSchedules = false,                     // Required only false is valid
    AdjustFutureSchedules = false                      // Required only false is valid
};
ResponseCreateSchedule result = CreateSchedule(request);
```

USE CASE: CHANGE PAYMENT SCHEDULES: DATE OF PAYMENT

** Not Currently Implemented **

Change the date payment collections will occur on an Account.

This will change the current and any future Payment Schedules that have been defined for the Account.

This function has slightly different effects depending on what Payment Frequency the Payment Schedules have set:

- If the Payment Frequency is in {Weekly, Fortnightly, Four Weekly}, the New Payment Date will define the day of the **week** the payment will be collected.
- If the Payment Frequency is in {Monthly, Bi Monthly, Quarterly}, the New Payment Date will define the day of the **month** the payment will be collected.

The date specified as the New Payment Date must be within the next payment cycle for the Account.

```
public ResponseAdjustDateOfPayment
   AdjustDateDayOfPayment(RequestAdjustDateOfPayment request)

var request = new RequestAdjustDateOfPayment () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // New Payment Date                      //
    NewPaymentDate = new DateTime()          // Required
};
ResponseAdjustDateOfPayment result = AdjustDateDayOfPayment(request);
```

USE CASE: CHANGE ACCOUNT: NEXT PAYMENT COLLECTION DATE

Change the date of the next payment collection for an Account.

This will only effect the next collection date and will have no effect on existing Payment Schedules.

```
public ResponseAdjustNextPaymentDate
   AdjustNextPaymentDate(RequestAdjustNextPaymentDate request)

var request = new RequestAdjustNextPaymentDate () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Next Payment Date                     //
    NextPaymentDate = new DateTime()         // Required > DateTime.Now
};
ResponseAdjustNextPaymentDate result = AdjustNextPaymentDate(request);
```

USE CASE: CHANGE ACCOUNT: DELETE PAY SCHEDULE

Allows to delete a pay schedule in case it hasn't started yet and it's not the only one of the account.

```
public ResponseDeletePaySchedule
   DeletePaySchedule(RequestDeletePaySchedule request)

var request = new RequestDeletePaySchedule () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    PayScheduleId = "Schedule id",
};
ResponseDeletePaySchedule result = DeletePaySchedule (request);
```

USE CASE: CHANGE PAYMENT COLLECTION: ADJUST CATCH UP AMOUNT

**** Not Currently Implemented ****

The Catch Up amount is an extra charge levied at each Payment Collection, in addition to the charge specified in the current effective Payment Schedule. The purpose of the Catch Up is to provide a mechanism for individual Accounts that fall into arrears, to catch up their payments to the contracted level. There is not automation in the catch up functionality. It is the responsibility of external DWS systems to manage the application and removal of catch ups.

```
public ResponseAdjustCatchUpPayment
   AdjustCatchUpPayment(RequestAdjustCatchUpPayment request)

var request = new RequestAdjustCatchUpPayment () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Catch-up amount                                  //
    CatchUpAmount = 9.99M,                              // Required
    CatchUpEndDate = new DateTime()                     // Optional
};
ResponseAdjustCatchUpPayment result = AdjustCatchUpPayment(request);
```

USE CASE: RECORD A PAYMENT RECEIVED FOR AN ACCOUNT OUTSIDE THE DEBITSUCCESS BILLING SYSTEM

A Consumer Company might collect a payment for an Account directly from the Customer and outside the Debitsuccess billing system. This payment still needs to be recorded against the Account in the Debitsuccess system.

