Payment Processor

Project Description

Payment Processor is a .Net proof of concept that provides a library of reusable components for building messages and executing units of work as a series of processing steps. Additionally, the project includes a simplified version of a TSYS integration and a web UI for testing transactions.

The proof of concept demonstrates building a typical formatted, serialized string such as the ones we construct for TSYS and other processors. However, messages can also be easily serialized to XML and JSON when those formats are required.

The base library design and the proof of concept make use of standard .Net best practices such as coding to interfaces and injecting dependencies. Care has been taken to ensure that classes and methods follow the single responsibility design principle.

Components

The main library contains components to define nested messages, map data values to the messages, serialize the messages as formatted data, and manage a workflow comprised of units of work.

AccessibleMessages

Each payment processor message structure is unique, so each processor's message structure will be custom to that processor. The AccessibleMessage<TMessage> component wraps custom message classes and uses reflection to expose a standard interface for managing formatting and serialization of the message.

Attributes

Payment Processor contains a set of attributes that can be applied to message fields to control formatting and serialization. For example, a DateFormatAttribute can be applied to a DateTime field in a message in order to control how the date value is formatted when serialized. The SerializationFormatter can be used to append a field terminator to a field during serialization or to control how a field is masked when securely serialized.

Mappers

The base library contains an IMapper interface and Mapper<TContext, TMessage> and ParentMapper< TContext, TMessage> base classes that can be used to define custom mappers for binding data to custom message classes.

Mappers are a spinoff of the Builder design pattern. Each mapper fully constructs a particular custom message class.

Process Context

During processing, artifacts are created that may need to be referenced in subsequent processing steps. For example, when the request is created it is needed by the unit of work that sends the request to the processor as well as by the unit of work that serializes the request and saves it to the database.

The Process Context acts as a repository for artifacts that are created at run time and that are needed throughout the payment processing workflow.

Process Runner

The Process Runner manages the workflow for processing a transaction. It calls each unit of work sequentially and manages the error state. Each processor integration will inherit from the process runner and define the units of work that are needed for interacting with that processor and with the gateway database. See the example code near the end of the document.

Serializers

Currently the library contains a string serializer that will support the kind of string messages that are used by Omaha, TSYS, and Worldpay. We'll need to add a JSON serializer and an XML serializer to fully support all of our processors. That work should be fairly trivial.

Transaction Model

The base library contains nested model classes that represent the data received in a transaction request payload. The payload is deserialized to a model, and model validations are called before processing is allowed to begin.

Using the Library

- Define the custom transaction class to be deserialized from the JSON transaction payload. This will inherit from the base transaction model; only the processor-specific merchant attributes differ between processors.
- Define the custom envelope class to be deserialized from the envelope stored in the database for certain transaction types. Every processor integration retains different data points, so envelopes need to be custom.
- Define the custom transaction context class that provides useful and DRY information that the mappers need to efficiently convert the transaction into a request message. This will inherit from the base transaction context; only the envelope and processor-specific merchant attributes differ between processors.
- Define custom message classes and add validation, formatting, and serialization attributes as needed.
- Define custom mapper classes for instantiating the message classes.
- Define classes to perform units of work; classes can inherit from the base unit of work. The framework is based on the concept that a complex activity such as processing a payment can be decomposed into small, incremental steps which are the units of work.
- Define the transaction runner which is responsible for executing the units of work in a workflow. The transaction runner can inherit from the base process runner. Here is example code that shows how clean and easy it is to set up a workflow. Note that the code only shows a few example steps. Several more steps are required to actually process a transaction.

```
protected override async Task StepsAsync()
{
    // TODO: add steps here
    RunStep<BuildTransactionContext>();
    RunStep<BuildMessage>();
    RunStep<SerializeMessage>();

    // await RunStepAsync<SendMessage>();
}
```