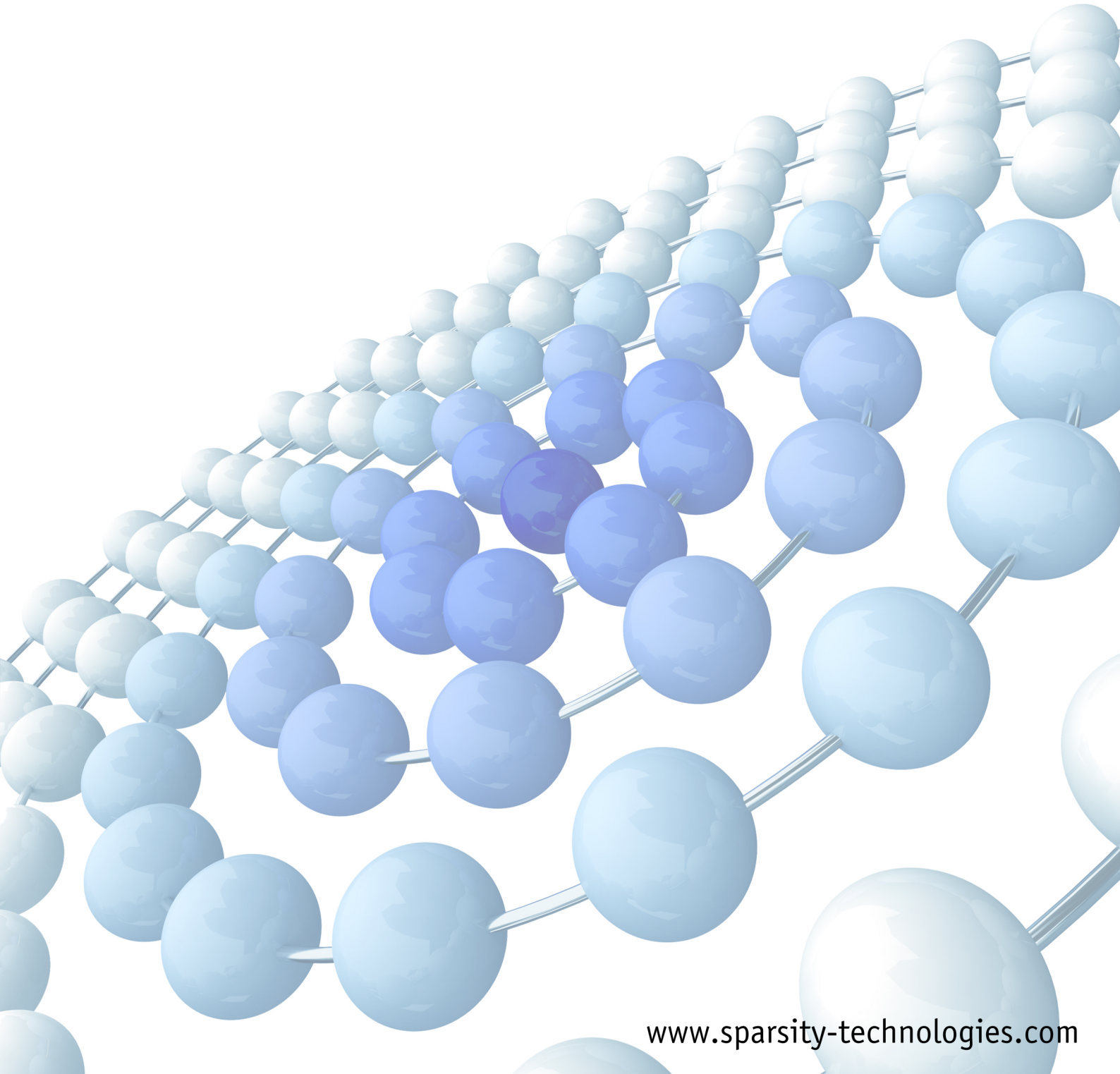
*dex

Starting Guide



www.sparsity-technologies.com

Contents

Getting started

This is the starting guide for DEX graph database. We will guide you during the creation of your first DEX graph, from downloading DEX to the execution of your graph. It should not take you much time but if you want to start faster, download DEX, download the example and run it now!.

If you are not familiar with graph databases or graph concepts, please visit this article.

The first thing you should know about the DEX graph you are going to build is that we define it as a labeled and directed attributed mutigraph. It is **labeled** because all the nodes and edges can have a type (a label) to classify them; **directed** because it supports edges with direction from the tail node to the head node, of course we support undirected edges too!; **attributed** because both nodes and edges can have one or more properties; and, finally, it is a **multigraph** because there are no restrictions on the number of edges between two nodes.

Figure 1 is an example of a DEX multigraph. Here there are two types of nodes (PEOPLE represented by a star icon and MOVIE shown as a clapperboard icon) which both have an attribute (called respectively NAME and TITLE) with a value for each of them. For instance you can see a *Scarlett Johansson* (NAME) node which will be of type PEOPLE (star icon). Also we can see two types of edges (DIRECTS shown in blue and CAST shown in orange). A directed edge has an arrow pointing to its head node (DIRECTS), while an undirected edge does not have any arrows (CAST). As many attributes as desired could be added to both edges and nodes.

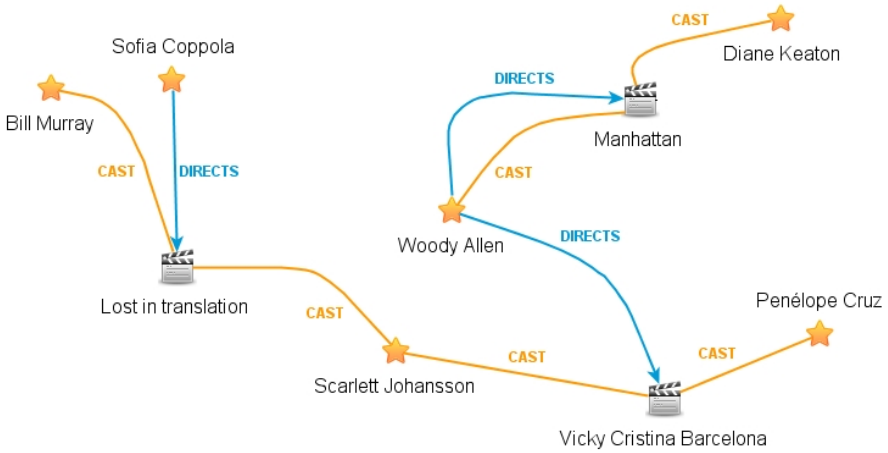


Figure 1: Hello DEX

Figure 1 above illustrates the DEX labeled and directed attributed multigraph definition. It has labels, as nodes and edges have types, is directed because edge DIRECTS has a certain direction and is a multigraph because node *Woody Allen* with node *Manhattan*, for instance, has two edges.

Following the steps in this guide will help you construct the graph illustrated in **Figure 1**.

It is also relevant to know that DEX is an embedded database, so from this point on, you should take into account the fact that installation and deployment of your graph database has to be made considering your programming language preference and framework.

Installation

Let's start with the installation of DEX graph database. DEX only needs to be downloaded and unpacked. Depending on which language you prefer, a few settings or considerations may need to be taken into account too.

By default DEX comes with a free personal evaluation license. Learn more about it in the last section of this chapter.

Download

DEX is available for several programming languages, so the first thing you should do is download the right package for your development platform.

All the packages can be downloaded here.

Available platforms:

- **Java:** This is the package for all new DEX Java projects.
- **Microsoft .NET:** DEX is natively available for .NET developers.
- **C++:** A C++ interface is also available. DEX core is C++, so we punch it another level to the API.
- **Jdex:** This is a legacy Java API that **should not be used in new projects**. We keep this API only for compatibility reasons, even though, if you are a DEX old timer you should consider switching to the new Java API.

Java developers can also get DEX through Apache Maven instead of manually downloading the packages. More information in DEX maven project.

Unpacking

Once you have downloaded DEX and unpacked it, the content should look like this:

- **doc:** This is the directory that contains DEX API **reference** documentation in html or another specific format for each platform.
- **lib:** A directory with DEX libraries. It may have subdirectories when different files are required regarding each operating system.
- **changelog.txt:** This is a text file with basic information on the latest release changes.
- **LICENSE.txt:** Contains DEX licensing information.

According to the programming language particularities, some packages may include additional contents.

Unpack the downloaded DEX package and it's ready to be used. No further installation is required.

However, some additional steps can be taken to make DEX usage easier for your platform.

Java

All the libraries required to develop a java application with DEX are contained in a jar file located at the `lib` directory (`dexjava.jar`).

If you are not using Maven, you may want to add the path to this file into the `CLASSPATH` environment variable. However, to avoid any misunderstandings, in this document we will explicitly use the file to compile the examples.

.NET

The .NET package contains two subdirectories in the lib directory (windows32 and windows64) for 32 or 64 bit systems. In each one the main library included is dexnet.dll. This is the library that you will need to include in your .NET projects.