```
public ResponsePaymentCollected
   PaymentCollected(RequestPaymentCollected request)
```

```
var request = new RequestPaymentCollected () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Payment Collected                                //
    Amount = 9.99M,                                     // Required
    DateOfPayment = new DateTime(),                     // Required
    Description = "a note about payment",               // Optional
};
ResponsePaymentCollected result = PaymentCollected(request);
```

USE CASE: INITIATE A REAL TIME CREDIT CARD PAYMENT FOR AN ACCOUNT

This message is used to initiate a real time card payment for an account using the Debitsuccess billing system as the payment services provider. This service is only available in certain merchant scenarios; please consult with your Debitsuccess Business Development Manager or Account Manager to discuss.

The ExternalTransactionIdentifier is used to uniquely identify the transaction and must be supplied by the client. Once a transaction is initiated, GetCardPaymentStatusForCustomerAccount is used with the ExternalTransactionIdentifier to retrieve the result of the transaction.

Debitsuccess is generally required to provide a payment service which has 3D Secure enabled. (see "Get Card Payment Status For Customer Account" section for details). This means that a client application must be able to submit and receive HTTP POSTs. This suggests that client applications will usually be web browser applications. Other client application architectures are possible but not as simple to implement.

In particular business cases, it may be possible to negotiate payment facilities which do not have 3D Secure enabled.

The response will contain one of three `TransactionStatus` values:

- **Processing** status means that the request has certainly been passed to the payment services provider.
- **ServiceBusy** status. The payment service provider is experiencing loading problems and cannot accept the transaction for processing. A transaction has certainly not been initiated and another call to ProcessCardPaymentForCustomerAccount with the same details is permitted.
- **NotSubmitted** status. This status will only be returned in the context of validation problems in the details passed or a wider problem with the Debitsuccess web services. The enclosing message will contain an error code and a note which should be reviewed. If the error code is "06" Debitsuccess technical support should be contacted. A transaction has certainly not been initiated.

ServiceBusy is anticipated to be a very rare occurrence for the client applications. The ServiceBusy status is supposed to be transient and indicates loading problems at the payment service provider. The interface between Debitsuccess and the payment services provider buffers ServiceBusy statuses by automatically retrying the transaction submission. Meanwhile, the client application will continue to receive Processing status. Only after retrying a number of times (currently configured as 6 times with 10 second intervals), the Debitsuccess web service will give up and return the ServiceBusy to the client application. In this event the client could ask the user to try again, or the transaction could be abandoned.

```
public ResponseProcessCardPaymentForCustomerAccount
   ProcessCardPaymentForCustomerAccount (RequestProcessCardPaymentForCustomerAccount request)

var request = new RequestProcessCardPaymentForCustomerAccount () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Card Payment Details                                                  //
    ExternalTransactionIdentifier = "theExternalTransactionReference",      // Required
    AccountHolder = "Card account holder name",                             // Required
    AccountHolder = "Card account holder name",                             // Required
    Amount = 9.99M,                                                         // Required
    CVC = "(3-4 digit) Card verification/security code",                    // Required
    CardNumber = " Credit/debit card number ",                              // Required
    CreditCardType = CreditCardType.Visa,                                   // Required
```

