Development Guide

AlgoTrader

Version 4.5

by Andy Flury, Oleg Kalnichevski, Ricardo Ribeiro, Robert Kolar, Marek Zganiacz, Jakub Chodorowicz, Rubén Fanjul, and Roger Langen

Table of Contents

List of Figures

List of Tables

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions. If not, alternative but equivalent typefaces are displayed.

1.1. Typographic Conventions

The following **typographic** conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

System input, including shell commands, file names and paths, and key caps and key-combinations are presented as follows.

> To see the contents of the file `my_next_bestselling_novel` in the current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the symbol connecting each part of a key-combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**-**Alt**-**F1** to switch to the first virtual terminal. Press **Ctrl**-**Alt**-**F7** to return to the X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph are presented as follows.

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles are presented as follows.

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

[1] https://pagure.io/liberation-fonts

> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to the document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all distinguishable by context.

Note the shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

Italics denotes text that does not need to be imputed literally or displayed text that changes depending on circumstance. Replaceable or variable text is presented as follows.

> To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and the username on that machine is john, type `ssh john@example.com`.
>
> The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `home` file system, the command is `mount -o remount /home`.
>
> To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`* .

Note the words in italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text entered when issuing a command or for text displayed by the system.

1.2. Pull-quote Conventions

Two commonly multi-line data types are set off visually from the surrounding text.

Output sent to a terminal is presented as follows:

```
books        Desktop   documentation  drafts  mss    photos   stuff  git
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are presented and highlighted as follows:

```


package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient {

  public static void main(String args[]) throws Exception {
    InitialContext iniCtx = new InitialContext();
    Object ref = iniCtx.lookup("EchoBean");
    EchoHome home   = (EchoHome) ref;
    Echo echo = home.create();

    System.out.println("Created Echo");

    System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
  }
}
```

1.3. Notes and Warnings

Finally, three visual styles are used to draw attention to information that might otherwise be overlooked.

Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but might lead to a missed out on a trick that makes life easier.

Introduction

This document provides additional information on the internal implementation of AlgoTrader for cases when clients which to make changes to the platform or extends its functionality.

Note

A source code license is required to update the internal parts of AlgoTrader

Building AlgoTrader

AlgoTrader can be built from its source either via command line or via Eclipse

> **Note**
>
> AlgoTrader based trading strategies can be developed and started without building AlgoTrader first

2.1. Command Line

To build AlgoTrader via command line please perform the following steps.

2.1.1. Git Checkout

If one hasn't installed git, please refer to git installation in the Reference Documentation (chapter 2.1.1. Prerequisites)

Perform a Git clone from the command line:

```
git clone https://gitlab.algotrader.ch/main/algotrader.git
```

> **Note**
>
> User name and password will be provided when signing up for an AlgoTrader license

2.1.2. Maven Build

Execute the following maven command to build all maven projects

```
mvn clean install
```

> **Note**
>
> When running the build process for the first time, this will take a few minutes since all maven dependencies have to be downloaded.

2.1.3. Docker Build

Execute the following Docker command to build the AlgoTrader Docker image:

```
docker build -t docker.algotrader.ch/algotrader/algotrader:latest .
```

2.2. Eclipse

To build AlgoTrader from within Eclipse please follow this process.

2.2.1. Git Checkout

- Inside Eclipse switch to the Java Perspective (`Windows --> Open Perspective --> Java`)
- Click File / Import / Git / Projects from Git / Clone URI
- Select the following URI *https://gitlab.algotrader.ch/main/algotrader.git*
- Enter User and Password (provided when licensing AlgoTrader)
- Click Next
- Select `master`
- Click Next
- Select Import existing projects and click Next
- Select the project `algotrader`
- Select the new project `algotrader`, right click and select Import / Maven / Existing Maven Projects and select:
  - `common`
  - `core`
  - `conf`
  - `launch`
- Click Finish

This will result in the following Eclipse projects:

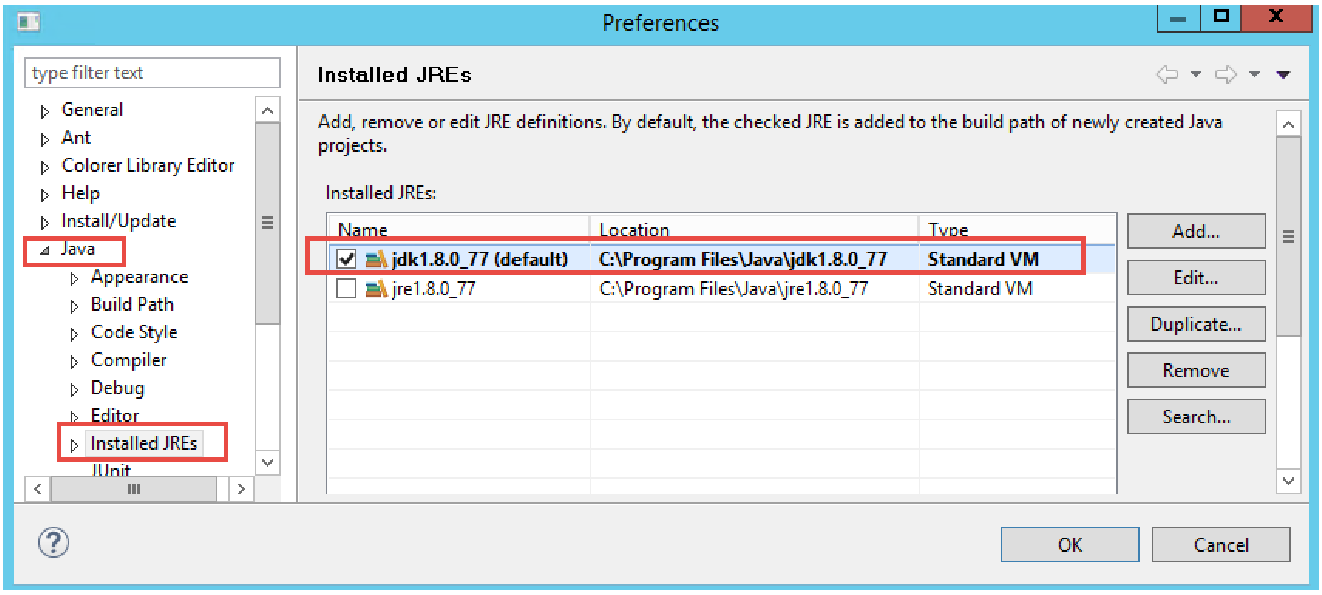
- `algotrader-common`
- `algotrader-core`
- `algotrader-conf`
- `algotrader-launch`

Note

The compilation will show errors, which should go away after the next section has been completed.

2.2.2. Maven Build

Before running the maven build from within Eclipse please make sure that the default Eclipse Java runtime environment is pointing to a Java JDK. To verify please go to Window / Preferences / Java / Installed JREs. If the default JRE is pointing to a Java JRE, then please add a reference to the Java JDK.



Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

Name	Location	Type
☑ jdk1.8.0_77 (default)	C:\Program Files\Java\jdk1.8.0_77	Standard VM
□ jre1.8.0_77	C:\Program Files\Java\jre1.8.0_77	Standard VM

Figure 2.1. Eclipse default JRE

To generate the code right click on the project `algotrader` inside Eclipse and select / Run As / Maven install. This will generate all maven modules.

Now refresh all projects. Eclipse will compile all java code automatically. In case there is an error message Project configuration is not up-to-date with pom.xml... on any of the projects the please select: `Maven->Update Project` from the project context menu.

2.2.3. Docker Build

The AlgoTrader Docker Image needs to be built from the command line (see above).

Domain Model

3.1. Security Visitors

The *Visitor Pattern*[1] is a way of separating an algorithm from an object structure on which it operates. Using this pattern it is possible to implement custom Logic per Entity without polluting the Entity code itself.

AlgoTrader provides the interface `ch.algotrader.visitor.EntityVisitor` which must be implemented by all Entity Visitors. Each Entity Visitor has two generic type parameters R and P. R is the return type (or `java.lang.Void`) returned by all visit methods and P is an arbitrary parameter object that can be added to the visit methods.

In addition there is the `ch.algotrader.visitor.PolymorphicEntityVisitor` which reflects the entire inheritance tree of all Securities. For example if there is no `visitFuture` method the `PolymorphicEntityVisitor` will automatically invoke the `visitSecurity` method.

The accept method of each Entity can be used to process an arbitrary Visitor like this:

```
entity.accept(MyVisitor.INSTANCE);
```

In AlgoTrader there are two Visitors available which are used by the AlgoTrader Server

InitializingVisitor

Is used to make sure certain Hibernate Entity References are initialized / loaded.

ScalingVisitor

is used to scale quantities and prices

SecurityVolumeVisitor

Is used to determine if a particular Security is supposed to report volumes

TickValidationVisitor

Used to validate a Tick by rules defined per Security

3.2. Data access objects (DAOs)

The AlgoTrader DAO framework of consists of several main components

BaseEntityI

`BaseEntityI` represents an abstract serializable persistent entity with a synthetic identifier of type long.

ReadOnlyDao

`ReadOnlyDao` represents an interface for common retrieval operations for entity classes.

[1] https://en.wikipedia.org/wiki/Visitor_pattern

ReadWriteDao

`ReadWriteDao` extends `ReadOnlyDao` and represents an interface for common retrieval and mutation operations.

AbstractDao

`AbstractDao` abstract class serves as a generic base class for data access classes. It provides the most common operations to retrieve, update and delete entities as well as to build HQL and native SQL queries.

It is possible to add custom DAOs to the platform. To accomplish this one needs to create a DAO interface extending either `ReadOnlyDao` or `ReadWriteDao`, add custom operations such as entity specific finders and then create a custom DAO class extending `AbstractDao` and implementing the custom DAO interface.

```
public class MyEntity implements BaseEntityI {

    private long id;
    private String name;

    @Override
    public long getId() {
        return this.id;
    }

    protected void setId(final long id) {
        this.id = id;
    }

    @Override
    public boolean isInitialized() {
        return true;
    }

    public String getName() {
        return this.name;
    }

    public void setName(final String name) {
        this.name = name;
    }
}
```

```
public interface MyEntityDao extends ReadWriteDao<MyEntity> {

    public MyEntity findByName(String name);

}
```

```
@Repository
public class MyEntityDaoImpl extends AbstractDao<MyEntity> implements MyEntityDao {

    public MyEntityDaoImpl(final SessionFactory sessionFactory) {
        super(MyEntity.class, sessionFactory);
    }

    @Override
    public Strategy findByName(final String name) {

        return findUniqueCaching(
          "from MyEntity where name = :name", QueryType.HQL, new NamedParam("name", name));
    }

}
```

HQL and SQL queries used by AlgoTrader DAO components are externalized and stored in `Hibernate.hbm.xml` file. This allows for better management and for easier re-use of queries.

```
<query name='Strategy.findByName'>
<![CDATA[
    from StrategyImpl
    where name = :name
]]>
</query>
```

Queries can be accessed from DAO classes or custom components by their names

```
public class StrategyDaoImpl extends AbstractDao<Strategy> implements StrategyDao {
    ...
    @Override
    public Strategy findByName(final String name) {
        return findUniqueCaching(
            "Strategy.findByName", QueryType.BY_NAME, new NamedParam("name", name));
    }
```

3.3. Services

3.3.1. Private Services

Table 3.1. Private Services

Service	Description
`AlgoOrderService`	Order Services responsible for handling of `AlgoOrders` (delegates to `AlgoOrderExecServices`)
`AlgoOrderExecService`	Abstract Base Class for all Algo Execution Order Services
`ExternalAccountService`	Abstract Base Class for all external Account Interfaces
`ExternalMarketDataService`	Abstract Base Class for all external Market Data Interfaces
`FixSessionService`	Exposes properties of FIX sessions
`ForexService`	Responsible for the FX Hedging functionality
`GenericOrderService`	Parent Class for all Order Services
`MarketDataPersistenceService`	Responsible for persisting Market Data to the database
`OrderExecutionService`	Responsible for handling of persistence and propagation various trading events such as order status update and order fills as well as maintaining order execution status in the order book.
`OrderPersistenceService`	Responsible for persisting `Orders` and `OrderStatus` to the database
`ReconciliationService`	Responsible for reconciliation of reports provided by the broker
`ResetService`	Responsible for resetting the DB state (e.g. before the start of a simulation)
`ServerLookupService`	Provides internal data lookup operations to other server side services
`SimpleOrderService`	Order Service responsible for handling of Simple Orders (delegates to `SimpleOrderExecServices`)
`SimpleOrderExecService`	Abstract Base Class for all Simple Order Execution Services
`StrategyPersistenceService`	Handles persistence of Strategy Entities
`TransactionPersistenceService`	Responsible for the persistence of Transactions, Positions updates and Cash Balance updates to the database
`TransactionService`	Responsible for handling of incoming Fills

3.3.2. Order Services

All `OrderServices` are derived from the `GenericOrderService`. All Order Services based on Fix are derived from `FixOrderService` which in terms has subclasses for Fix versions 4.2 and 4.4.

3.3.3. Market Data Services

All `MarketDataServices` are derived from the general `ExternalMarketDataService`. All `MarketDataServices` based on Fix are derived from `FixMarketDataService` which in terms has subclasses for Fix versions 4.2 and 4.4.

3.3.4. Historical Data Services

Historical Data Services are used to download aggregated Market Data Events from the Market Data Provider.

3.3.5. Reference Data Services

Reference Data Services are used to download current option and future chains as well as information about stocks.

Java Environment

4.1. AlgoTrader Project Structure

The Framework AlgoTrader consists of the following Sub-Projects:

algotrader
: the main project

algotrader-common
: contains java code accessible to trading strategies

algotrader-core
: contains internal java code needed by the AlgoTrader Server

bootstrap/algotrader-conf
: contains the configuration files needed by the AlgoTrader Server

bootstrap/algotrader-launch
: contains the Run Configurations

algotrader-archetype-esper
: Maven archetype for creating new AlgoTrader Esper based strategies

algotrader-archetype-simple
: Maven archetype for creating new AlgoTrader simple strategies

algotrader-test
: contains integration tests based on *JUnit*[1]

4.1.1. common project

the AlgoTrader common project has the following structure:

Table 4.1. common project

Directory	Description
`src/main/java`	manually created class files
`src/main/resouces`	manually created configuration files
`target/generated-sources/main/java`	generated class files
`src/test/java`	JUnit test cases
`src/test/resources`	test resources

[1] http://junit.org/

4.1.2. core project

the AlgoTrader core project has the following structure:

Table 4.2. core project

Directory	Description
`src/main/java`	manually created class files
`src/main/resouces`	manually created configuration files
`src/test/java`	JUnit test cases
`src/test/resources`	test resources
`bin`	shell start scripts
`files`	files which are created/exported by the system or imported into the system
`log`	log files

4.1.3. conf project

the AlgoTrader conf project has the following structure:

Table 4.3. conf project

Directory	Description
`src/main/resources`	all the properties files

4.1.4. launch project

the AlgoTrader conf project has the following structure:

Table 4.4. launch project

Directory	Description
`/`	all the Run Configuration files

4.1.5. strategy projects

Strategy projects typically have the following structure:

Table 4.5. strategy projects

Directory	Description
`src/main/java`	java code
`src/main/resources`	config files
`bin`	Run Configuration and shell start scripts

Directory	Description
`log`	log files

4.2. Java Packages & Classes

For a full list of java packages and classes please visit our *Javadoc*[2]

4.3. Maven Environment

AlgoTrader uses *Maven*[3] as its build management framework. Every project/module therefore has it's own pom.xml (Project Object Model) defining its structure and dependencies.

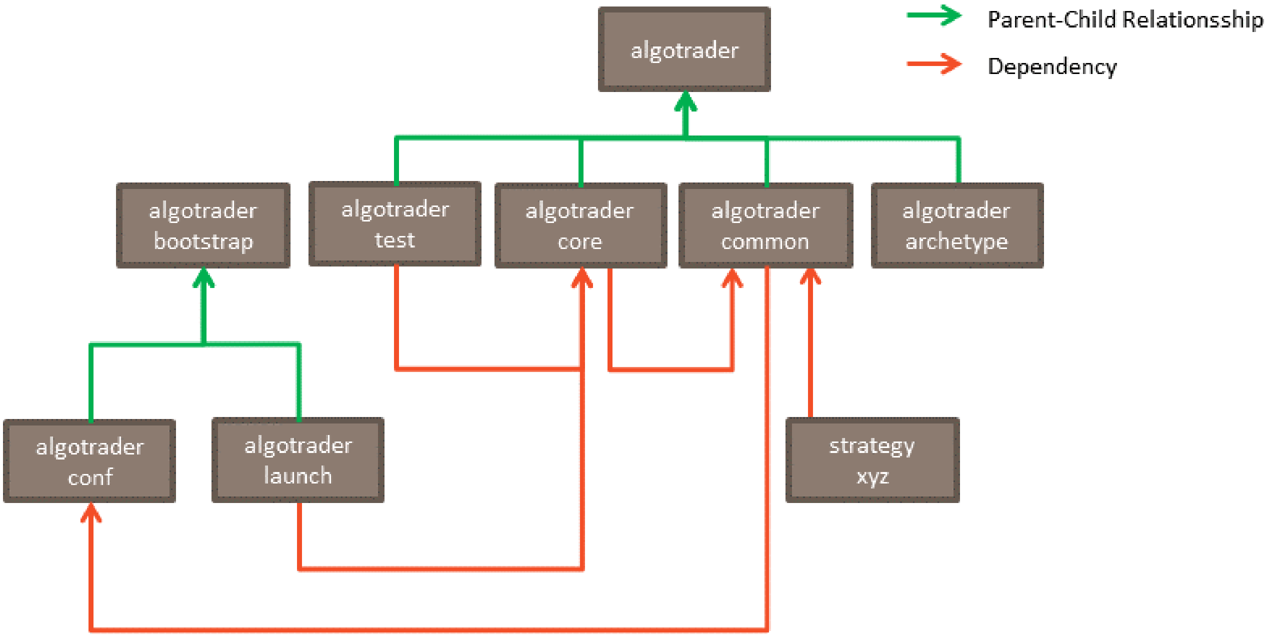


Figure 4.1. IB Services

4.3.1. Maven assemblies

AlgoTrader provide a maven assembly for the core module. The core assembly declares all components required to run AlgoTrader server in distributed mode.

The maven assembly definition file is located in `/algotrader-core/src/main/assembly/server-bin.xml`

The following command generates binary deployment packages from assembly descriptors.

