

# Sierra Chart Does not Support External Service API Components

- [Introduction](#)
  - [Reasons Client-Side API Components Are Not Supported](#)
  - [Encryption and Compression](#)
  - [Proposal for Using a Local Server Executable Program](#)
  - [FIX Protocol](#)
  - [Recommended Protocol](#)
  - [Minimum Required Quality of Data or Trading Service for Integration](#)
- 

## Introduction

---

Sierra Chart no longer is supporting client-side API components to connect to external Data or Trading services. The definition of a client-side API component is executable code usually in the form of a DLL (Dynamic Link Library) that links into the Sierra Chart executable or process that runs on a user's computer.

Even code which is in source code format and can be compiled by us, is also not acceptable unless it is some kind of encoding or decoding type of function library (an example would be Google protocol buffers).

Client-side API components which run as a separate executable on a user's computer, are acceptable for us to work with as long as they provide a TCP/IP or UDP socket method of communication. This is how the DTN IQ Feed and Interactive Brokers Trader Workstation work.

Sierra Chart will only be working with external Data or Trading services that provide a connection method which does not involve API components. This type of connection method involves a network socket and a standard or somewhat standardized communications protocol over the network socket which is clearly documented.

The protocols and data formats could be [DTC](#), FIX, FIX/FAST, Google Protocol Buffers, HTTP, HTTPS, XML, JSON, comma/tab delimited data or nonstandard data in binary/nontext or text format.

In the case of trading, the only 2 protocols that Sierra Chart will support is FIX or [DTC](#).

If you are the provider of a Data or Trading service, then we will no longer consider adding support for your Data or Trading service if it uses a client-side in-process API component.

## Reasons Client-Side API Components Are Not Supported

---

1. Server connection protocols which do not use an API component are completely independent of operating systems, programming languages, development environments and compiler versions. This allows us to interface to the external Data or Trading service no matter what operating system Sierra Chart will be running on and no matter what development environment or compiler version we are using.

This is especially important as we begin to work on support for other operating systems like Mac OS X and GNU/Linux.

We are not forced to work with a particular programming language, development environment, or compiler version.

2. It is routine with API components where we have had various compiling and linking problems with building our software due to conflicts and incompatibilities with the library files of an API component. We may be forced to use particular DLL files and other components which may create new sets of problems. All of this is completely eliminated by not using an API component.
3. We do not have to maintain proprietary API components for our project and update our project when a new API component version becomes available and understand and study all of proprietary changes to it. As has been previously indicated, by avoiding the use of APIs, we do not have to contend with compiling and linking problems between APIs and our own projects.
4. We do not have to study and learn a new API component. When working with multiple API components, it takes a significant amount of time to understand and learn them. Often little details with their behavior may not even be understood until years later causing a problem which affects our users. It is time-consuming and detrimental to us when we have to contend with bugs and design flaws that go unresolved from other developers. It is like working with a whole new language.
5. These API components follow no established standard and are often designed in such a way to work best with the developer's own software rather than third-party software. The design of an API component generally is bizarre in their design and nothing but a headache to understand. This makes them very difficult and time-consuming to integrate into our software. The integration of API components has never been proper because it is impossible to accomplish. There would always be stability and organizational problems with using them.
6. The model of how an API component works is proprietary, and is designed to follow a model of how the outside developer thinks is best, but is incompatible with other software programs. The integration of the API component can only be regarded as a hack. This is detrimental for us, and detrimental for the end-user. Quality and proper integration to a Data or Trading service cannot be achieved with in-process API components. This has been proven time and time and time again.
7. One point of great difficulty is when an API component makes a call back on a thread different than the thread that created the object.
  - It cannot be assumed that the client program is necessarily thread safe in every possible way and can easily handle the content of the callback on a background thread. Multi-threaded programming is highly complex programming and most programmers do not do it entirely safely. It is our

position that an API component must only make callbacks upon the thread which created the API object or called its "Connect" (or similar) function. The client program can then dedicate its own thread for creating the API object if it wants to use a separate thread for the API interaction and callbacks.

