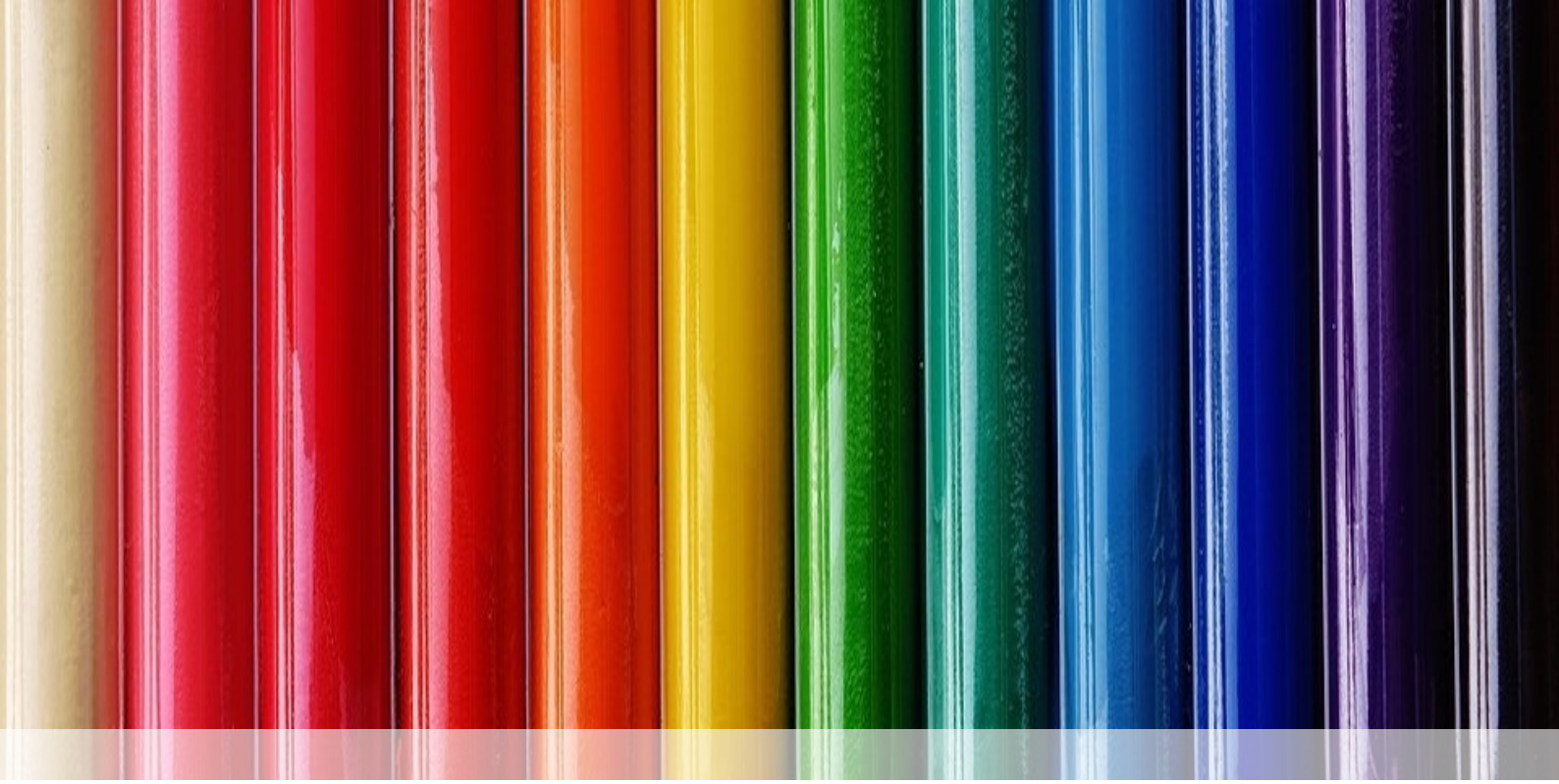


Manual

dataClay

Active objects across the network

version 2.7

Barcelona Supercomputing Center

support-dataclay@bsc.es

Release notes

2022, June: Release version 2.7

New features

- Support for Python 3.10
- Support for placing numpy array structures into Non-Volatile Memory devices
- Graph-based reference management to improve performance, resource consumption, and remove race conditions
- Configurable pickle protocol version for serialization of objects in Python

Improvements

- Updated dependency versions
- Optimizations in caching
- Improvements and optimizations in the export of objects via MQTT
- Other minor optimizations and bug fixes

Contents

I Getting started

1 Main Concepts 11

1.1 What is dataClay 11

1.2 Basic terminology 11

1.3 Execution model 12

1.4 Tasks and roles 12

1.5 Memory Management and Garbage Collection 12

1.6 Federation 13

2 My first dataClay application 15

2.1 HelloPeople: a first dataClay example 15

2.1.1 Java 15

2.1.2 Python 18

3 Application cycle 23

3.1 Account creation 24

3.2 Namespaces and class models 24

3.3 Datasets and data contracts 24

3.4 Using a registered class: getting its stubs 25

3.5 Build and run the application 25

3.6 Easier than it looks 25

II Java: Programmer API

4 Java API 29
4.1 dataClay API 29
4.2 Object store methods 30
4.2.1 Class methods 31
4.2.2 Object methods 32
4.3 Object oriented methods 34
4.3.1 Class methods 34
4.3.2 Object methods 35
4.4 Advanced methods 35
4.5 Error management 38
4.6 Memory Management and Garbage Collection 38
4.7 Replica management 38
4.8 Federation 40
4.8.1 dataClay API methods 43
4.8.2 Object methods 43
4.9 Further considerations 45
4.9.1 Importing registered classes 45
4.9.2 Non-registered classes 45
4.9.3 Third party libraries 45

III Python: Programmer API

5 Python API 49
5.1 dataClay API 49
5.2 Object store methods 50
5.2.1 Class methods 51
5.2.2 Object methods 52
5.3 Object oriented methods 53
5.3.1 Class methods 53
5.3.2 Object methods 54
5.4 Advanced methods 55
5.5 Error management 57
5.6 Memory Management and Garbage Collection 57
5.7 Replica management 58

5.8 Non-volatile Memory support 59
5.9 Federation 60
5.9.1 dataClay API methods . . . 62
5.10 Further considerations 64
5.10.1 Type annotation . . . 65
5.10.2 Non-registered classes . . . 65
5.10.3 Third party libraries . . . 66
5.10.4 Execution environment . . . 66

IV dataClay management utility

6 dataClay command line utility . . . 69
6.1 Accounts 69
6.2 Class models 69
6.3 Data contracts 71
6.4 Backends 72

V Installation

7 Deployment . . . 75
7.1 dataClay architecture 75
7.1.1 Logic Module . . . 76
7.1.2 Data Service . . . 76
7.2 Deployment with containers 76
7.2.1 Single node installation . . . 76
7.2.2 Cluster installation . . . 79
7.2.3 Enabling Python parallelism . . . 81
7.2.4 Tuning dataClay . . . 82
7.2.5 Singularity . . . 83
7.2.6 Memory Management and Garbage Collection . . . 83

8 Configuration . . . 85
8.1 Client libraries 85
8.2 Configuration files 85
8.2.1 Session properties . . . 85
8.2.2 Client properties . . . 86
8.3 Tracing 86
8.4 Federation with secure communications 88

VI Bibliography and index

Bibliography 93

Index 95

Getting started

1 Main Concepts 11
1.1 What is dataClay
1.2 Basic terminology
1.3 Execution model
1.4 Tasks and roles
1.5 Memory Management and Garbage Collection
1.6 Federation

2 My first dataClay application 15
2.1 HelloPeople: a first dataClay example

3 Application cycle 23
3.1 Account creation
3.2 Namespaces and class models
3.3 Datasets and data contracts
3.4 Using a registered class: getting its stubs
3.5 Build and run the application
3.6 Easier than it looks

1. Main Concepts

1.1 What is dataClay

dataClay [**MARTI2017129**, **MartiFraiz2017**] is a distributed object store that enables programmers to handle object persistence using the same model they use in their object-oriented applications, thus avoiding time consuming transformation between persistent and non persistent models. In other words, dataClay enables applications to store objects in the same format they have in memory. This can be done either using the standard *GET/PUT/UPDATE* methods of standard object stores, or by just calling the *makePersistent* method on an object that will enable applications to access it, in the same way, regardless of whether it is loaded in memory or persisted in disk (you just follow the object reference).

In addition, dataClay simplifies and optimizes the idea of moving computation close to data (see Section 1.3) by enabling the execution of methods in the same node where a given object is located. dataClay also optimizes the idea of sharing data and models (set of classes) between different users by means of storing the class (including method definition) together with the object.

1.2 Basic terminology

In this section we present a brief terminology that is used throughout the manual.

Object, as in object oriented programming, refers to a particular instance of a class.
dataClay application is any application that uses dataClay to handle its persistent data.
Backend is a node in the system that is able to handle persistent objects and execution requests. These nodes need to be running the dataClay platform on them. We can have as many as we need either for capacity or parallelism reasons.
Clients are the machines where dataClay applications run. These nodes can be very thin. They only need to be able to run Java or Python code and to have the dataClay lib installed.
dataClay object is any object stored in dataClay.
Objects with alias are objects that have been explicitly named (much in the same way we give names to files). Not all dataClay objects need to have an alias (a name). If an object has

an alias, we can access it by using its name. On the other hand, objects without an alias can only be accessed by a reference from another object.

Dataset is an abstraction where many objects are grouped. It is indented to simplify the task of sharing objects with other users.

Data model or **class model** consists of a set of related classes programmed in one of the supported languages.

Namespace is a dataClay abstraction aimed at grouping a set of classes together. Namespaces have two objectives: i) grouping related classes to ease the task of sharing them with other users and ii) avoiding clashing of class names. A namespace is similar to a Java/Python package.

1.3 Execution model

As we have mentioned, one of the key features of dataClay is to offer a mechanism to bring computation closer to data. For this reason, all methods of a dataClay object will not be executed in the client (application address space) but on the backend where dataClay stored the object. Thus, searching for an object in a collection will not imply sending all objects in the collection to the client, but only the final result because the search method will be executed in the backend. If the collection is distributed among different backends, any sub-method required to check whether objects match certain conditions or not, will be executed on the involved backends.

It is important to notice that this execution model does not prevent developers to use the standard object store model by using *GET/PUT/UPDATE* methods. In particular, a *GET* method (as *CLONE* in dataClay to match with object oriented terminology), will bring the object to the application address space and thus all methods will be executed locally. At this point, any application object either retrieved (*CLONED*) from dataClay or created by the application itself, can be either *PUT* into the system to save it or can be used to *UPDATE* an existing stored object.

1.4 Tasks and roles

In order to rationalize the different roles that take part in data-centric applications, such as the ones supported by dataClay, we assume two different roles.

Model providers design and implement class models to define the elements of data (data structure), their relationships, and methods (API) that applications can use to access and process it.

Application developers use classes developed by the model provider in order to build applications. These applications can either create and store new objects or access data previously created.

Although dataClay encourages these roles in the cycle of applications, they do not have to be declared as such and, of course, they can be assumed by a single person.

1.5 Memory Management and Garbage Collection

Every backend in dataClay maintains a daemon process that checks if memory usage has reached a certain threshold and, if this is the case, it flushes those objects that are not referenced into the underlying storage.

On the other hand, dataClay also performs background garbage collection to remove those objects that are no longer accessible. More specifically, dataClay deploys a distributed garbage collection service, involving all the backends, to periodically collect any object meeting the following conditions:

1. The object is not pointed by any other object.
2. The object has no aliases.
3. There is no user application referencing the object.
4. There is no backend accessing the object from a running execution method.

1.6 Federation

In some scenarios, such as edge-to-cloud deployments, part of the data stored in a dataClay instance has to be shared with another dataClay instance running in a different device. An example can be found in the context of smart cities where, for instance, part of the data residing in a car is temporarily shared with the city the car is traversing. This partial, and possibly temporal, integration of data between independent dataClay instances is implemented by means of dataClay's federation mechanism. More precisely, federation consists in replicating an object (either simple or complex, such as a collection of objects) in an independent dataClay instance so that the recipient dataClay can access the object without the need to contact the owner dataClay. This provides immediate access to the object, avoiding communications when the object is requested and overcoming the possible unavailability of the data source.

2. My first dataClay application

In order to better understand what dataClay is and how it is used, we present a very simple example (HelloPeople) where data is stored using dataClay. Sections 2.1.1 and 2.1.2 present this example both in Java and Python respectively and from two different perspectives. On the one hand, an Object Store approach with GET(CLONE)/PUT/UPDATE methods based on aliased objects, following Java API defined in Section 4.2 and Python API in Section 5.2). On the other hand, an Object Oriented approach with a reduced usage of aliasing and powered by references to persistent objects, with methods defined in Section 4.3 and Section 5.3.

Documented examples and demos can be found at:

https://github.com/bsc-dom/dataclay-demos

https://github.com/bsc-dom/dataclay-examples

2.1 HelloPeople: a first dataClay example

HelloPeople is a simple application that registers a list of people info into a persistent collection identified by an alias. Every time this application is executed, it first tries to load the collection by its alias, and if it does not exist the application creates it. Once the collection has been retrieved, or created, the given new person info is added to the collection and the whole set of people is displayed.

HelloPeople receives the following parameters:

- a string that identifies the name of the collection.
- a string with the name of the person to be inserted into the collection.
- an integer with the age of the person to be inserted into the collection.

2.1.1 Java

The following code snippets show the class model and two Java HelloPeople applications (Object Store and Object Oriented). The class model is the same for both applications, having a Person class that defines the info to be stored for each registered person: name and age; and the People

class that maintains a list of references to Person objects.

