Chapter 1
Metadata-Based Frameworks in the Context of Cloud Computing

Eduardo Martins Guerra and Ednelson Oliveira

Abstract In the context of cloud computing, one server is usually responsible to run multiple applications and a single application is spread across multiple servers. On the one hand, the applications need to be able to determine how the cloud environment should handle its execution, or even the execution of each one of its components. Yet, on the other hand, the applications should be decoupled from the middleware that executes them, enabling each one to evolve independently. Based on this scenario, it is possible to state that metadata-based frameworks are a suitable option for the interaction between the application and the services provided by the cloud, since it decouples the application from the environment and allows a transparent individual configuration of each class. The goal of this chapter is to describe the essence of metadata-based frameworks and how they can be applied to cloud computing. It brings several examples of cloud computing frameworks and describes some design practices for the framework structure, scenarios that are suitable for the metadata-based approach and best practices for metadata configuration. As a result, after reading this chapter, the reader should be able to understand the basic functioning of a metadata-based framework and why it is suitable for cloud applications.

Keywords Metadata • Metadata-based frameworks • Cloud-based services • Cloud patterns

E.M. Guerra (✉)
Laboratory of Computing and Applied Mathematics (LAC),
National Institute for Space Research (INPE), P. O. Box 515 – 12227-010,
São José dos Campos, SP, Brazil
e-mail: eduardo.guerra@inpe.br

E. Oliveira
Department of Computer Science, Aeronautical Institute of Technology (ITA),
Praça Marechal Eduardo Gomes, 50 Vila das Acassias,
São José dos Campos, São Paulo, Brazil
e-mail: ednelsonoliveira@gmail.com

1.1 Introduction

Cloud computing applications are executed across multiple servers, and this fact should be transparent to the developers, especially when the platform is provided as a service. Additionally, the application should interact with some cloud services, such as persistence and session management. To allow this transparency, the application should be decoupled from the cloud environment services. Since servers also execute many applications, another important requirement is to enable each application, or even each component, to configure how it should be handled by the cloud. Combining these two requirements, it is possible to conclude that the cloud provider should provide frameworks that, at the same time, abstract the cloud environment and allow a flexible configuration of how each application class should be handled.

Many frameworks for developing cloud applications, such as Gaelyk [5], Objectify [6], Play Framework [7] and Guice [8], use metadata to allow a fine-grained configuration of the application classes in the cloud environment. Gaelyk is a Groovy toolkit for web application development for Google App Engine (GAE) [5], which makes a classic use of metadata to map classes to persistent entities. Objectify implements the same mapping, but covering all features of the Google App Engine Datastore, using metadata also to define entities, relationships and listeners.

Another very interesting use for metadata can be found in Play Framework [7] where a web framework is deployed in GAE. It uses an annotation to schedule asynchronous jobs, which can run periodically or in an instant defined by expressions. Google Guice [8] uses metadata to inject dependencies from cloud services into application classes. It allows implementation classes to be programmatically bound to an interface and then injected into constructors, methods or fields using the @Inject annotation.

Metadata-based frameworks can be defined as frameworks that consume custom metadata from application classes to customize its behaviour [1]. Common approaches for metadata definition are XML documents, code annotations, code conventions and even databases. From the developer's perspective, the focus is on declarative metadata definition and not on method invocation or class extension. Recent studies reveal that one benefit in this approach is the decoupling between framework and application compared to other techniques [2]. This characteristic makes this kind of solution suitable for cloud application, where it is desirable to decouple the business rules from the infrastructure.

The goal of this chapter is to explore the characteristics of metadata-based frameworks and how they can be used for the development of cloud applications, considering the concept of platform as a service. Examples of real cloud frameworks are used to illustrate the practices presented. Additionally, it also presents some patterns that can be used to structure internally this kind of framework, architectural scenarios where this approach is used in cloud applications and best practices for metadata schema definition and metadata configuration. In brief, this chapter presents a complete set of practices considering distinct perspectives about metadata-based frameworks for cloud applications.

