# ASP.NET Web API Checking Account Register Service Demo with XML Datastore & Client "Tester" Console Application

## About

This project presents an ASP.NET Web API Checking Account Register Service Application that demos some CRUD commands (post, get) operating on data in the web application while serializing/deserializing to an XML Datastore. Also, a client “tester” console application tests the API requests to add and retrieve checking account data with the HTTP protocol.

## Description

This project emulates a checking account register that keeps tracks of your credits, debits, fees, and all transactions in one ledger.

## Architecture

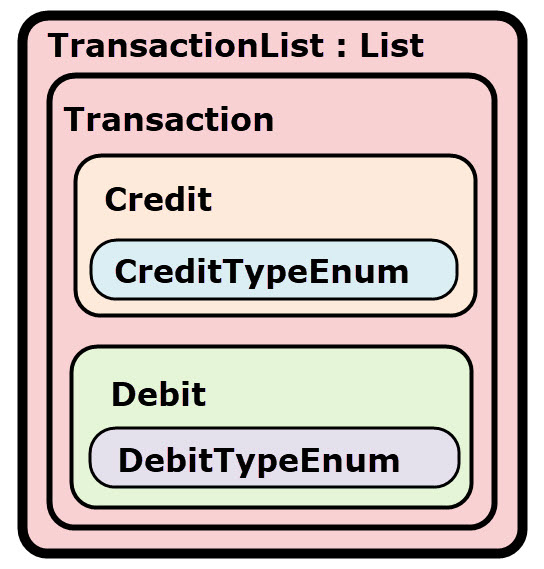
The demo project consists of these component topics:

* ASP.NET Web API Application
  + Update-to-Date Web API Components
  + XML Datastore
  + Web API Controller with Add & Get Methods/Actions
* Models Class Library and XML Serializable Attributes (Shared)
* Utilities Class Library (Shared)
* Web API Client “Tester to Service” Console Application
  + Post & Get Operations Testing using HTTP

## Models Class Library

The “Models” project in the Visual Studio solution is a common class library shared amongst the service and client. The common models help the service provide details on how a client can best consume the data in its object-orientated architecture. The client also uses the common Models library to know how to build type-safe objects to send to the service so that it can process the requests correctly.

### Data Architecture



The Checking Account Register service stores and retrieves records of data that represent money transactions of its customer. The *Transaction* object represents a general record of this money transaction. A collection of *Transaction* entities is stored in a *TransactionList* object that is an enumerable list and inherits from *List<Transaction>*. However, the Transaction record is further specified by a *Credit* or *Debit* type of transaction. The *Credit* and *Debit* objects inherit from the Transaction class which house common properties (ex: date, amount, etc..) common amongst the Credit and Debit child objects. Lastly, there are enumerations for both the Credit and Debit objects that organize the possible transaction categories (ex: cash, check, atm, etc.).

### Data Models with XML Serializable Attributes

#### Transaction

The Transaction class data model with XML attributes for serialization is shown below. The properties are self-explanatory by names. The Transaction class data model file is available in the GitHub repo here [].

The *Transaction* class also implements the IComparable<Transaction> and IComparer<Transaction> interfaces and sorts its objects with the criteria:

1. Date (in ascending order)
2. Transaction Type (Credits first, then Debits)
3. Amount (in ascending order)
4. Description (in ascending order)

[insert code here]

#### TransactionList

The TransactionList class data model with XML root attributes is shown below. The *Load()* method deserializes from the xml file into memory all the Transaction objects into a TransactionList object which inherits from List<Transaction> and returns the entire Datastore memory object to the caller. *Note: The Deserialization method is static and called on the class level.*

Serialization is done by the *Save()* method, where the caller is an *instance* object of this class type TransactionList containing the entire in memory Datastore of Transaction objects. The caller will be a TransactionList object that contains a List of Transaction objects per the class definition. Serialization writes its own in-memory Datastore of the instance object to a specified file location. The TransactionList class data model file is available in the GitHub repo here [].

[insert code here]

*Credit*

The Credit data model class inherits from the Transaction data model class common properties along with an enumeration *CreditTypeEnum* describing the type of credit. In the demo application, a transaction will either be type of Credit or Debit, not both, and not a generic Transaction object. The Credit class data model file is available in the GitHub repo here [].