Person.java - Person class

```
package model;

public class Person {
    String name;
    int age;

    public Person(String newName, int newAge) {
        name = newName;
        age = newAge;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

People.java - People class

```
package model;

import java.util.ArrayList;

public class People extends (DataClayObject) {
    private ArrayList<Person> people;

    public People() {
        people = new ArrayList<>();
    }

    public void add(final Person newPerson) {
        people.add(newPerson);
    }

    public String toString() {
        StringBuilder result = new StringBuilder("People: \n");
        for (Person p : people) {
            result.append(" - Name: " + p.getName());
            result.append(" Age: " + p.getAge() + "\n");
        }
        return result.toString();
    }
}
```

HelloPeopleOS.java - Object Store Hello People

```
package app;

import es.bsc.dataclay.api.DataClay;
import model.People;
import model.Person;

public class HelloPeopleOS {
    private static void usage() {
        System.out.println("Usage: application.HelloPeople <peopleAlias> <personName>
            <personAge>");
        System.exit(1);
    }
```

```
    public static void main(final String[] args) {
        try {
            // Check and parse arguments
            if (args.length != 3) {
                    usage();
            }
            final String peopleAlias = args[0];
            final String pName = args[1];
            final int pAge = Integer.parseInt(args[2]);

            // Init dataClay session
            DataClay.init();

            // Retrieve (or create) People collection
            People people = null;
            try {
                people = (People) People.dcCloneByAlias(peopleAlias);
                System.out.println("[LOG] Found People object with alias: " + peopleAlias);
            } catch (final Exception ex) {
                people = new People();
                System.out.println("[LOG] Created a NEW People object!");
            }

            // Check people contents (people iterated locally)
            System.out.println("[LOG] Current people");
            System.out.println(people);

            // Create a person and add it to the collection
            final Person person = new Person(pName, pAge);
            people.add(person);

            // Check people contents (people iterated locally)
            System.out.println("[LOG] Current people at client-side");
            System.out.println(people);

            try {
                // Update if object already exists
                People.dcUpdateByAlias(peopleAlias, people);
                System.out.println("[LOG] Updated existing people object");
            } catch (final Exception ex) {
                // Store it if does not exist
                people.dcPut(peopleAlias);
                System.out.println("[LOG] Stored people object");
            }

            // Retrieve stored people again to check changes
            people = (People) People.dcCloneByAlias(peopleAlias);
            System.out.println("[LOG] Current people at server-side after update");
            System.out.println(people);

            // Finish dataClay session
            DataClay.finish();

            // Exit
            System.exit(0);
        } catch (final Exception e) {
            System.exit(1);
        }
    }
}
```

HelloPeopleOO.java - Object Oriented HelloPeople

```
package app;

import es.bsc.dataclay.api.DataClay;
```

```java
import model.People;
import model.Person;

public class HelloPeopleOO {
    private static void usage() {
        System.out.println("Usage: application.HelloPeople <peopleAlias> <personName>
             <personAge>");
        System.exit(1);
    }

    public static void main(final String[] args) {
        try {
            // Check and parse arguments
            if (args.length != 3) {
                usage();
            }
            final String peopleAlias = args[0];
            final String pName = args[1];
            final int pAge = Integer.parseInt(args[2]);

            // Init dataClay session
            DataClay.init();

            // Access (or create) People collection
            People people;
            try {
                people = People.getByAlias(peopleAlias);
                System.out.println("[LOG] Found People object with alias " + peopleAlias);
            } catch (final Exception ex) {
                people = new People();
                people.makePersistent(peopleAlias);
                System.out.println("[LOG] Created a new People object with alias " + peopleAlias);
            }

            // Add new person to people (person object is persisted in the system)
            final Person person = new Person(pName, pAge);
            people.add(person);
            System.out.println("[LOG] Added a new person, current people:");
            // People is iterated remotely
            System.out.println(people);

            // Finish dataClay session
            DataClay.finish();

            // Exit
            System.exit(0);
        } catch (final Exception e) {
            System.exit(1);
        }
    }
}
```

2.1.2 Python

The following code snippets show the Python HelloPeople applications (Object Store and Object Oriented) and the class model. Analogously to Java model, *person.py* specifies the info to be registered for each person: name and age; and *people.py* maintains a list of references to person objects.

person.py - Person class

```python
from dataclay import DataClayObject, dclayMethod

class Person(DataClayObject):
    """
    @ClassField name str
```

```
    @ClassField age int
    """
    @dclayMethod(name='str', age='int')
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

people.py - People class

```
from dataclay import DataClayObject, dclayMethod

class People(DataClayObject):
    """
    @ClassField people list<model.classes.Person>
    """
    @dclayMethod()
    def __init__(self):
        self.people = list()

    @dclayMethod(new_person="model.classes.Person")
    def add(self, new_person):
        self.people.append(new_person)

    @dclayMethod(return_="str")
    def __str__(self):
        result = ["People:"]

        for p in self.people:
            result.append(" - Name: %s " % p.name)
            result.append(" - Age: %d " % p.age)

        return "\n".join(result)
```

hellopeople_os.py - Object Store HelloPeople

```
import sys

from dataclay.api import init, finish

# Init dataClay session
init()

from HelloPeople_ns.classes import Person, People

class Attributes(object):
    pass


def usage():
    print("Usage: hellopeople.py <colname> <personName> <personAge>")


def init_attributes(attributes):
    if len(sys.argv) != 4:
        print("ERROR: Missing parameters")
        usage()
        exit(2)
    attributes.collection = sys.argv[1]
    attributes.p_name = sys.argv[2]
    attributes.p_age = int(sys.argv[3])


if __name__ == "__main__":
    attributes = Attributes()
    init_attributes(attributes)
```

```
    # Retrieve (or create) people collection
    try:
        people = People.dc_clone_by_alias(attributes.collection)
        print("\n [LOG] Found existing people object with alias " + attributes.collection)
    except Exception:
        people = People()
        print("\n [LOG] Created a new People object!")

    # Check people contents (iterated locally)
    print("\n [LOG] Current people:")
    print(people)

    # Add a new person to people
    person = Person(attributes.p_name, attributes.p_age)
    people.add(person)
    print("\n [LOG] Current people at client-side")
    print(people)

    try:
        # Update persistent people object if it exists (notice that this is a class method)
        People.dc_update_by_alias(attributes.collection, people)
        print("\n [LOG] Updated existing people object")
    except:
        # Put the new object if it does not exist (notice that this is an object method)
        people.dc_put(attributes.collection)
        print("\n [LOG] Stored people object")

    # Retrieve from store to check contents
    people = People.dc_clone_by_alias(attributes.collection)
    print("\n [LOG] Current people at server-side:")
    print(people)

    # Close session
    finish()
    exit(0)
```

hellopeople_oo.py - Object Oriented Hello People

```
import sys

from dataclay.api import init, finish

# Init dataClay session
init()

from HelloPeople_ns.classes import Person, People

class Attributes(object):
    pass


def usage():
    print("Usage: hellopeople.py <colname> <personName> <personAge>")


def init_attributes(attributes):
    if len(sys.argv) != 4:
        print("ERROR: Missing parameters")
        usage()
        exit(2)
    attributes.collection = sys.argv[1]
    attributes.p_name = sys.argv[2]
    attributes.p_age = int(sys.argv[3])


if __name__ == "__main__":
```

```
attributes = Attributes()
init_attributes(attributes)

# Retrieve (or create) people object
try:
    # Trying to retrieve it using alias
    people = People.get_by_alias(attributes.collection)
    print("\n [LOG] Retrieved people's object with alias " + attributes.collection)
except Exception:
    people = People()
    people.make_persistent(alias=attributes.collection)
    print("\n [LOG] Persisted people's object with alias " + attributes.collection)

# Add new person to people (person object is persisted in the system)
person = Person(attributes.p_name, attributes.p_age)
people.add(person)
print("\n [LOG] Added a new person, current people:")
# Check final people contents (iterates remotely)
print(people)

# Close session
finish()
exit(0)
```

3. Application cycle

Now that we have created our first application in Java or Python by defining its class model (Person and People classes) and a main program (HelloPeople.java or hellopeople.py), we can detail the steps that need to be done in order for this application to run using dataClay and store its data in a persistent state. A graphical view of these steps is presented in Figure 3.1 and they are detailed in the following sections.

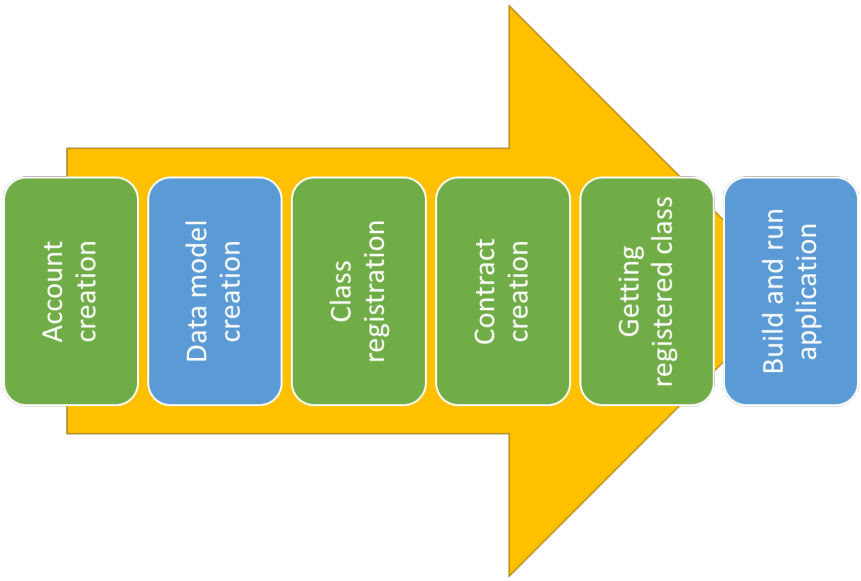


Figure 3.1: Application life cycle

3.1 Account creation

Everybody using dataClay needs to have its own account regardless of the role (model provider or application programmer). Currently accounts are identified by a string and are protected by a password. Accounts are the abstraction used to grant/deny privileges to create/use/access models and/or data.

Although dataClay foresees two different roles with respect to data, they can all be assumed by the same person and this is that case in the HelloPeople example: a single person (you) creates the model, creates application, and inserts the data.

Details on how accounts are created are presented in Chapter 6.

3.2 Namespaces and class models

The model provider is in charge of implementing the Java/Python class models. The involved classes are normally designed and implemented ignoring that they will be eventually used to store persistent objects in dataClay. Once the classes have been created and tested, the model provider needs to register them into dataClay to enable objects of these classes to be stored persistently.

Registering classes is important for three reasons: i) enables dataClay to automatically offer an optimal serialization of the objects instantiating any of these classes, ii) enables dataClay to execute class methods over the objects inside the backends without having to move data to the application, and iii) enables the sharing of classes among application developers in an easy and effective way.

Registering a class model only implies uploading its corresponding class files (e.g. .class files or .py files) into dataClay and defining into which namespace they should be added.

A **namespace** is a dataClay abstraction to group a set of classes together. Namespaces have two objectives: i) grouping related classes to ease the task of sharing them with other users and ii) avoiding class name clashing (for instance two users willing to register class models with names in common for a subset of their classes). That is, a namespace is a similar abstraction as a package in Java or module in Python.

Registering a class model can be easily performed by using the available tools described in Chapter 6 and has to be performed before any application tries to store objects of this class model into dataClay.

3.3 Datasets and data contracts

In the same way we grouped classes into namespaces to ease the task of sharing them, dataClay also has the dataset abstraction. A **dataset** is a set of objects that will be shared with other users as a whole. Objects inside a dataset can be of any class and there is no restriction to the number of objects or their size inside a dataset. Datasets are identified by a string defined by the creator of the dataset.

Datasets can be public or private. Public datasets can be accessed by any user and they will suffice in most scenarios. On the contrary, to access a private dataset its owner has to explicitly grant permission to it. This permission granting is achieved by means of data contracts . Contract creation can be easily performed by using the available tools described in Chapter 6. Once a data contract is created, dataClay will make sure that the user receiving the contract will have access to the objects included in the datasets of the contract. An application can access as many datasets as needed as will be shown in Chapter 8.2.

Notice that in the current public version of dataClay, only public datasets are available.

3.4 Using a registered class: getting its stubs

As we mentioned in Section 3.2, when we implement a class model we do not take into account anything about persistence, but when using it (as seen in the code in Section 2.1), persistent objects have methods such as `makePersistent` that have not been defined as part of the class. In order to be able to use such methods, and thus enable persistence of objects, the application needs to be linked with an automatically modified version of the classes. This modified version is what we refer as **stub classes** or simply **stubs**. A stub is a class file containing the modified version of the original class in order to be compatible with dataClay. It is important to understand that, besides the newly added methods, the rest of the class behaves just like the original one.

The stubs of a class are obtained using the available tools described in Chapter 6.

3.5 Build and run the application

To run a dataClay application, we just need i) to make sure that it is using the stubs instead of the original class files and ii) to create two configuration files that specify our account, datasets, stubs path and connection info. Next section shows an example of these configuration files and Section 8.2 describes further details.

3.6 Easier than it looks

Let's see how we can execute the examples in Sections 2.1.1 and 2.1.2.

First, we define two configuration files. Section 8.2 describes further details and where to place them.

The first one is named *session.properties* and the initialization method (`DataClay.init()` in Java or `api.init()` in Python) will automatically process it. This initialization process is detailed in Section 4.1 for Java and Section 5.1 for Python.

Here is an example:

```
Account=Alice
Password=AlicePass
DataSets=HelloPeopleDS
DataSetForStore=HelloPeopleDS
StubsClasspath=./stubs
```

A second file named *client.properties* contains the basic information for the network connection with dataClay:

```
HOST=127.0.0.1
PORT=11034
```

Now you can start using a simple dataClay command line utility intended for management operations. In this way, our class models can be registered, datasets with specific access rights can be defined, and our application can interact transparently with dataClay when using downloaded stubs (more details about the command line utility are explained in Chapter 6).

```
 # To begin with, create an account
dataclaycmd NewAccount Alice AlicePass

 # Create a dataset (with granted access)
 # to register stored objects on it
dataclaycmd NewDataContract Alice AlicePass myDataset

 # Register the class model in a certain namespace
 # Assuming Person.class or person.py is in ./modelClassDirPath
dataclaycmd NewModel Alice AlicePass myNamespace ./modelClassDirPath \
  <java | python>

 # Download the corresponding stubs for your application
dataclaycmd GetStubs Alice AlicePass myNamespace stubsDirPath
```

II Java: Programmer API

4 Java API 29
4.1 dataClay API
4.2 Object store methods
4.3 Object oriented methods
4.4 Advanced methods
4.5 Error management
4.6 Memory Management and Garbage Collection
4.7 Replica management
4.8 Federation
4.9 Further considerations

4. Java API

This chapter presents the Java API that can be used by applications divided into the following sections. First, in Section 4.1 we present the API intended to initialize and finish applications, as well as, to gather information about the system. In Section 4.2 we show the API for Object Store operations (GET(CLONE)/PUT/UPDATE). Next, in Section 4.3 we introduce extended methods to expand object store operations from an Object-oriented programming perspective. In Section 4.4 we show advanced extensions that will only be needed by a subset of applications. Finally, we present extra concepts such as error handling in Section 4.5, replica management in Section 4.7 and further considerations in Section 4.9.

Notice that current supported Java version is 1.8 (OpenJDK 8, reference implementation of Java SE 8).

4.1 dataClay API

In this section, we present a set of calls that are not linked to any given object, but are general to the system.

In Java, they can be called through *DataClay* main class by including the import:

```
import es.bsc.dataclay.api.DataClay
```

```
public static void  finish () throws DataClayException
```

Description:

Finishes a session with dataClay that has been previously created using init.

Exceptions:

If the session is not initialized or an error occurs while finishing the session, a DataClayException is thrown.

```
public static Map<BackendID, Backend>  getBackends ()
```

Description:

Retrieves the available backends in the system.

Returns:

A map with the available backends in the system indexed by their backend IDs.

Exceptions:

If the session is not initialized, a DataClayException is thrown.

```
public static void  init () throws DataClayException
```

Description:

Creates and initializes a new session with dataClay.

Environment:

`session.properties`: The configuration file can be optionally specified. Location of this file and its contents are detailed in Section 8.2.

Exceptions:

If any error occurs while initializing the session, a DataClayException is thrown.