All the other available libraries in the package are native dlls that dexnet.dll must be able to find at run time. Although it is not required, you may want to copy all these libraries to your system folder.

C++

The C++ lib folder contains native libraries for each available platform (Windows 32/64 bits, Linux 32/64 bits and MacOS 64 bits).

In Linux and MacOS, you may want to add the path to the correct subdirectory to your LD_LIBRARY_PATH environment variable.

In Windows you can copy the libraries to the system folders or just be sure to always include them in your projects.

The rest of files included in this package, like the ones in the includes directory, will be needed at compilation time. See Chapter 4.

License

Every DEX download comes with a limited personal evaluation license included that does not require any further configuration. The personal evaluation license allows the construction of graphs with up to 1M objects, which is more than enough to construct the example explained in this guide.

If you have commercial interest or need to deal with larger databases, contact us for licenses quotation on the latest release of DEX.

If you hold a license key, later in this document you will learn how to use your license for a project.

Hello DEX

If have reached this chapter you should now have DEX correctly installed in your computer and be ready to work in your development framework.

You have previously been introduced to DEX APIs and you should have chosen the language with which you feel most familiar. Here we will explain the complete construction of the HelloDEX application, including the construction of the database plus your first queries. For each step in the development we will show an example using all the available DEX programming languages; just focus on your chosen language.

Let's say Hello to DEX!

Setting up

The first step is the creation of a directory for this project and the new sample application in your development environment. We will come later in the compile and run chapter with certain modifications in the project in order to be able to run DEX.

As part of the setup, you could create a text file with the name "**dex.cfg**" (or any other name, if you explicitly load it) in the same folder where the executable file will be. This configuration file will establish the default DEX settings. This is not a required task, because you can modify these settings directly in the source code using the DexConfig class methods.

These are the most common settings that you may want to set in this file:

- **dex.license** : This option is used to set your commercial license key. By default you do not need to provide a license key.
- **dex.io.cache.maxsize** : Sets the maximum size for the cache (all pools) in megabytes. By default DEX uses all the available memory, leaving enough memory for other needs. If you are running several memory consuming applications at the same time you must consider adjusting this parameter.
- **dex.log.file** : Changes the log file path. The default value is dex.log.
- **dex.log.level** : The level of detail provided by the log file can be modified with this option. Valid values are: Off, Severe, Warning, Info, Config, Fine and Debug. The default level is Info. For HelloDEX you will not need to change this level.

A dex.cfg file could, for instance, look like this one:

```
dex.license=Your-license-key
dex.io.cache.maxsize=2048
dex.log.file=HelloDex.log
```

Where Your—license—key is the alphanumeric key provided by Sparsity Technologies when you acquire a license, the cache assigned for DEX is 2GB and the log file name is changed for HelloDEX.log.

Following this guide we will construct HelloDEX step by step, if you want to go faster, once you are done with the set up you can download the complete application, copy it in your sample application and jump to the compile and run chapter. However we recommend following the guide for a complete understanding of each of the steps of creating a graph database and querying it.

Let's now finish the set up by moving from the directories to your development framework and start coding. Before starting to create your database you should first include references to DEX: this is mandatory.

[Java]

```
import com.sparsity.dex.gdb.*;
```

[C#]

```
using com.sparsity.dex.gdb;
```

[C++]

```
#include "gdb/Dex.h"
#include "gdb/Database.h"
#include "gdb/Session.h"
#include "gdb/Graph.h"
#include "gdb/Objects.h"
#include "gdb/ObjectsIterator.h"
```

Create a DEX database

Now that we have the set up complete and we have started coding, let's create our first database. In this chapter we are going to create a simple database containing information about certain movies, their actors and directors.

The first thing you should do is to create a DexConfig object to establish the Dex main settings. That object will be created using the global DexProperties settings (initially loaded from a dex.cfg file). You do not have to change any setting directly here, but its creation is a must before creating the Dex object.

If you have a commercial license and you have not yet used it in the configuration file, you may set the license in the DexConfig using the setLicense method.

The newly created DexConfig object will be used to create a Dex object. Once you have created this object, you can create your databases.

A **new database** needs a file path (HelloDex.dex) and we can give it a name (HelloDex). This database file is where all the persistent information will be stored.

[Java]

```
DexConfig cfg = new DexConfig();
// The setLicense method is only performed if you have a license key and
// have not activated yet it using the configuration file (dex.cfg)
cfg.setLicense("Your license key");
Dex dex = new Dex(cfg);
Database db = dex.create("HelloDex.dex", "HelloDex");
```

[C#]

```
DexConfig cfg = new DexConfig();
// The setLicense method is only performed if you have a license key and
// have not activated yet it using the configuration file (dex.cfg)
cfg.SetLicense("Your license key");
Dex dex = new Dex(cfg);
Database db = dex.Create("HelloDex.dex", "HelloDex");
```

[C++]

```
DexConfig cfg;
// The setLicense method is only performed if you have a license key and
// have not activated yet it using the configuration file (dex.cfg)
cfg.SetLicense("Your license key");
Dex *dex = new Dex(cfg);
Database *db = dex->Create(L"HelloDex.dex", L"HelloDex");
```

Sessions and transactions

All the manipulation of a database must be enclosed within a session. A Session should be initiated from a Database instance. This is where you can get a Graph instance which represents the persistent graph (the graph database).

The Graph instance is needed to manipulate the Database as a graph.

Also, **temporary data** is associated to the Session, thus when a Session is closed, all the temporary data associated to that Session is removed too. Objects or Values instances or even session attributes are an example of temporary data.

You must take into account the fact that a Session is exclusive for a thread, we do not at all encourage that it be shared among threads as it will definitely raise unexpected errors.

A Session manages the transactions, allowing the execution of a set of graph operations as a single execution unit. A transaction encloses all the graph operations between the Session Begin and Commit methods, or just a single operation in autocommit mode (where no begin/commit methods are used).

Initially, a transaction starts as a read transaction and only when a method which updates the persistent graph database is found does it automatically become a write transaction. To become a write transaction it must wait until all other read transactions have finished.

Since HelloDEX is a simple example we are going to use autocommit because we don't need more complex transactions.

For more information about DEX transactions please take a look at the Session class in the reference documentation of your chosen DEX API.

[Java]

```
Session sess = db.newSession();
Graph g = sess.getGraph();
```

[C#]

```
Session sess = db.NewSession();
Graph g = sess.GetGraph();
```

[C++]

```
Session *sess = db->NewSession();
Graph *g = sess->GetGraph();
```

Set the schema

The HelloDEX sample application is a very simple movie database where we store information about movies, their actors and directors. All of them are nodes in the graph whereas the relationships between them are edges.

In a graph database we can represent movies, actors and directors using two node types:

- **MOVIE**: This type represents a movie.
- **PEOPLE**: This type represents a person involved in a movie, be they cast or crew member.

We can enrich theses by adding more information such as the movie title, year of production, or in the case of the people only their names. This information are attributes of the recently created node types:

For each **MOVIE** we may need the following attributes:

- **ID**: Unique identification for a movie.
- **TITLE**: Original title.
- **YEAR**: Year of production.

For **PEOPLE** we need these attributes:

- **ID**: Unique identification for a person.
- **NAME**: Complete name.

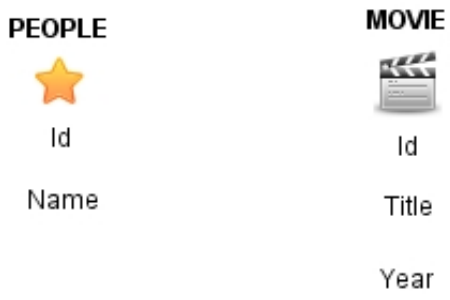


Figure 2: HelloDEX node schema

Note that the **ID** attribute is not required, but it is always useful to have a unique attribute value in order to identify each node, as the other attributes (name and title) can't be considered unique.

The **relationships** are represented by edges between the former two types of nodes. We need two types of edges:

- **CAST**: Represents the relationship between **PEOPLE** and **MOVIE** meaning that the person is part of the cast of that movie.
- **DIRECTS**: Represents the relationship between **PEOPLE** and **MOVIE** meaning that the person is the director of that particular movie.

The first one (**CAST**) is going to be an **undirected edge and it will have an attribute** (**CHARACTER**) to store the name of the role performed in that movie. In fact it is not a restricted edge, so we could use it between other types of nodes too. For example, later on we could add a new node type "ANIMAL" and use the same "CAST" edge type between "MOVIE" and "ANIMAL" nodes.

The other (**DIRECTS**) is going to be a **restricted directed edge without attributes**. It is restricted because it can only be used between **PEOPLE** and **MOVIE** nodes and directed because we only allow navigation from **PEOPLE** to **MOVIE**.

Create node types

Let's now construct the node types **MOVIE** and **PEOPLE** and then add the desired attributes to each type definition, which we have defined in our schema.