```
mvn clean package
```

[2] http://doc.algotrader.ch/javadoc/index.html
[3] http://maven.apache.org/

Binary deployment packages generated from Maven assemblies come in two varieties: tar.gz and zip, the former being more optimized for Unix-like operating systems while the latter being suitable for Windows platforms.

Code Generation

Java Entities, Entity Interfaces and Value Objects are created by the means of the *Hibernate Tools project*[1] using the *hbm2java code exporter*[2].

The Hibernate Tools project provides the code generator as an Eclipse plugin as well as a set of Ant tasks. AlgoTrader provides a custom maven plugin called `maven-codegen-plugin` which wraps the code generator.

```
<groupId>algotrader</groupId>
<artifactId>model-codegen-plugin</artifactId>
<name>Model code generator plugin</name>
<version>0.1.5</version>
```

The `maven-codegen-plugin` has been added to the file `/algotrader/common/pom.xml`

```
<plugin>
    <groupId>algotrader</groupId>
    <artifactId>model-codegen-plugin</artifactId>
    <version>0.1.5</version>
    <executions>
        <execution>
            <id>generate-model</id>
            <goals>
                <goal>codegen</goal>
            </goals>
            <configuration>
                <templates>
                    <template>
                        <file>pojo/Pojo.ftl</file>
                        <pattern>{package-name}/{class-name}.java</pattern>
                    </template>
                    <template>
                        <file>pojo/Interface.ftl</file>
                        <pattern>{package-name}/{class-name}I.java</pattern>
                    </template>
                    <template>
                        <file>pojo/ValueObject.ftl</file>
                        <pattern>{package-name}/{class-name}VO.java</pattern>
                    </template>
                    <template>
                        <file>pojo/ValueObjectBuilder.ftl</file>
                        <pattern>{package-name}/{class-name}VOBuilder.java</pattern>
```

[1] http://hibernate.org/tools/

[2] http://docs.jboss.org/tools/latest/en/hibernatetools/html/ant.html#d0e4821

```
                    </template>
                </templates>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The above configuration generates the following artifacts for each Java Entity defined in the Hibernate mapping files:

- Entity
- Entity interface
- Value Object
- Value Object Builder

The code generator uses Hibernate mapping files which are located in `/algotrader/common/src/main/resources` in combination with *Freemarker*[3] templates which are located in `/algotrader/common/pojo` . These templates are based on the original version supplied with the Hibernate Tools project but have been augmented to produces Java code needed by AlgoTrader. For this purpose several custom attributes have been added to Hibernate mapping files:

Table 5.1. Hibernate mapping files - custom meta attributes

Tag	Description
implements	The interface this Entity implements
generated-class	Name of the main Entity class file
class-code	Extra class code generated into Entity classes
interface-code	Extra class code generated into Entity interfaces
vo-code	Extra class code generated into Value Objects
class-description	Javadoc comment for classes
field-description	Javadoc comment for fields
use-in-equals	should this property be used in the equals method
property-type	the class to be used for this property

Generated code is placed under the directory `/algotrader/common//target/generated-sources/main/java`.

[3] https://freemarker.apache.org/

Database

6.1. Database scripts

- `src/main/resources/db/h2` H2 scripts loaded at runtime
  - `h2.sql` h2 database schema file
- `src/main/scripts/db/mysql` MySql scripts
  - `create-fix-messagestore.sql` creates database tables uses for Fix message persistence
  - `datetime_milli_precision.sql` converts all `datetime` fields to usage of millisecond (requires MySql 5.6.4)
  - `mysql-to-h2.sql` batch script to convert MySQL schema to h2 schema.
  - `mysql-to-h2-data.sql` batch script to convert MySQL data to h2 schema.
  - `reset-db.sql` reset script which resets the tables cash_balance, order, portfolio_value, subscription, transaction and position.

6.2. Transaction Handling

Using Spring Transaction Boundaries are declared on Services using the annotation *`@Transactional`*. Transaction Boundaries are handled by the `org.springframework.transaction.interceptor.TransactionInterceptor`. A typical declaration looks like this:

```
@Transactional(propagation = Propagation.REQUIRED)
public void saveTransaction(final Transaction transaction) {

}
```

Market Data

All Market Data Interfaces have a set of unique artifacts:

- Configuration Files (`conf-ib.properties` and `conf-bb.properties`)
- Adapter Classes (e.g. `IBAdapter`) responsible for management of Sessions. Adapters are available over JMX
- Session Classes (e.g. `IBSession`). Representing an individual Market Data Session
- Message Handler Classes (e.g. `BBMessageHandler`) responsible for receiving `MarketDataEvents` and propagating them into Esper
- Esper Modules (e.g. `market-data-ib`) responsible for processing and filtering of `MarketDataEvents`

Processing of Market Data is handled through the `MarketDataService`, which calls the market data provider specific `ExternalMarketDataService` implementations. Every market data service has to provide implementation of this interface e.g. (e.g. `IBMarketDataServiceImpl` or `BBMarketDataServiceImpl`).

The most important methods provided by the `MarketDataService` are `subscribe` and `unsubscribe`. Through the use of these methods new Market Data can be subscribed and unsubscribed. Subscribed securities are persisted within the DB-table *`subscription`*. The actual subscription of securities to the external broker is done through the market data provider specific `MarketDataService`.

Market data provider interfaces are responsible for receiving market data provider specific Market Data and sending them into the Esper Service Instance of the AlgoTrader Server. The Esper Service Instance will then convert these Events into generic `MarketDataEvents` (i.e. Ticks or Bars) which will be propagated to subscribed Strategies.