- In the cases where client-side API components have made callbacks on a thread different than the main program thread or the thread which created the object, this has required us to take a copy of the callback data and pass it to the primary thread for processing, except for basic market data.

This has involved rather sophisticated programming to accomplish reliably. Sierra Chart does use threads for certain purposes. However, other than for basic market data, Sierra Chart cannot safely handle callbacks in other threads. It is our contention that it must be avoided to use a separate thread for callbacks from an API component used by other programs. There is not anything wrong with separate threads to process network I/O. However, when interacting with an outside program, you should let the client program be in full control of threading and not create a separate thread that the client program never created, does not have implementation details of, does not fully understand, and has no control over.

- If thread synchronization is not done properly considering every possible scenario when 2 threads interact with a shared object, then this can lead to program instability. The use of multiple threads can lead to deadlocks which are very hard to isolate the source of. Multithreaded programming is very complex and most developers do not do it properly. Why you would introduce a massively complex model to developers most likely who are not highly experienced or knowledgeable with threading, makes no sense to us.
- If you believe that creating a thread in your API and making an uncontrolled callback with data on that thread into a client program is acceptable, then why is no such model like this followed with Windows Sockets for network communication? From our perspective, this comes across as substandard and the correct technical answer is that what you are doing is simply incorrect.

Have a look at how [Windows Sockets](#) works for both Overlapped I/O and asynchronous window notification messages. When a socket I/O operation completes or the socket is readable or writable, there are synchronized mechanisms to notify the thread that is using Windows sockets of the event. There is not an independent thread that does this in an uncontrolled or unsynchronized way.

Therefore, you might say why not use a synchronization object like a mutex or critical section to perform synchronization within our software. In order to do that, every object that your background thread interacts with either directly or indirectly, would have to be synchronized. This would massively increase the complexity of our and others programs, and hurt performance.

- This issue has been raised with a couple of API providers who have this

kind of callback behavior. In the case of a C++ client program, it is still not shown how the method is valid by using a background thread that you create which makes an event callback in an uncontrolled and unsynchronized way.

In order for your API to be friendly and flexible for all kinds of applications, you would have to implement more than one of the methods that Windows Sockets provides to notify of an event or an I/O completion. Rather than doing that, it make more sense for you to provide a direct socket connection to your services or follow our [Proposal for Using a Local Server Executable Program](#).

8. API components can and do throw exceptions for trivial things such as an invalid parameter or even an invalid symbol, rather than returning a clear error code indicating this kind of problem. This requires excessive code to trap exceptions and to recover from the error. Exceptions should be reserved for serious hardware faults only. And even then, there is a better model for handling exceptions rather than the current typical way in which they are, which is terminating an application. Software programs should not be throwing exceptions or certainly not throwing them for something trivial and passing them up to the client. An API component throwing an exception for something trivial, is only violently taking control away from the path of execution that we have established within a function. The question we always have, is what exactly would throw an exception, and what exactly is the exception type?
9. API components can and do have faults which will crash Sierra Chart and we have no control over that. API components often temporarily freeze programs when connecting and disconnecting. We have seen API components not properly and gracefully handle network communication. API components have peculiar behaviors that have to be understood and worked around. Another way of saying this, is that the use of client-side API components makes us look like incompetent programmers because of the problems they introduce into our software. This cannot be stressed enough.
10. Sierra Chart has no control over memory use of a client-side API component and these components can use excessive computer memory.
11. Sierra Chart has no control over the processing going on within a client-side API component and these components can use excessive CPU usage, freeze, and lock up .
12. Client-side API components have unusual and odd behaviors which cannot be understood or resolved by outside developers and cause various problems in our software.
13. There are no external API components to redistribute if an API component is not used.
14. We have full control over the process of connecting and disconnecting and we can handle it in a graceful and reliable way when not using an API component. We are able to connect and disconnect when needed. With an API component, we have no idea as to the status of its own internal network connection. You have hidden that from us. The concept that somehow you are making programming easy for people by handling the network communication, is untrue. We cannot stress that enough. Related to this, when there is a network communication error for direct network connections for services that allow a direct connection, Sierra Chart will clearly display the standard network error message, in the users own language, so it is clear to the user and to us what the source

of a network communication problem is.