There are three types of attributes:

- **Basic**: This type of attributes cannot be used for query operations (just get and set attribute values).
- **Unique**: Attributes that work as a primary key, which means that two objects cannot have the same value for an attribute (but NULL). They can be used for query operations.

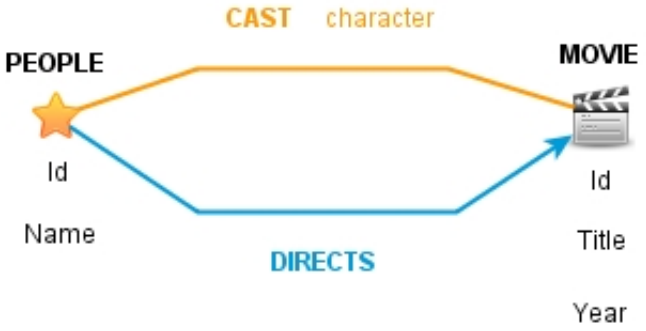


Figure 3: HelloDEX schema

- **Indexed**: Attributes that can be used for query operations.

Also you have to choose a datatype for the attributes. Available datatypes for attributes are: Boolean, Integer, Long, Double, Timestamp, String, Text and OID.

The **ID** attributes are going to be numeric (Long) unique values. The **TITLE** and **NAME** attributes are going to be String values and both are going to be Indexed. For the **YEAR** attribute we are going to use an Integer type and this is also going to be Indexed.

[Java]

```
// Add a node type for the movies, with a unique identifier and two indexed attributes
int movieType = g.newNodeType("MOVIE");
int movieIdType = g.newAttribute(movieType, "ID", DataType.Long, AttributeKind.Unique);
int movieTitleType = g.newAttribute(movieType, "TITLE", DataType.String, AttributeKind.Indexed);
int movieYearType = g.newAttribute(movieType, "YEAR", DataType.Integer, AttributeKind.Indexed);

// Add a node type for the people, with a unique identifier and an indexed attribute
int peopleType = g.newNodeType("PEOPLE");
int peopleIdType = g.newAttribute(peopleType, "ID", DataType.Long, AttributeKind.Unique);
int peopleNameType = g.newAttribute(peopleType, "NAME", DataType.String, AttributeKind.Indexed);
```

[C#]

```
// Add a node type for the movies, with a unique identifier and two indexed attributes
int movieType = g.NewNodeType("MOVIE");
int movieIdType = g.NewAttribute(movieType, "ID", DataType.Long, AttributeKind.Unique);
int movieTitleType = g.NewAttribute(movieType, "TITLE", DataType.String, AttributeKind.Indexed);
int movieYearType = g.NewAttribute(movieType, "YEAR", DataType.Integer, AttributeKind.Indexed);

// Add a node type for the people, with a unique identifier and an indexed attribute
int peopleType = g.NewNodeType("PEOPLE");
int peopleIdType = g.NewAttribute(peopleType, "ID", DataType.Long, AttributeKind.Unique);
int peopleNameType = g.NewAttribute(peopleType, "NAME", DataType.String, AttributeKind.Indexed);
```

[C++]

```
// Add a node type for the movies, with a unique identifier and two indexed attributes
type_t movieType = g->NewNodeType(L"MOVIE");
attr_t movieIdType = g->NewAttribute(movieType, L"ID", Long, Unique);
attr_t movieTitleType = g->NewAttribute(movieType, L"TITLE", String, Indexed);
attr_t movieYearType = g->NewAttribute(movieType, L"YEAR", Integer, Indexed);

// Add a node type for the people, with a unique identifier and an indexed attribute
type_t peopleType = g->NewNodeType(L"PEOPLE");
attr_t peopleIdType = g->NewAttribute(peopleType, L"ID", Long, Unique);
attr_t peopleNameType = g->NewAttribute(peopleType, L"NAME", String, Indexed);
```

Create edge types

Now that we have created our nodes let's add the relationships we have previously explained in our schema.

We have defined edge types **CAST** and **DIRECTS**. We stated during the schema explanation that **CAST** will be undirected while **DIRECTS** will be directed. Let's explain a little bit more about this classification of our edge types.

Directed edges have a node which is the *tail* (the source of the edge) and a node which is the *head* (the destination of the edge). In case of **undirected** edges, each node at the extreme of the edge plays two roles at the same time, head and tail. Whereas undirected edges allow navigation in any side, directed edges restrict the navigation to the direction of the edge.

Also, edges can be classified as restricted or unrestricted. **Restricted** edges define which must be the type of the *tail* and *head* nodes, thus edges will only be allowed between those specified type of nodes. In case of **unrestricted** edges, there is no restriction, and edges are allowed between nodes belonging to any type. It is important to note that restricted edges must be directed edges.

In addition, in our schema we have decided that CAST will have an attribute called **CHARACTER**. The **CHARACTER** attribute is going to be a String and we are going to set it as Basic. See the previous section for more information about the attribute types.

[Java]

```
// Add an undirected edge type with an attribute for the cast of a movie
int castType = g.newEdgeType("CAST", false, false);
int castCharacterType = g.newAttribute(castType, "CHARACTER", DataType.String, AttributeKind.Basic);

// Add a directed edge type restricted to go from people to movie for the director of a movie
int directsType = g.newRestrictedEdgeType("DIRECTS", peopleType, movieType, false);
```

[C#]

```
// Add an undirected edge type with an attribute for the cast of a movie
int castType = g.NewEdgeType("CAST", false, false);
int castCharacterType = g.NewAttribute(castType, "CHARACTER", DataType.String, AttributeKind.Basic);

// Add a directed edge type restricted to go from people to movie for the director of a movie
int directsType = g.NewRestrictedEdgeType("DIRECTS", peopleType, movieType, false);
```

[C++]

```
// Add an undirected edge type with an attribute for the cast of a movie
type_t castType = g->NewEdgeType(L"CAST", false, false);
attr_t castCharacterType = g->NewAttribute(castType, L"CHARACTER", String, Basic);

// Add a directed edge type restricted to go from people to movie for the director of a movie
type_t directsType = g->NewRestrictedEdgeType(L"DIRECTS", peopleType, movieType, false);
```

Adding data

Once the schema has been created the next step is to add data. It is interesting to note that the schema can be modified later with no great impact to the database, which is an important characteristic of the DEX graph database.

In the HelloDEX example we are adding enough information to be able to perform some simple queries afterwards.

Although is out of the scope of this guide, it is worth noting that you can use DEX loaders if you are dealing with large data sets.

Add nodes

We are going to add information about these movies:

- Lost in Translation
- Vicky Cristina Barcelona
- Manhattan

And some of the people that worked in these movies:

- Scarlett Johansson
- Bill Murray
- Sofia Coppola
- Woody Allen
- Penélope Cruz
- Diane Keaton

In a previous section we have seen that all this information is stored as nodes with attributes. So, we have to create 3 **MOVIE** nodes for the films, which will each have the attributes **ID**, **Title** and **Year**. Also we have to create 6 **PEOPLE** nodes for the cast and crew, where each will have the attributes **ID** and **Name**. Notice that we use the class Value to set the attributes values, and the same Value object is reused for all the attributes.

Figure 4: Adding nodes

[Java]

```
// Add some MOVIE nodes
Value value = new Value();

long mLostInTranslation = g.newNode(movieType);
g.setAttribute(mLostInTranslation, movieIdType, value.setLong(1));
g.setAttribute(mLostInTranslation, movieTitleType, value.setString("Lost in Translation"));
g.setAttribute(mLostInTranslation, movieYearType, value.setInteger(2003));
```

```
long mVickyCB = g.newNode(movieType);
g.setAttribute(mVickyCB, movieIdType, value.setLong(2));
g.setAttribute(mVickyCB, movieTitleType, value.setString("Vicky Cristina Barcelona"));
g.setAttribute(mVickyCB, movieYearType, value.setInteger(2008));

long mManhattan = g.newNode(movieType);
g.setAttribute(mManhattan, movieIdType, value.setLong(3));
g.setAttribute(mManhattan, movieTitleType, value.setString("Manhattan"));
g.setAttribute(mManhattan, movieYearType, value.setInteger(1979));


// Add some PEOPLE nodes
long pScarlett = g.newNode(peopleType);
g.setAttribute(pScarlett, peopleIdType, value.setLong(1));
g.setAttribute(pScarlett, peopleNameType, value.setString("Scarlett Johansson"));

long pBill = g.newNode(peopleType);
g.setAttribute(pBill, peopleIdType, value.setLong(2));
g.setAttribute(pBill, peopleNameType, value.setString("Bill Murray"));

long pSofia = g.newNode(peopleType);
g.setAttribute(pSofia, peopleIdType, value.setLong(3));
g.setAttribute(pSofia, peopleNameType, value.setString("Sofia Coppola"));

long pWoody = g.newNode(peopleType);
g.setAttribute(pWoody, peopleIdType, value.setLong(4));
g.setAttribute(pWoody, peopleNameType, value.setString("Woody Allen"));

long pPenelope = g.newNode(peopleType);
g.setAttribute(pPenelope, peopleIdType, value.setLong(5));
g.setAttribute(pPenelope, peopleNameType, value.setString("Penélope Cruz"));

long pDiane = g.newNode(peopleType);
g.setAttribute(pDiane, peopleIdType, value.setLong(6));
g.setAttribute(pDiane, peopleNameType, value.setString("Diane Keaton"));
```