```
    ExpiryDate = new DateTime(),                                    // Required
    Notes = " Any notes to be recorded against the payment ",       // Optional
    PayerAuthorisationResponse = "Value returned from 3DS HTTP POST",   // Required ONLY IF
responding to a Requires3Ds.
};
ResponsePaymentCollected result = PaymentCollected(request);
```

Note: Debitsuccess recommends that client organisations never store or log the Credit Card Number or CVC in any type of persistent storage. As a matter of good practice, this precaution helps safeguard clients' users' credit card information. Debitsuccess is a Level 1 PCIDSS compliant payment services provider. For further information about PCIDSS compliance please contact your Debitsuccess Business Development Manager or Account Manager.

USE CASE: GET CARD PAYMENT STATUS FOR CUSTOMER ACCOUNT

This message is used to retrieve the status of a card payment transaction initiated using ProcessCardPaymentForCustomerAccount after a Processing status has been received. Debitsuccess's current payment services provider usually produces a definitive result within 5 seconds.

There is a very small chance that a non-definitive status may be returned.

With 3D Secure enabled (the default), the client application may be required to follow the 3D Secure process, if a `TransactionStatus.Requires3ds` is returned.

The response will contain one of these `TransactionStatus` values:

- **Processing**. Waiting for a reply.
- **Authorised**. Payment have been authorised and recorded against the Debitsuccess account.
- **Declined**. Payment has been declined by the Debitsuccess payment services provider.
- **Requires3ds**. Client application must follow the 3DSecure procedure (see below).
- **StatusRequired**. This is a serious error condition which may be returned by the Debitsuccess payment services provider. It is anticipated that it will be an extremely rare case that a client application sees this status. The Debitsuccess payment services provider is saying that they are unable to return any information about the transaction. This includes whether the transaction is in progress, has been accepted or declined. This is supposed to be a transient condition dependent on the internal state of the Debitsuccess payment services provider's infrastructure. The Debitsuccess interface with our payment services provider, buffers the StatusRequired status by repeating get status requests over a period of time. Meanwhile, the client application will receive Processing status. Only after retrying a number of times (currently configured as 6 times with 10 second intervals), the Debitsuccess web service will give up and return the StatusRequired status to the client application. ***Should this ever happen, it is important to contact Debitsuccess technical support with the ExternalTransactionIdentifier. There is a risk that a payment has been processed successfully but not recorded against the account. The client's user should be informed that there has been a technical problem with their payment and that they will be contacted regarding it.*** It is always possible to call GetCardPaymentStatusForCustomerAccount again after StatusRequired has been returned. Any call *may* resolve the problem but a call to technical support is still recommended.

```
public ResponseProcessCardPaymentForCustomerAccount
   GetCardPaymentStatusForCustomerAccount (RequestGetCardPaymentStatusForCustomerAccount,
   request)

var request = new RequestGetCardPaymentStatusForCustomerAccount () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
```

```
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference",
    ////////////////////////////////////////

    // Card Payment Get Status Details                                    //
    ExternalTransactionIdentifier = "theExternalTransactionReference",    // Required
};
ResponsePaymentCollected result = PaymentCollected(request);
```

3D Secure Process

Should a Requires3Ds status be returned, our payment services provider requires the client application to obtain a token that proves the user has supplied additional security information to a system that is independent of the client application.

In this case, the PayerAuthenticationUrl and PayerAuthenticationRequest values in the response are returned populated.

The client application must then make an HTTP POST to the PayerAuthenticationUrl, including the PayerAuthenticationRequest as a POST parameter.

The HTTP POST to the PayerAuthenticationUrl has these POST parameters:

- **PaReq:** PayerAuthenticationRequest returned in `ResponseProcessCardPaymentForCustomerAccount`
- **TermUrl:** Url hosted by the client application which can receive HTTP POST with parameters. The client application will receive a PaRes which is the token that is used to complete the transaction.
- **MD:** An optional value which the client application can supply and which will be returned in the HTTP POST to TermUrl. This is its only significance.

Html Fragment which could be used to post to the 3D Secure verification page:

```
<form method="POST" action="{PayerAuthenticationUrl}">
<input type="hidden" name="PaReq" value="{PayerAuthenticationRequest}">
<input type="hidden" name="TermUrl" value="{Client App hosted URL which can receive HTTP POST parameters">
<input type="hidden" name="MD" value="{Optional value which will be returned as POST parameter to TermUrl">
</form>
```

Once the user has visited the 3D Secure authentication page, an HTTP POST will be made back to the TermUrl where the client application can extract the PaRes value and optionally the MD value from the POST parameters.

Once the client application has obtained the PaRes value, it can use the same details to recall the ProcessCardPaymentForCustomerAccount request, but with the PayerAuthorisationResponse set to the PaRes value.

Recalling the ProcessCardPaymentForCustomerAccount follows the same pattern as the original call. Once a Processing status is returned, GetCardPaymentStatusForCustomerAccount must be used to retrieve the final status of the transaction.

Test Data

The following Visa card numbers can be used with the Debitsuccess development web services. Other card details may result in a declined transaction. Valid until date must be in the future. Any or no cvc can be passed.

4111111111111111	Always returns Authorised.
4999999999999990	Always returns Declined.
4999999999999236	Always returns Declined.
4999999999999269	Always returns Declined.
4999999999999202	Always returns ServiceBusy.
4012001036275556	Uses 3D Secure, returns ServiceBusy.*
4012001038443335	Does not use 3D Secure, returns Authorised.*
4012001037141112	Uses 3D Secure, returns Authorised.*
4005559876540	Uses 3D Secure, returns Authorised.*
4012001037167778	Uses 3D Secure, returns Authorised.*
4012001037141369	Uses 3D Secure, returns Authorised.*
4024007187806731	Returns StatusRequired.

Any other numbers, or details presented with invalid end dates will get declined transactions.

Note: 3D Secure is not enabled in our platform as it is not required by our payment processor. The 3D secure cards above* may not behave as expected.

ASSURANCE

Assurance use cases involve retrieving information held about Accounts and activities relating to them from the Debitsuccess systems.

USE CASE: RETRIEVE ACCOUNT RECORDS

Retrieving Account records provides a complete snapshot of the information currently held for an Account.

The `ResponseRetrieveCustomerAccounts` is a complex data structure summarised in Figure 1.

In the Accounts collection, there will be one item for each Account. This will be the current information held by Debitsuccess as of the latest of three possible dates recorded in the item; the `DateAccountLoaded`, `LastUpdateDate` or the `DateAccountClosed`.

There are two methods for getting Account records from the DWS.

- Return a single record based on the `AccountReferenceNo` or `ExternalReferenceNo`.

```
public ResponseRetrieveCustomerAccounts
   RetrieveCustomerAccountsByAccountId(RequestRetrieveCustomerAccountsByAccountId request)