API components have various ways and codes to indicate communication problems, usually in ways which are not clear. This makes troubleshooting connection problems very difficult for us and our users.

15. Working with API components has proven in every case to greatly increase the development time due to bugs within the API client software, due to highly proprietary designs that have to be studied and understood, due to odd behaviors, and because they provide interfaces which are not compatible.

A proprietary API component which does not have reviewable source code, is a black box with all kinds of unknown behaviors. They do not make anything easier for us, only significantly more difficult. Do you realize, that in all cases you literally increase the development time with an API component as much as 20 times or higher by providing an API component rather than providing a direct socket and a simple set of standardized messages to exchange between the Client and the Server?

When you are working with professional developers who have exceptionally high programming standards and consider every possible detail to produce the very best software, a client-side API component is wholly unacceptable because we do not have control over it and we do not know how it works.

This point cannot be stressed enough. We have a **very exceptional negative view** of API components as having worked with them for more than 17 years and have seen nothing but problems from them as we have described in this section and have suffered severely from them. It is time for a change, and for this nonsense of API components to come to an end.

16. **Sierra Chart has a clear established policy that we do not work with API components.** This policy will be followed even to the detriment of not expanding business with services that only use client-side API components. We have a strong belief that client-side API components are going away because they simply make no reasonable sense and are not of acceptable quality, especially when you are working with professional developers like us.

We are clearly seeing this trend now with the move away from these components and greater use of FIX, DTC and Google Protocol Buffers with direct socket connections.

17. Another way to understand our position about in process API components is to consider how a web browser connects to a website. Does every website provide an API component? Of course not. Instead a web browser uses a network socket connection and uses the HTTP protocol. The specific implementation details of this is determined by the web browser and not by anyone else. Although the web browser will follow standards with its external interfaces to websites. But the internal implementation is as the web browser requires.

The author of this document has over 20 years of experience in this industry and working with API components. I can unequivocally say, that this practice of creating API components that run with an

address space of another program, is a substandard concept in the case of an external Data or Trading service, out of date (although was never reasonable to begin with), is not helpful, is disorganized, promotes substandard interfacing to your systems because the user of the API has no control over the input-output model being used, absolutely extremely problematic for each and every one of the above reasons, and does not make any reasonable sense that we have ever seen and will never see.

This is an open **complaint** about this practice. As we previously said, we are not supporting client-side API components any longer, if they run within the address space of a program. You need to recognize that you are an external service with systems located elsewhere on the Internet and what you are providing is a request and response based service and/or a streaming data service.

The only reasonable way to interface to your systems is using a network socket. Trying to wrap up your service into an API which runs into an address space of another program, never has worked well because simply the concept is incompatible with the type of service you are providing.

If you really want to know the truth, the author has had it with API components. API components make us look like idiots and incompetent programmers, have caused us to suffer massively with numerous problems, wasted programming time, wasted support time, and have caused us to waste enormous amounts of time understanding and working with stupid API components. It has been extremely detrimental, harmful and a very awful experience working with API components. Please do not mention them to us.

Many of our users have read this particular page, and those who understand the subject, 100% agree with what we have said.

## Encryption and Compression

---

In the case where encryption is required, SSL/TLS can be used. Sierra Chart supports SSL/TLS (secure sockets layer) through the use of Open SSL for encryption.

The freely available **stunnel** program can also be used for for SSL/TLS encryption. In our case, we do not need to rely on this because we have our own implementation.