[C#]

```
// Add some MOVIE nodes
Value value = new Value();

long mLostInTranslation = g.NewNode(movieType);
g.SetAttribute(mLostInTranslation, movieIdType, value.SetLong(1));
g.SetAttribute(mLostInTranslation, movieTitleType, value.SetString("Lost in Translation"));
g.SetAttribute(mLostInTranslation, movieYearType, value.SetInteger(2003));

long mVickyCB = g.NewNode(movieType);
g.SetAttribute(mVickyCB, movieIdType, value.SetLong(2));
g.SetAttribute(mVickyCB, movieTitleType, value.SetString("Vicky Cristina Barcelona"));
g.SetAttribute(mVickyCB, movieYearType, value.SetInteger(2008));

long mManhattan = g.NewNode(movieType);
g.SetAttribute(mManhattan, movieIdType, value.SetLong(3));
g.SetAttribute(mManhattan, movieTitleType, value.SetString("Manhattan"));
g.SetAttribute(mManhattan, movieYearType, value.SetInteger(1979));


// Add some PEOPLE nodes
long pScarlett = g.NewNode(peopleType);
g.SetAttribute(pScarlett, peopleIdType, value.SetLong(1));
g.SetAttribute(pScarlett, peopleNameType, value.SetString("Scarlett Johansson"));

long pBill = g.NewNode(peopleType);
g.SetAttribute(pBill, peopleIdType, value.SetLong(2));
g.SetAttribute(pBill, peopleNameType, value.SetString("Bill Murray"));

long pSofia = g.NewNode(peopleType);
g.SetAttribute(pSofia, peopleIdType, value.SetLong(3));
g.SetAttribute(pSofia, peopleNameType, value.SetString("Sofia Coppola"));

long pWoody = g.NewNode(peopleType);
g.SetAttribute(pWoody, peopleIdType, value.SetLong(4));
g.SetAttribute(pWoody, peopleNameType, value.SetString("Woody Allen"));

long pPenelope = g.NewNode(peopleType);
g.SetAttribute(pPenelope, peopleIdType, value.SetLong(5));
g.SetAttribute(pPenelope, peopleNameType, value.SetString("Penélope Cruz"));

long pDiane = g.NewNode(peopleType);
g.SetAttribute(pDiane, peopleIdType, value.SetLong(6));
g.SetAttribute(pDiane, peopleNameType, value.SetString("Diane Keaton"));
```

[C++]

```
// Add some MOVIE nodes
Value *value = new Value();

oid_t mLostInTranslation = g->NewNode(movieType);
g->SetAttribute(mLostInTranslation, movieIdType, value->SetLong(1));
g->SetAttribute(mLostInTranslation, movieTitleType, value->SetString(L"Lost in Translation"));
g->SetAttribute(mLostInTranslation, movieYearType, value->SetInteger(2003));

oid_t mVickyCB = g->NewNode(movieType);
g->SetAttribute(mVickyCB, movieIdType, value->SetLong(2));
g->SetAttribute(mVickyCB, movieTitleType, value->SetString(L"Vicky Cristina Barcelona"));
g->SetAttribute(mVickyCB, movieYearType, value->SetInteger(2008));

oid_t mManhattan = g->NewNode(movieType);
g->SetAttribute(mManhattan, movieIdType, value->SetLong(3));
g->SetAttribute(mManhattan, movieTitleType, value->SetString(L"Manhattan"));
g->SetAttribute(mManhattan, movieYearType, value->SetInteger(1979));


// Add some PEOPLE nodes
oid_t pScarlett = g->NewNode(peopleType);
g->SetAttribute(pScarlett, peopleIdType, value->SetLong(1));
g->SetAttribute(pScarlett, peopleNameType, value->SetString(L"Scarlett Johansson"));

oid_t pBill = g->NewNode(peopleType);
g->SetAttribute(pBill, peopleIdType, value->SetLong(2));
g->SetAttribute(pBill, peopleNameType, value->SetString(L"Bill Murray"));

oid_t pSofia = g->NewNode(peopleType);
g->SetAttribute(pSofia, peopleIdType, value->SetLong(3));
g->SetAttribute(pSofia, peopleNameType, value->SetString(L"Sofia Coppola"));

oid_t pWoody = g->NewNode(peopleType);
g->SetAttribute(pWoody, peopleIdType, value->SetLong(4));
g->SetAttribute(pWoody, peopleNameType, value->SetString(L"Woody Allen"));

oid_t pPenelope = g->NewNode(peopleType);
g->SetAttribute(pPenelope, peopleIdType, value->SetLong(5));
g->SetAttribute(pPenelope, peopleNameType, value->SetString(L"Penélope Cruz"));

oid_t pDiane = g->NewNode(peopleType);
g->SetAttribute(pDiane, peopleIdType, value->SetLong(6));
g->SetAttribute(pDiane, peopleNameType, value->SetString(L"Diane Keaton"));
```

Add edges

Now that we have all the nodes in the database we can start adding the relationships between them. As previously explained in the Set the schema section we are going to build two types of relationships depending on whether the person attached to a movie is part of the cast (edge **CAST**) or its director (edge **DIRECTS**).

We are going to add an edge of type **CAST** between each node of type **PEOPLE** and each node of type **MOVIE** when the person worked as an actor in that movie. Then, we will set the edge attribute **CHARACTER** as the name of the character played by that actor in the movie.

After this, we will create an edge of type **DIRECTS** between a node of type **PEOPLE** and a node of type **MOVIE** for the director of each movie.

[Java]

```
// Add some CAST edges
// Remember that we are reusing the Value class instance to set the attributes
long anEdge;
anEdge = g.newEdge(castType, mLostInTranslation, pScarlett);
g.setAttribute(anEdge, castCharacterType, value.setString("Charlotte"));

anEdge = g.newEdge(castType, mLostInTranslation, pBill);
g.setAttribute(anEdge, castCharacterType, value.setString("Bob Harris"));

anEdge = g.newEdge(castType, mVickyCB, pScarlett);
```

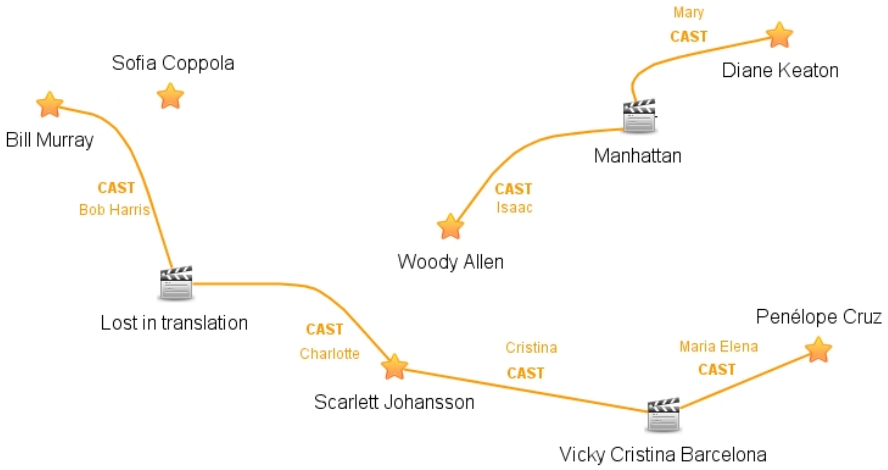


Figure 5: Adding CAST edges (Notice that we have omitted the attributes of the nodes)

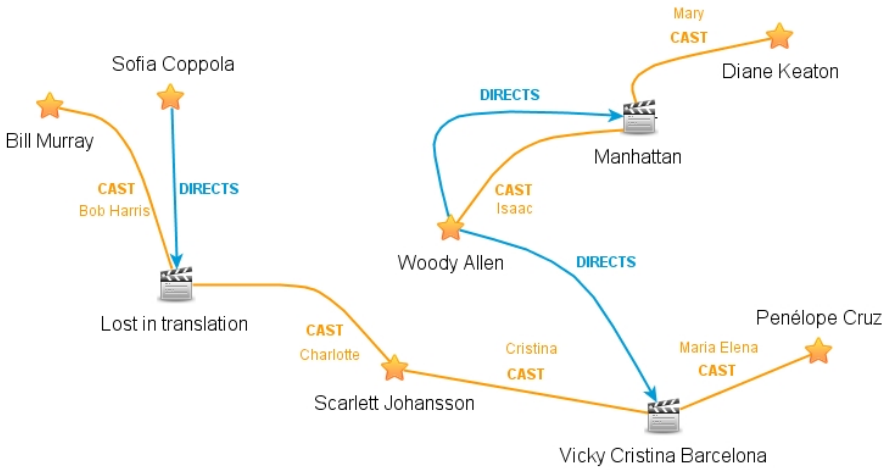


Figure 6: Adding DIRECTS edges (Notice that we have omitted the attributes of the nodes)

```
g.setAttribute(anEdge, castCharacterType, value.setString("Cristina"));

anEdge = g.newEdge(castType, mVickyCB, pPenelope);
g.setAttribute(anEdge, castCharacterType, value.setString("Maria Elena"));

anEdge = g.newEdge(castType, mManhattan, pDiane);
g.setAttribute(anEdge, castCharacterType, value.setString("Mary"));

anEdge = g.newEdge(castType, mManhattan, pWoody);
g.setAttribute(anEdge, castCharacterType, value.setString("Isaac"));



// Add some DIRECTS edges
anEdge = g.newEdge(directsType, pSofia, mLostInTranslation);

anEdge = g.newEdge(directsType, pWoody, mVickyCB);

anEdge = g.newEdge(directsType, pWoody, mManhattan);
```

