Welcome to the Fénix Framework project

Fénix Framework allows the development of Java-based applications that need a **transactional and persistent domain model**. Even though it was originally created to support the development of web applications, it may be used to develop any other kind of application that needs to keep a persistent set of data.

One of the major design goals underlying the development of the Fénix Framework is that it should provide a **natural programming model** to programmers used to develop plain Java programs without persistence. Unfortunately, the addition of persistence, typically backed up by a relational database management system, interferes with the normal coding patterns of object-oriented programming, because the need to access the database efficiently precludes the use of many of the powerful mechanisms available in the object-oriented paradigm.

Other frameworks and tools, often called Object/Relational Mappers, address this problem by giving the programmer ways to specify how objects are mapped into the relational database and by doing most of the heavy-lifting needed to access the data in the database. Yet, they fail to hide completely the presence of the database, meaning that the problem remains.

With the Fénix Framework, on the other hand, the database is completely hidden from the programmer. This has two consequences: (1) the programmer cannot control the mapping of objects to the database, and (2) there is no way to take advantage of database facilities such as joins or aggregate functions. In return, programmers may use, and are encouraged to do it, all of the normal object-oriented programming coding patterns.

Programmers using the Fénix Framework specify the structure of the application's domain model using a new domain-specific language created to that effect, the Domain Modeling Language (DML), and then develop the rest of the application in plain Java.

Quick start

To create a sample application pre-configured to work with the Fénix Framework, execute the following Maven command and follow the interactive mode:

```
mvn archetype:generate -DarchetypeGroupId=pt.ist
-DarchetypeArtifactId=fenix-framework-application-archetype-clean
-DarchetypeVersion=2.0
-DarchetypeRepository=https://fenix-ashes.ist.utl.pt/maven-public
```

The parameters that you can give to the clean application archetype (using the `-D` switch) are:

- `fenixFrameworkVersion`:
  - Description: The version of the Fenix Framework to use
  - Default: Same as the archetype's version.
- `backEndName`
  - Description: The name of the backend to use. Some possible values are `mem`, `ispn` and `ogm`.
  - Default: `mem`
- `backEndGroupId`
  - Description: The groupId of the backend module to use.
  - Default: `pt.ist`
- `backEndArtifactId`
  - Description: The artifactId of the backend module to use.
  - Default: `fenix-framework-backend-mem`
- `backEndVersion`
  - Description: The version of the backend module to use.
  - Default: Same as the archetype's version.
- `codeGeneratorClassName`
  - Description: The code generator class provided by the chosen backend. Some possible values for each backend are:
    - `mem` -> `pt.ist.fenixframework.backend.mem.MemCodeGenerator`
    - `ogm` -> `pt.ist.fenixframework.backend.ogm.OgmCodeGenerator`
    - `ispn` -> `pt.ist.fenixframework.backend.infinispan.InfinispanCodeGenerator`
  - Default: `pt.ist.fenixframework.backend.mem.MemCodeGenerator`

Hello world example

As a very simple example to illustrate the use of the Fénix Framework, let us create a Java application with a minimalist domain model.

In this application we model one new entity: The `Person`. We keep a collection of `people` by establishing a relation

between `Person` and `DomainRoot`. The `DomainRoot` is a built-in entity that represents a possible root for the domain objects' model. When the framework initializes, it automatically creates one instance of the DomainRoot (if it doesn't already exist). The Person type will have one instance for each person known.

Whenever the application is run we pass it the name of a new person and the application adds that person to its set of known people and asks each one of them to say "Hello".

Let us start by creating the basic application structure with:

```
mvn archetype:generate -DarchetypeGroupId=pt.ist
-DarchetypeArtifactId=fenix-framework-application-archetype-clean
-DarchetypeVersion=2.0
-DarchetypeRepository=https://fenix-ashes.ist.utl.pt/maven-public
-DgroupId=example -DartifactId=helloworld -Dversion=1.0-SNAPSHOT
```

Then, to define the domain model, edit the file in `src/main/dml/helloworld.dml` and write the following content:

```
package example;

class Person {
    String name;
}

relation KnownPeople {
    .pt.ist.fenixframework.DomainRoot playsRole root;
    Person playsRole people { multiplicity *; }
}
```

This defines the `Person` entity and adds a relation between `DomainRoot` and `Person`, where a `DomainRoot` may have any number of `Person` related to it. The package declaration is similar to the corresponding declaration in Java.