There also are socket libraries available which support SSL/TLS. Which further simplifies secure socket network communication. In the case of when using HTTPS, this kind of functionality is part of operating systems or programming libraries. So it is very accessible to any programmer.

In the case where compression is required, we support **zlib** compression. This is a widely used compression library which is easy to work with. Many programming libraries have built-in support for this.

## Proposal for Using a Local Server Executable Program

---

If you are a Data or Trading service provider currently providing an API component that runs within the address space of a program, and do not wish to support direct socket connections to your backend systems, then we offer the following proposal.

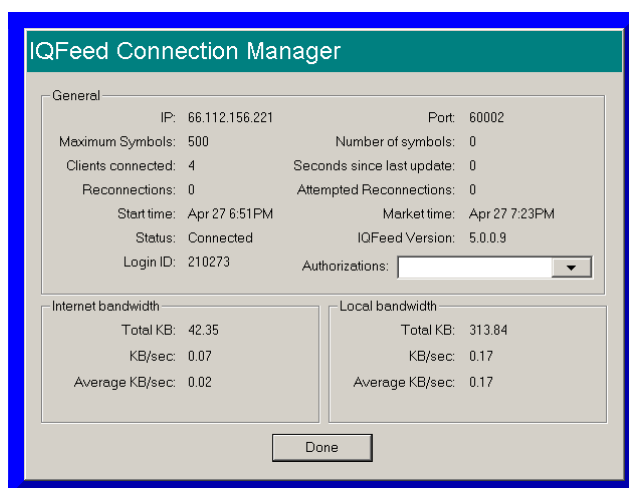
If you are open to providing a direct connection to your backend, then this is good and refer to the [Introduction](#) above instead of this proposal below.

The purpose of this proposal below is to put forth an alternative to the community, to using a direct socket connection to the backend. We tend to suspect the reasons that services that provide a client-side API that runs within the address space of a program to connect to a service's backend relates to encryption, compression and the ability to ID the local machine, which you feel for some programmers would be hard to implement. In the case of an encryption and compression, these are really not difficult to implement as explained in the [Encryption and Compression](#) section.

We have long recognized the inherent problems with everyone following their own protocol and compounding that with a proprietary black box DLL approach for interfacing. Especially if the black box uses something like this confusing and highly proprietary Microsoft COM (component object model). Which at this time is actually out of date and platform specific.

This proposal is meant to resolve this. We also hope that because Sierra Chart has established itself as one of the most reliable, stable and fastest programs out there in the industry, that we have significant credibility with our proposal. We hope that this proposal gains interest and becomes the standard when a direct connection to your backend systems is not offered.

1. Make your API a standalone executable with a small status window, rather than a DLL or library that runs within the address space of the program using it. We would launch this executable when we want to make a connection. The executable would automatically shut down when there are no local connections to it. Every time we make a connection, we always will run this executable. The executable needs to be designed to not allow a second instance of itself. Although if there are multiple installations of the executable on a system, each of those should be able to run independently. The status window can show connection state, a log and other information. Here is an example below.



2. This client-side executable server program will listen on a local TCP/IP socket using the specified port number passed to it on the command line by the client program that started the process.
3. The communication protocol over the local network socket will use the open specification Data and Trading Communications (DTC) protocol which is documented on the [DTC](#) page.
4. This kind of design completely eliminates incompatibilities involving differing compiler versions that you use, and we use. You would no longer have to provide any

documentation related to versioning, compiling, and linking or provide support for various issues which arise from these. By working with a network socket protocol you are working with a completely language and operating system independent protocol.

5. With this kind of design, your server executable program can serve multiple programs running on the same computer. This minimizes bandwidth usage to your remote server. You can even look at the address of the incoming socket connection and restrict connections only to the localhost.
6. This kind of design is exactly how DTN IQ Feed and the Interactive Brokers Trader Workstation (TWS) API work. The Interactive Brokers TWS is a very successful and widely used API proving the sensibility of this design.