[C#]

```
// Add some CAST edges
// Remember that we are reusing the Value class instance to set the attributes
long anEdge;
anEdge = g.NewEdge(castType, mLostInTranslation, pScarlett);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Charlotte"));

anEdge = g.NewEdge(castType, mLostInTranslation, pBill);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Bob Harris"));

anEdge = g.NewEdge(castType, mVickyCB, pScarlett);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Cristina"));

anEdge = g.NewEdge(castType, mVickyCB, pPenelope);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Maria Elena"));

anEdge = g.NewEdge(castType, mManhattan, pDiane);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Mary"));

anEdge = g.NewEdge(castType, mManhattan, pWoody);
g.SetAttribute(anEdge, castCharacterType, value.SetString("Isaac"));



// Add some DIRECTS edges
anEdge = g.NewEdge(directsType, pSofia, mLostInTranslation);

anEdge = g.NewEdge(directsType, pWoody, mVickyCB);

anEdge = g.NewEdge(directsType, pWoody, mManhattan);
```

[C++]

```
// Add some CAST edges
// Remember that we are reusing the Value class instance to set the attributes
oid_t anEdge;
anEdge = g->NewEdge(castType, mLostInTranslation, pScarlett);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Charlotte"));

anEdge = g->NewEdge(castType, mLostInTranslation, pBill);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Bob Harris"));

anEdge = g->NewEdge(castType, mVickyCB, pScarlett);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Cristina"));

anEdge = g->NewEdge(castType, mVickyCB, pPenelope);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Maria Elena"));

anEdge = g->NewEdge(castType, mManhattan, pDiane);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Mary"));

anEdge = g->NewEdge(castType, mManhattan, pWoody);
g->SetAttribute(anEdge, castCharacterType, value->SetString(L"Isaac"));



// Add some DIRECTS edges
anEdge = g->NewEdge(directsType, pSofia, mLostInTranslation);
```

```
anEdge = g->NewEdge(directsType, pWoody, mVickyCB);

anEdge = g->NewEdge(directsType, pWoody, mManhattan);
```

First queries

If you have successfully completed all the previous steps in this chapter you have now created your first graph, congratulations! The graph should look exactly like Figure 7.

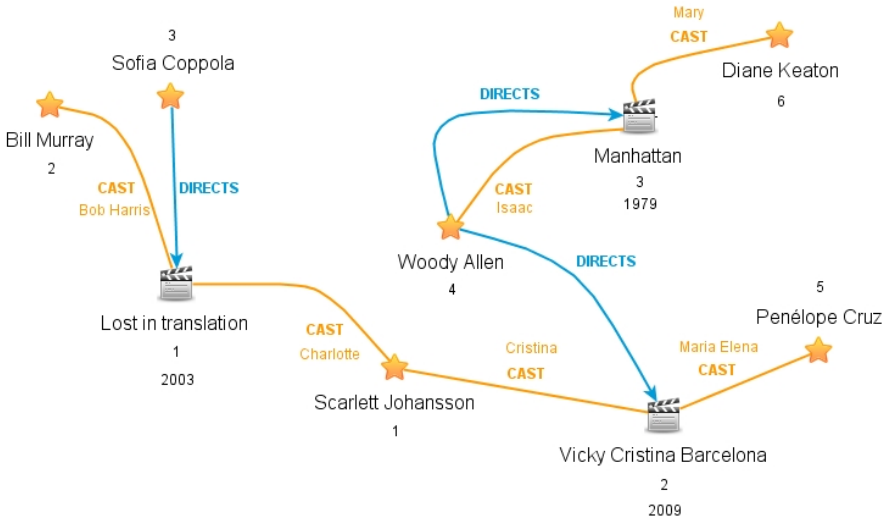


Figure 7: HelloDEX complete graph

Although you have accomplished the main objective of this guide we encourage you to follow the final steps: querying the graph and the necessary procedure of closing your graph database.

Let's propose a simple example that illustrates how to query a DEX graph database. For instance you may be interested in finding out who acted in movies directed by Woody Allen and also acted in movies directed by Sofia Coppola. We are able to establish the connection between these two excellent directors.

We could start by searching for the "Woody Allen" node using the FindObject method. However, as we have just created the graph we know that we already have that information in a variable called pWoody because we kept it when adding the node.

As we already have the "Woody Allen" node, our first query is to obtain the collection of movies directed by him. For each of his movies, there is an edge of type **DIRECTS** that starts from his node to a node of type **MOVIE**. To retrieve this information we will use the Neighbors method against the "Woody Allen" node through the edge **DIRECTS**. As we are only interested in the head (hence the movie) of this directed edge we truncate this retrieval to "Outgoing" only.

[Java]

```
// Get the movies directed by Woody Allen
Objects directedByWoody = g.neighbors(pWoody, directsType, EdgesDirection.Outgoing);
```

[C#]

```
// Get the movies directed by Woody Allen
Objects directedByWoody = g.Neighbors(pWoody, directsType, EdgesDirection.Outgoing);
```

[C++]

```
// Get the movies directed by Woody Allen
Objects *directedByWoody = g->Neighbors(pWoody, directsType, Outgoing);
```

The result of the query is an Objects class instance. It stores a collection of Dex object identifiers as a set; in this way we do not obtain duplicated elements.

Figure 8: Movies directed by Woody Allen

Now that we have found all the movies directed by Woody Allen, we can use them to find the people that acted in them. To do so, we can use the Neighbors operation again, but from the collection of movies of Woody Allen and using the **CAST** edge type. In this case, we will use Any edge direction because it is not a directed edge.

All the Objects instances should be closed when no longer needed, therefore we can close the directedbyWoody collection just after retrieving its cast.

[Java]

```
// Get the cast of the movies directed by Woody Allen
Objects castDirectedByWoody = g.neighbors(directedByWoody, castType, EdgesDirection.Any);

// We don't need the directedByWoody collection anymore, so we should close it
directedByWoody.close();
```

[C#]

```
// Get the cast of the movies directed by Woody Allen
Objects castDirectedByWoody = g.Neighbors(directedByWoody, castType, EdgesDirection.Any);

// We don't need the directedByWoody collection anymore, so we should close it
directedByWoody.Close();
```

[C++]

```
// Get the cast of the movies directed by Woody Allen
Objects *castDirectedByWoody = g->Neighbors(directedByWoody, castType, Any);

// We don't need the directedByWoody collection anymore, so we should delete it
delete directedByWoody;
```

Figure 9: Cast in movies by Woody Allen

We now have all the people that acted in movies directed by Woody Allen (Figure 9) but we only wanted to know who also acted in movies directed by Sofia Coppola.

To match the cast in movies directed by Woody Allen with the cast in movies directed by Sofia Coppola we have to repeat the same queries previously performed for Woody Allen: the first is the retrieval of movies of Sofia Coppola followed by the retrieval of the cast of each of her movies. The result is shown in Figure 10.