Now, this DML code takes care of defining the structural aspects of the domain model, but has no behavior in it. For instance, we would like that each `Person` could say hello. Also, in our example, we will be instantiating a `Person` with a constructor that receives the name of the person and the instance of the `DomainRoot`.

To implement the behavior, we write it in plain old Java. So, for the `Person`, we may write (in Java, now):

```
package example;

import pt.ist.fenixframework.DomainRoot;

public class Person extends Person_Base {

    public Person(String name, DomainRoot root) {
        setName(name);
        setRoot(root);
    }

    public void sayHello() {
        System.out.println("Hello, I'm " + getName() + ".");
    }
}
```

The `Person` class extends the `Person_Base` class, which is an abstract class generated by the Fénix Framework that implements the structural aspects of the `Person` entity (likewise for any other entity defined in a DML file).

To conclude this example, we just have to write our `HelloWorld` application, which, as mentioned, will create the new people and then, ask all of them to say hello. This can be done with:

```
package example;

import pt.ist.fenixframework.Atomic;
import pt.ist.fenixframework.DomainRoot;
import pt.ist.fenixframework.FenixFramework;

public class HelloWorld {

    // FenixFramework will try automatic initialization when first accessed
    public static void main(String [] args) {
        try {
            addNewPeople(args);
            greetAll();
        } finally {
            // ensure an orderly shutdown
```

```
            FenixFramework.shutdown();
        }
    }

    @Atomic
    private static void addNewPeople(String[] args) {
        DomainRoot root = FenixFramework.getDomainRoot();
        for (String name : args) {
            new Person(name, root);
        }
    }

    @Atomic
    private static void greetAll() {
        DomainRoot root = FenixFramework.getDomainRoot();
        for (Person p : root.getPeople()) {
            p.sayHello();
        }
    }
}
```

Notice the use of the `@Atomic` annotation to demarcate the transactions. Operations involving domain objects must always be performed inside a transaction.

And that's all. Now, you just have to build it and run it. The complete example is available for download here. After decompressing it, you can execute it with:

```
mvn clean package exec:java -Dexec.mainClass="example.HelloWorld" -Dexec.args="John"
```

By default, the Fénix Framework is configured to use the in-memory backend, so, in the previous example, the application state is not persisted between executions. Here is the same example, now using the Infinispan backend to store the domain objects persistently.

How to use Fénix Framework

Introduction

The Fénix Framework is useful for applications that need to have a transactional and persistent domain model.

Typically, a domain model is programmed in Java by using Java classes to implement the domain model's entities. Relationships between entities, however, do not map that easily into Java constructs. Instead, they are typically implemented in each of the participating entities' classes, either as references to other objects, or as collections of objects, or both.

Moreover, classes corresponding to domain model's entities have other requirements that are not common to classes implementing other types of objects in the application. For instance, their objects need to be persistent and are typically shared among many concurrent threads. So, these classes need to be implemented specially, taking these requirements in consideration.

Given the special nature and needs of the domain model, in the Fénix Framework the domain model is defined using a new language that was specifically created to allow the definition of the structural aspects of a domain model. This language is the DML (Domain Modeling Language). A domain model defined in the DML is then compiled into the corresponding Java classes that correctly implement that domain model structure in such a way that allows the programmers to further define the entities' behaviors in plain Java.

The Fénix Framework stores the application's entities in a storage that depends on the selected Backend. It does that automatically and transparently to the programmer.

So, in a nutshell, to use the Fénix Framework programmers have to do the following:

1. Define the structure of the domain model in DML;
2. Write the rest of the program in plain old Java;
3. Use the `@Atomic` annotation on the methods that should execute atomically;
4. Somewhere in the Java code, initialize the framework. Since version 2.0, the framework attempts automatic initialization when first accessed (read more);
5. Configure your storage. This will be used to store the application's data;
6. Build and run the application.

Have a look at the [Quick Start page][QuickStart] for a guide to create your first Fénix Framework-based application. In the following we describe how to integrate the Fénix Framework in your application development cycle.

Build and run your application

The Fénix Framework is provided as a set of JAR files, which should be made available to the application both during compile and runtime. At compile time, the Fénix Framework is required to generate the backend-specific code of the domain entities. At runtime it is used both by the generated code and by the application code. Here we describe the general procedure to setup the Fénix Framework to develop an application.