[insert code here]

*CreditTypeEnum*

The CreditTypeEnum is a simple enumeration for description credit type transactions (ex: cash) and will automatically serialize/deserialize without explicit XML attributes. The CreditTypeEnum class data model file is available in the GitHub repo here [].

[insert code here]

*Debit*

The Debit data model class inherits from the Transaction data model class common properties along with an enumeration *DebitTypeEnum* describing the type of debit, check number, and fee. In the demo application, a transaction will either be type of Credit or Debit, not both, and not a generic Transaction object. The Debit class data model file is available in the GitHub repo here [].

[insert code here]

*DebitTypeEnum*

The DebitTypeEnum is a simple enumeration for description Debit type transactions (ex: Check) and will automatically serialize/deserialize without explicit XML attributes. The DebitTypeEnum class data model file is available in the GitHub repo here [].

[insert code here]

## ASP.NET Web API Application

This project demos the ASP.NET Web API portion only, not the MVC. I did not build a fancy front-end webpage for the user interface. However, the project did build help documentation for the Web API service from the XML comments in the controller code. The API help document page can be access from the ASP.NET Web API main page and clicking on “help” from the main menu (see Demo section). See the entire ASP.NET Web API Application project code directory in the GitHub repo here [].

### Update-to-Date Web API Components

Packages from *Microsoft.AspNet.WebApi* were installed since the Web API updates are released out of band from the .Net framework. The Web API application should be deployed with the most recent and complete Web API package so that the project is not dependent on the localized .Net framework code of Web API, wherever it is hosted. *Note: Use the package manager to update the installed packages before a deployment.* See the project package code directory in the GitHub repo here [].

### XML Datastore

This data for this demo project is serialized/deserialized with the simple file “*transactions.xml*” stored in the *App\_Data* directory of the web application project. To see it in the solution explorer you may need to click on the “*Show All Files*”. The file’s *Build Action* property, is set as “Content”. See the section “*Web API Controller with CRUD methods/actions*” below, to learn how the controller serialized/deserialized to this XML file. See the demo transactions.xml file in the GitHub repo here [].

[insert sample code]

### Web API Controller with Get and Post CRUD Methods/Actions

The *TransactionsController* class file is available in the GitHub repo here []. It is specifically an ApiController and very specific to the Web API engine and implements the API methods or actions that a client requested.

The Web API supports four types of Get (Read) requests:

* Get All Transactions
* Get Transaction in a Date Range
* Get Credit Transactions by Type
* Get Debit Transactions by Type

Also, it supports two types of Post (add) requests:

* Add Credit Transaction
* Add Debit Transaction

#### FilePath

The FilePath property stores the file name and location of the Datastore xml file where data is to be serialized/deserialized. For this demo project, it is stored in the *App\_Data* directory of the web application project. The Controller calls the *TransactionList* static class method to deserialize into an instance object and also calls upon an instance object method to serialize to file (*see the TransactionList data model above*).

[insert sample code]

#### Get “Read” All Transactions

This method loads all transactions from the memory store into a TransactionList object (List<Transaction>), sorts the list per the implemented interface on the Transaction object, and returns to the caller.

[insert sample code]

#### Get “Read” All Transactions in Date Range

This method loads all transactions from the memory store into a TransactionList object (List<Transaction>), performs a LINQ query with the supplied input Date parameters, sorts the list per the implemented interface on the Transaction object, and returns to the caller.

[insert sample code]

#### Get “Read” All Credit Transactions

This method loads all transactions from the memory store into a TransactionList object (List<Transaction>), and performs a LINQ query to obtain first the Credit type transactions and then those with the Credit Enumeration matching the query input parameter. It then sorts the list per the implemented interface on the Transaction object, and returns to the caller.

[insert sample code]

#### Get “Read” All Debit Transactions

This method loads all transactions from the memory store into a TransactionList object (List<Transaction>), and performs a LINQ query to obtain first the Debit type transactions and then those with the Debit Enumeration matching the query input parameter. It then sorts the list per the implemented interface on the Transaction object, and returns to the caller.