In the case of IQ Feed, the incoming and outgoing message format over the socket, is a new line terminated and comma delimited text string. This is not efficient and binary encoding should be used instead.

By using a local server executable program we have not observed any significant additional CPU overhead with this kind of design on multicore CPUs. While we have not done actual performance timing in regards to the additional delay introduced by using a local network socket, it is believed that this would be insignificant and down into the microsecond level.

When using the [DTC Protocol](#), the communication between the local client and server will be extremely efficient.

7. Another advantage to this kind of design is that you allow programs from any language to interface to your services. You do not need to have a .NET version, a C++ version, a Java version, or whatever version of your API. You only need to have a different version of your executable for whatever operating systems you wish to support.

For example, the Interactive Brokers Trader Workstation software is Java and we have always connected to it using a socket from the Sierra Chart native C++ program. The two programs have never caused any interference to each other.

8. It may be your position that socket programming is difficult, and that providing a client-side API with functions that can be called and callback functions, that you are making things easier for the programmer. As we have stated in the [Reasons Client-Side API Components Are Not Supported](#) section, for us that is most certainly not the case and things become dramatically more difficult.

In our experience socket programming is easy. However, in the case of when using [DTC](#), wrapper classes which handle the network socket communication are in the process of becoming available. More on this in the next list item below.

If your current API makes a callback on another thread, then you have significantly increased the complexity of using your API requiring many advanced programming considerations, at least for C++. Therefore, you have not made the interface to your systems easier. This is explained in the [Reasons Client-Side API Components Are Not Supported](#) section.



9. In order to alleviate your users of socket programming, what we recommend doing is to write a class in the most popular languages which handles the socket communication to your executable server program. These classes will provide the Request functions and Response callbacks. These classes will also handle launching the server executable program. In the case of [DTC](#), these classes are becoming available. The user would take the source code for the class for their language and they would just simply include it into their project and use it in the most appropriate way.
10. In the case of web browsers, they recognize the importance of using a separate process to run code developed by other developers. Google Chrome will run every webpage within a separate process (Quote: *Chrome uses a "multiple processes architecture", which means its processes are designed to work independent of one another. So issues in one tab should not affect the performance of other tabs or the overall responsiveness of the browser.*). Opera and Firefox use plug-in containers that run as a separate process as well. This is a very important point which gives this proposal sensibility and credibility.
11. We recognize that we could wrap your API library component into a executable ourselves and establish a socket connection to it. However, we would not do that for many of the same reasons stated in [Reasons Client-Side API Components Are Not Supported](#). Additionally, it only makes sense that the service provider is fully responsible for all of the code running within the local server process and provide a direct local socket connection to their service.

There are 2 existing client-side API components that we work with (One of them being Rithmic) where it is less efficient for us to wrap the API component ourselves rather than the original developer. The reason for this is that every time a trade occurs, there is a separate call back from their API with this one single trade. When that trade is received, we would write into the local socket. We would have to do a separate write for every single trade. In the case where there is hundreds of symbols being followed, or there is very high volume for a particular symbol, this is a lot of writing into the local socket. This is definitely not efficient and causes excessive CPU usage. Whereas if the original API developer were to do this, they could implement this more efficiently. When they read from the remote socket to their remote server, they may have a buffer full of 100 trades received in that moment, they could then fill out a outbound memory block with all of those 100 trades, use overlapped I/O, and pass that to the local socket write function. Since we do not have access to the API source code, do not wish to be involved in another developers code, it is most efficient that the API component developer, follow this proposed design rather than us patching it.

12. It is our position, that if this is the only method you provide for interfacing to your systems, assuming you do not provide a direct connection to your backend systems, it will be embraced by everyone, and they will be very pleased with it. There can only be acceptance of the model described here. You should not hesitate to offer this out of fear of somehow it not being accepted. IQ Feed, a high-performance data feed uses it. And this proposal is coming from us and we have established ourselves as a provider of a very well respected and high-performance trading and charting program. We have already established a position that if you provide a client-side API component which runs in the address space of a program, we will not use your service. In addition to all of

this, you are going to greatly simplify support for yourselves because your only responsibility is to support your client-side server executable program.

