Sales Tax Service Design Notes

Nicholas Dragotas (Feb 2022)

Contents

[Processing Design Overview 1](#_Toc95950569)

[Overall Process and Design 1](#_Toc95950570)

[General Sales Tax Service Process 2](#_Toc95950571)

[TaxJar Specific Request Processing 2](#_Toc95950572)

[Performance Considerations 3](#_Toc95950573)

[Interface Design Notes 4](#_Toc95950574)

[Defining Unified/Common Data Structures for Request Input and Output 4](#_Toc95950575)

[Not Using Exceptions in the Sales Tax Service API 4](#_Toc95950576)

[Sales Tax Service Async API 4](#_Toc95950577)

[Improving Design Ideas 5](#_Toc95950578)

[Class Per Request 5](#_Toc95950579)

[Abstract Factory Pattern 5](#_Toc95950580)

[Other Considerations 5](#_Toc95950581)

[Logging 5](#_Toc95950582)

# Processing Design Overview

## Overall Process and Design

We need a C#/.NET Sales Tax Service class that allows client code to query sales tax info from various online tax data sources/providers using a unified/common interface. As a start, the Tax Service will provide a generic way for a client to request:

* Sales Tax Rates for Location
* Sales Tax Amount for Order

When interacting with the Sales Tax Service, the client will specify the tax data source provider for the tax data retrieval.

The Sales Tax Service interface will specify a unified/common way to specify the query data in a way that is generally agnostic to any specific provider. The client will need no knowledge of the request data and mechanics for specific tax data sources/providers. Thus, the client will provide the same data format for a query regardless of the tax data source provider. The Sales Tax Service will use this common data to generate provider specific request data and send the request to the provider.

The interface will include the superset of data required by various providers. If a provider does not require a specific data item, the client can leave that item empty.

Similarly, the interface will specify a unified/common way to return the sales tax request/query results. The Sales Tax Service will furnish the client with a unified/common sales tax output format regardless of tax data source provider selected. Thus, after receiving a request response from a provider, the Sales Tax Service will extract specific data from the response and transform it into the unified/common sales tax data. The client will need no knowledge of the sales tax data format used by a specific tax data source/provider.

The interface will specify a unified/common way to indicate success/failure results. If the tax query was successful, the Tax Service will translate the query results into the common generic format. If there was an issue with the tax query, the Tax Service will provide the client with a generic error result as well as the specific error message from the provider. Logging should capture more specifics on the error as well as provider specific error messages.

## General Sales Tax Service Process

The overall Sales Tax Service process is as follows

* Client performs the following:
* Initiates interaction with Sales Tax Service.
* Client specifies sales tax provider to use.
* Populates request/query data using common/unified format (e.g. structure/class or JSON).
* Calls Sales Tax Service to perform request and provides request/query common/unified data.
* Sales Tax Service does the following after receiving client call/query:
* Performs initial request data validation.
* Uses common/unified data from client to generate the specific provider request data.
* Sends request/query to specific provider.
* Monitor/handles communication issues.
* If request failed, logs provider specific issues and error messages. Then returns a generic error code as well as provider specific error messages.
* If request/query receives a successful response, extracts data from provider response and transforms it into a common/unified format. E.g. if the response is a JSON or XML string, it is parsed to extract specific data fields and values. Then a common/unified response data element is created (e.g. class/structure or JSON structure) and it is populated with the data fields and values extracted.
* Returns to client request status and common/unified data.

## TaxJar Specific Request Processing

Here is a description of request processing by TaxJar.

Processing of Sales Tax Rate requests from TaxJar:

* STS performs generic validation of common/unified request input data.
* TaxJar expects the input data for a tax rate request as URL query parameters in an HTTP GET request. Thus, the STS TaxJar Calculator will take the unified/common request data from the client (e.g. zip, country, etc.) and form a URL with the query arguments created from the data.
* An HTTP GET request is sent to TaxJar with the URL and query parameters specifying zip, and optionally country, state, city, and street.
* If the HTTP request fails, the STS TaxJar Calculator should return a generic error state as well as any available error messages (either from the HTTP request mechanism or from within the TaxJar response). The STS in turn will return the error states and messages to the client and log error info.
* If the HTTP succeeds, TaxJar returns tax data in JSON format. At this point a transformer/translator in the TaxJar Calculator would extract the data from the JSON format and populate a common/unified data format (a struct/class or common JSON format).
* The common/unified format tax rate data is returned to the client.

Processing of Sales Tax Calculation requests from TaxJar:

* STS performs generic validation of common/unified request input data.
* TaxJar expects the input data for a tax calculation request as a JSON string to be submitted via HTTP POST request. Thus, the STS TaxJar Calculator will take the unified/common request data from the client (e.g. to/from zip, taxable amount, shipping, etc.) and create the JSON string expected by TaxJar.
* An HTTP POST request is sent to TaxJar with the JSON data.
* If the HTTP request fails, the STS TaxJar Calculator should return a generic error state as well as any available error messages (either from the HTTP request mechanism or from within the TaxJar response). The STS in turn will return the error states and messages to the client and log error info.
* If the HTTP succeeds, TaxJar returns tax data in JSON format. At this point a transformer/translator in the TaxJar Calculator extracts the data from the JSON format and populates a common/unified data format (a struct/class or common JSON format).
* The common/unified format tax rate data is returned to the client.