Adapters

8.1. Fix Interface

The Fix infrastructure consists of the following classes:

Table 8.1. Fix Infrastructure

Class / Interface	Description
`Session`	A `Session` represents a connection to a broker / exchange / market data provider
`Application`	For each `Session` an `Application` object is created. It will forward incoming messages to the corresponding `MessageHandlers`
`DefaultFixApplication`	
`FixApplicationFactory`	Is responsible for the creation of `Applications`
`DefaultFixApplicationFactory`	
`FixMultiApplicationSessionFactory`	Creates a `Session` and `Application` using the specified `FixApplicationFactory` according to the following steps: • lookup the `FixApplicationFactory` by its name • create an `Application` • create a `DefaultSessionFactory` • create a `Session`
`ExternalSessionStateHolder`	Represents the current state of a `Session` (i.e. `DISCONNECTED`, `CONNECTED`, `LOGGED_ON` and `SUBSCRIBED`)
`DefaultFixSessionStateHolder`	
`MarketDataFixSessionStateHolder`	A `ExternalSessionStateHolder` for market data sessions that will subscribe to securities as soon as the session is logged on.
`FixOrderIdGenerator`	Generator for Fix Order Ids. The default implementation reads the last Order Ids from the Fix log on start-up.
`DefaultFixOrderIdGenerator`	
`FixAdapter`	Management Adapter for the Fix environment. Allows the creation of dynamic sessions, sending Messages and managing Order Ids.
`DefaultFixAdapter`	
`ManagedFixAdapter`	Manageable implementation of a `FixAdapter` (based on JMX)
`FixEventScheduler`	QuickFix/J currently supports daily sessions (with a daily session 7 times a week) and weekly sessions (with one
`DefaultFixEventScheduler`	

Class / Interface	Description
`EventPattern`	weekly session). However some brokers (e.g. JP Morgan) use daily sessions during workdays. To accomplish this scenario, AlgoTrader allows creation of a weekly `logon/logoff` event (e.g. Mo 08:00:00 and Fr 18:00:00) using Esper Statements
`FixSocketInitiatorFactoryBean`	A Spring Factory Bean that creates the `SocketInitiator` necessary for all Fix Sessions.
`Fix42MarketDataMessageHandler`	Message Handler for incoming Fix market data messages. These classes need to be sub classed by the corresponding market data interface. Messages are propagated into the Esper Engine.
`Fix44MarketDataMessageHandler`	
`Fix42OrderMessageHandler`	Message Handler for incoming Fix Execution Reports. These classes need to be sub classed by the corresponding order interface. Messages are propagated into the Esper Engine.
`Fix44OrderMessageHandler`	

8.2. Bloomberg

The Bloomberg infrastructure consists of the following classes:

Table 8.2. Bloomberg Infrastructure

Class / Interface	Description
`BBAdapter`	Management Adapter for the Bloomberg environment. Allows to start and stop Bloomberg Sessions
`BBConstants`	Bloomberg Constants used by the Bloomberg interface
`BBIdGenerator`	Generator for Bloomberg Request Id's
`BBMessageHandler`	Message Handler for incoming Bloomberg messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.
`BBMarketDataMessageHandler`	
`BBSession`	Represents a connection to the Bloomberg service

8.3. IB Native Interface

The IB infrastructure consists of the following classes:

Table 8.3. IB Infrastructure

Class / Interface	Description
`IBSession`	An `IBSession` represents a connection to the TWS or IB Gateway

Class / Interface	Description
`IBSessionStateHolder`	Represents the current state of a `Session` (i.e. `DISCONNECTED`, `CONNECTED`, `IDLE`, `LOGGED_ON` and `SUBSCRIBED`)
`DefaultIBSessionStateHolder`	
`IBAdapter`	Management Adapter for the IB environment. Allows to connect and disconnect IB Sessions as well as retrieval of the current `ConnectionState`
`DefaultIBAdapter`	
`AbstractIBMessageHandler`	Message Handler for incoming IB messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.
`DefaultIBMessageHandler`	
`IBOrderMessageFactory`	Factory for order messages.
`DefaultIBOrderMessageFactory`	
`IBExection`	Details of an individual order execution
`IBExections`	Collection (internally represented by a map) of all order executions
`IBPendingRequest`	Details of an individual data request such as historic data or contract details
`IBPendingRequests`	Collection (internally represented by a map) of all pending data requests

8.4. QuantHouse

The QuantHouse infrastructure consists of the following classes:

Table 8.4. QuantHouse Infrastructure

Class / Interface	Description
`QHAdapter`	Management Adapter for the QuantHouse environment.
`QHMessageHandler`	Message Handler for incoming QuantHouse messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.
`QHSessionStateHolder`	Holds the current state of the QuantHouse session

8.5. Binance

Table 8.5. Main service classes

Service class name	Functionality
`BNCOrderServiceImpl`	Order submission
`BNCMarketDataServiceImpl`	Market data feed
`BNCReferenceDataServiceImpl`	Reference data - instruments

Service class name	Functionality
`BNCAccountServiceImpl`	Account info

Table 8.6. Main classes

Class name	Functionality
`BNCAdapter`	Main adapter class

`BNCOrderMessageHandler,`
`BNCMarketDataMessageHandler`
	Live market data subscription connectors

`BNCServiceWiring, BNCWiring`	Spring config files

8.6. Bitfinex

Table 8.7. Main service classes

Service class name	Functionality
`BFXOrderServiceImpl`	Order submission
`BFXMarketDataServiceImpl`	Market data feed
`BFXReferenceDataServiceImpl`	Reference data - instruments
`BFXAccountServiceImpl`	Account info retrieval

Table 8.8. Main classes

Class name	Functionality
`BFXRestAdapter`	Main adapter class
`BFXWebSocketAdapter, BFXMessageHandler`	WebSockets connectivity logic
`BFXServiceWiring, BFXWiring`	Spring config files

8.7. Bitflyer

Table 8.9. Main service classes

Service class name	Functionality
`BFLOrderServiceImpl`	Order submission
`BFLMarketDataServiceImpl`	Market data feed
`BFLReferenceDataServiceImpl`	Reference data - instruments
`BFLAccountServiceImpl`	Account info

Table 8.10. Main classes

Class name	Functionality
`BFLRestAdapter`	Main adapter class

Class name	Functionality
`BFLPubNubAdapter`, `BFLMarketDataMessageHandler`	Live market data subscription connectors
`BFLServiceWiring, BFLWiring`	Spring config files

8.8. BitMEX

Table 8.11. Main service classes

Service class name	Functionality
`BMXOrderServiceImpl`	Order submission: Market, Limit, Stop and Stop-Limit orders with DAY, GTC, FOK or IOC Time in Force
`BMXMarketDataServiceImpl`	Market data feed
`BMXReferenceDataServiceImpl`	Reference data - instruments
`BMXAccountServiceImpl`	Account balances and info retrieval

Table 8.12. Main classes

Class name	Functionality
`BMXRestAdapter`	Main adapter class
`BMXWebSocketAdapter`, `BMXMarketDataMessageHandler`, `BMXOrderMessageHandler`	Live market data subscription and order status update connectors
`BMXServiceWiring, BMXWiring`	Spring config files

8.9. Bitstamp

Table 8.13. Main service classes

Service class name	Functionality
`BTSFixOrderServiceImpl`	Order submission
`BTSFixMarketDataServiceImpl`	Market data feed
`BTSReferenceDataServiceImpl`	Reference data - instruments
`BTSAccountServiceImpl`	Account balances and info retrieval

Table 8.14. Main classes

Class name	Functionality
`BTSRestAdapter`	Main adapter class
`BTSFixMessageHandler`	Facade class for handling Fix messages

Class name	Functionality
`BTSServiceWiring, BTSWiring, BTSFixServiceWiring, BTSFixWiring`	Spring config files

8.10. CoinAPI

Table 8.15. Main service classes

Service class name	Functionality
`CNPHistoricalDataServiceImpl`	Historical data
`CNPMarketDataServiceImpl`	Market data feed
`CNPReferenceDataServiceImpl`	Reference data - instruments

Table 8.16. Main classes

Class name	Functionality
`CNPRestAdapter`	Main adapter class
`CNPWebSocketAdapter, CNPMessageHandler`	WebSockets connectivity logic
`CNPServiceWiring, CNPWiring`	Spring config files

8.11. Coinigy

Table 8.17. Main service classes

Service class name	Functionality
`CNGOrderServiceImpl`	Order submission
`CNGMarketDataServiceImpl`	Market data feed
`CNGReferenceDataServiceImpl`	Reference data - instruments
`CNGAccountServiceImpl`	Account info