1.2 Frameworks and Metadata

To understand how a metadata-based framework can be useful on a cloud environment, it is important to understand the concepts about frameworks and metadata definition. That knowledge is important to understand how frameworks can be internally structured to allow behaviour specialization and extension using classic object-oriented techniques and metadata. This section also explores the alternatives on metadata definition and the basic functioning of a metadata-based framework.

1.2.1 Framework Concepts

A framework can be considered an incomplete software with some points that can be specialized to add application-specific behaviour, consisting in a set of classes that represents an abstract design for a family of related problems. It is more than well-written class libraries, which are more application independent and provide functionality that can be directly invoked. A framework provides a set of abstract classes that must be extended and composed with others to create a concrete and executable application. Those classes can be application-specific or taken from a class library, usually provided along with the framework [9].

The main purpose of a framework is to provide reuse in the application, but in a larger granularity than a class. This reuse of the design provided by a framework is defined by its internal interfaces and the way that the functions are divided among its components. It can be considered more important than the source code reuse. According to Jacobsen and Nowack [10], the reuse in a framework is performed in three levels: analysis, design and implementation. The flexibility which makes possible the application behaviour specialization is important to enable its usage in multiple contexts.

Another important characteristic of a framework is the inversion of control [11]. Framework runtime architecture enables the definition of processing steps that can call applications handlers. This allows the framework to determine which set of application methods should be called in response to an external event. The common execution flow is an application to invoke the functionality on an external piece of software and not the opposite. On the other hand, using the inversion of control approach, the framework, and not the application, is responsible for the main execution flow. This is also known as the Hollywood Principle [12]: *Don't call us, we'll call you.*

A framework can contain points, called hot spots, where applications can customize their behaviour [13]. Each type of behaviour which can be customized in a framework is called variability, and they represent domain pieces that can change among applications. Points that cannot be changed are called frozen spots. Those points usually define the framework general architecture, which consists in its basic components and the relationships between them. This section presents the two different types of hot spots that respectively use inheritance and composition to enable the application to add behaviour. New approaches in framework development can use other kinds of hot spots, such as reflective method invocation [14] and metadata definition [1].

1.2.2 Metadata Definition

Metadata is an overloaded term in computer science and can be interpreted differently according to the context. In the context of object-oriented programming, metadata is information about the program structure itself such as classes, methods and attributes. A class, for example, has intrinsic metadata like its name, its superclass, its interfaces, its methods and its attributes. In metadata-based frameworks, the developer also must define some additional application-specific or domain-specific metadata.

Even in this context, metadata can be used for many purposes. There are several examples of this, such as source code generation [15], compile-time verifications [16, 17] and class transformation [18]. The metadata-based components consume metadata at runtime and use it for framework adaptation. This distinction is important because the same goal could be achieved using different strategies [19].

The metadata consumed by the framework can be defined in different ways. Naming conventions [20] use patterns in the name of classes and methods that have a special meaning for the framework. To exemplify this, there are the JavaBeans specification [21], which uses method names beginning with ‘get’ and ‘set’, and the JUnit 3 [22], which interprets methods beginning with ‘test’ as test cases implementation. Ruby on Rails [23] is an example of a framework known by the naming conventions usage. Other information can also be used on conventions, such as variable types, method parameters and other class characteristics.

Conventions usage can save a lot of configurations, but it has a limited expressiveness. For some scenarios, the metadata needed are more complex and naming conventions are not enough. An alternative can be setting the information programmatically in the framework, but it is not used in practice in the majority of the frameworks. Another option is metadata definition in external sources, like XML files and databases. The possibility to modify the metadata at deploy time or even at runtime without recompiling the code is an advantage of this type of definition. However, the definition is more verbose because it has to reference and identify program elements. Furthermore, the distance that configuration keeps from the source code is not intuitive for some developers.

Another alternative that has become popular in the software community is the use of code annotations, which is supported by some programming languages like Java [24] and C# [25]. Using this technique, the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to the source code. The use of code annotations is a technique called attribute-oriented programming [26].

Fernandes et al. [4] presented a study about how the different types of metadata definition are suitable for different framework requirements. Their study analyses how simple it is to use and develop a framework with some requirements about metadata. The requirements considered are metadata extension, existence of more than one metadata schema per class in different contexts and runtime metadata modification.