Example: Using dataClay api - distributed people

```
import es.bsc.dataclay.api.DataClay;
import model.Person;

// Open session with init()
DataClay.init();

List<Person> people = new ArrayList<>();
people.add(new Person("Alice", 32));
people.add(new Person("Bob", 41));
people.add(new Person("Charlie", 35));

// Retrieve backend information with getBackends()
Map<BackendID, Backend> backends = DataClay.getBackends();

BackendID[] backendArray = backends.keySet().toArray();
int numBackends = backends.size();
int i = 0;
for (Person p : people) {
  p.dcPut(backendArray[i % numBackends]);
  i++;
}

// Close session with finish()
DataClay.finish();
```

4.2 Object store methods

Object store methods are those related to common *GET(CLONE)/PUT/UPDATE* operations as introduced in Sections 1.3 and 2.

Given that these three operations are very common in class model definition (e.g. get/put operations in collections), we prepend the “dc” prefix to prevent an unexpected behavior due to potential overriden operations. Notice that *get* is named as *dcClone* to match OO terminology.

This section focuses on a set of static methods that can be called directly from *DataClayObject* class.

4.2.1 Class methods

The following methods can be called from any downloaded stub as static methods. In this way, the user is allowed to access persistent objects by using their aliases (see Section 4.2.2 to see how objects are persisted with an alias assigned). Aliases prevent objects to be removed by the Garbage Collector, thus an operation to remove the alias of an object is also provided. More details on how the garbage collector works can be found in Section 1.5.

Notice that the examples provided assume the initialization and finalization of user's session with methods described in previous section Section 4.1.

```
public static <T> dcCloneByAlias (String alias [, boolean recursive])
throws DataClayException
```

Description:

Retrieves a copy of current object from dataClay. Fields referencing to other objects are kept as remote references to objects stored in dataClay, unless the recursive parameter is set to *True*.

Parameters:

alias: alias of the object to be retrieved.
recursive: When this is set to True, the default behavior is altered so not only current object but all of its references are also retrieved locally.

Returns:

A new object instance initialized with the field values of the object with the alias specified.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using dcCloneByAlias method

```
Person newPerson = new Person("Alice", 32);
newPerson.dcPut("student1");
Person retrieved = Person.dcCloneByAlias("student1");
assertTrue(retrieved.getName().equals(newPerson.getName())
```

```
public static void dcUpdateByAlias (String alias,
DataClayObject fromObject) throws DataClayException
```

Description:

Updates the object identified by specified alias with contents of *fromObject*.

Parameters:

alias: alias of the object to be retrieved.
fromObject: the base object which contents will be used to update target object with alias specified.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.
If object identified with given alias has different fields than *fromObject*, a DataClayException is raised.

Example: Using dcUpdateByAlias method

```
Person newP = new Person("Alice", 32);
newP.dcPut("student1");
Person newValues = new Person("Alice Smith", 35);
Person.dcUpdateByAlias("student1", newValues);
Person clonedP = Person.cloneByAlias("student1");
assertTrue(clonedP.getName().equals(newValues.getName()));
```

4.2.2 Object methods

This section expands Section 4.2.1 with methods that can be called directly from object stub instances. That is, stub classes are adapted to extend a common dataClay class called DataClayObject, which provides the following methods.

```
public <T> dcClone (([boolean recursive]) throws DataClayException
```

Description:

Retrieves a copy of current object from dataClay. Fields referencing to other objects are kept as remote references to objects stored in dataClay.

Parameters:

`recursive:` When this is set to True, the default behavior is altered so not only current object but all of its references are also retrieved locally.

Returns:

A new object instance initialized with the field values of current object. Non-primitive fields or sub-objects are also copied by creating new objects.

Exceptions:

If current object is not persistent, a DataClayException is raised.

Example: Using dcClone method

```
Person p = new Person("Alice", 32);
p.dcPut("student1");
Person copy = p.dcClone()
assertTrue(copy.getAge() == p.getAge())
```

```
public void dcPut () throws DataClayException
public void dcPut (String alias [, BackendID backendID])
throws DataClayException
public void dcPut (String alias [, boolean recursive])
throws DataClayException
public void dcPut (String alias, BackendID backendID
[, boolean recursive]) throws DataClayException
```

Description:

Stores an aliased object in the system and assigns an OID to it. Notice this method allows specifying a certain backend. In this regard, the `DataClay.LOCAL` field can be set as a constant for a specific backendID (as detailed in Section 8.2). To use this field from your application, you have to add the proper import: `import es.bsc.dataClay.api.DataClay`.

Parameters:

`alias:` a string that will identify the object in addition to its OID. Aliases are unique in the system.

`backendID:` identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. When `DataClay.LOCAL` is used, the object is created in the backend specified as local in the client configuration file.

`recursive:` when this flag is True, all objects referenced by the current one will also be made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:

If there is a stored object with the same alias, a DataClayException is raised.
If a backend is specified and it is not valid, a DataClayException is raised. Use getBackends (4.1) to obtain valid backends.

Example: Using dcPut method

```
Person p = new Person("Alice", 32);
p.dcPut("student1", DataClay.LOCAL);
assertTrue(p.getLocation().equals(DataClay.LOCAL));
```

```
public void dcUpdate (DataClayObject fromObject) throws DataClayException
```

Description:

Updates current object with contents of *fromObject*.

Parameters:

`fromObject:` the base object which contents will be used to update target object with alias specified.

Exceptions:

If object to be updated is not persistent, a DataClayException is raised.
If the object has different fields than *fromObject*, a DataClayException is raised.

Example: Using dcUpdate method

```
Person p = new Person("Alice", 32);
p.dcPut("student1");
Person newValues = new Person("Alice Smith", 35);
p.dcUpdate(newValues);
assertTrue(p.getName().equals(newValues.getName()));
```

4.3 Object oriented methods

Besides object-store operations, dataClay also offers a set of methods to enable applications work in a more Object-oriented fashion.

In Object-oriented programming objects are connected by using navigable associations (object references). In dataClay, applications might have objects containing fields associated with other persistent objects through remote object references. Therefore, a set of extended methods are provided to expand object store methods presented in previous section 4.2.

4.3.1 Class methods

```
public static void  deleteAlias  (String alias)
throws DataClayException
```

Description:

Removes the alias linked to an object. If this object is not referenced starting from a root object and no active session is accessing it, the garbage collector will remove it from the system.

Parameters:

`alias:` alias to be removed.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using deleteAlias

```
Person newPerson = new Person("Alice", 32);
newPerson.makePersistent("student1");
...
Person.deleteAlias("student1");
```

```
public static void <T>  getByAlias  (String alias)
throws DataClayException
```

Description:

Retrieves an object reference of current stub class corresponding to the persistent object with alias provided.

Parameters:

`alias:` alias of the object.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using getByAlias

```
Person newPerson = new Person("Alice", 32);
newPerson.makePersistent("student1");
Person refPerson = Person.getByAlias("student1");
assertTrue(newPerson.getName().equals(refPerson.getName())
```

4.3.2 Object methods

In Object-oriented programming, aliases are not required if we can refer to an object by following a navigable association from another object. Therefore, the following method is similar to *dcPut* but offering the possibility to register an object without an alias.

```
public void makePersistent () throws DataClayException
public void makePersistent (BackendID backendID)
throws DataClayException
public void makePersistent (String alias, [BackendID backendID])
throws DataClayException
public void makePersistent (String alias, [boolean recursive])
throws DataClayException
public void makePersistent (BackendID backendID, [boolean recursive])
throws DataClayException
public void makePersistent (String alias, BackendID backendID,
[boolean recursive]) throws DataClayException
```

Description:

Stores an object in dataClay and assigns an OID to it.

Parameters:

alias: a string that will identify the object in addition to its OID. Aliases are unique in the system. If no alias is set, this object will not have an alias, will only be accessible though other object references.

backendID: identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. When `DataClay.LOCAL` is used, the object is created in the backend specified as local in the client configuration file.

recursive: when this flag is True, all objects referenced by the current one will also be made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:

If an alias is specified and there is a stored object with the same alias, a DataClayException is raised.

If a backend is specified and it is not valid, a DataClayException is raised. Use getBackends (4.1) to obtain valid backends.

Example: Using makePersistent method

```
Person p = new Person("Alice", 32);
p.makePersistent("student1", DataClay.LOCAL);
assertTrue(p.getLocation().equals(DataClay.LOCAL));
```

4.4 Advanced methods

In this section we present advanced methods that are also inherited from DataClayObject class. These methods are not intended to be used by standard programmers, but by runtime and library developers or expert programmers.

```
public Set<BackendID>  getAllLocations () throws DataClayException
```

Description:

Retrieves all locations where the object is persisted/replicated.

Returns:

A set of backend IDs in which this object or its replicas are stored.

Exceptions:

If the object is not persistent, a DataClayException is raised.

Example: Using getAllLocations

```
Person p1 = Person.getByAlias("personalias");
Set<BackendID> locations = p1.getAllLocations();
if (!locations.contains(DataClay.LOCAL)) {
 p1.newReplica(DataClay.LOCAL);
}
```

```
public BackendID  getLocation () throws DataClayException
```

Description:

Retrieves a location of the object.

Returns:

Backend ID in which this object is stored. If the object is not persistent (i.e. it has never been persisted) this function will fail.

Exceptions:

If the object is not persistent, a DataClayException is raised.

Example: Using getLocation

```
Person p1 = Person.getByAlias("student1");
p1.makePersistent(DataClay.LOCAL);
assertTrue(p1.getLocation().equals(DataClay.LOCAL));
```

```
public BackendID  newReplica () throws DataClayException
public BackendID  newReplica (boolean recursive) throws DataClayException
public BackendID  newReplica (BackendID backendID
[,boolean recursive]) throws DataClayException
```

Description:

Creates a replica of the current object.
It is important to notice that dataClay does not take care of replica synchronization. Details on how such synchronization can be achieved are described in Section 4.7.

Notice that the replication of an object includes the replication of its subobjects (references) as the default behavior (i.e. recursive is True by default). Therefore, some objects (including current object) might be already present in the destination backend. These

objects will be ignored from replication, since a backend cannot have two replicas of the same object. But it is ensured that, after a correct execution of this method, a full copy of the current object (and all its subobjects if recursive) is present in the returned backend (same as backendID if user specifies it).

Parameters:

backendID: ID of the backend in which to create the replica. If null, a random backend is chosen. When `DataClay.LOCAL` is used, the object is replicated in the backend specified as local in the client configuration file.
recursive: when this flag is True, all objects referenced by the current one will also be replicated (except those that are already present in the destination backend). When this parameter is not set, the default behavior is to perform a recursive replica.

Returns:

The ID of the backend in which the replica was created.

Exceptions:

If the object is not persistent, a DataClayException is raised.
If a backend is specified and it is not valid, a DataClayException is raised. Use getBackends (4.1) to obtain valid backends.

Example: Using newReplica

```
Person p1 = Person.getByAlias("student1");
// replicating object and referenced objects
// from one of its locations to LOCAL
p1.newReplica(p1.getLocation(), DataClay.LOCAL);
```

```
public Object  runRemote (BackendID location,
String opID, Object[] params) throws DataClayException
```

Description:

Executes a specific method on a particular backend. Notice that currently this method is intended for synchronization purposes, as can be seen in section 4.7. Check that section for a proper example.

Parameters:

location: Backend where the method must be executed. When `DataClay.LOCAL` is used, the execution request is sent to the backend specified as local in the client configuration file.
opID: ID of the method to be executed.
params: The regular parameters of the method.

Returns:

The expected result from the execution of the specified method.

Exceptions:

If this object is not persistent, a DataClayException is raised.
If location specified is not valid, a DataClayException is raised. Use getBackends (4.1) to obtain valid backends.

4.5 Error management

Besides *DataClayException* raised from *DataClayObject* methods or dataClay API methods as exposed along this chapter, exceptions raised from methods of your class models while running on a dataClay backend are also forwarded to end-user applications.

However, notice that current version of dataClay does not allow you to register your own exception classes (i.e. as part of your data model), so methods enclosed in your data model can only throw language built-in exceptions.

4.6 Memory Management and Garbage Collection

In section 1.5 we introduced the routines that aim to optimize memory and disk usage in the backends.

In Java, users cannot deallocate objects manually so in dataClay we do not provide a direct operation to do so. However, since we add an extra layer for persistence we have to ensure that the Java Garbage Collector (GC) does not remove loaded objects before they are synchronized with the underlying storage. To this end, a dataClay thread periodically checks if the memory usage reaches a certain threshold and, when this is the case, objects are firstly flushed to persistent storage in a way that Java GC can collect them.

On the other hand, a Global Garbage Collector keeps track of global reference counters in a per object basis. Considering the conditions that an object has to meet in order to be removed, as stated in section 1.5, its associated reference counter not only counts which objects are pointing to it, but also how many aliases it has or the applications and running methods that are using it.

In some cases, certain objects used in long-running applications, such as services, are only needed during a limited period of time, and it may be convenient to free the space they are taking. The following method can be used to notify the Global Garbage Collector that an object is no longer needed by the application or service and, thus, it can be garbage-collected:

```
public final void  sessionDetach ()
```

Description:

Dissociates current object from the current session.

Parameters:

4.7 Replica management

Given that each object or piece of data may potentially need a different consistency model, dataClay will not synchronize objects. On the other hand, it will offer mechanisms for the model developer to include it as part of the model in an easy way, and how to be able to import the consistency model form another class already defined.

The first way to guarantee the consistency level required by a replicated object is to add the needed code in all setters/getter of the class. Although this is a feasible option is quite impractical if we need to add this code to all classes we want to build. Fir this reason, dataClay also offers a mechanism to add arbitrary code (from a static class) to be executed before or after a given method. This mechanism, explained in detail in this section, will enable programmers to build their consistency model once (or use a predefined one) and use it in any of their classes without modifying the class itself.

In this section, we present how to add consistency code into existing classes.

Let us assume that we have our class Person:

```
public class Person {
  String name;
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Once this class is registered and with the proper permissions and stubs, an application that uses it might look like this:

```
public class App {
  public static void main(String[] args) {
    DataClay.init();
    Person p = new Person("Alice", 42);

    p.makePersistent("student1");
    p.newReplica();

    p.setAge(43);
    System.out.println(p.getAge());
  }
}
```

With no consistency policies, the printed message would show an unpredictable age for Alice, since methods *setAge* and *getAge* are executed in a random backend among the locations of the object.

In order to overcome this problem, dataClay provides a mechanism to define synchronization policies at user-level. In particular, class developers are allowed to define three different annotations to customize the behavior of attribute updates:

```
    @Replication.InMaster
    @Replication.BeforeUpdate(method="...", clazz="...")
    @Replication.AfterUpdate(method="...", clazz="...")
```

The *InMaster* annotation forces the update operation to be handled from the master location. The default master location of an object is the backend where the object was originally stored.

On the other hand, *BeforeUpdate* and *AfterUpdate* define extra behavior to be executed before or after the update operation. The *method* argument specifies an operation signature of a static class method. The *clazz* argument refers to the class where such a static class method is implemented. In this way, the developer is allowed to define an action to be triggered before the update operation, and an action to be taken after the update operation.

Let us resume our previous example. Assuming that the *name* attribute is never modified (e.g. private setter), we want, however, that every time the age is updated the change is propagated to all the replicas. Empowering Person class with the proper annotation, we can intervene updates of attribute *age* to perform the update synchronization:

```
public class Person {
  String name;

  @Replication.InMaster
  @Replication.AfterUpdate(method="replicateToSlaves",
```

```
                                clazz="model.SequentialConsistency")
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Following the example, and as part of the class model, the proposed *SequentialConsistency* class can be implemented as follows:

```
package model;

import java.util.Set;

import api.BackendID;
import serialization.DataClayObject;

public class SequentialConsistency {
  public static void replicateToSlaves(DataClayObject o, String setter, Object[] args) {
    Set<BackendID> locations = o.getAllLocations();
    for (BackendID replicaLocation : locations) {
      if (!replicaLocation.equals(o.getMasterLocation())) {
        o.runRemote(replicaLocation, setter, args);
      }
    }
  }
}
```

In this example, the master replica leads a sequential consistency model by synchronizing the contents with secondary replicas.