13. **Automatic Updating:** Another advantage to using a separate executable which provides access to your Data and/or Trading services is that you can build into it, automatic updating of your program files so you can be sure that all users are running the latest files. You do not have to rely upon outside developers to do any updating. Sending outside developers library files that have to be compiled into a program, is a very inefficient way of distributing an update.
14. You will have the ability to include pop-up messages that you want to send your users if you have a separate server executable.
15. When using a separate process to interface to your backend systems, when the client-side developer has a problem in their program with a memory leak or an exception, then both sides will know for certain that the problem is within the developers own program and not your API because your API is in a separate process. This is also another big advantage to using a separate process.

---

## FIX Protocol

---

The [FIX Protocol](#) was meant as a solution to some of the issues explained on this page. While for us implementing our own FIX message handler for sending and receiving messages is no problem at all and what we have done especially being that we have stringent programming standards, it can be difficult for an individual programmer writing something for their own use or for a small user base.

For this reason it is clear that Data and Trading services that want to provide as much access to their services as possible to the average programmer, avoid the use of FIX to access their systems. Although some of them do support it as an alternative.

There are FIX engines out there with make integration easier to FIX, but we have found they are of poor quality, difficult to use, do not meet our standards, or they are prohibitively expensive.

The other problem with FIX that we have heard is that it is not efficient for market data. Our experience in a native C++ program with our own FIX processing functions, is that the overhead of FIX is of virtually no significance. And definitely would be of no significance if a separate CPU core is used for message processing. However, we have no experience with using FIX on the server side serving hundreds or thousands of clients.

The solution to these issues with FIX, is to use the open specification Data and Trading Communications (DTC) protocol which is documented on the [DTC](#) page.

---

## Recommended Protocol

---

For the technical specifications which detail an ideal protocol supported by Sierra Chart, refer to the open specification [Data and Trading Communications Protocol](#).

---

## Minimum Required Quality of Data or Trading Service for Integration

---

For Sierra Chart to undertake development effort to integrate to a Data or Trading service, the following are the requirements:

- The service must be consistently reliable and have a proven track record.
- The service must use a protocol-based method of integration and this protocol must be thoroughly documented.
- The interface documentation needs to be up-to-date, accurate, orderly and easily accessible and found.
- The protocol must be logically and orderly designed.
- The service must be well engineered.
- The service must be well supported.
- The service must be under active development and maintenance.
- The service must be willing to accept requests for improvements or changes and implement those which are regarded as critical in order to accomplish a reliable and quality integration between Sierra Chart and that service.
- In the case of trading we will only work with the FIX and DTC Protocols.
- There must be no in-process API component method of integration or a DLL that we would have to create that runs within a separate process provided by the services provider.
- If there is not a not a direct connection to a remote server, then this is considered a negative and will require special consideration.
- When requesting market data for a symbol, or submitting an order for a symbol, the symbol itself must be used and not some other "identifier" of some kind which has to be requested from the server ahead of time.

The above are considered critical in order for Sierra Chart users and for Sierra Chart development to have a reasonable working relationship with that service and not get burdened with spending time on continuous problem resolutions which is very detrimental for us and for the users.

If the Data or Trading service does not meet the above requirements, but will work towards generally meeting those requirements in a reasonable amount of time, then this will satisfy our requirements as well.

And it must be understood by Trading services that an unreliable service or poorly designed interface to that service can and does result in financial losses. It is for this reason, that we will not accept substandard trading interfaces. A trading interface must be based upon the FIX or DTC Protocols. We will not accept anything else as a matter of policy.

---

\*Last modified Wednesday, 22nd February, 2023.