1.2.3 Metadata-Based Frameworks

Metadata-based frameworks can be defined as frameworks that process their logic based on the metadata of the classes whose instances they are working with [1]. In these frameworks, the developer must define, into application classes, additional domain-specific or application-specific metadata to be consumed and processed by the framework.

The use of metadata changes the way frameworks are built and how they are used by software developers. In an interview motivated by the 15 years of the book *Design Patterns: Elements of Reusable Object-Oriented Software* [27], when asked about how the metadata approach replaces or complements the patterns in the book, Erich Gamma answered the following [28]:

> While they complement the patterns in DP (referring to the patterns in the book) it can indeed be the case that meta-programming can replace the design pattern used in a design. The evolution of JUnit 3 to JUnit 4 comes to mind. JUnit 3 was a small framework that used several patterns like Composite, Template Method and Command. JUnit 4 leverages the Annotations meta-programming facilities introduced in J2SE 5.0. The use of the patterns disappeared and the framework evolved into a small set of annotations plus a test runner infrastructure that executes the annotated Java code.

In metadata-based frameworks, some variable points in the framework processing are determined by class metadata. Reflective algorithms must be generic and, in some cases, they cannot be applied due to more specific requirements for some classes. Metadata can be used to configure specific behaviours when the framework is working with that class.

The developer's perspective in the use of those frameworks has a stronger interaction with metadata configuration than with method invocation or class specialization. In traditional frameworks, the developer must extend its classes, implement its interfaces and create hook classes for behaviour adaptation. He also has to create instances of those classes, setting information and hook class instances. Using metadata-based frameworks, programming focus is on declarative metadata configuration and the method invocations in framework classes are smaller and localized.

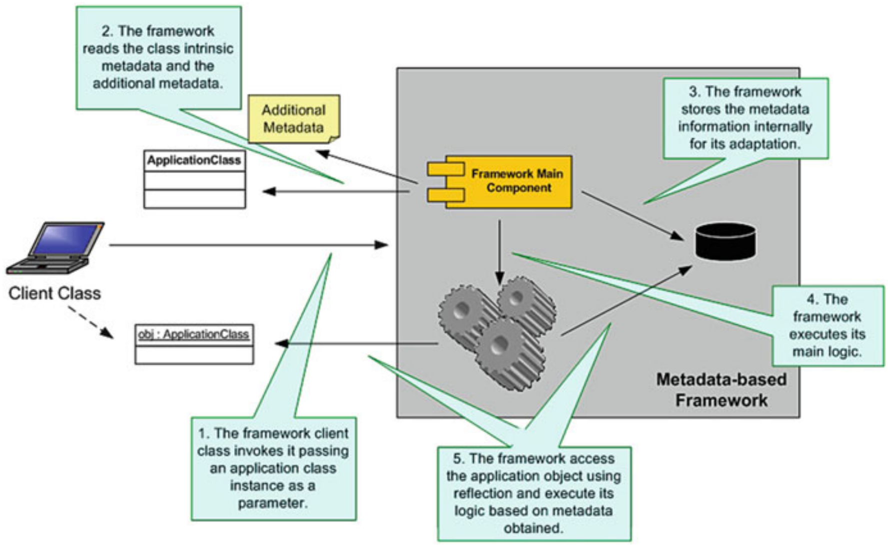
2. The framework reads the class intrinsic metadata and the additional metadata.
Additional Metadata
ApplicationClass
Framework Main Component
3. The framework stores the metadata information internally for its adaptation.
Client Class
obj : ApplicationClass
4. The framework executes its main logic.
Metadata-based Framework
1. The framework client class invokes it passing an application class instance as a parameter.
5. The framework access the application object using reflection and execute its logic based on metadata obtained.

Fig. 1.1 Basic execution process of a metadata-based framework

Figure 1.1 presents the basic processing steps in a metadata-based framework. It starts with the framework main component being called by the application and passing an application object as a parameter. It is important to notice that this first step can be triggered more transparently using aspects or dynamic proxies. Then, the framework reads class-intrinsic metadata using introspection and additional metadata, like in annotations or in XML files. Cached information can also be used to avoid unnecessary readings.