Table 8.18. Main classes

Class name	Functionality
`CNGRestAdapter`	A REST client for the Coinigy REST API endpoint. Allows placing Limit and Stop Limit orders as well as retrieval of reference and account data.
`CNGSocketClusterAdapter, CNGMarketDataMessageHandler`	Socket Cluster client for the Coinigy WebSocket API endpoint that provides real-time market data feeds. Message Handler for incoming market data updates received through the WebSocket channels.
`CNGServiceWiring, CNGWiring`	Spring config files

8.12. CoinMarketCap

Table 8.19. Main service classes

Service class name	Functionality
`CMCHistoricalDataServiceImpl`	Daily historical data
`CMCReferenceDataServiceImpl`	Reference data - instruments

Table 8.20. Main classes

Class name	Functionality
`CMCRestAdapter`	Main adapter class
`CMCWiring, CMCServiceWiring`	Spring config files

Execution Algos

9.1. Development of Execution Algos

Additional Execution Algos can be added to the system with minimal effort. Execution Algos consist of the following artifacts

- a subclass of `AlgoOrder`. An `AlgoOrder` should be a plain old Java bean and contain no execution logic.
- an object that represents the state of an algo order execution which needs to subclass `AlgoOrderStateVO`. Please note that if state objects contains elements that are potentially threading unsafe, access to those elements must be synchronized!
- an esper module (optional). This module can optionally provide statements to cancel a child order, modify a child order, send the next child order, etc.
- an implementation of `AlgoOrderExecService` interface. It is generally recommended to subclass `AbstractAlgoOrderExecService` and implement its protected methods that represent various algo specific handling logic. The `#handleValidateOrder` method must implement algo specific order validation logic. The `#handleSendOrder`, `#handleModifyOrder` and `#handleCancelOrder` method must implement algo specific order execution, modification and cancellation logic respectively.
- a corresponding entry in the `ch.algotrader.enumeration.OrderType` class.
- a Spring wiring within `ch.algotrader.wiring.core.ServiceWiring`.

  Custom `AbstractAlgoOrderExecService` implementation can also tap into event streams pertaining to the algo order execution. The `#handleChildFill` and `#handleChildOrderStatus` methods can used to provide custom handling of fills and status events for child orders executed by the algo and to update its internal state. The `#handleOrderStatus` method can be used to update the internal state in response to transition of the algo order from one execution phase to another

  In addition the class can implement any additional logic needed in conjunction with custom Esper statements.

> **Note**
>
> It is important that `AlgoOrderExecService` implementations are built to be state-less. They must store all details pertaining to execution of algo orders in their respective state objects.

The `OrderService` is aware of all `AlgoOrderExecService` instances declared in the Spring application context of the server process. Custom `AlgoOrderExecService` implementations also get automatically recognized as long as they are declared in the same application context. The `OrderService` delegates handling of individual orders to their respective algo service based on the order type. It is important for classes implementing `AlgoOrderExecService` to correctly implement its `#getAlgoOrderType` method.

Spring Services

AlgoTrader is built on top of the Spring Framework, which uses `BeanFactory` and `ApplicationContext` to locate Spring Beans (= AlgoTrader-Services).

The *Spring*[1] web site provides *documentation*[2] such as *'The IoC container'*[3] as an introduction.

AlgoTrader provides the class `ch.algotrader.ServiceLocator` which will instantiate the adequate `BeanFactories` & `ApplicationContexts` for a given operational mode depending on the specified *`BEAN_REFERENCE_LOCATION`*.

In Simulation mode the AlgoTrader Server as well as the Strategy run inside the same JVM.

In Live-Trading mode the AlgoTrader Server and strategies can be run in different JVMs. Through the use of `RmiServiceExporters` and `RmiProxyFactoryBean`, Strategies can call Services from the AlgoTrader Server. Behind the scenes this is handled transparently through RMI.

Please see *Remoting and web services using Spring*[4] for further details.

10.1. Wiring Factories

AlgoTrader provides the following Wiring Factories, which are instantiated by the `ServiceLocators`:

Table 10.1. Bean Reference Factories

Wiring Factory	Description
`LocalWiringFactory`	used when no remoting or strategy related functionality is needed (e.g. `HistoricalDataStarter`)
`EmbeddedWiringFactory`	used in Live Trading Mode when running in embedded mode
`ServerWiringFactory`	used in Live Trading Model on the server side
`ClientWiringFactory`	used by the Strategies in Live Trading Mode to connect to Services through RMI
`SimulationWiringFactory`	used in simulation

10.2. ApplicationContext

AlgoTrader provides the following Wiring Classes and Application Context XML-Files :

[1] http://spring.io

[2] https://docs.spring.io/spring/docs/4.3.18.RELEASE/spring-framework-reference/htmlsingle/

[3] https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans

[4] https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#remoting

Table 10.2. Application Context Files

ApplicationContext	Description	Examples
`CommonWiring`	contains common beans	Configuration, JMX Management, Esper Engine, Event Dispatching, Logging & Web Configuration
`ServerWiring`	contains server bean definitions	ActiveMQ, SSL & REST Controllers
`CoreWiring`		Caching, Configuration, DAOs, Data Source and Transaction Management, Hibernate, Live Cycle Manager, Server Esper Engine, Simulation & InfluxDB
`AdapterWiring`	contains adapter related beans	market data and trading adapters
`ExternalWiring`	contains external services	market data and trading services
`ClientServicesWiring`	contains all client service beans	`SubscriptionService`, `MarketDataCache`, `LiveCycleManager`, `CacheManager` & `LookupService`
`EmbeddedJMSWiring`	contains JMS embedded beans	Embedded JMS
`ClientJMSWiring`	contains JMS client beans	Remote JMS
`JMSWiring`	contains JMS server beans	Server JMS
`applicationContext-export-remoteServices.xml`	contains all `RmiServiceExporters` to make Services remotely available	
`applicationContext-import-remoteServices.xml`	contains all `RmiProxyFactoryBean` to call remote Services from the Strategies through RMI	
`applicationContext-client-*.xml`	to be provided by the Strategies	Strategy services, JMS queues
`applicationContext-env.xml`	contains environment specific bean definitions	Mail Settings, Reconciliation Dispositions

The following table shows which Wiring Classes and `ApplicationContext` is referenced by which Wiring Factory:

Table 10.3. Application Context References

ApplicationContext	Local	Embedded	Server	Client	Simulation
`CommonWiring`	x	x	x	x	x

ApplicationContext	Local	Embedded	Server	Client	Simulation
`ServerWiring`		x	x		
`CoreWiring`	x	x	x		x
`AdapterWiring`	x	x	x		
`ExternalWiring`	x	x	x		
`ClientServicesWiring`		x		x	
`EmbeddedJMSWiring`					
`ClientJMSWiring`				x	
`JMSWiring`			x		
`applicationContext-export-remoteServices.xml`			x		
`applicationContext-import-remoteServices.xml`				x	
`applicationContext-client-*.xml`		x		x	x
`applicationContext-environment.xml`	x	x	x		

10.3. Abstract Services

For many use cases abstract services are in place which can be extended for different broker interfaces.

For abstract services which have only one active implementation (through profiles), an alias can be defined for the concrete service (e.g. `iBHistoricalDataService`). A typical Spring Bean Alias definition looks like this:

```
@Profile("bBHistoricalData")
@Bean(name = {"bBHistoricalDataService", "historicalDataService"})
public HistoricalDataService createBBHistoricalDataService(
  final BBAdapter bBAdapter,
  final SecurityDao securityDao,
  final BarDao barDao) {
    return new BBHistoricalDataServiceImpl(bBAdapter, securityDao, barDao);
}
```

At runtime this service can now be accessed through its alias (e.g. *`historicalDataService`*)

For abstract services which might have more than one active implementation (through profiles), aliases are not available. In this case the following method can be used to look up all available concrete services that extend the abstract service (see `OrderService` for an example):

```
ServiceLocator.instance().getServices(OrderExecService.class)
```

10.4. Service initialization order

`InitializingServiceI` interface represents an abstract service that requires special initialization after it has been created and fully wired in the Spring application context. The life-cycle manager automatically detects such services and calls their `InitializingServiceI#init()` method before proceeding with initialization of strategy engines and deployment of strategy modules. This helps ensure that all external interfaces are fully initialized prior to strategy activation. `InitializationPriority` annotation can be used to explicitly mark a service either as a part of the platform core or as a part of an external broker interface. Core services have higher priority than all other services and get initialized before all others.