Some considerations merit the attention of model developers:

The master location of an object can be checked with the method *getMasterLocation()*.
The method name specified in the annotations is always implemented as a *public static void* operation, which receives the context info about the original method that triggered the action. This context info consists of:

- A dataClay object reference. Object in which the original method is being executed. In our example, a reference to Person object.
- The method itself. An identifier that dataClay can manage. In our example, the *setAge* method identifier.
- The arguments received by the method. In our example, the new age to be set.

For convenience, the implementation of the *SequentialConsistency* class in the example can be used by including the import:

```
import es.bsc.dataclay.util.replication.Replication
```

4.8 Federation

In some scenarios, such as edge-to-cloud environments, part of the data stored in a dataClay instance has to be shared with another dataClay instance running in a different device. An example can be found in the context of smart cities where, for instance, part of the data residing in a car is temporarily shared with the city the car is traversing. This partial, and possibly temporal, integration of data between independent dataClay instances is implemented by means of dataClay's federation mechanism. More precisely, federation consists in replicating an object (either simple or complex, such as a collection of objects) in an independent dataClay instance so that the recipient dataClay

can access the object without the need to contact the owner dataClay. This provides immediate access to the object, avoiding communications when the object is requested and overcoming the possible unavailability of the data source.

An object can be federated with an unlimited number of other dataClay instances. Additionally, a dataClay instance that receives a federated object can federate it with other dataClay instances.

Federated objects can be synchronized in all dataClay instances sharing them, in such a way that only those parts of the data that change are transferred through the network in order to avoid unnecessary transfers. This is achieved analogously to the synchronization of replicas stored among different backends of a single dataClay, as explained below.

To federate an object, both the source and the target dataClay must have the same data model registered. This is achieved by importing the model from the target dataClay, or from another dataClay instance holding the same model as the target dataClay. This process is done through the methods *RegisterDataClay* and *ImportModelsFromExternalDataClay* (as well as the usual *GetStubs*) before the execution of the application (see Section 6).

In this section we present how to manage federation of objects that instantiate Java classes.

Assume we have our class Person:

```
public class Person {
  String name;
  int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

An application that federates an object of this class with another dataClay might look like this:

```
import es.bsc.dataclay.api.DataClay;

public class App {
  public static void main(String[] args) {
    DataClay.init();
    otherDC = DataClay.registerDataClay(host, port)

    Person p = new Person("Alice", 42);
    p.makePersistent("person1");

    p.federate(otherDC);
  }
}
```

The first step is to make both dataClay instances aware of each other by means of the *registerDataClay* method, explained in section 4.8.1. The dataClay instance id returned by this call is used as a parameter for the *federate* call on the object to indicate the dataClay instance that will receive the federated object. As explained above, note that both dataClay instances must have the same data model registered. At this point, an application accessing the dataClay instance *otherDC* can execute the following code:

```
public class App {
  public static void main(String[] args) {
    DataClay.init();
```

```
    Person p = Person.getByAlias("person1");

    System.out.println(p.name);
  }
}
```

The secondary dataClay has actually performed a replica of Person object aliased *person1*. From now on, this replica can be used in the execution environment of any of the backends of the secondary dataClay, as any other object created in *otherDC*.

A user-defined behaviour can optionally be attached to the class of the object to be federated, which will be executed upon reception of the object in the target dataClay instance. To do this, a method *whenFederated* must be implemented in the corresponding class, for instance:

```
public class Person {
  String name;
  int age;

 public Person(String name, int age) {
    this.name = name;
    this.age = age;

 public whenFederated() {
    PersonList pl = PersonList.getByAlias("persons");
    pl.add(this);
  }
}
```

In this way, the application accessing the target dataClay instance can use the collection *pl* to get all the available objects of class *Person* at any time. Notice that *pl* is not a federated object, but a collection residing in the target dataClay instance that includes objects federated from the source dataClay (as well as possibly other objects created in the target dataClay instance).

```
public class App {
  public static void main(String[] args) {
    ...

    PersonList pl = new PersonList();
    pl.makePersistent("persons");
    ...
    length = pl.size();
    ...
  }
}
```

Federated objects can be synchronized using the same mechanisms provided to synchronize replicas within a dataClay instance, as explained in 4.7. To implement customized synchronization mechanisms on federated objects, the methods to be used are *getFederationTargets*, which returns the identifiers of the dataClay instances where the object is federated, and *getFederationSource*, which returns the source dataClay instance of a federated object in the current dataClay. Also, the method *setInBackend* is provided to execute a setter method on the replica of the object that is stored in the specified dataClay instance. The description of these methods can be found in section 4.8.2.

For convenience, to synchronize federated objects following a sequential consistency policy, the method *synchronizeFederated* in the same *SequentialConsistency* class can be used.

Both the source and the target dataClay instance can stop sharing an object by calling the *unfederate*

method on the federated object. Then, the replica in the target dataClay will be eventually removed by the garbage collector unless it has an alias or it is referenced by another object. In any case, it will cease to be synchronized with the original object.

Analogously to federation, the method *whenUnfederated* can be implemented in the corresponding class to execute a customized behaviour in the target dataClay instance when an object is unfederated (for instance, removing the object from the *PersonList* in the example above, so that the object can be garbage-collected.

In the following we present the API provided by dataClay to manage the federation of objects between dataClay instances. It comprises a set of methods that are part of the dataClay API to manage the connection between different dataClay instances, as well as object methods to manage the federation of objects. Recall that methods from the dataClay API can be called through *DataClay* main class by including the import:

```
import es.bsc.dataclay.api.DataClay
```

4.8.1 dataClay API methods

```
public static DataClayInstanceID  getDataClayID ([String host, String port])
```

Description:

Retrieves the ID of the dataClay instance accessible in *host*, *port*, or of the current dataClay instance if there are no parameters.

Parameters:

host: host where the dataClay instance is located.
port: port where the dataClay instance is listening.

Returns:

The ID of the current dataClay instance, or of the dataClay instance located in *host*, *port*.

```
public static DataClayInstanceID  registerDataClay (String host, String port)
```

Description:

Makes the current dataClay instance aware of another dataClay instance accessible in *host* and *port*, and returns its ID.

Parameters:

host: host where the dataClay instance to be registered is located.
port: port where the dataClay instance to be registered is listening.

Returns:

The ID of the dataClay instance located in *host*, *port*.

4.8.2 Object methods

```
public void  federate (DataClayInstanceID dcID [,boolean recursive])
```

Description:

Federates current object with another dataClay instance, replicating it in any of its backends.

Parameters:

dcID: ID of the external dataClay. It must be previously registered.
recursive: when this flag is TRUE, all objects (recursively) referenced by the current one will also be federated (except those that are already present in the destination dataClay). This parameter is optional, default value is TRUE.

Example: Using federate

```
DataClayID otherDC = DataClay.getDataClayID(host, port);
Person p1 = Person.getByAlias("person1");
// federating object and subobjects to otherDC (previously registered)
p1.federate(otherDC);
```

`public void federateToBackend (BackendID backend [,boolean recursive])`

Description:

Federates current object with another dataClay instance, replicating it in the indicated backend.

Parameters:

backendID: ID of a backend in an external dataClay instance, which must be previously registered.
recursive: when this flag is TRUE, all objects (recursively) referenced by the current one will also be federated (except those that are already present in the destination dataClay). This parameter is optional, default value is TRUE.

`public DataClayInstanceID getFederationSource ()`

Description:

Retrieves the ID of the dataClay instance where the object is federated from.

Returns:

A DataClayInstanceID which is the source of this federated object. It is null if the object is not federated.

`public Set<DataClayInstanceID> getFederationTargets ()`

Description:

Retrieves the IDs of all the dataClay instances where the object is federated.

Returns:

A set of DataClayInstanceID objects in which this object is federated. It is empty if the object is not federated.

Example: Using getFederationTargets

```
Person p1 = Person.getByAlias("personalias");
// using getFederationTargets to check if p1 is federated
Set<DataClayInstanceID> federation = p1.getFederationTargets();
```

```
return (!federation.size() == 0)
```

```
public void  setInDataClayInstance (DataClayInstanceID dcID,
ImplementationID setterID, Object[] params)
```

Description:

Executes a setter on a particular dataClay where the object is federated.

Parameters:

dcID: dataClay instance where the method must be executed.
setterID: ID of the setter to be executed.
params: The parameters of the method.

```
public void  unfederate ([DataClayInstanceID dcID] [,boolean recursive])
```

Description:

Unfederates current object (and referenced objects) with the indicated dataClay instance. If no *dcID* is specified, the object is unfederated from all the instances where it lives.

Parameters:

dcID: ID of the external dataClay. It must be previously registered.
recursive: when this flag is TRUE, all objects (recursively) referenced by the current one will also be unfederated. This parameter is optional, default value is TRUE.

4.9 Further considerations

This section exposes some particularities that are coupled to current dataClay requirements or limitations.

4.9.1 Importing registered classes

In order to use certain classes from registered data models you will have to specify the imports for the corresponding stubs. To this end, you have to ensure that Java *classpath* includes the path of your stubs directory when running your applications.

4.9.2 Non-registered classes

Non-registered mutable types (such as Java built-in collections) are opaque to dataClay. Thus, when a registered class has one of such objects (as a field) and this mutable object is modified from outside its containing class, the changes in the mutable object may not be reflected.

For example, given a Class A with a field b of type B, and B has a field list of type `ArrayList`. After executing the instruction `this.b.list.add(x)` from a method in A, the list may not contain the new element x. To solve this, the class model should define a method in class B containing the instruction `b.list.add(x)` and call it from class A.

4.9.3 Third party libraries

Sometimes using third-party libraries from registered data models is not trivial, thus if you experience such problems, please contact us by email: `support-dataclay@bsc.es`

III Python: Programmer API

5 Python API 49
5.1 dataClay API
5.2 Object store methods
5.3 Object oriented methods
5.4 Advanced methods
5.5 Error management
5.6 Memory Management and Garbage Collection
5.7 Replica management
5.8 Non-volatile Memory support
5.9 Federation
5.10 Further considerations

5. Python API

This chapter presents the Python API that can be used by applications divided into the following sections. First, in Section 5.1 we present the API intended to initialize and finish applications, as well as, to gather information about the system. In Section 5.2 we show the API for Object Store operations (GET(CLONE)/PUT/UPDATE). Next, in Section 5.3 we introduce extended methods to expand object store operations from an Object-oriented programming perspective. In Section 5.4 we show advanced extensions that will only be needed by a subset of applications. Finally, we present extra concepts such as error handling in Section 5.5, replica management in Section 5.7 and further considerations in Section 5.10.

5.1 dataClay API

In this section, we present a set of calls that are not linked to any given object, but are general to the system.

In Python, they can be called through *dataclay.api* with the proper import:

```
from dataclay.api import finish, init, get_backends
```

```
def  finish ():
```

Description:

Finishes a session with dataClay that has been previously created using init.

Exceptions:

If the session is not initialized or an error occurs while finishing the session, a DataClayException is thrown.

```
def get_backends ():
```

Description:

Retrieves the available backends in the system.

Returns:

A map with the available backends in the system indexed by their IDs.

If the session is not initialized, a DataClayException is thrown.

```
def init (config_file='./cfgfiles/session.properties'):
```

Description:

Creates and initializes a new session with dataClay.

Environment:

`session.properties`: The configuration file can be optionally specified. Location of this file and its contents are detailed in Section 8.2.

Exceptions:

If any error occurs while initializing the session, a DataClayException is thrown.

Example: Using dataClay api - distributed people

```
from itertools import cycle
from dataclay.api import finish, init, get_backends

# Open session with init()
init()

from model import Person

student1 = Person(name="Alice", age=32)
person2 = Person(name="Bob", age=41)
person2 = Person(name="Charlie", age=35)
people = [p1, p2, p3]

# Retrieve backend information with get_backends()
backends = get_backends().keys()

# Round robin of persons in backends
for person, backend in zip(people, cycle(backends)):
    person.dc_put(backend_id=backend)

# Close session with finish()
finish()
```

5.2 Object store methods

Object store methods are those related to common *GET(CLONE)/PUT/UPDATE* operations as introduced in Sections 1.3 and 2.

Given that these three operations are very common in class model definition (e.g. get/put operations in collections), we prepend the "dc" prefix to prevent an unexpected behavior due to potential overriden operations. Notice that *get* is named as *dc_clone* to match OO terminology.

This section focuses on a set of static methods that can be called directly from *DataClayObject* class.

5.2.1 Class methods

The following methods can be called from any downloaded stub as class methods. In this way, the user is allowed to access persistent objects by using their aliases (see Section 5.2.2 to see how objects are persisted with an alias assigned). Aliases prevent objects to be removed by the Garbage Collector, thus an operation to remove the alias of an object is also provided. More details on how the garbage collector works can be found in Section 1.5.

Notice that the examples provided assume the initialization and finalization of user's session with methods described in previous section Section 5.1.

```
def dc_clone_by_alias (cls, alias, recursive=False):
```

Description:

Retrieves a copy of current object from dataClay. Fields referencing to other objects are kept as remote references to objects stored in dataClay, unless the recursive parameter is set to *True*.

Parameters:

alias: alias of the object to be retrieved.
recursive: When this is set to True, the default behavior is altered so not only current object but all of its references are also retrieved locally.

Returns:

A new object instance initialized with the field values of the object with the alias specified.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using dc_clone_by_alias method

```
new_person = Person(name="Alice", age=32)
new_person.dc_put("student1")
retrieved = Person.dc_clone_by_alias("student1")
assert retrieved.get_name() == new_person.get_name()
```

```
def dc_update_by_alias (cls, alias, from_object):
```

Description:

Updates the object identified by specified alias with contents of *from_object*.

Parameters:

alias: alias of the object to be retrieved.
from_object: the base object which contents will be used to update target object with alias specified.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.
If object identified with given alias has different fields than *from_object*, a DataClayException is raised.

Example: Using dc_update_by_alias method

```
new_person = new Person(name="Alice", age=32)
new_person.dc_put("student1")
new_values = Person(name="Alice Smith", age=35)
Person.dc_update_by_alias("student1", new_values)
cloned_person = Person.dc_clone_by_alias("student1")
assert cloned_person.get_name() == new_values.get_name()
```

5.2.2 Object methods

This section expands Section 5.2.1 with methods that can be called directly from object instances. That is, stub classes are adapted to extend a common dataClay class called DataClayObject, which provides the following methods.