This information is somehow stored inside the framework for a further use. It can store the meta-information or create and configure hook classes based on them. After that, the framework calls its main logic, which uses the read metadata to adapt its behaviour and introspection to access and modify the application object. Not all the metadata-based frameworks follow exactly this process, but it captures a good abstraction of how they work.

In the metadata-based approach, the metadata can be considered as one kind of hot spot since the framework changes its behaviour based on it. Usually a framework has only one defined behaviour for each instantiation; however, using metadata it can have a distinct behaviour for each application class received. Internally, the framework can use the other presented techniques for behaviour adaptation, but it configures them based on each class metadata. One advantage of this approach is to allow a granular and manageable configuration of the framework variabilities.

An important drawback of such kind of framework is the indirection caused by the usage of metadata. Since the behaviour is generated by metadata configuration, an error or inconsistence in class metadata can cause an unexpected result. Since metadata has a declarative nature, this error is hard to debug and find if the framework does not provide a comprehensive error message.

1.3 Cloud Framework Examples

The goal of this section is to present some examples of metadata-based frameworks that can be applied to cloud architectures to exemplify the usage of metadata for such frameworks. It is not in the scope of this chapter to perform a comparative study about these frameworks or to present all their features.

Among many metadata-based frameworks and APIs designed for regular enterprise applications, there are some that are not supported by GAE. However, there are others that are supported with restrictions, such as JPA [40], and those that work fully without any change, such as Guice [8].

Because of the wide use of GAE, frameworks for the exclusive use of this platform raised, such as Gaelyk and Objectify. But there is a trend of creating frameworks that abstracts the particularities of a cloud architecture and work also in other environments, such as Play Framework. The following sections present some existing metadata-based frameworks used for cloud architectures, focusing on how metadata is used in the client code and how they are consumed.

1.3.1 Gaelyk

Gaelyk Framework [5] is a Groovy Web Framework for GAE, which makes use of metadata to map classes to persistent entities and inject various services in client classes. Listing 1.1 illustrates the usage of annotation @GaelykBinding, which indicates this class should be injected by the framework.

The framework uses dynamic features of the Groovy Language to get all classes with @GaelykBinding and, at compile time, injects GAE services, such as DatastoreService, MemcacheService and MailService. Gaelyk implements Active Record Pattern [29], which adds the data access logic in the domain object. As it is shown in Listing 1.2, for an ordinary class to become a persistent entity, it needs to add annotations in the client class and in their fields, identifying keys, not indexed and transient fields. The @Entity annotation is consumed by the compiler, which injects CRUD methods, such as save, delete, find and others. When some of these methods are called, the framework uses the

```
//One needs only use this annotation
//to Inject GAE services
@GaelykBindings
class WeblogService {
   def numberOfComments(post) {
      // the datastore service is available
      datastore.execute {
         select count from comments where postId == post.id
      }
   }
}
```

Listing 1.1 Usage example of @GaelykBinding

```
@Entity
class Person {
   @Key String login
   String firstName
   String lastName
   @Unindexed String bio
   @Ignore String getFullName() {"$firstName $lastName"}
}
```

Listing 1.2 Example of how to define an entity using Gaelyk

```
public class Address {
   @Id Long id;
   String street;
   String city;
}

public class Person {
   @Id Long id;
   String name;
   @NotSaved String street;
   @NotSaved String city;
   Key<Address> address;

   @PrePersist
   void onSave(Objectify ofy){
      if (this.street != null || this.city != null){
         this.address =
           ofy.put(new Address(this.street, this.city));
      }
   }
}
```

Listing 1.3 Example of how to define an entity using Objectify

field annotations to convert the object into a Google DataService entity. Only after this conversion, the persistence operation is really executed.