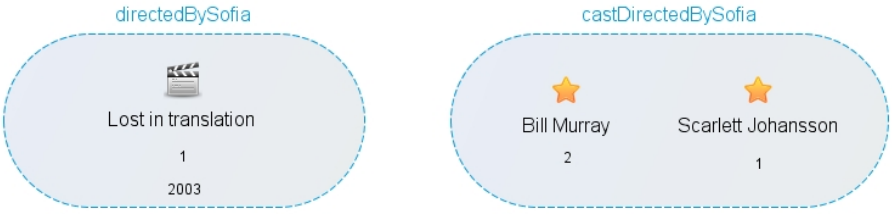


Figure 10: Movies and cast for Sofia Coppola

[Java]

```
// Get the movies directed by Sofia Coppola
Objects directedBySofia = g.neighbors(pSofia, directsType, EdgesDirection.Outgoing);

// Get the cast of the movies directed by Sofia Coppola
Objects castDirectedBySofia = g.neighbors(directedBySofia, castType, EdgesDirection.Any);

// We don't need the directedBySofia collection anymore, so we should close it
directedBySofia.close();
```

[C#]

```
// Get the movies directed by Sofia Coppola
Objects directedBySofia = g.Neighbors(pSofia, directsType, EdgesDirection.Outgoing);

// Get the cast of the movies directed by Sofia Coppola
Objects castDirectedBySofia = g.Neighbors(directedBySofia, castType, EdgesDirection.Any);

// We don't need the directedBySofia collection anymore, so we should close it
directedBySofia.Close();
```

[C++]

```
// Get the movies directed by Sofia Coppola
Objects *directedBySofia = g->Neighbors(pSofia, directsType, Outgoing);

// Get the cast of the movies directed by Sofia Coppola
Objects *castDirectedBySofia = g->Neighbors(directedBySofia, castType, Any);

// We don't need the directedBySofia collection anymore, so we should delete it
delete directedBySofia;
```

In the collections called castDirectedByWoody and castDirectedBySofia we now have all the cast in movies directed by each director respectively. But the objective of the query was to find out who acted in movies directed by both of them. To achieve this we only need to calculate the intersection of these two sets of **PEOPLE** nodes.

[Java]

```
// We want to know the people that acted in movies directed by Woody AND in movies directed by Sofia.
Objects castFromBoth = Objects.combineIntersection(castDirectedByWoody, castDirectedBySofia);

// We don't need the other collections anymore
castDirectedByWoody.close();
castDirectedBySofia.close();
```

[C#]

```
// We want to know the people that acted in movies directed by Woody AND in movies directed by Sofia.
Objects castFromBoth = Objects.CombineIntersection(castDirectedByWoody, castDirectedBySofia);

// We don't need the other collections anymore
castDirectedByWoody.Close();
castDirectedBySofia.Close();
```

[C++]

```
// We want to know the people that acted in movies directed by Woody AND in movies directed by Sofia.
Objects *castFromBoth = Objects::CombineIntersection(castDirectedByWoody, castDirectedBySofia);

// We don't need the other collections anymore
delete castDirectedByWoody;
delete castDirectedBySofia;
```

Remember that you should close the Objects when you are not going to use them anymore.

We think we may have the result that we were looking for. But how do we check the information from that Objects collection? You must use ObjectsIterator that will traverse all the elements inside the set. With this iterator we can get all the node identifiers in the result (**PEOPLE** node identifiers), one by one. Then, for each one, we can get their attributes. We are only interested in the name of the actor.

Here you have it! Scarlett Johansson is the link between both directors.

[Java]

```
// Say hello to the people found
ObjectsIterator it = castFromBoth.iterator();
while (it.hasNext())
{
    long peopleOid = it.next();
    g.getAttribute(peopleOid, peopleNameType, value);
    System.out.println("Hello " + value.getString());
}
// The ObjectsIterator must be closed
it.close();

// The Objects must be closed
castFromBoth.close();
```

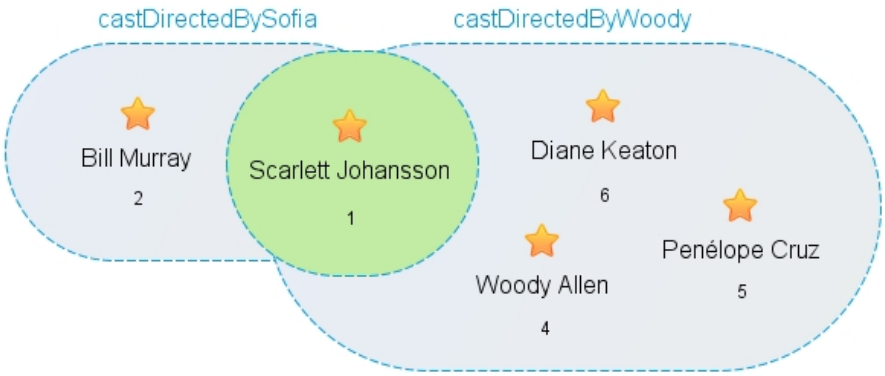


Figure 11: Link between Woody Allen and Sofia Coppola

[C#]

```
// Say hello to the people found
ObjectsIterator it = castFromBoth.Iterator();
while (it.HasNext())
{
    long peopleOid = it.Next();
    g.GetAttribute(peopleOid, peopleNameType, value);
    System.Console.WriteLine("Hello " + value.GetString());
}
// The ObjectsIterator must be closed
it.Close();

// The Objects must be closed
castFromBoth.Close();
```

[C++]

```
// Say hello to the people found
ObjectsIterator *it = castFromBoth->Iterator();
while (it->HasNext())
{
    oid_t peopleOid = it->Next();
    g->GetAttribute(peopleOid, peopleNameType, *value);
    std::wcout << L"Hello " << value->GetString() << std::endl;
}
// The ObjectsIterator must be deleted
delete it;

// The Objects must be deleted
delete castFromBoth;
```

Again we have reused the Value class instance to get the value of the **NAME** attribute.

Note that the ObjectsIterator should also be closed before closing the Objects collection.

Closing the database

This guide is almost finished. You have now performed all the tasks necessary to create a graph with its schema, add data and query this data afterwards. There is only one inevitable and very important final step: the proper closing of the database.

You must take into account the fact that all the collections and iterators should be closed first. You can close them now, or an even better programming practice is to close them as soon as they are no longer needed.

To close DEX, once collections and iterators have been closed, you must first close the Session (or sessions) which will free all the temporary information stored in it, then close the Database to proceed to the closure of the Dex instance.

[Java]

```
sess.close();
db.close();
dex.close();
```

[C#]

```
sess.Close();
db.Close();
dex.Close();
```

[C++]

```
delete sess;
delete db;
delete dex;
```

Compile and run

To compile and run your DEX application you must take into account your development framework.

Java

In the Installation chapter we have seen how to download and unpack the java package to get the **dexjava.jar** file. You need to include that jar in you development environment project.

If you don't use an IDE, you just need to add the **dexjava.jar** file to the classpath. So, you can compile and run the HelloDEX application with these simple commands:

```
$ javac -cp dexjava.jar HelloDex.java
$ java -cp dexjava.jar;. HelloDex
```

With Maven

If you use Apache Maven, then it is even easier. DEX is in the maven central repository, adding the dependency to the correct DEX version into your "pom.xml" file should be enough:

```
<dependency>
  <groupId>com.sparsity</groupId>
  <artifactId>dexjava</artifactId>
  <version>4.6.0</version>
</dependency>
```

.NET

.NET developers have the following two different options to build a .NET application.

MS Visual Studio users

If you are a MS Visual Studio IDE developer, you need to add the reference to the DEX .NET library (**dexnet.dll**) in your project and set build platform to the appropiate specific platfrom (x64 or x86). Please, check that in the "Configuration Manager" from the "BUILD" menu of Visual Studio, the "Platform" selected for the build is NOT "Any CPU". If it's "Any CPU", you must select "New" to set a "x86" or "x64" build target.

Since all the other libraries included in the package are native libraries that will be loaded at runtime, they must be available too. And the MS Visual C runtime must be installed on all the computers running your application.

The best option is to copy all the other ".dll" files into the same directory where your application executable file will be.

Using the development environment, this can be done using the option Add existing Item, choosing to see Executable files and selecting the other three native libraries (dex.dll, dexnetwarp.dll and stlport.dll).

Next you must select the three libraries in the Solution Explorer window and set the property Copy to Output as Copy Always for all three native libraries.

Instead of copying the native libraries to the application target directory, an alternative would be to put all the native ".dll" files into your Windows system folder (System32 or SysWOW64 depending on your Windows version).

Now you are ready to build and run the application like any Visual Studio project.