[insert sample code]

#### Post “Create” Credit Transaction

The Post() method in the controller creates a record. It simply takes the HTTP Request’s content body and attempts to add the Credit object to the XML Datastore. It will first load the XML Datastore to temporary memory “TransactionList”, add the HTTP Request’s Credit object, then call the Save() method on the instance TransactionList object to serialize and save to the XML file Datastore.

[insert sample code]

#### Post “Create” Debit Transaction

The Post() method in the controller creates a record. It simply takes the HTTP Request’s content body and attempts to add the Debit object to the XML Datastore. It will first load the XML Datastore to temporary memory “TransactionList”, add the HTTP Request’s Debit object, then call the Save() method on the instance TransactionList object to serialize and save to the XML file Datastore.

[insert sample code]

## Web API Client “Tester to Service” Console Application

A simple console application was created to test the Checking Account Register ASP.NET Web API service capabilities. The Checking Account Register ASP.NET Web API publishes a help page in the project build settings that make public the available API methods and instructions on how to consume it (see Demo section). The purpose of the client was to test consumption of all the available methods in the Web API. The client console “tester” application project code is available on GitHub here [].

### CRUD Operations Testing

The main Program class in the Client Console Application “Tester” executes CRUD (post, read) operations on the Checking Account Register Web API Service. It utilizes the ClientHelper class (see Utilities Class Library below) to execute commands with appropriate parameters (which may include Transaction model data) on the service. The code for the Program class is available in the client project on GitHub here []. Note: It specifies in the code the Checking Account Register Web API Service address as a constant for use in testing the API. The Demo section shows screen captures of the client console application testing the service and the results printed to the console.

[insert sample code]

## Utilities Class Library

The Utilities class library is a project with multiple classes that make user data entry, menu navigation, enumerations, text manipulations, streaming, and building http response/request objects a lot easier than coding from scratch. *Note: I did not develop the classes for this framework class project included for our labs. We would not had been capable to code this in a week’s deadline for the class assignment. Instead, we were expected to learn about and how to utilize this framework and integrate into our client application to test the Web API service.*

### Helper Classes

Only the main Web API helper classes that help build HttpClient objects are discussed below.

#### HttpResults

The HttpResults data model class helps to encapsulate and simplify the Web API Response processing. The code for the HttpResults class is available in the client project on GitHub here [].

