Creating a New Data Provider

When connecting the Marketcetera Automated Trading Platform to an external data source, like a market data provider

Step-by-step guide

1. Create a class that implements `org.marketcetera.marketdata.core.provider.MarketDataProvider`
2. Add that class to the application context

There, wasn't that easy? Like all things in life worth getting, though, the devil is in the details. Let's take a closer look.

Implement MarketDataProvider

You can just create a class that implements `MarketDataProvider`, however, it's probably going to be easier to create a class that extends `org.marketcetera.marketdata.core.provider.AbstractMarketDataProvider` (which itself implements the necessary interface) instead. So, do that. Then, implement the following abstract methods:

doStart

Just like it sounds! Implement the steps necessary to start the adapter. This usually includes doing some initialization on libraries from the provider, opening sockets, etc. The details differ depending on the external source.

doStop

Like doStart, but the opposite. Shut it all down.

doMarketDataRequest

Execute a market data request from the provider. You're going to be handed two objects: `MarketDataRequestAtom` and `MarketDataRequest`. The request atom is a symbol/content tuple, like "top-of-the-book for IBM", e.g. Just make data flow from the provider for that. This method will be invoked for each atom generated from a request. The complete request is provided for context as well, but just generate data for the atom.

Almost all market data providers are asynchronous and streaming. So, return from this method as soon as the request is made, don't wait for market data to start arriving. More on that later. The superclass takes care of reference counting, etc, so you don't have to.

doCancel

Cancel a market data request subscription for the given atom.

And that's (almost) it!

There are two more pieces of information you need:

1. Before your adapter can return events, it needs to call `AbstractMarketDataProvider.addSymbolMapping`. This method maps the symbol in an atom to a subclass of `org.marketcetera.trade.Instrument`, as in:

   ```
   addSymbolMapping("IBM",new Equity("IBM"));
   ```

   This allows the parent class to associate the atom to a particular instrument.
2. To return events, call `AbstractMarketDataProvider.publishEvents` using the `org.marketcetera.event.Content` from the atom that generated the request and the `org.marketcetera.trade.Instrument` you associated the atom with above.

Add to Application Context

The easiest way to add your provider to the application context is to add a bean to `strategyengine.xml` like this:

```
<bean id="myFeed" class="com.marketcetera.marketdata.myfeed.MyFeed">
  <property name="providerRegistry" ref="marketDataManager"/>
</bean>
```

The `providerRegistry` points to the existing `MarketDataManager` implementation, which is how Marketcetera knows about your provider. You can also set other properties you might need here, like hostname, port, username, password, etc. Just add matching getters/setters to the class and additional property elements.

That really is it!

To make it easy to resolve symbols to instruments, use the symbol resolver built into Marketcetera. To make it show up in your provider, add this to your bean:

```
<bean id="myFeed" class="com.marketcetera.marketdata.myfeed.MyFeed">
  <property name="providerRegistry" ref="marketDataManager"/>
  <property name="symbolResolver" ref="symbolResolverService"/>
</bean>
```

and this to your feed class:

```
public void setSymbolResolver(SymbolResolverService inService)
{
    symbolResolverService = inService;
}
public SymbolResolverService getSymbolResolverService()
{
    return symbolResolverService;
}
private SymbolResolverService symbolResolverService;
```

To use it:

```
protected void doMarketDataRequest(MarketDataRequest inCompleteRequest, MarketDataRequestAtom
inRequestAtom)
{
    Instrument resolvedInstrument = symbolResolverService.resolveSymbol(inRequestAtom.getSymbol);
    addSymbolMapping(inRequestAtom.getSymbol(), resolvedInstrument);
    // do whatever to make the data flow - remember the instrument above, you'll need it
}
```

You can adapt the behavior of the `SymbolResolverService` if you want. Look at `application.xml` for the bean that defines it. You can add your own symbol resolvers if you want.

- Creating a New Data Provider