1.3.2 Objectify

An alternative to develop a persistence layer at GAE is the framework Objectify [6], which implements a metadata mapping between classes and the GAE persistent storage. Its main differential is that it covers all the Google DataService features. As previous example, it is necessary to add annotations in client code to identify fields with a distinct behaviour from default. Listing 1.3 shows an example of entities defined with Objectify. In this example, it is possible to observe that annotations are used to indicate how the framework should handle each field on persistence operations. It is also possible to define callback methods, which are called at certain times by the framework. In Listing 1.3, the method with the @PrePersist annotation will be called before it is saved on the data storage.

```
@OnApplicationStart
public class Initializer extends Job {
   public void doJob() {
      //do something
   }
}

@On("0 0 6 * * ?")
public class DailyReportJob extends Job {
   public void doJob() {
      //create a report
   }
}
```

Listing 1.4 Example of how to schedule jobs in Play Framework

To configure a class as an entity, it is not necessary to add an annotation in the class, nor to configure the class name in an XML file. On Objectify the class needs to be registered previously to its usage. This registration can be done by the invocation of the method register() in the class ObjectifyService.

When the application code registers a persistent class, ObjectifyService reads all annotations and stores them in memory. When a CRUD method is called, the framework uses the metadata to convert the client entity into a data store entity. After that, Objectify invokes the Google DataService methods passing the parameters according to the metadata retrieved.

1.3.3 Play Framework

Play Framework [7] is Java and Scala [30] web framework that enables to deploy applications on the cloud application platforms Heroku [31] and GAE. This framework abstracts the characteristics of the cloud environment, allowing the application to be deployed also on dedicated servers. Regardless of the deployment option, it provides a single way to schedule asynchronous jobs with annotations. Listing 1.4 shows examples of how to do that. For instance, to schedule a job to run at start time, it is necessary to mark the class with @OnApplicationStart annotation. It is also possible to schedule a job to run at a specific instance, like the example presented in Listing 1.4 that creates a daily report at 6:00 AM.

1.3.4 Miscellaneous

The examples of metadata-based framework presented in the previous sections have focused at persistence, dependence injection and scheduling in framework designed specifically to execute in cloud architectures. This section enumerates some other examples that can be applied to cloud application. Jersey [32], for instance, can be used to map using metadata class methods to restful web services that can be accessed remotely. On the web tier, the framework VRaptor [33] can be used in the development of web controllers, using annotations to determine which requests each method should handle. Hibernate Validator [34] uses metadata, defined as annotations or XML documents, to define constraints to validate application class instances. At last, JAXB [35] is an API which maps application classes to XML documents, also using metadata.

Finally, it is important to emphasize that the main goal of the Java EE 7 specification [36], which is in development, is to allow enterprise applications to be deployed in dedicated servers or in cloud environments. It is for this reason it is possible to speculate that some specifications that already integrate the stack will be adjusted, and others will be created in order to support the cloud requirements. It is expected that applications that follow the standard should be able to be ported between different application servers and cloud providers. Since the current Java EE specification provides a metadata-based API, the techniques presented in this chapter will be very important to develop its features for cloud providers.

1.4 Internal Structure

The internal architecture of a framework is composed of hot spots and frozen spots which respectively represent points with fixed functionality and points where behaviour can be extended and adapted. The potential of reuse of a framework is directly related to its capacity to adapt to different requirements and applications. This is achieved by providing hot spots at the right places, allowing the application to extend its behaviour when necessary.

In frameworks that aim to provide functionality for cloud-based applications, the flexibility requirements can cover different kinds of needs. The first one is to enable each application to adapt the framework behaviour to its needs, considering that many applications will share the same resources. The second is to enable an application to be built independently from the cloud providers, allowing each one to adapt the implementation according to its infrastructure. And finally, the third is to enable the evolution of the cloud services without affecting the deployed applications.

This section is based on a pattern language that studied several metadata-based frameworks and identified recurrent solutions on them [1]. The practices presented focus mainly on metadata reading and processing, providing alternatives to extend behaviour on each mechanism. It is important to state that, despite all practices can be used successfully on the same framework, they should be introduced according to the framework needs.