```
def dc_clone (recursive=False)
```

Description:

Retrieves a copy of current object from dataClay. Fields referencing to other objects are kept as remote references to objects stored in dataClay.

Parameters:

recursive: When this is set to True, the default behavior is altered so not only current object but all of its references are also retrieved locally.

Returns:

A new object instance initialized with the field values of current object. Non-primitive fields or sub-objects are also copied by creating new objects.

Exceptions:

If current object is not persistent, a DataClayException is raised.

Example: Using dc_clone method

```
new_person = Person(name="Alice", age=32)
new_person.dc_put("student1")
copy = new_person.dc_clone()
assert copy.get_age() == new_person.get_age()
```

```
def dc_put (self, alias, backend_id=None, recursive=True):
```

Description:

Stores an aliased object in the system and assigns an OID to it. Notice that next method allows specifying a certain backend. In this regard, the `api.LOCAL` field can be set as a constant for a specific backendID (as detailed in Section 8.2). To use this field from your application, you have to add the proper import: `from dataclay import api`.

Parameters:

alias: a string that will identify the object in addition to its OID. Aliases are unique in the system.

backendID: identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. When `api.LOCAL` is used, the object is created in the backend specified as local in the client configuration file.

recursive: when this flag is True, all objects referenced by the current one will also be

made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:

If there is a stored object with the same alias, a DataClayException is raised.
If a backend is specified and it is not valid, a DataClayException is raised. Use get_backends (5.1) to obtain valid backends.

Example: Using dc_put method

```
new_person = Person(name="Alice", age=32)
new_person.dc_put("student1", api.LOCAL)
locations = list(p1.get_all_locations())
assert api.LOCAL in locations
```

```
def dc_update (self, from_object)
```

Description:

Updates current object with contents of *from_object*.

Parameters:

`from_object:` the base object which contents will be used to update target object with alias specified.

Exceptions:

If object to be updated is not persistent, a DataClayException is raised.
If the object has different fields than *from_object*, a DataClayException is raised.

Example: Using dc_update method

```
new_person = Person(name="Alice", age=32)
new_person.dc_put("student1")
new_values = Person(name="Alice Smith", age=35)
new_person.dc_update(new_values)
assert new_person.get_name() == new_values.get_name()
```

5.3 Object oriented methods

Besides object store operations, dataClay also offers a set of methods to enable applications work in a more Object-oriented fashion.

In Object-oriented programming objects are connected by using navigable associations (object references). In dataClay, applications might have objects containing fields associated with other persistent objects through remote object references. Therefore, a set of extended methods are provided to expand object store methods presented in previous section 5.2.

5.3.1 Class methods

```
def delete_alias (cls, alias):
```

Description:

Removes the alias linked to an object. If this object is not referenced starting from a root object and no active session is accessing it, the garbage collector will remove it from the

system.

Parameters:

`alias:` alias to be removed.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using delete_alias

```
new_person = Person(name="Alice", age=32)
new_person.make_persistent("student1")
...
Person.delete_alias("student1")
```

```
def get_by_alias (cls, alias):
```

Description:

Retrieves an object reference of current stub class corresponding to the persistent object with alias provided.

Parameters:

`alias:` alias of the object.

Exceptions:

If no object with specified alias exists, a DataClayException is raised.

Example: Using get_by_alias

```
new_person = Person(name="Alice", age=32)
new_person.make_persistent("student1")
ref_person = Person.get_by_alias("student1")
assert new_person.get_name() == ref_person.get_name()
```

5.3.2 Object methods

In Object-oriented programming, aliases are not required if we can refer to an object by following a navigable association from another object. Therefore, the following method is similar to *dc_put* but offering the possibility to register an object without an alias.

```
def make_persistent (self, alias=None, backend_id=None, recursive=True):
```

Description:

Stores an object in dataClay and assigns an OID to it.

Parameters:

`alias:` a string that will identify the object in addition to its OID. Aliases are unique in the system. If no alias is set, this object will not have an alias, will only be accessible though other object references.

`backend_id:` identifies the backend where the object will be stored. If this parameter is missing, then a random backend is selected to store the object. When `api.LOCAL` is used,

the object is created in the backend specified as local in the client configuration file.
recursive: when this flag is True, all objects referenced by the current one will also be made persistent (in case they were not already persistent) in a recursive manner. When this parameter is not set, the default behavior is to perform a recursive makePersistent.

Exceptions:

If an alias is specified and there is a stored object with the same alias, a DataClayException is raised.
If a backend is specified and it is not valid, a DataClayException is raised. Use get_backends (5.1) to obtain valid backends.

Example: Using make_persistent

```
p1 = Person(name="Alice", age=32)
p1.make_persistent("student1", api.LOCAL)
assert p1.get_location() == api.LOCAL
```

5.4 Advanced methods

In this section we present advanced methods that are also inherited from DataClayObject class. These methods are not intended to be used by standard programmers, but by runtime and library developers or expert programmers.

```
def get_all_locations (self):
```

Description:

Retrieves all locations where the object is persisted/replicated.

Returns:

A set of backend IDs in which this object or its replicas are stored.

Exceptions:

If the object is not persistent, a DataClayException is raised.

Example: Using get_all_locations

```
new_person = Person(name="Alice", age=32)
new_person.make_persistent("student1", api.LOCAL)
locations = list(p1.get_all_locations())
assert api.LOCAL in locations
```

```
def get_location (self):
```

Description:

Retrieves a location of the object.

Returns:

Backend ID in which this object is stored. If the object is not persistent (i.e. it has never been persisted) this function will fail.

Exceptions:

If the object is not persistent, a DataClayException is raised.

Example: Using get_location

```
new_person = Person(name="Alice", age=32)
new_person.make_persistent("student1", api.LOCAL)
assert new_person.get_location() == api.LOCAL
```

`def new_replica (self, backend_id=None, recursive=True):`

Description:

Creates a replica of the current object.
It is important to notice that dataClay does not take care of replica synchronization. Details on how such synchronization can be achieved are described in Section 5.7.

Notice that the replication of an object includes the replication of its subobjects (references) as the default behavior (i.e. recursive is True by default). Therefore, some objects (including current object) might be already present in the destination backend. These objects will be ignored from replication, since a backend cannot have two replicas of the same object. But it is ensured that, after a correct execution of this method, a full copy of the current object (and all its subobjects if recursive) is present in the returned backend (same as backend_ID if user specifies it).

Parameters:

`backend_id:` ID of the backend in which to create the replica. If null, a random backend is chosen. When `api.LOCAL` is used, the object is replicated in the backend specified as local in the client configuration file.
`recursive:` when this flag is True, all objects referenced by the current one will also be replicated (except those that are already present in the destination backend). When this parameter is not set, the default behavior is to perform a recursive replica.

Returns:

The ID of the backend in which the replica was created.

Exceptions:

If the object is not persistent, a DataClayException is raised.
If a backend is specified and it is not valid, a DataClayException is raised. Use get_backends (5.1) to obtain valid backends.

Example: Using new_replica

```
p1 = Person.get_by_alias("student1")
# replicating object and referenced objects
# from one of its locations to LOCAL
p1.new_replica(api.LOCAL)
```

`def run_remote (backend_id, operation_name, params)`

Description:

Executes a specific method on a particular backend. Notice that currently this method is intended for synchronization purposes, as can be seen in section 5.7. Check that section

for a proper example.

Parameters:

`backend_id,:` Backend where the method must be executed. When `api.LOCAL` is used, the execution request is sent to the backend specified as local in the client configuration file.
`operation_name:` Method to be executed.
`params:` The regular parameters of the method.

Returns:

The expected result from the execution of the specified method.

Exceptions:

If this object is not persistent, a DataClayException is raised.
If location specified is not valid, a DataClayException is raised. Use get_backends (5.1) to obtain valid backends.

5.5 Error management

Besides *DataClayException* raised from *DataClayObject* methods or dataClay API methods as exposed along this chapter, exceptions raised from methods of your class models while running on a dataClay backend are also forwarded to end-user applications.

However, notice that current version of dataClay does not allow you to register your own exception classes (i.e. as part of your data model), so methods enclosed in your data model can only throw language built-in exceptions.

5.6 Memory Management and Garbage Collection

In section 1.5 we introduced the routines that aim to optimize memory and disk usage in the backends.

In Python, users cannot deallocate objects manually so dataClay does not provide a direct operation to do that. However, since we add an extra layer for persistence we have to ensure that Python does not remove objects before they are synchronized with the underlying storage. To this end, a dataClay thread periodically checks if the memory usage reaches a certain threshold and, when this is the case, objects are firstly flushed to persistent storage in a way that the Python GC can collect them.

On the other hand, a Global Garbage Collector keeps track of global reference counters in a per object basis. Considering the conditions that an object has to meet in order to be removed, as stated in section 1.5, its associated reference counter not only counts which objects are pointing to it, but also how many aliases it has or the applications and running methods that are using it.

In some cases, certain objects used in long-running applications, such as services, are only needed during a limited period of time, and it may be convenient to free the space they are taking. The following method can be used to notify the Global Garbage Collector that an object is no longer needed by the application or service and, thus, it can be garbage-collected:

```
def  session_detach (self)
```

Description:

Dissociates current object from the current session.

Parameters:

5.7 Replica management

Given that each object or piece of data may potentially need a different consistency model, dataClay will not synchronize objects. On the other hand, it will offer mechanisms for the model developer to include it as part of the model in an easy way, and how to be able to import the consistency model form another class already defined.

The first way to guarantee the consistency level required by a replicated object is to add the needed code in all setters/getter of the class. Although this is a feasible option is quite impractical if we need to add this code to all classes we want to build. Fir this reason, dataClay also offers a mechanism to add arbitrary code (fom a static class) to be executed before or after a given method. This mechanism, explained in detail in this section, will enable programmers to build their consistency model once (or use a predefined one) and use it in any of their classes without modifying the class itself.

In this section, we present how to add consistency code into existing classes.

Let us assume that we have our class Person:

```
class Person(DataClayObject):
    """
    @ClassField name str
    @ClassField age int
    """
    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Once this class is registered and with the proper permissions and stubs, an application that uses it might look like this:

```
# Initialize dataClay
from dataclay.api import init, finish, get_backends
init()

from model.classes import *

if __name__ == "__main__":
    p = Person("foo", 100)
    backends = get_backends().keys()

    p.make_persistent(backend_id=backends[0])
    p.new_replica(backend_id=backends[1])

    p.age = 1000
    print(p.age)
    finish()
```

With no consistency policies, the printed message would show an unpredictable age for Alice, since getters and setters are executed in a random backend among the locations of the object.

In order to overcome this problem, dataClay provides a mechanism to define synchronization policies at user-level. In particular, class developers are allowed to define an annotation to customize the behavior of attribute updates:

```
@dclayReplication(inMaster='...', beforeUpdate='...', afterUpdate='...')
@ClassField name type
```

The *inMaster* annotation forces the update operation to be handled from the master location if set to true. The default master location of an object is the backend where the object was originally stored.

On the other hand, *beforeUpdate* and *afterUpdate* define extra behavior to be executed before or after the update operation. Their value corresponds to an operation signature of a method, which can belong to the same class or can be inherited from a mixin. In this way, the developer can define an action to be triggered before the update operation, and/or an action to be taken after the update operation.

The *ClassField* annotation defines which fields of the class have to apply the defined behavior.

Let us resume our previous example. Assuming that the *name* attribute is never modified (e.g. private setter), we want, however, that every time the *age* is updated the change is propagated to all the replicas. Empowering Person class with the proper annotations, we can intervene updates of attribute *age* to perform the update synchronization, which has been implemented in the class *SequentialConsistencyMixin*, extended by *Person*:

```
class Person(DataClayObject, SequentialConsistencyMixin):
    """
    @dclayReplication(afterUpdate='synchronize', inMaster='True')
    @ClassField age int
    """

    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Following the example, and as part of the class model, the proposed *SequentialConsistencyMixin* class can be implemented as follows:

```
class SequentialConsistencyMixin(object):

    @dclayMethod(attribute="str", value="anything")
    def synchronize(self, attribute, value):
        for exeenv_id in self.get_all_locations().keys():
            if exeenv_id != master_location:
                self.run_remote(exeenv_id, attribute, value)
```

In this example, the master replica leads a sequential consistency model by synchronizing the contents with secondary replicas.

Some considerations merit the attention of model developers:

The master location can be checked through the field *master_location*.

The method whose name is specified in the annotations is implemented in the model class or a superclass and receives the attribute name to be set and its new value.

For convenience, the implementation of the *SequentialConsistencyMixin* class in the example can be used by including the import:

```
from dataClay.contrib.synchronization import SequentialConsistencyMixin
```

5.8 Non-volatile Memory support

Non-Volatile Memory devices (NVM), also known as persistent memories, are storage devices that blur the line between DRAM and SSD.

There are two strong motivations for supporting NVM devices in dataClay:

They have much better performance than drives, available at a fraction of the price-per-byte cost of DRAM. This allows dataClay to manage big datasets – datasets that would not fit in DRAM– without incurring in the high performance penalties of storing and accessing them to/from drives.
They are byte-addressable, which enables in-place execution (i.e., direct load/store accesses are possible and efficient, without moving data to DRAM). Being able to execute code in-place can boost the data locality; this perfectly combines with the idea of a system such as dataClay, which is itself improving data locality.

dataClay provides the *npp2nvm library* to facilitate placing numpy array structures into NVM devices, so that they can be directly accessed during execution.

5.9 Federation

In some scenarios, such as edge-to-cloud environments, part of the data stored in a dataClay instance has to be shared with another dataClay instance running in a different device. An example can be found in the context of smart cities where, for instance, part of the data residing in a car is temporarily shared with the city the car is traversing. This partial, and possibly temporal, integration of data between independent dataClay instances is implemented by means of dataClay's federation mechanism. More precisely, federation consists in replicating an object (either simple or complex, such as a collection of objects) in an independent dataClay instance so that the recipient dataClay can access the object without the need to contact the owner dataClay. This provides immediate access to the object, avoiding communications when the object is requested and overcoming the possible unavailability of the data source.

An object can be federated with an unlimited number of other dataClay instances. Additionally, a dataClay instance that receives a federated object can federate it with other dataClay instances.

Federated objects can be synchronized in all dataClay instances sharing them, in such a way that only those parts of the data that change are transferred through the network in order to avoid unnecessary transfers. This is achieved analogously to the synchronization of replicas stored among different backends of a single dataClay, as explained below.

To federate an object, both the source and the target dataClay must have the same data model registered. This is achieved by importing the model from the target dataClay, or from another dataClay instance holding the same model as the target dataClay. This process is done through the methods *RegisterDataClay* and *ImportModelsFromExternalDataClay* (as well as the usual *GetStubs*) before the execution of the application (see Section 6).

In this section, we present how to manage federation of objects that instantiate Python classes.

Assume we have our class Person:

```
class Person(DataClayObject):
    """
    @ClassField name str
    @ClassField age int
    """
    @dclayMethod(name="str", age="int")
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

An application that federates an object of this class with another dataClay might look like this:

```
# Initialize dataClay
from dataclay.api import init, finish, register_dataclay

init()

from model.classes import *

if __name__ == "__main__":

    other_dc = register_dataclay(host, port)

    p = Person('Alice', 42)

    p.make_persistent('person1')

    p.federate(other_dc)

    finish()
```

The first step is to make both dataClay instances aware of each other by means of the *register_dataclay* method, explained in section 5.9.1. The dataClay instance id returned by this call is used as a parameter for the *federate* call on the object to indicate the dataClay instance that will receive the federated object. As explained above, note that both dataClay instances must have the same data model registered. At this point, an application accessing the dataClay instance *other_dc* can execute the following code:

```
# Initialize dataClay
from dataclay.api import init, finish

init()

from model.classes import *

if __name__ == "__main__":
    p = Person.get_by_alias('person1')

    assert p.get\_name() == 'Alice'

    finish()
```

The secondary dataClay has actually performed a replica of Person object aliased *person1*. From now on, this replica can be used in the execution environment of any of the backends of the secondary dataClay, as any other object created in *other_dc*.