var request = new RequestRetrieveCustomerAccountsByAccountId () {

    // Setup Request Properties             //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
};
ResponseRetrieveCustomerAccounts result = RetrieveCustomerAccountsByAccountId(request);
```

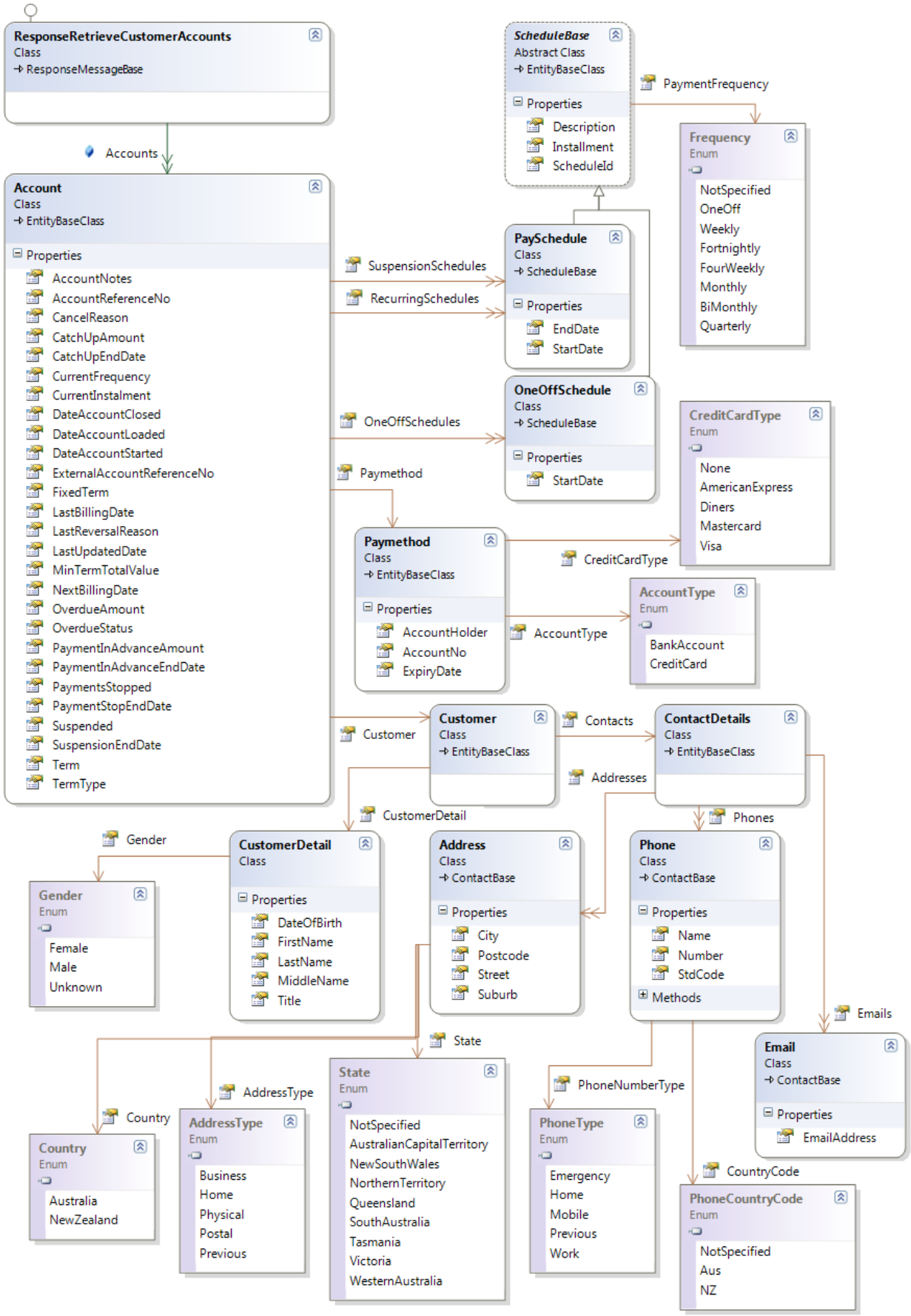
- Return a number of Accounts where their `DateAccountLoaded`, `LastUpdateDate` or `DateAccountClosed` properties fall within a supplied range of dates.

```
public ResponseRetrieveCustomerAccounts
   RetrieveCustomerAccountsForDateRange(RequestRetrieveCustomerAccountsForDateRange request)