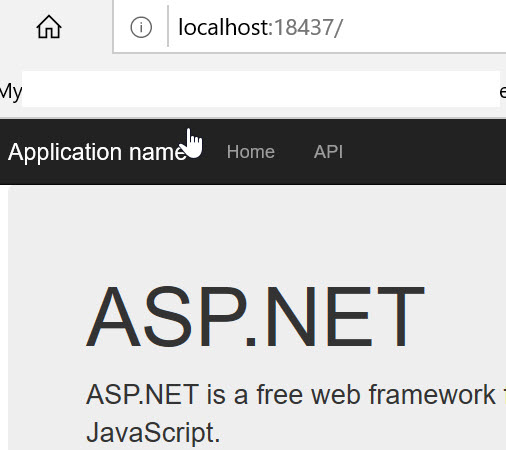
#### ClientHelper

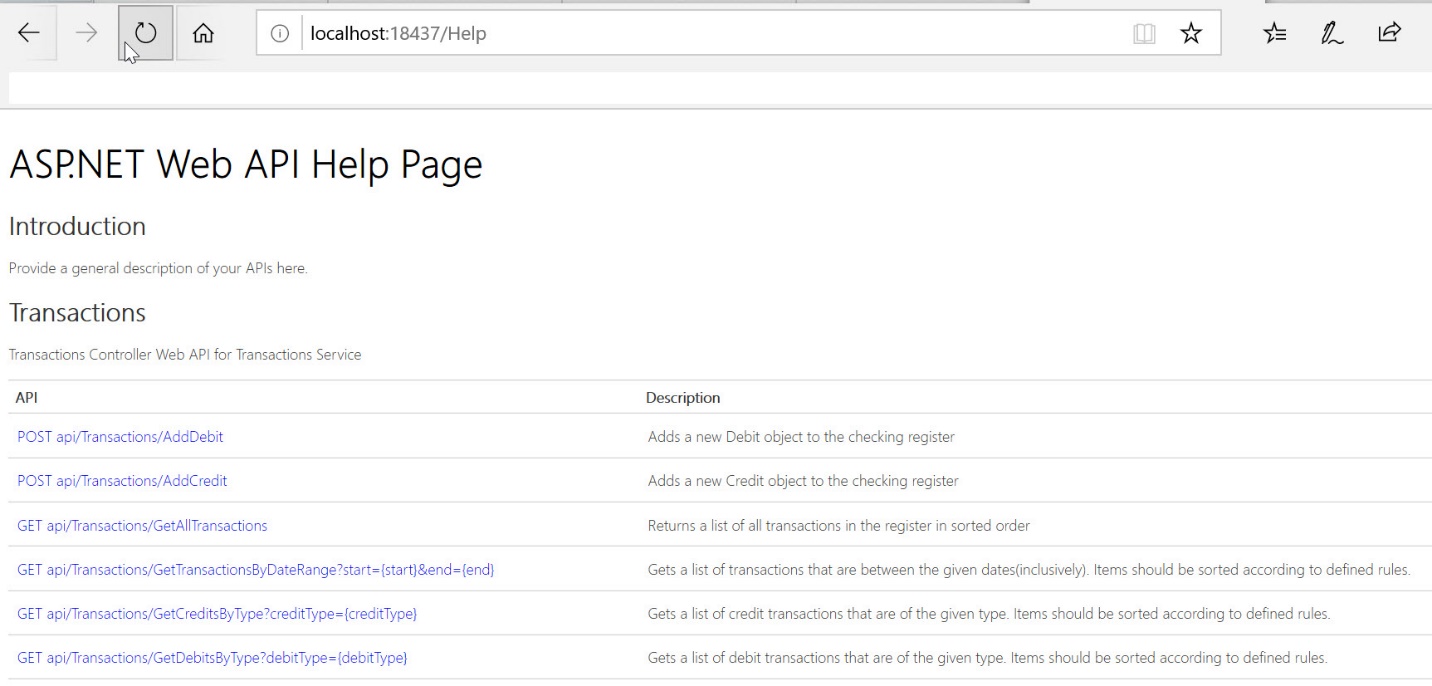
The ClientHelper class is a generic framework that simplifies the HTTP Request/Response processing and operations required by the client “tester” application to formally build request objects and process the responses from a service. The code for the ClientHelper class is available in the client project on GitHub here [].

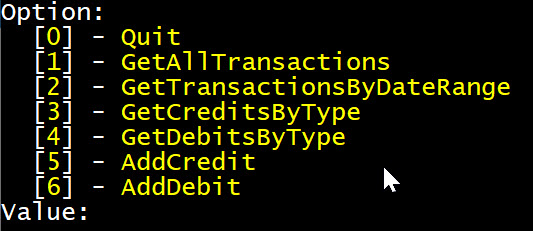
A summary of the capabilities and methods are shown in the list below:

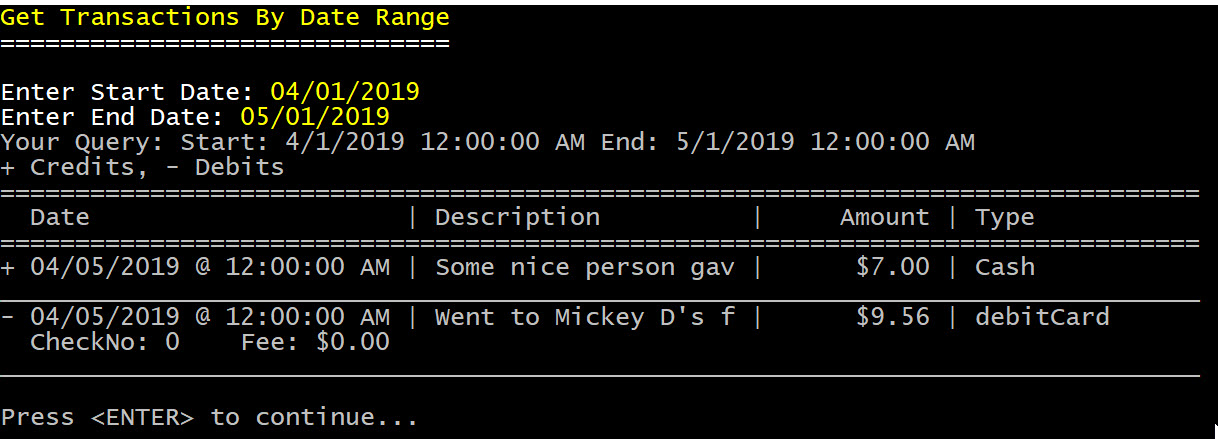
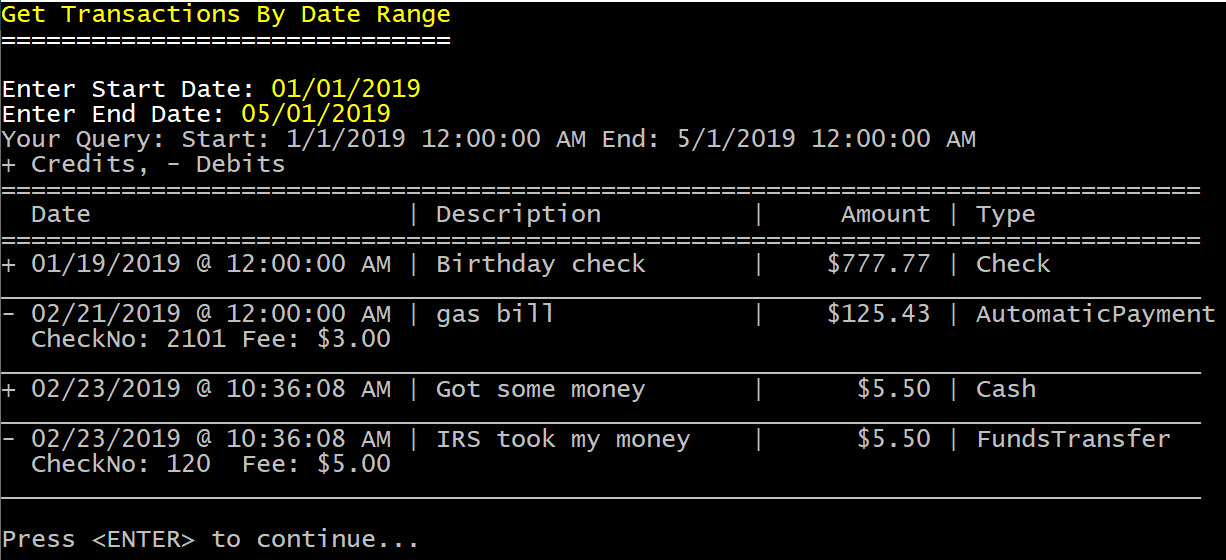
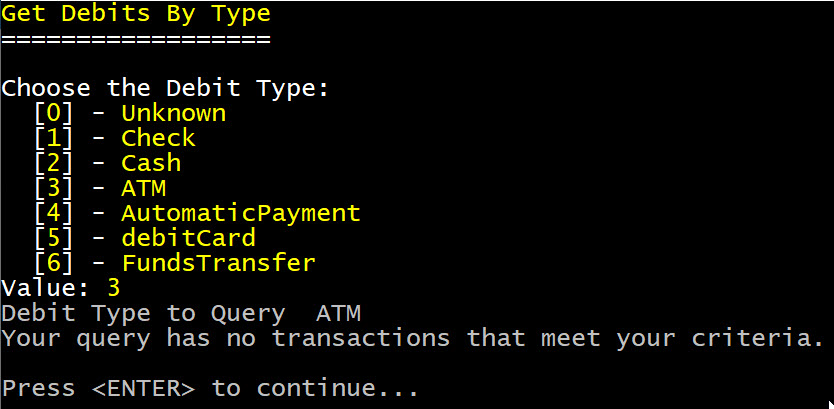
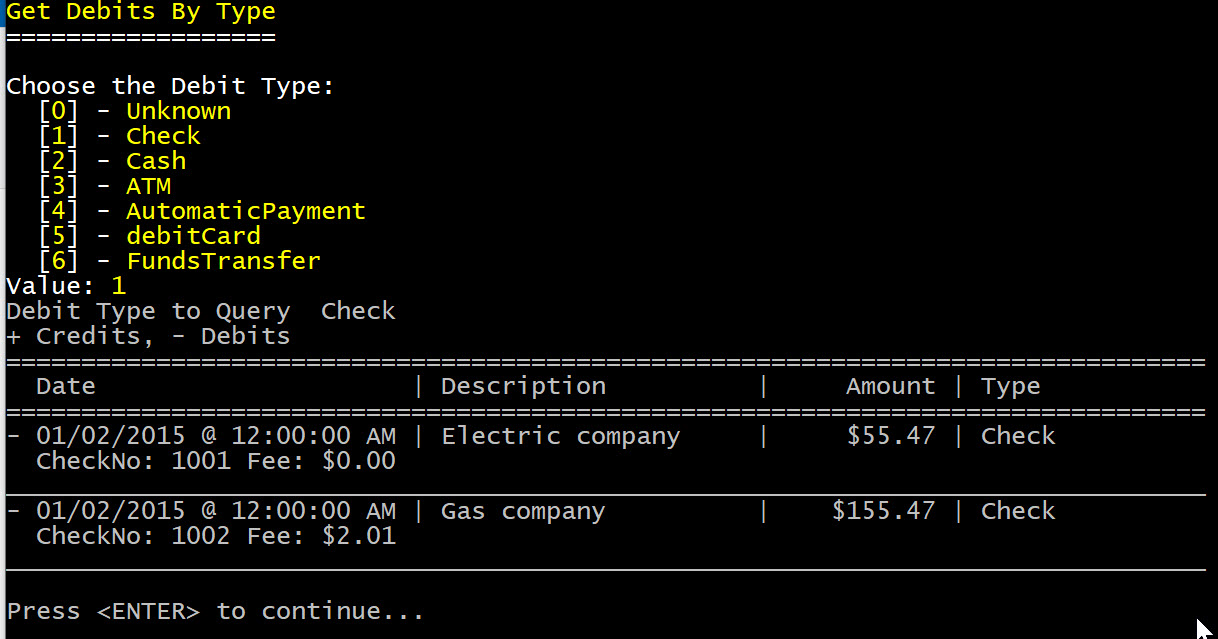
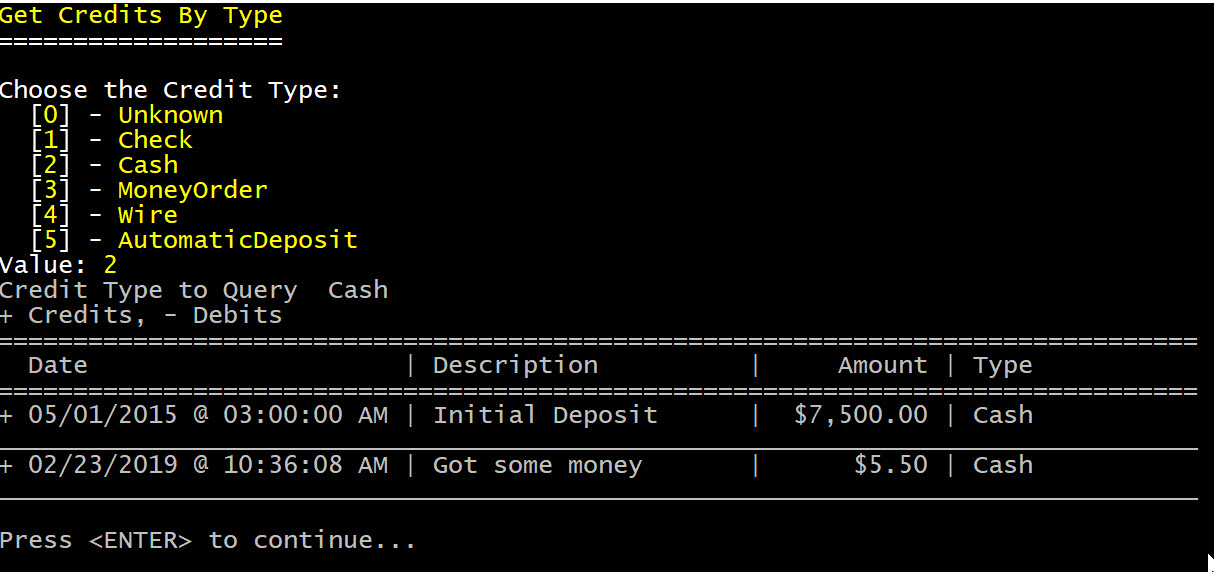
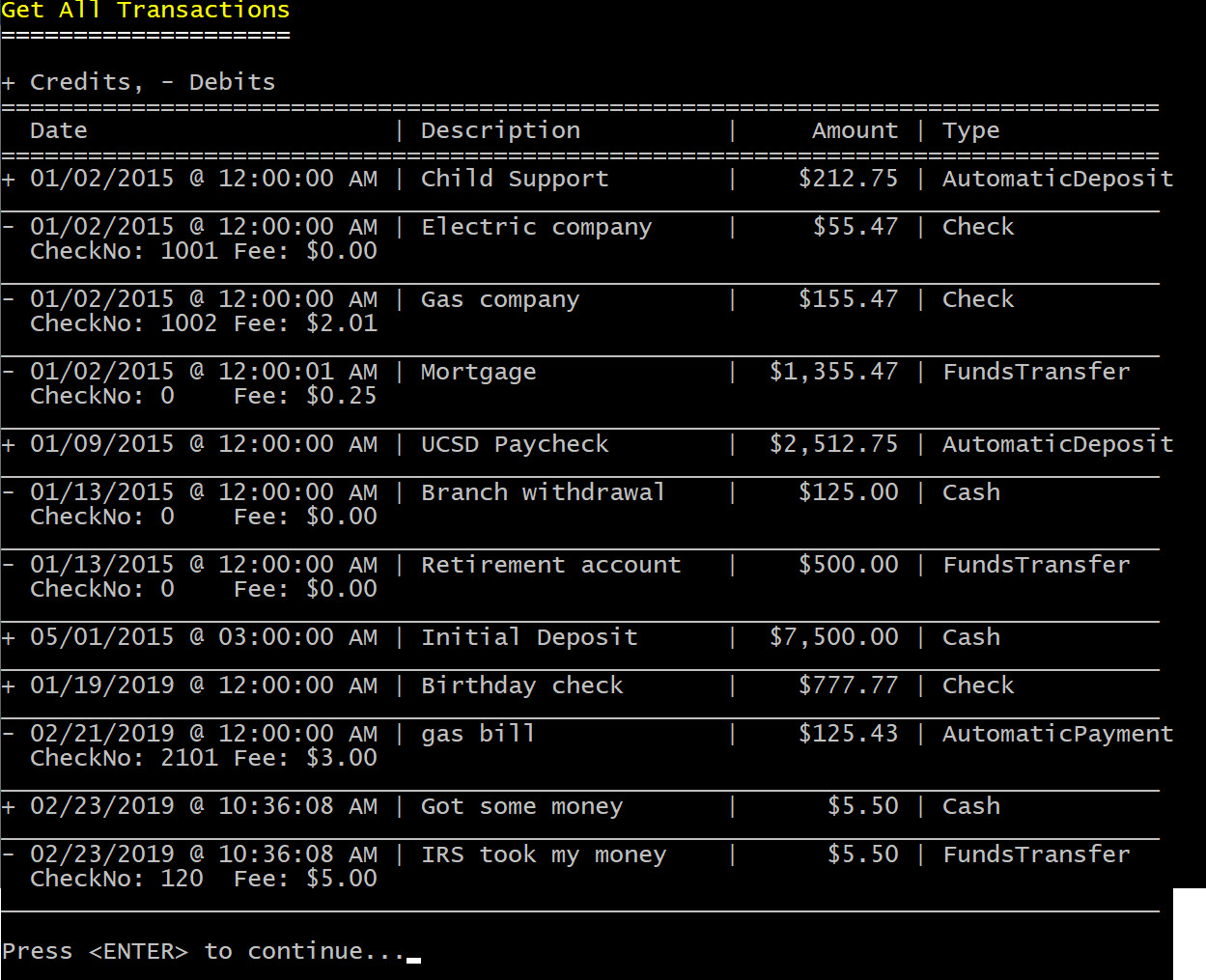