A user-defined behaviour can optionally be attached to the class of the object to be federated, which will be executed upon reception of the object in the target dataClay instance. To do this, a method *when_federated* must be implemented in the corresponding class, for instance:

```
class Person(DataClayObject):
     ...

     @dclayMethod()
     def when_federated():
     pl = PersonList.get_by_alias('persons')
     pl.add(self);
  }
}
```

In this way, the application accessing the target dataClay instance can use the collection *pl* to get

all the available objects of class *Person* at any time. Notice that *pl* is not a federated object, but a collection residing in the target dataClay instance that includes objects federated from the source dataClay (as well as possibly other objects created in the target dataClay instance).

```
...
  if __name__ == "__main__":
    pl = Person()
    pl.make_persistent('persons')
    ...
    length = len(pl)
    ...
```

Federated objects can be synchronized using the same mechanisms provided to synchronize replicas within a dataClay instance, as explained in 5.7. To implement customized synchronization mechanisms on federated objects, the methods to be used are *get_federation_targets*, which returns the identifiers of the dataClay instances where the object is federated, and *get_federation_source*, which returns the source dataClay instance of a federated object in the current dataClay. Also, the method *set_in_backend* is provided to execute a setter method on the replica of the object that is stored in the specified dataClay instance. The description of these methods can be found in section **??**.

For convenience, to synchronize federated objects following a sequential consistency policy, the method *synchronize_federated* in the same *SequentialConsistencyMixin* class can be used.

Both the source and the target dataClay instance can stop sharing an object by calling the *unfederate* method on the federated object. Then, the replica in the target dataClay will be eventually removed by the garbage collector unless it has an alias or it is referenced by another object. In any case, it will cease to be synchronized with the original object.

Analogously to federation, the method *when_unfederated* can be implemented in the corresponding class to execute a customized behaviour in the target dataClay instance when an object is unfederated (for instance, removing the object from the list in the example above, so that the object can be garbage-collected.

In the following we present the API provided by dataClay to manage the federation of objects between dataClay instances. It comprises a set of methods that are part of the dataClay API to manage the connection between different dataClay instances, as well as object methods to manage the federation of objects. Recall that methods from the dataClay API can be called through *dataclay.api* with the proper import, for instance:

```
from dataclay.api import finish, init, register_dataclay
```

5.9.1 dataClay API methods

```
def  get_dataclay_id ([host, port]):
```

Description:

Retrieves the ID of the dataClay instance accessible in *host*, *port*, or of the current dataClay instance if there are no parameters.

Parameters:

host: host where the dataClay instance is located.
port: port where the dataClay instance is listening.

Returns:

The ID of the current dataClay instance, or of the dataClay instance located in *host*, *port*.

```
def register_dataclay (host, port):
```

Description:

Makes the current dataClay instance aware of another dataClay instance accessible in *host* and *port*, and returns its ID.

Parameters:

host: host where the dataClay instance to be registered is located.
port: port where the dataClay instance to be registered is listening.

Returns:

The ID of the dataClay instance located in *host*, *port*.

```
def federate (self, dc_id, recursive=True):
```

Description:

Federates current object with another dataClay instance, replicating it in any of its backends.

Parameters:

dc_id: ID of the external dataClay. It must be previously registered.
recursive: when this flag is TRUE, all objects (recursively) referenced by the current one will also be federated (except those that are already present in the destination dataClay).

Example: Using federate

```
other_dc = get_dataclay_id(host, port);
p1 = Person.get_by_alias("person1");
# federating object and subobjects to other_dc (previously registered)
p1.federate(other_dc);
```

```
def federate_to_backend (self, backend_id, recursive=True):
```

Description:

Federates current object with another dataClay instance, replicating it in the indicated backend.

Parameters:

backend_id: ID of a backend in an external dataClay instance, which must be previously registered.
recursive: when this flag is TRUE, all objects (recursively) referenced by the current one will also be federated (except those that are already present in the destination dataClay).

```
def get_federation_source (self):
```

Description:

Retrieves the ID of the dataClay instance where the object is federated from.

Returns:

The id of the dataClay instance that is the source of this federated object. It is null if the object is not federated.

```
def get_federation_targets (self):
```

Description:

Retrieves the IDs of all the dataClay instances where the object is federated.

Returns:

A set of DataClayInstanceID objects in which this object is federated. It can be empty if it is not federated.

Example: Using get_federation_targets

```
from dataclay import api
newPerson = Person.get_by_alias('Alias')
dataclays = list(p1.get_federation_of_object())
assert api.LOCAL in dataclays
```

```
def set_in_dataclay_instance (self, dc_id, operation_name, params):
```

Description:

Executes a setter on a particular dataClay where the object is federated.

Parameters:

`dc_id:` dataClay instance where the method must be executed.
`operation_name:` ID of the setter to be executed.
`params:` The parameters of the method.

```
def unfederate (self, [dc_id], recursive=True):
```

Description:

Unfederates current object (and referenced objects) with the indicated dataClay instance. If no *dc_id* is specified, the object is unfederated from all the instances where it lives.

Parameters:

`dc_id:` ID of the external dataClay. It must be previously registered.
`recursive:` when this flag is TRUE, all objects (recursively) referenced by the current one will also be unfederated.

5.10 Further considerations

This section exposes some particularities that are coupled to current dataClay requirements or limitations.

5.10.1 Type annotation

Python uses dynamic typing and does not provide the concept of symbol table, therefore dataClay asks the model provider to explicitly specify the types for class registration.

To this end, fields, methods (argument and return types) and imports must be annotated, as you can notice in the examples of previous section 5.7 about replica management.

In particular, fields and imports are defined as part of the *docstring* of the class, whereas methods are annotated using *decorators*.

In the case of imports, two tags are provided to define imports as shown in the examples below:

```
@dataClayImport numpy as np
@dataClayImportFrom itertools import cycle
```

In the case of fields and methods, they are annotated with tags `@ClassField` and `@dclayMethod` as shown, for instance, in People class from HelloPeople application:

```
class People(DataClayObject):
  """
  @ClassField people list<HelloPeople_ns.classes.Person>
  """
  @dclayMethod()
  def __init__(self):
      self.people = list()

  @dclayMethod(new_person="HelloPeople_ns.classes.Person")
  def add(self, new_person):
      self.people.append(new_person)

  @dclayMethod(return_="str")
  def __str__(self):
      result = ["People:"]

      for p in self.people:
          result.append(" - Name: %s, age: %d" % (p.name, p.age))

      return "\n".join(result)
```

Notice that in case that a class requires types (classes) from your data models (registered or pending to register), you must provide a valid namespace as a prefix for your annotations. If the defined type belongs to a namespace that is pending to register (maybe because you are currently defining it) then you must ensure that the prefix used is the same as the namespace you will provide during the class model registration process.

In the example above, *people* field is specified as a list of **`HelloPeople_ns.classes.Person`** objects being **`HelloPeople_ns`** the namespace to be used for the HelloPeople class model.

5.10.2 Non-registered classes

Non-registered mutable types (such as Python dictionaries or numpy arrays) are opaque to dataClay. Thus, when a registered class has one of such objects (as a field) and this mutable object is modified from outside its containing class, the changes in the mutable object may or may not be reflected.

For example, given a Class A with a field b of type B, and B has a list field. After executing the instruction `self.b.list.add(x)` from a method in A, the list may or may not contain the new element x. To solve this, the class model should define a method in class B containing the instruction `b.list.add(x)` and call it from class A.

5.10.3 Third party libraries

Sometimes using third-party libraries from registered data models is not trivial, thus if you experience such problems, please contact us by email: `support-dataclay@bsc.es`

5.10.4 Execution environment

Multithreading is supported both at the client side and at the server side. However, due to the CPython Global Interpreter Lock implementation details, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you have Python-pure CPU-intensive parallel workloads which are being executed in methods of dataClay persisted objects, then you may experience serialization of executions (and thus, loss of performance).

In Section 7.2.3 we explain how to face this problem, but if you still have some doubts or further requirements for your applications, please contact us (`support-dataclay@bsc.es`) and we will provide you the proper solutions.

IV dataClay management utility

6 dataClay command line utility 69

6.1 Accounts
6.2 Class models
6.3 Data contracts
6.4 Backends

6. dataClay command line utility

In this chapter we present *dataclaycmd*, the dataClay command line utility intended to be used for management operations such as accounting, class registering, or contract creation. The methods of the *dataclaycmd* can be executed through the *run* command of Docker or Singularity. Examples can be found in https://github.com/bsc-dom/dataclay-examples.

6.1 Accounts

In this section we present the options offered by the dataClay command line utility in order to manage user accounts.

`NewAccount  newaccount_name newaccount_pass`

Description:

Registers a new account in the system.

Parameters:

`newaccount_name`: name of the new account to be created.
`newaccount_pass`: password of the new account.

6.2 Class models

In this section we present the options offered by the dataClay command line utility in order to manage classes such as registering a class model or obtaining the corresponding class stubs.

`NewModel`
`user_name user_pass namespace_name class_path language`

Description:

Registers all classes contained in the class path provided. It is assumed that the class path is the main directory of the data model to be registered, and in case of containing subdirectories these represent different packages/modules.
If the namespace where you want to include your class model does not exist, it is created.
One of the supported languages must be chosen to specify which classes of the class path provided will be registered as part of the model.
Notice that, in order to register classes using stubs of other namespaces or just previous registered classes, the corresponding stubs must be contained in the folder so dataClay will be able to annotate the association between classes (e.g. between an already registered and the new one).

Parameters:

`user_name`: user registering the model.
`user_pass`: user's password.
`namespace_name`: namespace where the model will be registered.
`class_path`: class path where classes are registered.
`language`: one of the supported languages (currently, *python* or *java*).

`GetStubs`

```
user_name user_pass namespace_name stubs_path
```

Description:

Retrieves the stubs of a certain class model registered in a namespace.

Parameters:

`user_name`: user requesting the stubs.
`user_pass`: user's password.
`namespace_name`: namespace of the class model to be retrieved.
`stubs_path`: folder where the downloaded stubs will be stored.

`NewNamespace`

```
user_name user_pass namespace_name class_path language
```

Description:

Registers a new namespace in the system.

Parameters:

`user_name`: user registering the namespace.
`user_pass`: user's password.
`namespace_name`: name of the new namespace.
`language`: one of the supported languages (currently, *python* or *java*).

`GetNamespaces`

```
user_name user_pass
```

Description:

Retrieves the namespaces the user has access to.

Parameters:

user_name: user registering the namespace.
user_pass: user's password.

`ImportModelsFromExternalDataClay`
`host port namespace`

Description:

Imports all classes from the indicated namespace of the dataClay instance running in the specified host and port. The external dataClay instance must have been previously registered by means of the method `RegisterDataClay`.

Parameters:

host: host where the dataClay instance from which the models must be imported is located.
port: port where the dataClay instance from which the models must be imported is listening.
namespace: namespace from which the models must be imported.

`RegisterDataClay`
`host port`

Description:

Makes the current dataClay instance aware of another dataClay instance accessible in host and port

Parameters:

host: host where the dataClay instance to be registered is located.
port: port where the dataClay instance to be registered is listening.

6.3 Data contracts

In this section we present the options offered by the dataClay command line utility in order to manage datasets and data contracts.

`NewDataContract`
`user_name user_pass dataset_name beneficiary_name`

Description:

Registers a data contract to grant a user access to a specific private dataset.
Only the owner of the dataset may grant users access to it.
If the dataset does not exist, it is created as a new dataset owned by the user registering this contract.

Parameters:

user_name: user that owns the dataset.
user_pass: user's password.

dataset_name: name of the dataset that will be shared through this contract.
beneficiary_name: user account that will benefit from this contract.

`NewDataset`
`user_name user_pass dataset_name dataset_type`

Description:

Registers a dataset to define its objects to be provided under same contract constraints. If the dataset is private, only the owner of the dataset may grant users access to it through a data contract.

Parameters:

user_name: user that owns the dataset.
user_pass: user's password.
dataset_name: name of the dataset.
dataset_type: either 'public' or 'private'.

`GetDatasets`
`user_name user_pass`

Description:

Retrieves the datasets the user has access to.

Parameters:

user_name: user that owns the dataset.
user_pass: user's password.

6.4 Backends

In this section we present other utilities that can be used to retrieve information from dataClay.

`GetBackends` `user_name user_pass language`

Description:

Retrieves backend names and hosts. Backend name can be used for LocalBackend in config file Section 8.2.

Parameters:

user_name: user requesting backend info.
user_pass: user's password.
language: one of the supported languages (currently, *python* or *java*).

V Installation

7 Deployment 75
7.1 dataClay architecture
7.2 Deployment with containers

8 Configuration 85
8.1 Client libraries
8.2 Configuration files
8.3 Tracing
8.4 Federation with secure communications

7. Deployment

This chapter explains how to perform a dataClay installation. First, we briefly describe the dataClay architecture for the better understanding of its different components and their interactions. Thereafter, we show how to deploy a minimum dataClay installation based on Dockers, and how to extrapolate this basic setup to more complex scenarios.

In case you have some other needs not addressed in this chapter, please contact us by email:

`support-dataclay@bsc.es`

7.1 dataClay architecture

The architecture of dataClay is composed by two main components: the Logic Module and the Data Service. The Logic Module is a central repository that handles object metadata and management information. The Data Service is a distributed object store that handles object persistence and execution requests.

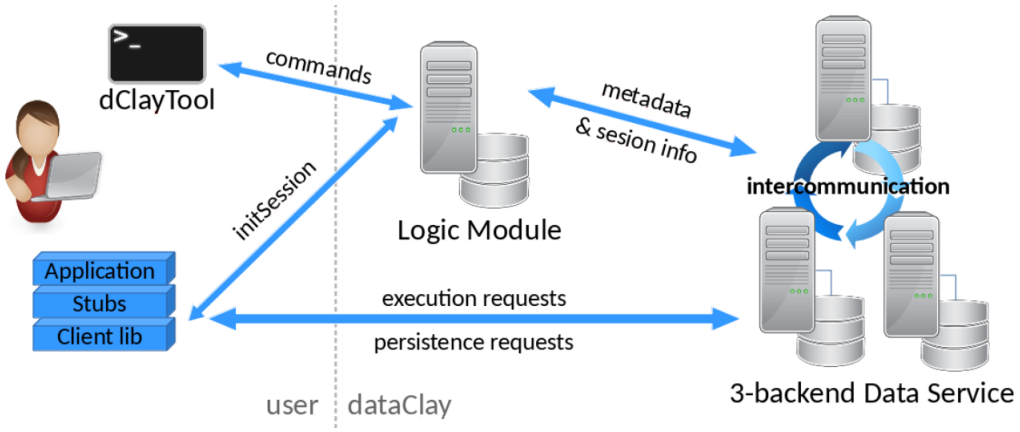


Figure 7.1: dataClay overview

7.1.1 Logic Module

The Logic Module is a unique centralized component that keeps track of every object metadata such as: its unique identifier, (replica) locations, and the dataset it is associated with.