var request = new RequestRetrieveCustomerAccountsForDateRange () {

    // Setup Request Properties             //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // Date Range                           //
    StartDate = new DateTime(),             // Required
    EndDate = new DateTime()                // Optional
};
ResponseRetrieveCustomerAccounts result = RetrieveCustomerAccountsForDateRange(request);
```


ResponseRetrieveCustomerAccounts
Class
ResponseMessageBase
Accounts
Account
Class
EntityBaseClass
Properties
AccountNotes
AccountReferenceNo
CancelReason
CatchUpAmount
CatchUpEndDate
CurrentFrequency
CurrentInstalment
DateAccountClosed
DateAccountLoaded
DateAccountStarted
ExternalAccountReferenceNo
FixedTerm
LastBillingDate
LastReversalReason
LastUpdatedDate
MinTermTotalValue
NextBillingDate
OverdueAmount
OverdueStatus
PaymentInAdvanceAmount
PaymentInAdvanceEndDate
PaymentsStopped
PaymentStopEndDate
Suspended
SuspensionEndDate
Term
TermType
SuspensionSchedules
RecurringSchedules
OneOffSchedules
Paymethod
Customer
ScheduleBase
Abstract Class
EntityBaseClass
Properties
Description
Installment
ScheduleId
PaymentFrequency
Frequency
Enum
NotSpecified
OneOff
Weekly
Fortnightly
FourWeekly
Monthly
BiMonthly
Quarterly
PaySchedule
Class
ScheduleBase
Properties
EndDate
StartDate
OneOffSchedule
Class
ScheduleBase
Properties
StartDate
CreditCardType
Enum
None
AmericanExpress
Diners
Mastercard
Visa
Paymethod
Class
EntityBaseClass
Properties
AccountHolder
AccountNo
ExpiryDate
CreditCardType
AccountType
AccountType
Enum
BankAccount
CreditCard
Customer
Class
EntityBaseClass
Contacts
ContactDetails
Class
EntityBaseClass
Addresses
CustomerDetail
Phones
Gender
CustomerDetail
Class
Properties
DateOfBirth
FirstName
LastName
MiddleName
Title
Address
Class
ContactBase
Properties
City
Postcode
Street
Suburb
Phone
Class
ContactBase
Properties
Name
Number
StdCode
Methods
Gender
Enum
Female
Male
Unknown
Emails
Email
Class
ContactBase
Properties
EmailAddress
State
State
Enum
NotSpecified
AustralianCapitalTerritory
NewSouthWales
NorthernTerritory
Queensland
SouthAustralia
Tasmania
Victoria
WesternAustralia
AddressType
AddressType
Enum
Business
Home
Physical
Postal
Previous
Country
Country
Enum
Australia
NewZealand
PhoneNumberType
PhoneType
Enum
Emergency
Home
Mobile
Previous
Work
CountryCode
PhoneCountryCode
Enum
NotSpecified
Aus
NZ

USE CASE: RETRIEVE PAYMENT HISTORY

Each Account has a Payment History which details financial transactions made against an account.

There are two methods for getting Payment Histories from the DWS.

- Return the Payment History for a single Account based on the `AccountReferenceNo` or `ExternalReferenceNo`.

```
public ResponseRetrievePaymentHistory
   GetPaymentHistoryByAccountId(RequestRetrievePaymentHistoryByAccountId request)

var request = new RequestRetrievePaymentHistoryByAccountId () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },

    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
};
ResponseRetrievePaymentHistory result = GetPaymentHistoryByAccountId(request);
```

- Return the Payment Histories for all Accounts associated with the requesting User, where the transaction date falls in a supplied range of dates.

```
public ResponseRetrievePaymentHistory
   GetPaymentHistoryForDateRange(RequestRetrievePaymentHistoryForDateRange request)

var request = new RequestRetrievePaymentHistoryForDateRange () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },

    RequestInitiator = "an external identifier",

    // Date Range                            //
    StartDate = new DateTime(),              // Required
    EndDate = new DateTime()                 // Optional
};
ResponseRetrievePaymentHistory result = GetPaymentHistoryForDateRange(request);
```

USE CASE: RETRIEVE AUDIT TRAIL

** Not Currently Implemented **

Each Account has an Audit Trail which details all changes made to an account.

There are two methods for getting Audit Trails from the DWS.

- Return the Audit Trail for a single Account based on the `AccountReferenceNo` or `ExternalReferenceNo`.

```
public ResponseRetrieveAuditTrail
   GetAuditHistoryByAccountId(RequestRetrieveAuditTrailByAccountId request)

var request = new RequestRetrieveAuditTrailByAccountId () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
};
ResponseRetrieveAuditTrail result = GetAuditHistoryByAccountId(request);
```

- Return the Audit Trails for all Accounts associated with the requesting User, where the audited changes fall in a supplied range of dates.

```
public ResponseRetrieveAuditTrail
   GetAuditHistoryForDateRange(RequestRetrieveAuditTrailForDateRange request)

var request = new RequestRetrieveAuditTrailForDateRange () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

    // Date Range                            //
    StartDate = new DateTime(),              // Required
    EndDate = new DateTime()                 // Optional
};
ResponseRetrieveAuditTrail result = GetAuditHistoryForDateRange(request);
```

USE CASE: RETRIEVE CALL HISTORY

Each Account has a Call History which details all communications with a Customer about their Account made by Debitsuccess customer services or credit control functions.

There are two methods for getting Call Histories from the DWS.

- Return the Call History for a single Account based on the `AccountReferenceNo` or `ExternalReferenceNo`.