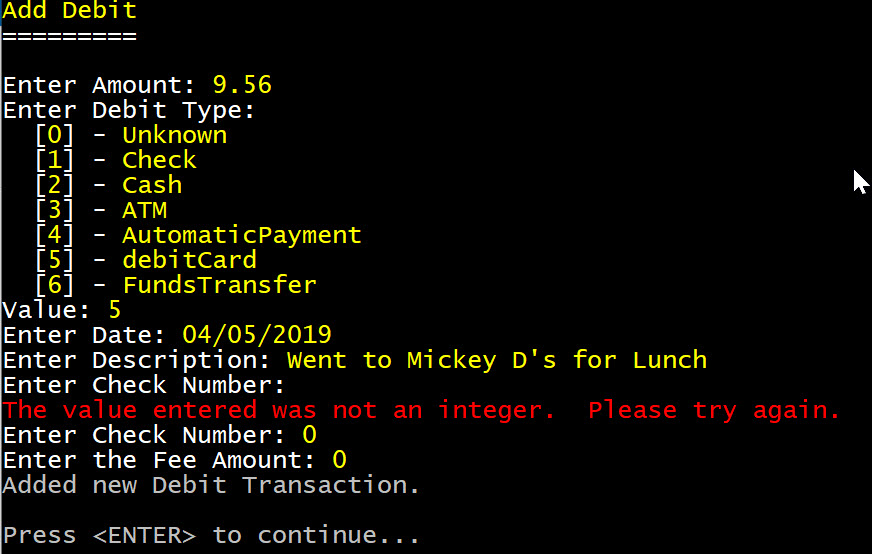
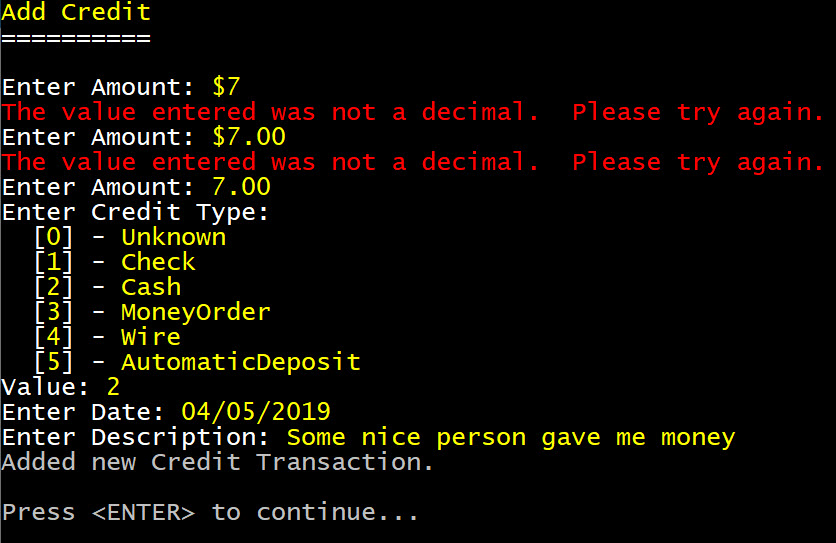
* Gets an HttpClient object to connect to the given service URL
* Gets the media type for the given serialization mode
* Modifies the headers to support the given serialization mode
* Determines the contents of the http response message
* Parses an Http response message
* Executes a GET on the given Url
* Executes an HTTP method on the given Url
* Executes a POST on the given Url
* Executes a PUT on the given Url
* Executes a DELETE on the given Url
* Executes a DELETE on the given Url without header inputs or outputs

## Demo









## Code

The entire project code repository is available on GitHub here.