In addition, the Logic Module is in charge of management info, comprising: accounting, namespaces and datasets, permissions (contracts) and registered class models. That is, the information that can be registered in the system using the dataClay command line utility as shown in section 6.

Furthermore, the Logic Module is the entry point for any user application, which must authenticate in the system to create a working session and gain permission to interact with the components of the Data Service.

7.1.2 Data Service

The Data Service is deployed as a set of multiple backends. Any of these backends handles a subset of objects as well as the execution requests aiming to them. This means that every backend has an *Execution Environment* for all supported languages (currently Java and Python) and an associated *Storage System* to handle object persistence. In the case of Python, where multi-threading cannot be managed as Java does, it is possible to deploy multiple *Execution Environments* sharing a single *Storage System* thus enabling applications to exploit parallelism.

In order to enable *Execution Environments* to handle any kind of object and execution request, the Logic Module is in charge to deploy registered classes to every Data Service backend. In this way, every backend can load stored objects as class instances and execute class methods on them corresponding to upcoming execution requests.

This means that when an application initializes a session with dataClay, it first establishes a connection with the Logic Module and obtains information about the available Data Service backends. At this point, the application is enabled to interact with the Data Service backends through stub classes (retrieved with the dataClay command line utility, section 6) by submitting execution requests directly to them.

7.2 Deployment with containers

Keeping in mind the dataClay architecture, hereafter we show how to deploy a dataClay installation based on Docker containers.

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.

In this way, we populate different Docker images corresponding to the main components of the dataClay architecture: the Logic Module and the Data Service. The latter actually comprises one image for each supported language since the corresponding execution requests are handled by separated containers (one per language).

In order to retrieve these images and orchestrate dataClay services properly, following sections show different scenarios based on the standard *docker-compose* tools.

7.2.1 Single node installation

This first dataClay installation assumes that all services will run locally on a single node (e.g. your own laptop).

To this end, we will guide you through the installation process by looking at the details of the following *docker-compose* YAML definition:

```
version: '3.4'
services:
  dcinitializer:
    image: "bscdataclay/initializer"
    command: "--wait-for-python-ds␣1␣--wait-for-java-ds␣1"
    depends_on:
      - logicmodule
      - dsjava
      - dspython
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - USER=${USER:-defaultUser}
      - PASS=${PASS:-defaultPass}
      - DATASET=${DATASET:-defaultDS}
      - GIT_PYTHON_MODELS_URLS=https://gitlab.bsc.es/model.git
      - PYTHON_NAMESPACES=Namespace
    healthcheck:
      interval: 5s
      retries: 10
      test: [ "CMD-SHELL", "/dataclay-initializer/health_check.sh" ]

  logicmodule:
    image: "bscdataclay/logicmodule"
    ports:
      - "11034:11034"
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATACLAY_ADMIN_USER=admin
      - DATACLAY_ADMIN_PASSWORD=admin
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dsjava:
    image: "bscdataclay/dsjava"
    ports:
      - "2127:2127"
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS1
      - DATASERVICE_JAVA_PORT_TCP=2127
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython:
    image: "bscdataclay/dspython"
    ports:
      - "6867:6867"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATASERVICE_PYTHON_PORT_TCP=6867
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

Before starting, the first step is to download the required images executing the following command from the directory where this *docker-compose* file resides:

```
> docker-compose pull
```

At this point, following subsections detail the different parts of the file and which ones can be customized.

Initializer

The *initializer* service facilitates the initialization of a dataClay instance by enabling the user to indicate the classes to be registered, which may reside locally or in a remote repository. For this container, the host and port of the Logic Module must be indicated in the corresponding environment variables `LOGICMODULE_HOST` and `LOGICMODULE_PORT_TCP` (see next subsection), as well as all the information required to register the classes. In particular, the user name and password must be given in variables `USER` and `PASS`. The dataset where objects will be created corresponds to variable `DATASET`, and the namespace where the model will be registered shall be indicated in `JAVA_NAMESPACES` or `PYTHON_NAMESPACES`. Finally, the classes to be registered can be indicated in the following environment variables: `JAVA_MODELS_PATH` or `PYTHON_MODELS_PATH`, if they are stored locally, or in the variables `GIT_JAVA_MODELS_URLS/GIT_PYTHON_MODELS_URLS`, and `GIT_JAVA_MODELS_PATHS/GIT_PYTHON_MODELS_PATHS` if they are available in a GIT repository. In all cases several values can be indicated, separated by spaces.

Alternatively, models can also be imported from other dataClay instances which already have them registered. For this, the following environment variables shall be used:

```
IMPORT_MODELS_FROM_EXTERNAL_DC_HOSTS
IMPORT_MODELS_FROM_EXTERNAL_DC_PORTS
IMPORT_MODELS_FROM_EXTERNAL_DC_NAMESPACES
```

Logic Module

The *logicmodule* service corresponds to the Logic Module. It is possible to customize the default Logic Module port, which is currently set to 11034 through environment variable `LOGICMODULE_PORT_TCP` and is mapped to host in `ports: - "11034:11034"`.

In this way, the Logic Module will publish its service at: `localhost:11034`. In section 8.2 it is described how to define the proper configuration files for user's applications considering this info.

Data Service Backend - Java

The *dsjava* service corresponds to the Java container of the Data Service Backend. Every Data Service backend is tagged with a name so containers for all supported languages can be defined as part of the same backend. This is specially useful to, for instance, define a unique database shared by different execution environments. In this case, the Java container is configured to be part of the Data Service backend called *DS1* as defined via the `DATASERVICE_NAME` variable. Furthermore, it is also necessary to specify the port that will be used to handle Java execution requests: `DATASERVICE_JAVA_PORT_TCP=2127`.

Finally, we also need to specify the address of the Logic Module to enable this Java container to populate its service. To this end, we use the same variables and values as in the Logic Module service (*logicmodule*): `LOGICMODULE_HOST=logicmodule` and `LOGICMODULE_PORT_TCP=11034`.

Data Service Backend - Python

For Python, we only need to attach the container to a Data Service backend with Java support. In this case, we attach the Python container to *DS1* Data Service backend through the `DATASERVICE_NAME` variable. Furthermore, it is also necessary to specify the port that will be used to handle Java execution requests: `DATASERVICE_PYTHON_PORT_TCP=6867`.

Finally, we also need to specify the address of the Logic Module to enable this Python container to populate its service. To this end, we use the same variables and values as in the Logic Module service (*logicmodule*): `LOGICMODULE_HOST=logicmodule` and `LOGICMODULE_PORT_TCP=11034`.

7.2.2 Cluster installation

If you want to deploy dataClay on a cluster of N nodes, you can create different *docker-compose* files for each node depending on the setup you want.

In this section we describe a setup for a 3 node cluster with 1 node for the Logic Module and 2 nodes for Data Service backends. You can easy extrapolate this scenario to more complex ones, but always keeping in mind the following considerations/constraints for the current version of dataClay:

1. The Logic Module is unique in the system. This means that only one of the nodes should have a docker-compose file with the Logic Module section.
2. In this case, all services must be exposed using "host" network mode in order to make them visible and discoverable between different nodes and from the client application.

Node 1 - Logic Module

The docker-compose file for the first node defining the Logic Module. Notice that using host network we do not need to map its port.

```
version: '3.4'
services:
  logicmodule:
    image: "bscdataclay/logicmodule"
    ports:
      - "11034:11034"
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATACLAY_ADMIN_USER=admin
      - DATACLAY_ADMIN_PASSWORD=admin
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]
```

Node 2 - Backend 1

This node runs the Data Service backend *DS1*, as specified through the `DATASERVICE_NAME` variable.

Notice that the only variable that needs to be manually defined is `LOGICMODULE_HOST`, which will be the host name of Node 1 where the Logic Module is deployed.

```
version: '3.4'
services:
 dsjava:
    image: "bscdataclay/dsjava"
    ports:
      - "2127:2127"
```

```
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS1
      - DATASERVICE_JAVA_PORT_TCP=2127
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython:
    image: "bscdataclay/dspython"
    ports:
      - "6867:6867"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATASERVICE_PYTHON_PORT_TCP=6867
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

Node 3 - Backend 2

Analogously to Node 2, this node runs the Data Service backend *DS2*, as specified through the `DATASERVICE_NAME` variable.

Finally, the only variable that needs to be manually defined is `LOGICMODULE_HOST` with the host name of the Node 1 where Logic Module is deployed.

```
version: '3.4'
services:
 dsjava2:
    image: "bscdataclay/dsjava"
    ports:
      - "2128:2128"
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS2
      - DATASERVICE_JAVA_PORT_TCP=2128
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython2:
    image: "bscdataclay/dspython"
    ports:
      - "6868:6868"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS2
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
```

```
      - DATASERVICE_PYTHON_PORT_TCP=6868
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

7.2.3 Enabling Python parallelism

In Section 5.10.4 we explain that the implementation details of the CPython Global Interpreter Lock forces that only one thread can execute Python code at once. However, and as introduced in Section 7.1.2, we can mitigate this problem by configuring dataClay to deploy multiple Python execution environments (backends) on a single node. The example below shows two Python execution environments that will load/store objects from the same Data Service *DS1* ((*dspython1, dspython2*).

```
version: '3.4'
services:
  logicmodule:
    image: "bscdataclay/logicmodule"
    ports:
      - "11034:11034"
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATACLAY_ADMIN_USER=admin
      - DATACLAY_ADMIN_PASSWORD=admin
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dsjava:
    image: "bscdataclay/dsjava"
    ports:
      - "2127:2127"
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS1
      - DATASERVICE_JAVA_PORT_TCP=2127
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython:
    image: "bscdataclay/dspython"
    ports:
      - "6867:6867"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATASERVICE_PYTHON_PORT_TCP=6867
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

```
  dspython2:
    image: "bscdataclay/dspython"
    ports:
      - "6868:6868"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATASERVICE_PYTHON_PORT_TCP=6868
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

7.2.4 Tuning dataClay

dataClay allows to tune some specific settings (detailed in next sections) using either environment variables or a property file located on a specific path.

In the application/client side the default path of this file is `./cfgfiles/global.properties` and can be also defined via the environment variable `DATACLAY_GLOBAL_CONFIG`.

In the server-side the default path is the same, but following previous examples with Docker containers we must define a *volume* to load it. Given the first docker-compose file for a local installation, and assuming that there is a property file located at `./cfgfiles/global.properties` (relative to the docker-compose file), the volume can be mounted in a per-service basis as illustrated below:

```
version: '3.4'
services:
  logicmodule:
    image: "bscdataclay/logicmodule"
    ports:
      - "11034:11034"
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATACLAY_ADMIN_USER=admin
      - DATACLAY_ADMIN_PASSWORD=admin
    volumes:
      - ./prop/global.properties:/usr/src/dataclay/javaclay/cfgfiles/global.properties:ro
      - ./prop/log4j2.xml:/usr/src/dataclay/javaclay/log4j2.xml:ro
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dsjava:
    image: "bscdataclay/dsjava"
    ports:
      - "2127:2127"
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS1
      - DATASERVICE_JAVA_PORT_TCP=2127
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    volumes:
      - ./prop/global.properties:/usr/src/dataclay/javaclay/cfgfiles/global.properties:ro
      - ./prop/log4j2.xml:/usr/src/dataclay/javaclay/log4j2.xml:ro
    healthcheck:
       interval: 5s
```

```
      retries: 10
      test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython:
    image: "bscdataclay/dspython"
    ports:
      - "6867:6867"
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATASERVICE_PYTHON_PORT_TCP=6867
    volumes:
      - ./prop/global.properties:/usr/src/dataclay/pyclay/cfgfiles/global.properties:ro
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

7.2.5 Singularity

dataClay can also be deployed using Singularity, by using the `singularity pull` command (for more information see the Singularity official guide):

```
singularity pull docker://bscdataclay/logicmodule
singularity pull docker://bscdataclay/dsjava
singularity pull docker://bscdataclay/dspython
```

Each container can be orchestrated by using Singularity Compose (e.g. by manually porting the configurations contained in the `docker-compose.yml` example files).

7.2.6 Memory Management and Garbage Collection

In section 1.5 we have introduced that dataClay runs some processes to keep memory and disk usage in a healthy state. On the one hand, flushing objects from memory to disk when memory usage reaches a certain threshold. On the other hand, keeping track of reference counters to detect objects that are no longer accessible so they can be removed from the system.

To control the impact of these processes on the system performance, dataClay provides the administrator with the capability to configure the following parameters via environment variables or the *global.properties* file. Notice that time parameters are always expressed in milliseconds, except stated otherwise.

property	default value	description
MEMMGMT_CHECK_TIME_INTERVAL	5000 (5 seconds)	Periodicity to check memory usage.
MEMMGMT_EASE_FRACTION	0	Minimum fraction for an object to be in memory.
MEMMGMT_MIN_OBJECT_TIME	200	Minimum time for an object to be in memory.
MEMMGMT_PRESSURE_FRACTION	0.7 (70%)	Fraction of memory usage from which to consider that it is under pressure.
GLOBALGC_CHECK_REMOTE_PENDING	2000	Interval to check pending references to notify to other nodes.
GLOBALGC_CHECK_TIME_INTERVAL	86400000 (1 day)	Periodicity to check and collect objects from underlying storage.
GLOBALGC_COLLECT_TIME_INTERVAL	10000	Interval to check objects to clean.
GLOBALGC_COLLECTOR_INITIAL_DELAY_HOURS	0	Garbage collector initial delay in hours.
GLOBALGC_MAX_OBJECTS_TO_COLLECT_ITERATION	1000	Maximum number of objects to clean per iteration.
GLOBALGC_PROCESS_COUNTINGS_INTERVAL	1000	Interval to check pending references to count.

8. Configuration

8.1 Client libraries

In order to connect your applications with dataClay services you need a client library for your preferred programming language.
If you are developing a Java application, you can add the following dependency into your pom file to install the Java client library for dataClay version 2.5:

```
<dependency>
  <groupId>es.bsc.dataclay</groupId>
  <artifactId>dataclay</artifactId>
  <version>2.5.1</version>
</dependency>
```

In case you are developing a Python application, you can easily install the Python module with *pip* command:

```
> pip install dataClay
```

8.2 Configuration files

The basic client configuration for an application is the minimum information required to initialize a session with dataClay. To this end two different files are required: the *session.properties* file and the *client.properties* file.

8.2.1 Session properties

This file contains the basic info to initialize a session with dataClay. It is automatically loaded during the initialization process (`DataClay.init()` in Java or `api.init()` in Python) and its default path is `./cfgfiles/session.properties`. This path can be overridden by setting a

different path through the environment variable DATACLAYSESSIONCONFIG.

Here is an example:

```
Account=MyAccount
Password=MyPassword
StubsClasspath=/home/me/myapp/stubs
DataSetForStore=MyDataset
DataSets=MyDataset,OtherDataSet
LocalBackend=DS1
% DataClayClientConfig=/home/me/myapp/client.properties
```

Account and Password properties are used to specify user's credentials.