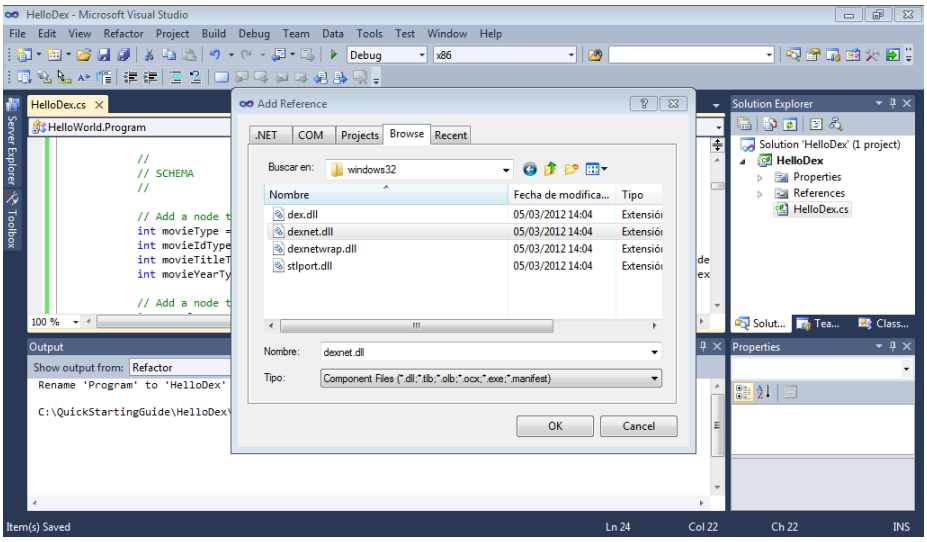


Figure 12: .Net compilation - adding reference

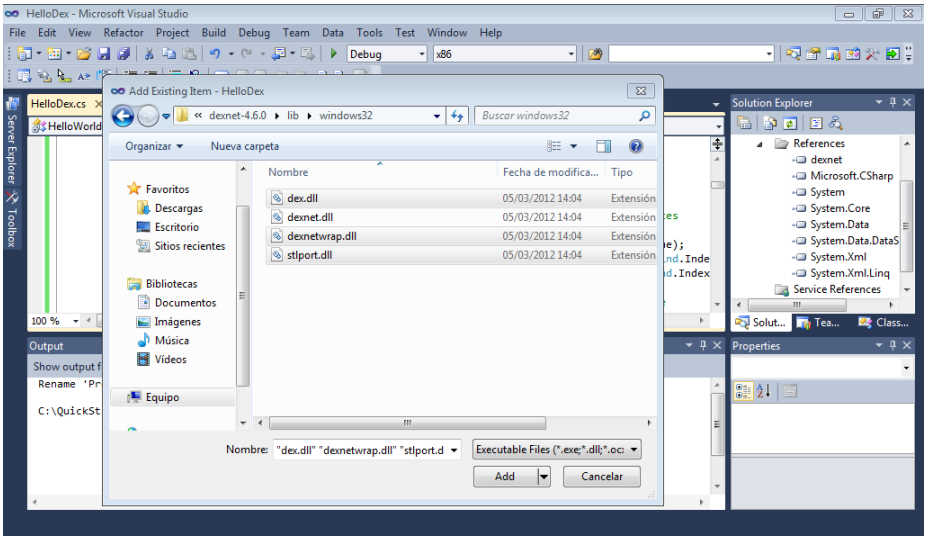


Figure 13: .Net compilation - adding existing item

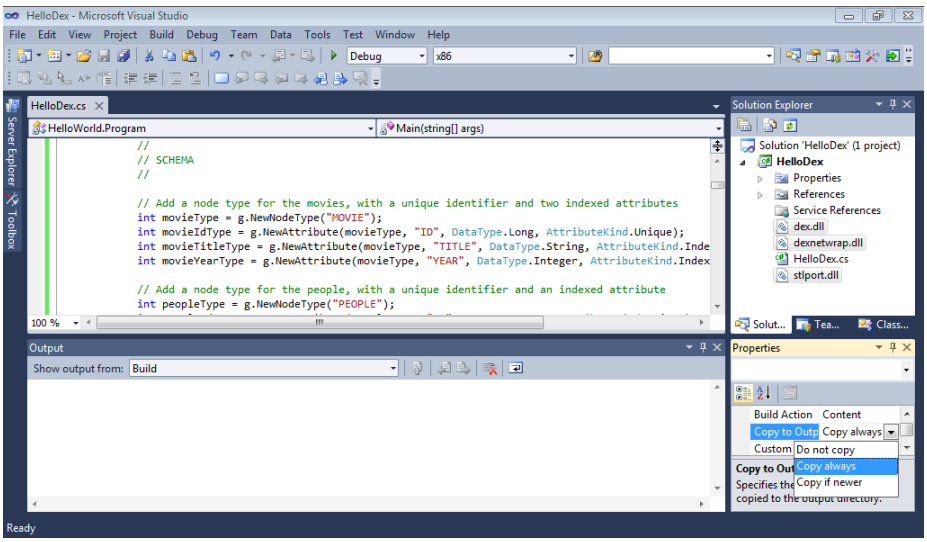


Figure 14: .Net compilation - copy to output

Command-line users

If you just want to quickly test the HelloDEX sample application, you can use the command line. First setup your compiler environment with the vsvars32.bat file if you are using a 32 bit MS Visual Studio.

```
> call "C:\Program Files\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat"
```

or with the vcvarsall.bat file if you are using a 64 bit MS Visual Studio.

```
> call "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" amd64
```

Then compile and run the application (assuming all the libraries have already been copied to the same directory):

```
> csc /out:HelloDex.exe /r:dexnet.dll HelloDex.cs
> HelloDex.exe
```

C++

The DEX C++ interface contains include files and dynamic libraries in order to compile and run an application using DEX. The general procedure is to first add the include directories to your project, then link with the supplied libraries corresponding to your operating system and finally copy them to any place where they can be loaded at runtime (common places are the same folder as the target executable file or your system libraries folder).

Let's have a look at a more detailed description of this procedure in the most common environments.

Remember that the package should already be unpacked in a known directory (see chapter 2).

Windows

If your development environment is Microsoft Visual Studio, your first step should be to add to the Additional include directories, C++ general property of your project and **dex** subdirectory of the includes-folder from the DEX package.

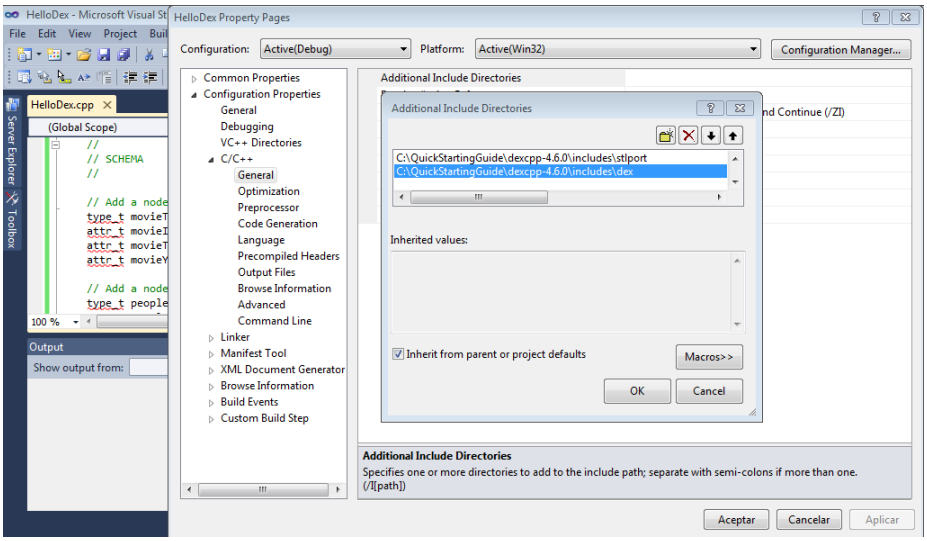


Figure 15: C++ compilation - include directories

This must also be done with the library directory, so the Additonal library directories linker general property must be edited to add the correct subdirectory of the DEX **lib** folder for your operating system.

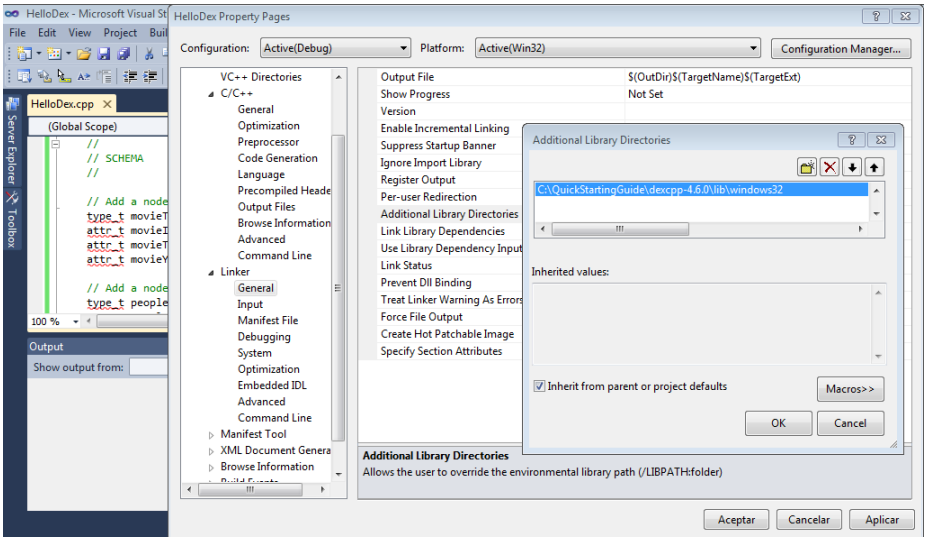


Figure 16: C++ compilation - add library directories

After this, you should add the **dex** ".lib" library to the Additional Dependencies linker input property.