Events and Messaging

AlgoTrader provides a sophisticated event dispatching and messaging sub system. In Simulation Mode as well as Embedded Mode Event Propagation takes places within the JVM. In Distributed Live Trading Mode Event Propagation from the AlgoTrader Server to the strategies (and between strategies) happens via JMS & *ActiveMQ*[1]

11.1. Embedded ActiveMQ message broker

AlgoTrader makes use of an embedded instance of ActiveMQ message broker for message dispatch and delivery. It presently supports three transports by default:

- VM: for internal message delivery
- TCP: for message delivery to strategies running in distributed mode
- WebSockets: for message delivery to the HTML5 front-end. WebSockets transport can also be used for message delivery to any arbitrary external application that supports WebSockets transport and STOMP messaging protocols.

11.2. STOMP messaging over WebSockets transport

AlgoTrader employs STOMP messaging protocol over WebSockets transport to implement multi-topic, multi-client message delivery based on the Publish-Subscribe pattern. AlgoTrader acts as a message producer that generates messages representing various system or trading related events and publishes them to predefined topics. Browsers running the HTML5 UI client (and potentially any external application supporting STOMP over WebSockets) act as message consumers that subscribe to message topics of interest such as market data, orders, order status updates, position changes, executed transactions and so on. Consumers express their interest in a particular type of event by subscribing to message topics. Consumers should no longer need to filter out unwanted messages. They are expected to subscribe only to a subset of messages they are interested in.

For further details on the STOMP protocol please visit the *STOMP website*[2].

AlgoTrader publishes events to multiple event topics.

Table 11.1. Event topics

Topic format	Type of events
tick.<security_id>	`TickVO`
order.<strategy_name>.<order_int_id>	`OrderVO`

[1] http://activemq.apache.org/

[2] https://stomp.github.io/

Topic format	Type of events
order-status.<strategy_name>.<order_int_id>	`OrderStatusVO`
transaction.<strategy_name>.<uuid>	`TransactionVO`
position.<strategy_name>.<id>	`PositionVO`
cash-balance.<strategy_name>.<id>	`CashBalanceVO`
market-data-subscription.<strategy_name>.<security_id>.<feed_type>	`MarketDataSubscriptionVO`
log-event.<priority>	`LogEventVO`

Topics are organized by name spaces. A consumer wishing to receive market data for security with id 12 only can subscribe to the following topic:

```
tick.12
```

A consumer wishing to receive market data all securities can subscribe to the following wild card topic.

```
tick.*
```

Strategy specific events are organized by strategy name. A consumer wishing to receive order status updates for the order with internal id 10 issued by strategy MY_STRATEGY can subscribe to the following topic

```
order-status.MY_STRATEGY.10
```

A consumer wishing to receive order status updates for all orders issued by strategy MY_STRATEGY can subscribe to the following wild card topic

```
order-status.MY_STRATEGY.*
```

The `*` wild card selects all elements within the same namespace

A consumer wishing to receive order status updates for all orders of all strategies can subscribe to the following wild card topic

```
order-status.>
```

The > wild card selects all topics within the same namespace and their sub-namespaces.

In order to ensure optimal performance of HTML5 clients AlgoTrader can throttle market data event delivered by the WebSockets transport. The embedded message broker by default attempts to ensure that the total rate of events per connection does not exceed 50 per second. At the same time instruments with infrequent market data updates are not throttled if their total event rate is below 0.1 per second (less that one event every 10 seconds).

Throttling rates can be adjusted by changing the following configuration parameters:

```
activeMQ.maxRatePerConnection = 50

activeMQ.minRatePerConsumer = 0.1
```

In JavaScript STOMP messages can be consumed like this:

```
<html>
<head>
    <script src="https://unpkg.com/@stomp/stompjs@4.0.6/lib/stomp.min.js"></script>
    <script type="text/javascript">
        var ws = new WebSocket("ws://localhost:61614", "stomp");
        var stompClient = Stomp.over(ws);
        stompClient.connect({}, function(frame) {
            stompClient.subscribe('/topic/tick.*', function(message){
                console.log(JSON.parse(message.body));
            });
        });
    </script>
</head>
</html>
```

For further details please visit the *STOMP JavaScript documentation*[3].

11.3. Embedded Jetty HTTP server

In addition to RMI transport AlgoTrader provides a RESTful interface over HTTP/S. RESTful endpoints serve only a subset of AlgoTrader functionality primarily required for HTML5 front-end. While being a subset it nonetheless represents the core functionality of the platform.

HTTP/HTTPS transport is powered by the embedded Jetty HTTP server and REST endpoints are managed by Spring Web framework.

11.4. RESTful interface

RESTful endpoints largely expose the same interface as Spring services exposed via RMI. REST controllers must follow RESTful semantic and also use immutable value objects for input / output representation.

AlgoTrader RESTful controllers serve several purposes:

- they provide request / response mapping to RESTful endpoints and enforce a contract conforming with the principles of RESTful interface;

[3] http://jmesnil.net/stomp-websocket/doc/

- they de-serialize endpoint input to immutable value objects and if necessary convert them to input structures expected by the services;
- they convert service output structures to immutable value objects that can be serialized by the endpoints;
- they optionally perform additional input validation and mapping;
- they map service exception to endpoint responses with an appropriate error status;

The following RESTful controller provides a list of all accounts available in the system:

```
@CrossOrigin
@RequestMapping(path                                =                                 "/
account", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
public List<AccountVO> getAccounts() {

    return lookupService.getAllAccounts().stream()
            .map(Account::convertToVO)
            .collect(Collectors.toList());
}
```

In this example the `@CrossOrigin` annotation marks endpoint as permitting cross origin requests. The `@RequestMapping` annotation defines various aspects of request / response mapping: *`path`* attribute defines path element of the request URI, *`method`* attribute defines the request method (such as GET, POST, PUT or DELETE), *`produces`* attribute defines expected media type of response body. The endpoint method implementation performs conversion of `Account` Entity objects to `AccountVO` objects, which are then serialized to JSON data stream by the framework.

For more detailed explanation of REST controllers and Web annotations please refer to Spring documentation.

AlgoTrader uses SWAGGER to document the individual REST endpoints, see:

11.5. JSON data binding

AlgoTrader uses a consistent message format based on JSON for the event topics described above as well for RESTful endpoints. The use of JSON as a message format and WebSockets / HTTP as transports enables interoperability with a wide variety of modern development platforms and languages.

Using the JSON messages in combination with the AlgoTrader RESTful endpoints and WebSocket/STOMP topics any popular development language can be used to build trading strategies making use of the AlgoTrader platform, for example:

- C#
- C++
- Python

- R
- MatLab
- JavaScript / NodeJS

HTTP GET endpoints can easily be queried via the Browser. To retrieve a JSON formatted list of all accounts open to the following URL in the Browser:

```
http://localhost:9090/rest/account
```

In addition one case use *Curl*[4], a popular utility for execution of HTTP requests. The following example shows how to retrieve a JSON formatted list of all accounts by executing HTTP `GET` request:

```
$ curl -X GET http://localhost:9090/rest/account -i
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.6.v20151106)

[{"id":100,"name":"IB_NATIVE_TEST","active":true,"broker":"IB",...},
{"id":101,"name":"IB_FIX_TEST","active":false,"broker":"IB",...},
{"id":103,"name":"DC_TEST","active":false,"broker":"DC",...},
{"id":104,"name":"JPM_TEST","active":false,"broker":"JPM",...},
{"id":105,"name":"RT_TEST","active":false,"broker":"RT",...},
{"id":106,"name":"LMAX_TEST","active":true,"broker":"LMAX",...},
{"id":107,"name":"FXCM_TEST","active":false,"broker":"FXCM",...},
{"id":108,"name":"CNX_TEST","active":false,"broker":"CNX",...}]
```

Similarly one can request AlgoTrader to subscribe to market data for security with id 11 by executing HTTP `PUT` request:

```
$ curl -X PUT -H "content-type: application/json"
  http://localhost:9090/rest/subscription/marketdata/subscribe \
  -d "{\"strategyName\":\"SERVER\",\"securityId\":11,\"subscribe\":true}" -i
HTTP/1.1 200 OK
Content-Length: 0
Server: Jetty(9.3.6.v20151106)
```

11.6. EventDispatcher

The `EventDispatcher` API represents a platform wide communication interface capable of submitting events to multiple `Engine` instances and event listeners both inside the same JVM as well as to separated JVMs. The `EventDispatcher` acts as an event bus for the AlgoTrader platform and individual strategies. The following Recipients are available

[4] https://curl.haxx.se/

Table 11.2. Event Recipients

Scope	Local			Remote
	Server Engine	Strategy Engines	Event listeners	External processes
ALL	X	X	X	X
ALL_LOCAL	X	X	X	
ALL_LOCAL_STRATEGIES		X	X	
ALL_LOCAL_LISTENERS			X	
ALL_STRATEGIES		X	X	X
SERVER_LISTENERS	X		X	
SERVER_ENGINE_ONLY	X			
ALL_LISTENERS			X	X
REMOTE_ONLY				X

Note

AlgoTrader uses immutable value objects to represent events sent to strategy Engines and strategy processes.