StubsClasspath defines a path where the stub classes can be located. That is, the path where the dataClay command line utility (exposed in section 6) saved our stub classes after calling GetStubs operation.

DataSetForStore specifies which dataset the application will use in case a *makePersistent* request is produced to store a new object in the system, and DataSets provide information about the datasets the application will access (normally it includes the DataSetForStore).

LocalBackend defines the default backend that the application will access when using either DataClay.LOCAL in Java or api.LOCAL in Python (examples of this can be found in API sections 4 and 5).

8.2.2 Client properties

This file contains the minimum service info to connect applications with dataClay. It is also loaded automatically during the initialization process and its default path is ./cfgfiles/client.properties, which can be overriden by setting the environment variable DATACLAYCLIENTCONFIG.

Here is an example:

```
HOST=localhost
TCPPORT=11034
```

As you can see, it only requires two properties to be defined: HOST and TCPPORT; comprising the full address to be resolved in order to initialize a session with dataClay from your application.

8.3 Tracing

dataClay provides a built-in tracing system to generate tracefiles of an application execution. This is achieved using Extrae (https://tools.bsc.es/extrae).

For each service, Extrae keeps track of the events in an intermediate file (with .mpit extension). At the end of the execution, all intermediate files are gathered and merged by Extrae in order to create the final trace, encoded in a Paraver file (.prv) (https://tools.bsc.es/paraver)

In order to enable Extrae tracing in dataClay, the application must activate it. We must write Tracing=True in the session.properties file:

```
Account=MyAccount
Password=MyPassword
StubsClasspath=/home/me/myapp/stubs
DataSetForStore=MyDataset
DataSets=MyDataset,OtherDataSet
```

```
LocalBackend=DS1
Tracing=True
```

Additionally, we need to modify dataClay's `docker-compose.yml` to add `-tracing` command:

```
version: '3.4'
services:
  logicmodule:
    image: "bscdataclay/logicmodule"
    command: --tracing
    ports:
      - "11034:11034"
    environment:
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
      - DATACLAY_ADMIN_USER=admin
      - DATACLAY_ADMIN_PASSWORD=admin
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dsjava:
    image: "bscdataclay/dsjava"
    command: --tracing
    ports:
      - "2127:2127"
    depends_on:
      - logicmodule
    environment:
      - DATASERVICE_NAME=DS1
      - DATASERVICE_JAVA_PORT_TCP=2127
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/javaclay/health_check.sh"]

  dspython:
    image: "bscdataclay/dspython"
    command: --tracing
    depends_on:
      - logicmodule
      - dsjava
    environment:
      - DATASERVICE_NAME=DS1
      - LOGICMODULE_PORT_TCP=11034
      - LOGICMODULE_HOST=logicmodule
    healthcheck:
       interval: 5s
       retries: 10
       test: ["CMD-SHELL", "/usr/src/dataclay/pyclay/health_check.sh"]
```

Now we can start dataClay and run our application with tracing. Once finished, traces will be generated and stored in `‘pwd‘/traces` directory. Those traces are ready to be used by Paraver (https://tools.bsc.es/paraver)

dataClay Extrae traces can be used together with COMPSs (https://compss.bsc.es). Each node/service has an Extrae task ID defined. This task ID is used to define different threads and lines in Paraver visualization. It means that in COMPSs you will have defined task IDs for master and workers (task ID = 0 for master, task ID = 1 for first worker, task ID = 2 for second worker, . . .).

dataClay needs to use the first available task ID which is `task ID = COMPSs workers + 1`. The session.properties file must be modified by adding the option `ExtraeStartingTaskID=taskID`

with the appropriate *taskID*.

```
Account=MyAccount
Password=MyPassword
StubsClasspath=/home/me/myapp/stubs
DataSetForStore=MyDataset
DataSets=MyDataset,OtherDataSet
LocalBackend=DS1
Tracing=True
ExtraeStartingTaskID=9
```

Once the application is finished, traces will be generated and stored in ‘pwd‘/traces directory.
The versions currently supported are Extrae 3.5.4 and COMPSs 2.6.

8.4 Federation with secure communications

In this section we explain how to secure dataClay communications between different dataClay instances.

In a federated environment, different dataClays are communicating to each other via the LogicModule service.

The current implementation of dataClay provides support to client certificates. Thus, we use a Traefik reverse-proxy https://docs.traefik.io/ to check client certificates, and also to avoid publishing dataClay ports.

An example of the *docker-compose.yml* file with reverse-proxy is as follows:

```
version: '3.4'

volumes:
  dataclay-certs:
    driver: local

services:

  certificates_initializer:
    image: "dataclaydemos/certificate-initializer"
    build:
      context: .
      dockerfile: cert.Dockerfile
    environment:
      - CERTIFICATE_AUTHORITY_HOST=${CERTIFICATE_AUTHORITY_HOST}
    volumes:
      - dataclay-certs:/ssl/:rw
    healthcheck:
      test: bash -c "[ -f /ssl/dataclay-agent.crt ]"
      timeout: 1s
      retries: 20

  proxy:
    image: traefik:v1.7.17
    depends_on:
      - certificates_initializer
    restart: unless-stopped
    command: --docker --docker.exposedByDefault=false
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - /home/docker/dataclay/traefik.toml:/traefik.toml
      - dataclay-certs:/ssl:ro
    ports:
      - "80:80"
```

```
    - "443:443"


logicmodule:
  image: "bscdataclay/logicmodule:develop.jdk11-alpine"
  command: "--debug"
  depends_on:
    - proxy
  environment:
    - LOGICMODULE_HOST=logicmodule
    - LOGICMODULE_PORT_TCP=11034
    - DATACLAY_ADMIN_USER=admin
    - DATACLAY_ADMIN_PASSWORD=admin
    - LM_SERVICE_ALIAS_HEADERMSG=logicmodule
    - SSL_CLIENT_TRUSTED_CERTIFICATES=/ssl/dataclay-ca.crt
    - SSL_CLIENT_CERTIFICATE=/ssl/dataclay-agent.crt
    - SSL_CLIENT_KEY=/ssl/dataclay-agent.pem
  volumes:
    - dataclay-certs:/ssl/:ro
  labels:
    - "traefik.enable=true"
    - "traefik.backend=logicmodule"
    - "traefik.frontend.rule=Headers:␣service-alias,logicmodule"
    - "traefik.port=11034"
    - "traefik.protocol=h2c"
  stop_grace_period: 5m
  healthcheck:
    interval: 5s
    retries: 10
    test: ["CMD-SHELL", "/home/dataclayusr/dataclay/health/health_check.sh"]


dsjava:
  image: "bscdataclay/dsjava:develop.jdk11-alpine"
  depends_on:
    - logicmodule
    - proxy
  environment:
    - DATASERVICE_NAME=DS1
    - DATASERVICE_JAVA_PORT_TCP=2127
    - LOGICMODULE_PORT_TCP=11034
    - LOGICMODULE_HOST=logicmodule
  stop_grace_period: 5m
  healthcheck:
    interval: 5s
    retries: 10
    test: ["CMD-SHELL", "/home/dataclayusr/dataclay/health/health_check.sh"]

dspython:
  image: "bscdataclay/dspython:develop-alpine"
  command: "--debug"
  depends_on:
    - logicmodule
    - dsjava
  environment:
    - DATASERVICE_NAME=DS1
    - LOGICMODULE_PORT_TCP=11034
    - LOGICMODULE_HOST=logicmodule
    - DATASERVICE_PYTHON_PORT_TCP=6867
    - DEBUG=True
  stop_grace_period: 5m
  healthcheck:
     interval: 5s
     retries: 10
     test: ["CMD-SHELL", "/home/dataclayusr/dataclay/health/health_check.sh"]
```

With the following *traefik.toml* example:

```
debug = false
defaultEntryPoints = ["http", "https"]
[entryPoints]
    [entryPoints.http]
    address = ":80"
        [entryPoints.http.redirect]
        entryPoint = "https"

   [entryPoints.https]
     address = ":443"
   [entryPoints.https.tls]
     [entryPoints.https.tls.clientCA]
     files = ["/ssl/dataclay-ca.crt"]
     optional = false

         [entryPoints.https.tls.defaultCertificate]
     certFile = "/ssl/dataclay-agent.crt"
     keyFile = "/ssl/dataclay-agent.pem"

     # For secure connection on frontend.local
     [[entryPoints.https.tls.certificates]]
     certFile = "/ssl/dataclay-agent.crt"
     keyFile = "/ssl/dataclay-agent.pem"
```

Note that ports are not published in *docker-compose.yml* and we configure Traefik by adding labels to the *logicmodule* service.

Note that we configured our application to use the certificates via the following environment variables in *logimodule*:

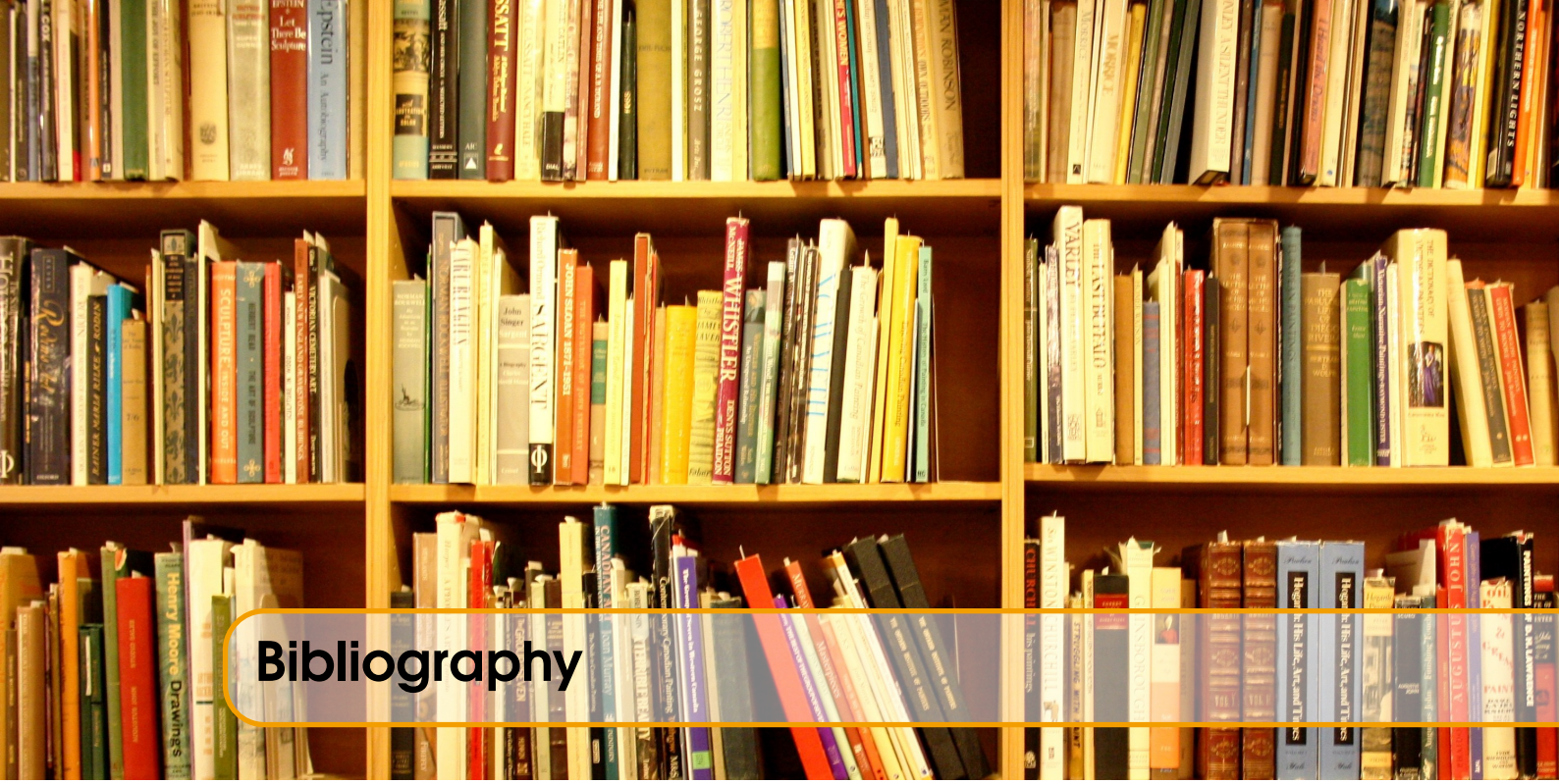
property	**default value**	**description**
LM_SERVICE_ALIAS_HEADERMSG	logicmodule	Add to the message the header service-alias (used to filter in traefik) .
SSL_TARGET_AUTHORITY	proxy	Override target authority (usually traefik service name).
SSL_CLIENT_TRUSTED_CERTIFICATES	None	Path to CA certificate.
SSL_CLIENT_CERTIFICATE	None	Path to Client certificate.
SSL_CLIENT_KEY	None	Path to Client key.

Alternatively, this values could be provided in the *global.properties* file.

VI Bibliography and index

Bibliography 93

Index 95



Bibliography

Index

account, 24, 69
account creation, 24
alias, 11, 32, 34, 35, 52–54
api.init(), 85
application cycle, 23
application developer, 12

backend, 11, 76

class, 69
class registration, 24
class stub, 25
client, 11
client.properties, 25, 85
contract, 71
contract, data, 24

data contract, 24
data model, 12
data service, 75, 76
dataClay application, 11
dataClay cmd, 69
dataClay object, 11
DataClay.init(), 25, 85
dataset, 12, 24
dc_clone, 52
dc_clone_by_alias, 51
dc_put, 52
dc_update, 53
dc_update_by_alias, 51
dcClone, 32
dcCloneByAlias, 31
dcPut, 32
dcUpdate, 33
dcUpdateByAlias, 31
delete_alias, 53
deleteAlias, 34
docker, 76
docker-compose, 76
dockers, 75

error management, 38, 57
execution model, 12

federate, 43, 63
federate_to_backend, 63
federateToBackend, 44
federation, 13, 40, 60
finish, 29, 49

garbage collection, 12, 38, 57, 83
get_all_locations, 55
get_backends, 50
get_by_alias, 54
get_dataclay_id, 62
get_federation_source, 64
get_federation_targets, 64
get_location, 55
getAllLocations, 36
GetBackends, 72
getBackends, 30
getByAlias, 34

getDataClayID, 43
GetDatasets, 72
getFederationSource, 44
getFederationTargets, 44
getLocation, 36
GetNamespaces, 70
GetStubs, 70

HelloPeople, 15

ImportModelsFromExternalDataClay, 71
init, 30, 50
installation, 75

logic module, 75, 76

make_persistent, 54
makePersistent, 25, 35
management operations, 25
memory management, 12, 38, 57, 83
model provider, 12

namespace, 12, 24
new_replica, 56
NewAccount, 69
NewDataContract, 71
NewDataset, 72
NewModel, 69
NewNamespace, 70
newReplica, 36
nvm, 59

object, 11
Object Store, 30, 50

register_dataclay, 63
RegisterDataClay, 71
registerDataClay, 43
replica management, 38, 58
roles, 12
run_remote, 56
runRemote, 37

session.properties, 25, 85
session_detach, 57
sessionDetach, 38
set_in_dataclay_instance, 64
setInDataClayInstance, 45
stub, 25
stub class, 25

unfederate, 45, 64