```
public ResponseRetrieveCalls
   GetCallsHistoryByAccountId(RequestRetrieveCallsByAccountId request)

var request = new RequestRetrieveCallsByAccountId () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },

    RequestInitiator = "an external identifier",

    // At least one of the following must be set:
    AccountReferenceNo = "theInternalReference",
    ExternalAccountReferenceNo = "theExternalReference"
};
ResponseRetrieveCalls result = GetCallsHistoryByAccountId(request);
```

- Returns the Call Histories for all Accounts associated with the requesting User, where the calls fall in a supplied range of dates.

```
public ResponseRetrieveCalls
   GetCallsHistoryForDateRange(RequestRetrieveCallsForDateRange request)

var request = new RequestRetrieveCallsForDateRange () {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },

    RequestInitiator = "an external identifier",

    // Date Range                            //
    StartDate = new DateTime(),              // Required
    EndDate = new DateTime()                 // Optional
};
ResponseRetrieveCalls result = GetCallsHistoryForDateRange(request);
```

USE CASE: RETRIEVE FACILITY ACCOUNT CONFIGURATION

Every facility account has a preset contract values like fees and commission rates. These can be retrieved by asking for a specific contract prefix.

```
public ResponseGetFacilityAccountConfiguration
   GetFacilityAccountConfiguration(RequestGetFacilityAccountConfiguration request)

var request = new RequestGetFacilityAccountConfiguration() {

    // Setup Request Properties              //
    User = new User() { Username="yourUsername", Password="yourPassword" },
    ContractPrefix = "yourServicePrefix",
    RequestInitiator = "an external identifier",

};
ResponseGetFacilityAccountConfiguration result = GetFacilityAccountConfiguration(request);
```

USE CASE: ADD COMMISSION VALUE TO PAYMENT AMOUNT

Every transaction taken by Debitsuccess will incur a commission fee. The commission can be different depending upon what type of transaction has taken place. There are three types of transaction, which are derived from the payment method used:

- Direct Debit
- Standard Credit Card (Visa/Mastercard)
- Other Credit Card (American Express/Diners)

There are also two types of commission calculation:

- Per Transaction: This adds the rate as a flat fee to the transaction value.
- Percentage: This uses the rate to add a percentage of the transaction value to the transaction value.

A business may want to pass the commission onto the end customer rather than pay it themselves. To achieve this, the instalment amount provided must include the commission on top, which will then be taken out by us leaving the desired amount. The following example shows how this can be done:

```
// determine the amount that you want to receive
decimal amount = 20;
// get the commission configuration
var result = GetFacilityAccountConfiguration(request);
// get the payment method for the account
var payMethod = GetPayMethodDetails(details);
// determine the correct commission configuration for the payment method
decimal rate;
ChargeType type;
if (payMethod.PayMethod.AccountType == AccountType.BankAccount)
{
    // for bank account use DD
    rate = result.DDCommissionRate;
    type = result.DDCommissionType;
}
else if (payMethod.PayMethod.CreditCardType == CreditCardType.Visa
    || payMethod.PayMethod.CreditCardType == CreditCardType.Mastercard)
{
    // for standard cards use CC
    rate = result.CCCommissionRate;
    type = result.CCCommissionType;
}
else
{
    // for other cards use CO
    rate = result.COCommissionRate;
    type = result.COCommissionType;
}
// add the commission to the amount, this will be deducted by us
switch (type)
{
    case ChargeType.PerTransaction:
        // for a flat fee just add the rate
        amount += rate;
        break;
    case ChargeType.Percentage:
        // for a percentage work out amount that when percentage is
        // taken out the original amount will remain
        // i.e. newAmount * (100 - %) / 100 = originalAmount
        amount *= 100 / (100 - rate);
        break;
}
```