11.7. Event listeners

`EventListener` represents a generic communication interface to receive events from multiple event producers both in-process and remote. `EventListenerRegistry` interface represents a registry of event listeners used internally by the AlgoTrader Server process as well as individual strategy processes. One can register listeners for arbitrary event classes, which enables strategies to generate custom events either through Esper statements or in Java code and consume them internally or propagate them to other strategy processes.

The AlgoTrader platform provides a number of event listeners for common event types such as market data events, order events, external session events, life-cycle events, and a few others. Components that implement those event interfaces which are declared in the Spring application context get automatically registered with the platform upon initialization..

Table 11.3. Standard event listener classes

`BarEventListener`	receives `BarVO` events generated from tick events by individual strategies or fed from an external source
`EntityCacheEventListener`	receives `EntityCacheEvictionEventVO` generated by the cache manager
`FillEventListener`	receives `FillVO` events generated by trading interface adapters

`GenericEventListener`	receives `GenericEventVO` events generated by strategies
`GenericTickEventListener`	receives `GenericTickVO` events generated by market data interface adapters or fed from an external source
`LifecycleEventListener`	receives `LifecycleEventVO` generated by the life-cycle manager
`OrderCompletionEventListener`	receives `OrderCompletionVO` events generated by the Server Engine
`OrderEventListener`	receives `OrderVO` events generated by the order service
`OrderStatusEventListener`	receives `OrderStatusVO` events generated by trading interface adapters
`PositionEventListener`	receives `PositionNutationVO` events generated by the transaction service
`QueryCacheEventListener`	receives `QueryCacheEvictionEventVO` generated by the cache manager
`QuoteEventListener`	receives `QuoteVO` events (`BidVO` or `AskVO`) generated by market data interface adapters or fed from an external source
`SessionEventListener`	receives `SessionEventVO` generated by market data and trading interface adapters
`TickEventListener`	receives `TickVO` events generated by market data interface adapters or fed from an external source
`TradeEventListener`	receives `TradeVO` events generated by market data interface adapters or fed from an external source
`TransactionEventListener`	receives `TransactionVO` events generated by the transaction service

11.8. JMS Destinations

Events are delivered to strategies via JMS. The following JMS Destinations are defined by the system

- one market data topic (`MARKETDATA.TOPIC`) defined inside `applicationContext-server.xml`: A Topic where all market data events are pushed into. On every event the *`securityId`* is set as a property, which can be used by the strategies to select market data events for securities subscribed to.

- one strategy queue per strategy (`XXX.QUEUE`) defined inside `applicationContext-client-xxx.xml`: A strategy specific Queue for messages like `OrderStatus`, `Fills` & `Transactions`. As JMS queues are persistent, messages will be delivered, even if a strategy was down, at the time of message creation.

- one generic topic (`GENERIC.TOPIC`) defined inside `applicationContext-server.xml`: A Topic for Generic Messages. Any strategy can send messages into this Topic. On every event the *`className`* of the event is set as a property, which can be used by the strategy to select event types that it is subscribed to.

Since market data events and generic events are pushed into two topics that are available to all strategies, strategies have to select appropriate messages on their own. This is the job of the `SubscriptionService`. It will modify the selectors on `MessageListenerContainer` accordingly and invoke the corresponding methods on the (server-side) `MarketDataService` (e.g. to request market data for additional securities).

Important

Strategies should never call the `MarketDataService` directly but instead call the `SubscriptionService`.

Configuration and Preferences API

12.1. Config Providers

AlgoTrader provides extensive support for configuration and customization of platform functions as well as of strategy specific settings.

The cornerstone of the configuration and preference APIs is the `ConfigProvider` interface that can be used to obtain arbitrary typed configuration parameters.

```
public interface ConfigProvider {

    <T> T getParameter(String name, Class<T> clazz);

    Set<String> getNames();
}
```

`DefaultSystemConfigProvider` is the default implementation of `ConfigProvider` based on Spring `ConversionService` and is internally backed by a thread safe Map. The default provider makes use of `ConversionService` conversion framework to convert the content of the internal parameter map to the desired type. One can customize the process of parameter conversion by using a custom `ConversionService` implementation.

`ConfigParams` is a utility facade for `ConfigProvider` exposing a set of getter methods for common data types such `Boolean`, `Integer`, `Long`, `Double`, `BigDecimal`, `URL` and `URI`. This class can be used by trading strategies that need to dynamically resolve configuration parameters at runtime.

`DefaultConfigLoader` is used to read configuration parameters from property files.

12.2. Config Beans

AlgoTrader also provides commonly used parameters in a form of plain Java beans referred to as Config beans. Common configuration are represented by `CommonConfig`. Core platform parameters are represented by `CoreConfig`. Instances of these classes are immutable and can be shared by multiple components and multiple threads of execution.

`ConfigBeanFactory` class can be used to create instances of Config beans based on configuration parameters using `@ConfigName` constructor parameter annotations. This factory is used to build standard `CommonConfig` and `CoreConfig` but it can also be used to build arbitrary Config beans for a trading strategy using the following convention

```
public final class StratConfig {
    private final String textParam;
    private final boolean boolParam;
    private final BigDecimal decimalParam;

    public StratConfig(
        @ConfigName(value = "my.text") final String textParam,
        @ConfigName(value = "my.bool") final boolean boolParam,
       @ConfigName(value = "my.decimal", optional = true) final BigDecimal decimalParam) {

        this.textParam = textParam;
        this.boolParam = boolParam;
        this.decimalParam = decimalParam;
    }

    public String getTextParam() {
        return textParam;
    }

    public boolean isBoolParam() {
        return boolParam;
    }

    public BigDecimal getDecimalParam() {
        return decimalParam;
    }
}
```

Each constructor parameter of a Config bean must be annotated with `@ConfigName` containing the parameter name. The config parameter type will be inferred from the constructor argument type. If a parameter is null able and might be undefined in the config property files it can be marked as `optional`.

Standard platform Config beans such as `CommonConfig` and `CoreConfig` are declared in the Spring application context and get automatically injected into all beans that require configuration. One can also add strategy specific Config beans using the following bean definition:

```
<bean id="stratConfig" class="ch.algotrader.config.spring.ConfigBeanFactoryBean">
    <constructor-arg index="0" ref="configLocator"/>
    <constructor-arg index="1" value="my.strategy.StratConfig"/>
</bean>
```

Standard as well as strategy specific Config beans can be conveniently accessed using Spring SPEL expressions to wire other beans in the same Spring application context.

```
<bean id="MyObject" class="...">
    <constructor-arg value="#{@stratConfig.requestUri}"/>
```

```
</bean>
```

In addition it is possible to reference individual beans (e.g. config beans) directly within Spring wired classes

```
private @Value("#{@configParams.accountId}") long accountId;
```

12.3. Config Locator

Even though it is preferable to dependency injection services provided by Spring application context to obtain configuration details required by custom components, in certain cases it may be necessary for unmanaged beans to get hold of Config beans. This can be done through the global `ConfigLocator`

```
ConfigParams configParams = ConfigLocator.instance().getConfigParams();
CommonConfig commonConfig = ConfigLocator.instance().getCommonConfig();
StratConfig stratConfig = ConfigLocator.instance().getConfig(StratConfig.class);
```

Processes and Networking

13.1. Processes

The following Services and Process are used by the system:

Table 13.1. Services and Processes

Service / Process	Description
AlgoTrader Server	This is the main AlgoTrader process
Strategies	In Live Trading Mode each strategy can run in its own Java process or within the same process as the AlgoTrader Server (one strategy only). In simulation mode, the strategies run within the same process as the AlgoTrader Server
MySql	Main database process. Alternatively to using MySql the embedded in-memory database H2 can be run within the process of the AlgoTrader Server in simulation mode.
InfluxDB	The InfluxDB database used for storage of historical data

If the AlgoTrader Server and the strategies are running within separate processes, individual strategies can be stopped / altered / restarted independent of each other and the AlgoTrader Server.