1.4.1 Metadata Reading and Processing Decoupling

Some metadata-based frameworks consume metadata and execute its behaviour at the same time. The coupling between these two concerns can prevent the introduction of extensions on both mechanisms. So, when designing this kind of framework,

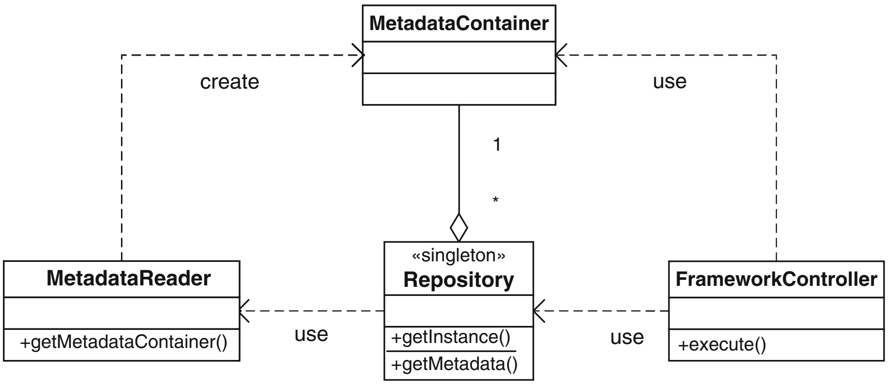

Fig. 1.2 Decoupling between metadata reading and processing

the first requirement to be considered is the decoupling between metadata reading and execution. To achieve this, a solution can be the introduction of a class that represents metadata at runtime and is used to exchange data between these two mechanisms. A representation of this solution is presented in Fig. 1.2.

The MetadataReader is the class responsible to read metadata wherever it is defined and to create an instance of MetadataContainer representing metadata. Further, the MetadataContainer is accessed by the FrameworkController, which in this scenario has the role to receive the framework client calls and execute the main functionality. The MetadataContainer became the protocol between the other components, allowing their decoupling.

This decoupling is also important to allow the MetadataContainer to be stored and reused, avoiding unnecessary metadata readings. In Fig. 1.2 this solution is represented by the class Repository, which can intermediate the communication between FrameworkController and MetadataReader. This class can create a cache of the instances of MetadataContainer already retrieved, improving framework performance after the first call. The Repository can also be a central component where metadata can be retrieved easily by all the framework components.

1.4.2 Flexibility on Metadata Reading

By the introduction of a component responsible to read metadata, it is possible to apply solutions to this mechanism transparently from the other parts of the framework. As presented previously in this chapter, there are several ways to define metadata, such as code conventions, code annotations and external sources (XML documents, databases). Depending on the application requirements, a different metadata definition strategy can be more suitable. For instance, the usage of code annotations are less verbose and closer to the source code, but an external source should be used when you need to be able to change configurations at deploy time without recompiling the code.

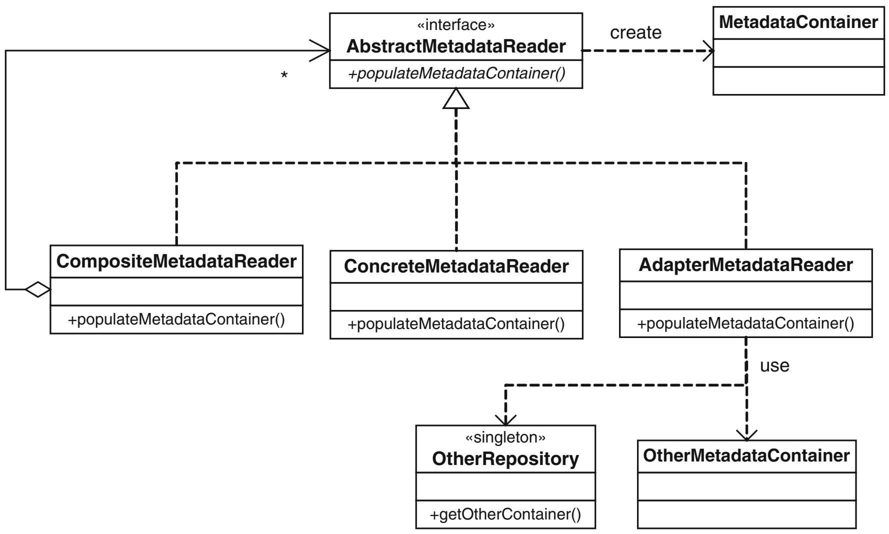

Fig. 1.3 Providing flexibility on metadata reading

To allow the flexibility on metadata reading, the metadata reader can be defined by an interface which can have more than one implementation. Each implementation can provide logic to read metadata from a different kind of source or with a distinct data schema. The structure of this solution is presented in Fig. 1.3. The interface AbstractMetadataReader represents an abstraction of a metadata reader component, and the ConcreteMetadataReader represents an implementation of it.