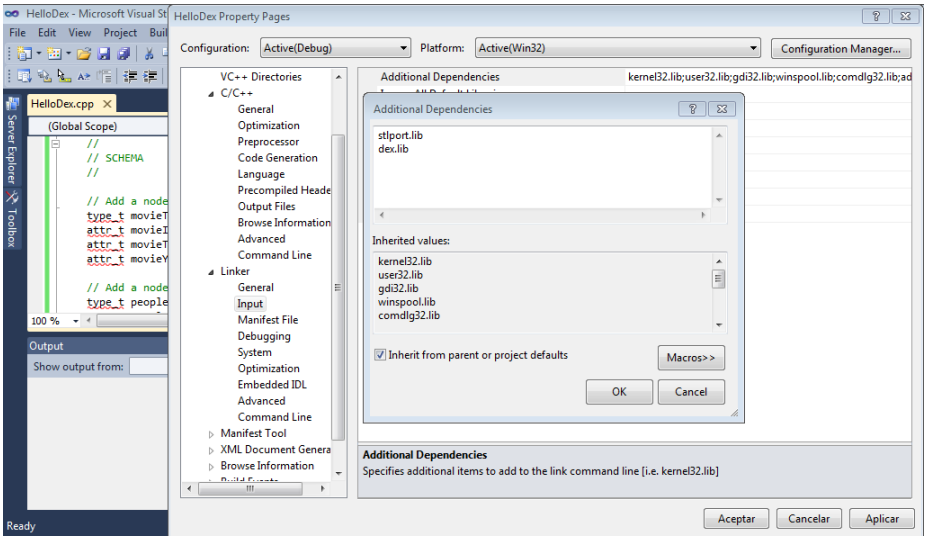


Figure 17: C++ compilation - add dependencies

Finally make sure that the dll files can be found at run time. An easy way to do this is to add a post-process in your project to copy the dll files to the same output folder where your application executable will be built.

An alternative would be to simply put all the native ".dll" files into your Windows system folder (System32 or SysWOW64 depending on your Windows version).

Now you are ready to build and run the application like any Visual Studio project.

Finally, if you just want to quickly test the HelloDEX sample application, you can use the command line. First setup your compiler environment with the vsvars32.bat file if you are using a 32-bit MS Visual Studio.

```
> call "C:\Program Files\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat"
```

or with the vcvarsall.bat file if you are using a 64-bit MS Visual Studio.

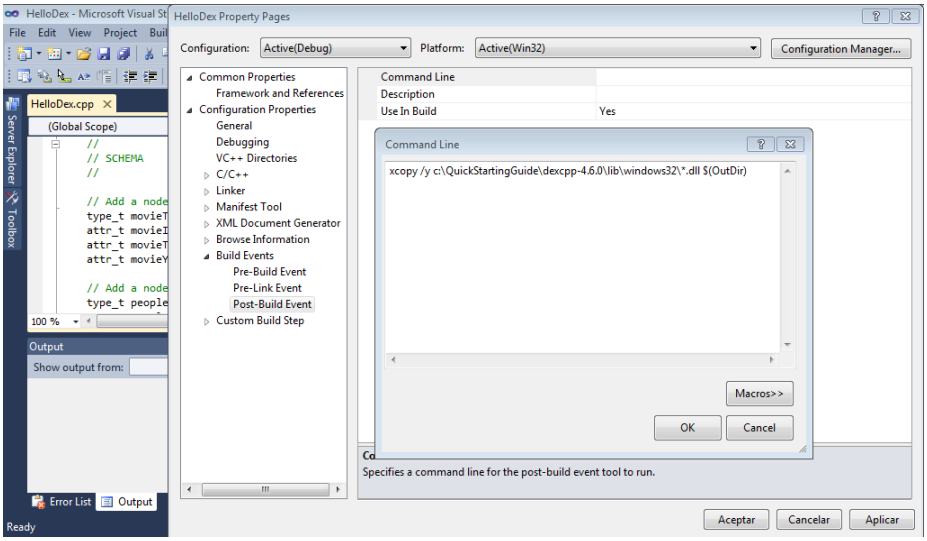


Figure 18: C++ compilation - post build event

```
> call "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" amd64
```

Then compile and run the application (example on a 32-bit Windows):

```
> cl /I"path_to_the_unpacked_dex\includes\dex" /D "WIN32" /D "_UNICODE" /D "UNICODE" /EHsc /MD /c
    HelloDex.cpp

> link /OUT:"HelloDex.exe" HelloDex.obj /LIBPATH:"path_to_the_unpacked_dex\lib\windows32" "dex.lib"

> xcopy /y "path_to_the_unpacked_dex\lib\windows32\*.dll" .

> HelloDex.exe
```

Linux/MacOS

In this guide we are not going to focus on any specific integrated development environment for Linux or Mac OS because it is beyond the scope of the guide. Instead we will give an explanation of the procedure which you can adapt to the specifics of your development environment.

- In the **includes** directory, there is the subdirectory **dex** that must be added as include search directory in your project.
- The **lib** directory contains a subdirectory for each operating system available. You should add the correct directory for your computer as a link search directory in your project.
- To link your application, the **dex** and your **pthread** libraries must be used in this order.
- Finally you may need to add the directory where the libraries can be found to the **LD_LIBRARY_PATH** environment variable to be sure that they will be found at runtime.

Finally, if you just want to quickly test the HelloDEX sample application, you can use the command line.

```
$ g++ -I/path_to_the_unpacked_dex/includes/dex -o HelloDex.o -c HelloDex.cpp

$ g++ HelloDex.o -o HelloDex -L../lib/linux64 -ldex -lpthread

$ export LD_LIBRARY_PATH=/path_to_the_unpacked_dex/lib/linux64/

$ ./HelloDex
```

Download examples

Here you can download the *HelloDEX* sources including all the examples which have been explained in this starting guide.

If you have followed all the steps up to this point you should have created a graph database which looks exactly like **Figure 7**.

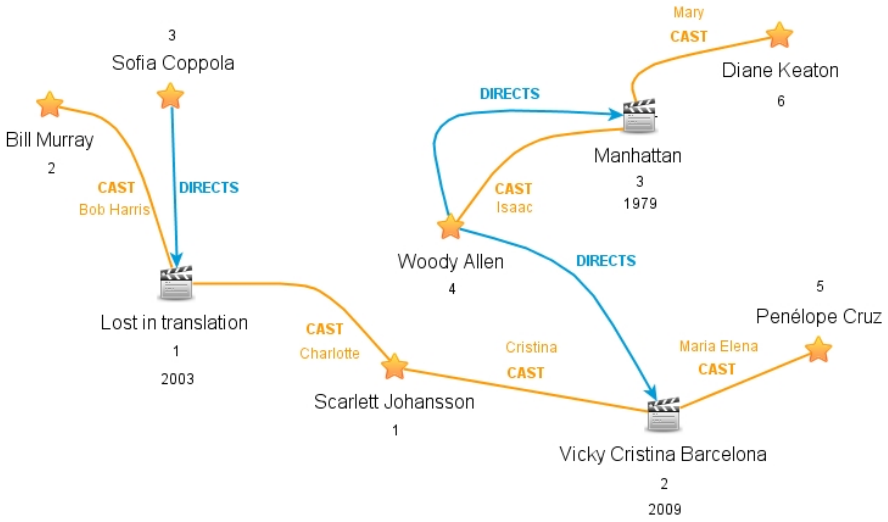


Figure 7: HelloDEX complete graph

You can also directly download *HelloDEX* sources which, once you run them, will construct the same graph.

HelloDEX first creates a DEX graph database (see Chapter 3-section 2), then creates the schema (see Chapter3-section 4), adds data creating nodes and edges and their attributes (see Chapter3-section 5) and finally queries this data (see Chapter3-section 6).

Queries included in the example retrieve neighbors from some nodes. For instance *all the movies directed by Woody Allen*, which will be the neighbors of Woody Allen through the DIRECTS edge. Or other more complex examples include retrieving *all the people who acted both in movies directed by Woody Allen and in movies directed by Sofia Coppola*.

Choose your *HelloDEX* download language:

- Java: Hello DEX in Java
- .Net: Hello DEX for .Net
- C++: Hello DEX in C++

Support

This is the final section of the DEX starting guide. We have guided you through the entire process of creating your first graph with DEX. Moreover, you have added data to your graph and finally queried it.

We encourage you to learn more about the advanced features of DEX, practice with the rest of available queries & functionalities, and build your own application.

While developing do not forget to consult the DEX **reference manuals**. Reference manuals are included in the `doc` directory of the DEX package. You can also directly consult the information in the documentation section of our website choosing your preferred programming language.

Also in the documentation section of the Sparsity Technologies website you can find **tutorials** that will give you further details about DEX. If you are a Java programmer, we highly recommend taht you to take a look at this seminar tutorial, which is another take on the most commonly used DEX functionalities.

One of our main support channels is the DEX Technical Area **forum** where code examples are displayed and questions resolved. Here you will find DEX advanced programmers who will answer to your questions. Do not hesitate to share any doubts you may have, however small.

Finally do not forget to **follow us** on twitter, facebook and linkedin. You can share your doubts and thoughts there too!.