13.2. Sockets

Table 13.2. Sockets

Socket	Description	Default Ports
RMI Exporter	Remote access to Spring services	1199
ActiveMQ TCP	ActiveMQ TCP transport	61616
ActiveMQ WS	ActiveMQ WebSockets transport	61614
ActiveMQ WSS	ActiveMQ Secure WebSockets transport	61613
HTTP	Jetty HTTP transport	9090
HTTPS	Jetty HTTPS transport	9443
MySql	MySql database connection	3306
InfluxDB	InfluxDB connection	8086
IB Gateway	defined by IB Gateway configuration	4001
Bloomberg Terminal	BBComm.exe	8194
Fix	Fix Connections	varies

13.3. RMI

The system defines an RMI services through Spring Remoting (RMI Registry 1199): `RmiServiceExporter` (defined in `applicationContext-export-remoteServices.xml`)

Hibernate Sessions and Caching

AlgoTrader uses Hibernate for accessing and persisting objects to the database.

14.1. Hibernate Caching

In order to prevent having to access the database on every single request, Hibernate provides two types of caches:

First Level Cache

The First Level Cache is always associated with the current Session. Hibernate uses this cache by default. Its main purpose is to reduce the number of SQL queries needed to execute within a given transaction. Instead of updating after every modification done to an object separately, it updates the database only once at the end of the transaction. At the end of a Session, the attached First Level Cache will be destroyed. In case an Object, that is already loaded into the current First Level Cache, is modified outside the Session, the First Level Cache will not get notified of the change.

Second Level Cache

The Second Level Cache is always associated with the Session Factory. While processing transactions, Hibernate stores objects at the Session Factory level, so that those objects will available to the entire system. Whenever a new query is executed, Hibernate will first check with the Second Level Cache to see whether the objects are available in the Cache.

Both First and Second Level Cache require a Hibernate Session. Creation of a Session is usually very quick (a few milliseconds). This mechanism is therefore fine for any request-response based system. However this approach is not feasible for a trading application. A trading application typically receives several thousand market data events per second. Ideally these market data events have to be matched to the latest data stored in the database (e.g. Security related information, current Positions, executed Trades, etc.). Opening a new Hibernate Session for every market data event, to synchronize related objects (like corresponding Security), is much too expensive!

For this purpose AlgoTrader introduces a Level-Zero Cache

14.1.1. Level-Zero Cache

AlgoTrader Level-Zero Cache is an additional Caching Level on top of Hibernate First and Second Level Cache which has the following features:

- Level-Zero Cache is a pure Java based Cache
- Level-Zero Cache does not require an active Hibernate Session
- Objects available inside the Level-Zero Cache will be delivered instantaneously and do not introduce any additional latency
- Level-Zero Cache does refresh objects at the same time a database update occurs

- Level-Zero Cache is a read-only Cache, it does not provide any sort update functionality. Changes to Entities retrieved from the Level-Zero Cache will never be persisted to the database. In order to modify objects in the database Spring services and Hibernate DAOs have to be used
- Level-Zero Cache preserves object identity, so graphs and cyclical references are allowed. Therefore objects retrieved from the Level-Zero Cache can be compared using the `equals()` method but also using the comparison operator ==.
- Level-Zero Cache provides ad hoc initialization of Hibernate Proxies and Persistent Collections. Newly initialized Proxies and Persistent Collections will be added to the Cache automatically.
- Level-Zero Cache does not provide any Passivation or Eviction. All Objects stay in memory. It is therefore not recommended to use the Level-Zero Cache for Objects that are only needed one time, especially if there is a large number of those objects.
- Level-Zero Cache is available to both the JVM containing the AlgoTrader Server as well as all Strategy JVMs.
- No Proxies, no Byte Code Instrumentation and no Annotations are needed for Level-Zero Cache to work

By using the Level-Zero Cache it is possible to work on fully up-to-date versions of Entities without introducing any latency penalties.

Access to the Level-Zero Cache is provided by the class `ch.algotrader.cache.CacheManagerImpl` which is exposed as a Spring Bean named `cacheManager`. The `CacheManagerImpl` provides these public methods to access the Level-Zero Cache:

`get`
: gets an Entity of the given *`clazz`* by the defined *`id`*

`getAll`
: gets all Entities of the given *`clazz`*

`contains`
: checks whether an Entity of the given *`clazz`* and *`id`* is in the Cache

`initialzeProxy`
: lazy-initializes the give *`key`* of the specified *`entity`*

`initialzeCollection`
: lazy-initializes the give *`collection`* of the specified *`entity`*

`find`
: performs the given HQL *`query`* by passing defined *`namedParameters`*

`find`
: performs the given HQL *`query`* by passing defined *`maxResults`* (passing zero will return all elements) *`namedParameters`*

`findUnique`
: performs the given unique HQL *`query`* by passing defined *`namedParameters`*

`clear`
: clears the entire cache

Like Hibernate First and Second Level Cache the AlgoTrader Level-Zero Cache will first check if the requested object is available inside the Cache. If not, the object will be retrieved via the `ch.algotrader.hibernate.GenericDao` and stored in the Cache.

The class `EntityCache` is responsible for caching of Entities (handled by the `EntityHandler`) and Entity-Collections (handled by `CollectionHandler`). When adding a new Entity to the cache, the `EntityHandler` and `CollectionHandler` traverse the entire object graph of initialized Entities and Collections and store all of them in separate nodes of the internal Hash Map Cache. Java reflection is used for the object graph traversal (on new objects or object updates) .

The class `QueryCache` is responsible for caching of Hibernate Query Results. Query Results are cached in reference to all involved tables. So whenever one of those referenced tables is modified, the corresponding Query Cache Entry is removed (detached).

Through the class `ch.algotrader.wiring.server.CacheWiring` `net.sf.ehcache.event.CacheEventListenerAdapter` are registered with each Ehcache element. These are either `EntityCacheEventListener` for Entity caches or `CollectionCacheEventListener` for Collection Caches. These `EventListeners` emit `CacheEvictionEvents` via the AlgoTrader `EventDispatcher` to `CacheManager` of the Server and Strategies (running in remote JVMs) to get notified every time an Entity / Collection is updated or the `UpdateTimestampsCache` (related to the Hibernate `StandardQueryCache`) has been updated.

Note

The `LookupService` uses the `CacheManager` for all lookup operations instead of using Hibernate DAOs directly.

Logging

15.1. Custom UI LogEventAppender

This is a special customized appender which allows to send log events to the UI. The log events are sent via JMS/STOMP and log levels and loggers are configurable. For example you could define a particular logger (e.g. some specific class) or use the Root logger configured for the desired log level, e.g. INFO or WARN. In order to have multiple loggers with multiple log levels, separate appenders must be created each with it's own filter. The sample configuration below will create two UI appenders - one with level WARN (and above), another with INFO. Loggers defined in "Loggers" section reference these appenders in such a way that INFO log entries from `PortfolioServiceImpl` as well as all the entries with level WARN and above will be sent to the UI

Log entries are wrapped inside JMS message and get propagated via WebSocket STOMP protocol to the UI. In order to consume the log message the client will have to be subscribed to specific JMS topic ("/topic/log-event."). See StompAPIUtils.js in the HTML client for the subscription logic example.

```
<Appenders>
    <LogEvent name="LogEventWARN">
        <ThresholdFilter level="WARN" onMatch="ACCEPT" onMismatch="DENY"/>
    </LogEvent>
    <LogEvent name="LogEventINFO">
        <ThresholdFilter level="INFO" onMatch="ACCEPT" onMismatch="DENY"/>
    </LogEvent>
</Appenders>
<Loggers>
    <Root level="debug">
        <AppenderRef ref="LogEventWARN"/>
    </Root>
    <Logger name="ch.algotrader.service.PortfolioServiceImpl">
        <AppenderRef ref="LogEventINFO" />
    </Logger>
<Loggers>
```