The framework is developed using Maven, so if you use Maven to build your application, you can just depend on the framework's artifacts that you need, by adding them to your `pom.xml`:

```
<dependencies>
    <!-- add dependencies for the desired backends -->
    <dependency>
        <groupId>pt.ist</groupId>
        <artifactId>fenix-framework-backend-infinispan</artifactId>
        <version>2.0-cloudtm</version>
    </dependency>
</dependencies>
```

These artifacts are available via the Fénix Framework Nexus repository, so you need to add it to your configuration:

```
<pluginRepositories>
    <pluginRepository>
        <id>fenix-ashes-maven-repository</id>
        <url>https://fenix-ashes.ist.utl.pt/maven-public</url>
    </pluginRepository>
</pluginRepositories>

<pluginRepositories>
    <repository>
        <id>fenix-ashes-maven-repository</id>
        <url>https://fenix-ashes.ist.utl.pt/maven-public</url>
    </repository>
</repositories>
```

Additionally, you will probably want to hook the `dml-maven-plugin` to your build process, so that your domain classes get properly generated and post-processed. This can be achieved by adding the plugin to the build phase.

```
<build>
    <plugins>
        <plugin>
            <groupId>pt.ist</groupId>
            <artifactId>dml-maven-plugin</artifactId>
            <version>${fenixframework.version}</version>
            <configuration>
                <codeGeneratorClassName>${fenixframework.code.generator}</codeGeneratorClassName>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>generate-domain</goal>
                        <goal>post-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The `fenixframework.code.generator` property, shown in the previous listing, could be set in your properties section of the POM file, as per the following example:

```
<properties>
    <fenixframework.code.generator>
        pt.ist.fenixframework.backend.infinispan.InfinispanCodeGenerator
    </fenixframework.code.generator>
    <!-- alternative value could be pt.ist.fenixframework.backend.ogm.OgmCodeGenerator -->
</properties>
```

Just make sure that `fenixframework.version` property is set in accordance with the version used for the other Fénix Framework modules. This ensures that you use a plugin that matches the version of the framework you are using.

The steps described above are all that is necessary to be able to develop using the Fénix Framework. The Maven build system will automatically download the required artifacts and the plugins will hook to the correct phases of your build.

Additionally, you may opt to compile the framework from source. It can be downloaded from GitHub and packaged with:

```
git clone git://github.com/fenix-framework/fenix-framework.git
cd fenix-framework
mvn package
```

The previous sequence produces the artifacts, i.e. one JAR file for each Fénix Framework sub-module. To use the Fénix Framework in your application, these packaged JAR files are required. They can be installed to the local Maven repository with:

```
mvn install
```

If you use any system other than Maven, first make sure that the jars resulting from the execution of `mvn package` are visible in your application's build and runtime classpaths, and then check for any additional dependencies. You can check for dependencies using the Maven dependency plugin (even if you don't use Maven in your application):

```
mvn dependency:list
```

At this point your application should be set up correctly to integrate the Fénix Framework in its build dependencies. **If you are not using Maven** to build you application, then you need to take care of two additional steps. The first is to run the DML Compiler (Java program `pt.ist.fenixframework.DmlCompiler`) to generate the source base classes before compiling your own code, and the second is to run the post-processor (Java program `pt.ist.fenixframework.core.FullPostProcessDomainClasses`) on your compiled classes. Both tools come available with the Fénix Framework code in the `fenix-framework-dml-compiler` sub-module.

Finally, when deploying your application ensure that Fénix Framework and all of its dependencies are available in the `CLASSPATH`.

Domain Modelling Language (DML)

The Domain Modeling Language (DML) is a micro-language designed specifically to implement the structure of a domain model. It has constructs for specifying both entity types and associations between entity types.

DML has a Java-like syntax to be easy to learn by Java programmers.

This page will document the language, but until then, you may find basic documentation in this excerpt of the PhD thesis that introduced the DML. Also, more advanced (and newly added) aspects of the language are available in this doc.

Please note that the syntax for the association declaration has changed with regard to the text in the thesis: The keyword `relation` is used instead of the keyword `association`.

Built-in value types in the Fénix Framework

The built-in value types (meaning that you can use them without declaring them) are the following:

- The Java primitive types: `boolean`, `byte`, `char`, `short`, `int`, `float`, `long`, and `double`.
- The wrapper types for primitive Java types: `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Float`, `Long`, and `Double`. These types are typically used when an entity's field is optional (because it may be `null` in this case).
- The `String` type.
- The `Serializable` type.
- The `bytearray` type (it will map into the Java type `byte[]`).
- To represent dates and related types, there are the following built-in value-types, from the JodaTime(http://joda-time.sourceforge.net/) API: `DateTime`, `LocalDate`, `LocalTime`, and `Partial`.

Transaction Support

The Fénix Framework provides transaction support to applications through three mechanisms:

- standard *begin*, *commit* and *rollback* operations common to any transactional system;
- Atomic invocation of a *Callable*;
- Atomic invocation of methods that use the `@Atomic` annotation.

Low-level mechanism for transaction management

The `pt.ist.fenixframework.TransactionManager` interface provides the low-level API for managing a transaction. Using this low-level API the programmer has a fine-grained control over the transaction, but at the expense of added complexity.

The framework's `TransactionManager` extends the `javax.transaction.TransactionManager` interface, which means that any application already using the Java Transactions API can be easily ported to use the Fénix Framework.

Starting a new transaction requires the invocation of the `begin()` method. The transaction created will be active, within the current thread, until either a commit or a rollback occurs.

When the programmer wants to finish the transaction, making any changes available to others he must invoke the `commit()` method on the `TransactionManager`. This method will commit any changes performed on the active transaction, within the current thread, and close the transaction.

If the programmer sees fit to abnormally terminate the current transaction, he can do so through the `rollback()` method. This method will close the transaction, discarding any changes that may exist.

Here's an example code:

```
import pt.ist.fenixframework.FenixFramework;
import pt.ist.fenixframework.TransactionManager;

public class SomeClass {
    // ...

    void someMethod() {
        // non-transactional code

        TransactionManager tm = FenixFramework.getTransactionManager();
        tm.begin();
        boolean txcommitted = false;
        try {
            // transactional code
            tm.commit();
            txcommitted = true;
        } finally {
            if (! txcommitted) {
                tm.rollback();
            }
        }

        // non-transactional code
    }

    // ...
}
```

Transactional commands

Another way to provide transaction support is to program the behaviour that needs to be transactional in the call() method of a `java.util.concurrent.Callable`.

```
import java.util.concurrent.Callable;
import pt.ist.fenixframework.TransactionManager;

public class SomeClass {
    // ...

    void someMethod() {
        // non-transactional code

        FenixFramework.getTransactionManager().withTransaction(new Callable() {
                @Override
                public Object call() {
                    // transactional code
                }
            });

        // non-transactional code
    }

    // ...
}
```

@Atomic annotation

The easiest way to execute a method in its own transaction is to simply annotate it with the `@Atomic` annotation. During the post-compile phase the annotated methods are surrounded by transaction management code using the lower-level

APIs.

```
import pt.ist.fenixframework.Atomic;

public class SomeClass {
    //...

    @Atomic
    void atomicMethod() {
        // transactional code
    }

    // ...
}
```

BackEnds

Regardless of which backend the programmer uses, the Fénix Framework provides automatic management of bidirectional relations between domain entities modelled in DML (for example, when adding an object to a collection, both ends are updated).

Bundled with Fénix Framework 2.0, there are currently two backend implementations: The *Infinispan Direct backend*, and the *OGM backend*. Both backends provide bindings to the underlying storage provided by Infinispan's cache. However, each backend provides a different mapping between the objects and their key-value representation.

Infinispan Direct backend

The Infinispan Direct backend leverages on the fact that Infinispan's cache is already build into the application's memory space, and thus uses very lightweight domain objects. These objects' attributes and relations to other objects don't take up any additional memory space. Whenever the value of any attribute or relation is requested, the generated code fetches the value directly from the corresponding cache entry in Infinispan. As such, the objects themselves, are *hollow*, in the sense that they only contain their primary key attribute.

OGM Backend

The OGM backend relies on Hibernate Core's engine, stacked with the Hibernate Object/Grid Mapper to do the actual load/store operations from/to Infinispan. As such, the code generation for domain entities creates typical *POJO*s together with a standard Java Persistence API mapping.

STM-based backends

The first implementations of the Fénix Framework (up to version 2.0) have no notion of replaceable backend, and support only an STM-based transactional system (using JVSTM). The main development branch supported only data storage on top of MySQL using OJB as data mapper. There was, however, parallel work that supported some NoSQL storage systems (HBase, Berkeley DB, Infinispan) to a limited extent. Namely, on top of NoSQL databases, Fénix Framework only supports a single node, whereas on top of MySQL it can run multiple application servers.

We are currently working to develop and STM-based backend on the new architecture. We plan to provide storage support for at least MySQL and Infinispan.