# Performance Considerations

Because most of my experience involved embedded or desktop systems, I am conscious of performance considerations.[[1]](#footnote-1) Cloud platforms provide a considerable amount of computing resources. Although there is no need to be paranoid, I think you still want to be considerate with how code scales (I’ve deployed software on AWS and have seen the bills).

In this case we should consider how software clients interact with the Sales Tax Service (STS) class/component. For simplicity, I made the assumption that the number of STS clients does not correspond to the number of real-time online orders. I don’t think you want to create an instance of the STS class for each request. Thus, I assume there is a “system” that receives all orders and places them in one or more queues processed by one or more STS clients. Each client creates a single instance of the STS class. Then it reads from the queue and invokes STS sequentially.

Now, if each order/tax request represents an STS client, we would need a different design. We don’t want to create a separate STS instance for each request. We would have to change the design so that a single instance would handle concurrent requests asynchronously.

# Interface Design Notes

## Defining Unified/Common Data Structures for Request Input and Output

Part of the Sales Tax Service interface is a unified/common way to specify request input as well as response output. I am using structures or classes to define the unified/common format for simplicity.

One of the advantages with structs/classes is that you don’t need field name validation (if you get a field name wrong, the compiler catches it). If you use a dictionary/map/hash, or JSON or any type of string, you need to validate the field names extensively. Troubleshooting issues stemming from mistyped field name in a string can be hard.

The disadvantage with using stucts/classes is that if STS needs to become a microservice with REST interface, you’ll need a string format like JSON).

## Not Using Exceptions in the Sales Tax Service API

The SalesTaxService API calls are defined to return error status. This is considered “old school”. I did this out of habit really and did not think about using exceptions until later. More modern APIs use exceptions instead of returning error codes and Microsoft encourages using exceptions as well. Changing the API to use exceptions is easy enough to implement: define an STR Failure Exception class and include error data. Then throw an exception instance and remove the code returning error codes. At the same time, there are a couple of things to consider.

First, how often are exceptions possible? One thousand exceptions within a short time may not be an issue, but 500K may affect performance. Is there a scenario involving that many exceptions? I’ll admit that I am biased by embedded C++ situations where I had to replace exceptions with return codes months after the code was completed due to performance issues. From what I see online, C#/.NET has not improved compared to C++ environments - exceptions can take 10+K times longer. Thinking about it, this is not a surprise. But the question is will we ever get that many exceptions? So, assume that there is a sale is going on and somehow there are 100K orders submitted within seconds. Assume, also, that the TaxJar service happened to be down. Will the Sales Tax Service use a different provider or will it throw 100K exceptions? What is the impact on the system? Is this scenario reasonable? If not, use exceptions.

Another thing to consider is the future potential need to convert the Sales Tax Service into a microservice with a REST HTTP API. But, on second thought, that would not be as big of a change to do – we could simply write a wrapper around Sales Tax Service that would catch exceptions convert error data and include it in HTTP responses.

## Sales Tax Service Async API

The Sales Tax Service API could be changed to support asynchronous processing, allowing clients to send multiple concurrent requests and use “await” to process all responses asynchronously. The HTTP request calls I used are asynchronous and I am using “await” up to a certain point. However, I did not propagate this up to the API level for simplicity.

# Improving Design Ideas

## Class Per Request

Currently there are only two tax requests: rate and calculation. Thus, a single Tax Calculator class can contain the code handling both requests. If the number of tax requests becomes larger (e.g. more than 5), we may want to separate code handling each request into a separate class.

## Abstract Factory Pattern

Here is another design idea on how one could use the “Abstract Factory” pattern. I’ve used this pattern before extensively. However, in this situation I do not necessarily recommend this (I don’t promote overdesign). I bring it up just for the sake of discussion and to show that I am familiar with OO patterns.

We could split each processing done for a specific tax data provider into implementations of the following interfaces each of which has one method: “DoProcess”.

* ITaxRateInputGeneration
* ITaxRateSendRequest
* ITaxRateOutputProcessing
* ITaxCalcInputGeneration
* ITaxCalcSendRequest
* ITaxCalcOutputProcessing

Generic code would use an Abstract Factory to generate instances of the above interfaces for a specific service. E.g. for TaxJar, the Abstract Factory would return the following objects implementing the above interfaces respectively.

* TaxJarRateInputGeneration
* TaxJarRateSendRequest
* TaxJarRateOutputProcessing
* TaxJarCalcInputGeneration
* TaxJarCalcSendRequest
* TaxJarCalcOutputProcessing

So, the generic code would have members of ITaxRateInputGeneration etc. containing objects of TaxJarRateInputGeneration etc. To service a request for Tax Rate, the generic code would call methods of the above objects in the following sequence:

* ITaxRateInputGeneration.DoProcess(…)
* ITaxRateSendRequest.DoProcess(…)
* ITaxRateOutputProcessing.DoProcess(…)

# Other Considerations

## Logging

I created a basic logger that outputs in STDOUT to keep things simple. Logging facilities should be simple but in my experience they are not, so I did not look into C# loggers. (It’s been so long that I do not even remember the logger I used for C# in the past. My most recent logging experience is with Java using Log4J and one more logger – both pretty complex.)

1. Software that can potentially create thousands of object instances in less than a second (or throws just as many exceptions), can lead to serious issues even in cloud environments. Years ago I had to fix a C# application which gradually consumed 6+ GBs of memory and crashed its system. It looked like a memory leak - not an issue with .NET and garbage collection. The issue was that the code was creating and destroying objects faster than the garbage collector could release memory. [↑](#footnote-ref-1)