Based on the presented structure, it is also possible to create a chain of metadata readers, enabling more than one metadata source to be considered at the same time. In Fig. 1.3, the pattern Composite [27] is used on the class CompositeMetadataReader to implement this reading sequence. A Chain of Responsibility [27] is another option for this implementation. That solution enables the introduction of metadata readers that reads only a partial portion of metadata. This enables the application to create metadata readers that can consider domain-specific code conventions to infer part of the information. That also allows the existence of classes like the AdapterMetadataReader, which obtain metadata from the Repository of other frameworks, avoiding metadata redundancy and duplication.

1.4.3 Metadata Schema Extension

An important flexibility requirement that a metadata-based framework can have is to enable the extension of the metadata schema associated to an extension on the framework behaviour. In other words, the application should be able to create new types of metadata elements and to execute application classes when that

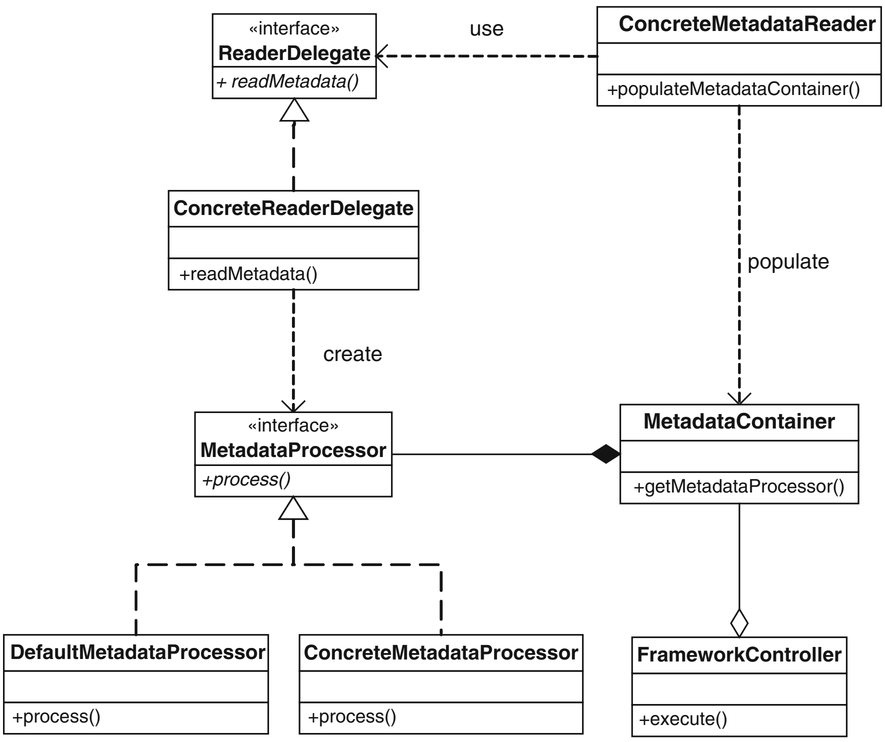

Fig. 1.4 Framework structure to enable metadata extension

```
@AssociatedDelegateReader(AnnotationReader.class)
public @interface ApplicationAnnotation{
    String value();
}
```

Listing 1.5 Example of association between annotation and ReaderDelegate

piece of metadata is processed by the framework. The structure necessary to enable metadata extension as a hot spot is presented in Fig. 1.4.

During the metadata reading process, a ConcreteMetadataReader should delegate the metadata reading of each piece of metadata to an associated class, which in this work is called ReaderDelegate. The concept of metadata piece can vary with the context and with the metadata definition strategy. For instance, if code annotations are being used, the metadata piece can be a single annotation. As another example, if metadata is defined in an XML document, a metadata piece can be an XML element.

Listing 1.5 presents an example of how an annotation can be associated to its delegate metadata reader. A framework annotation, in this example @Associated-DelegateReader, can be used to define the ReaderDelegate implementation which

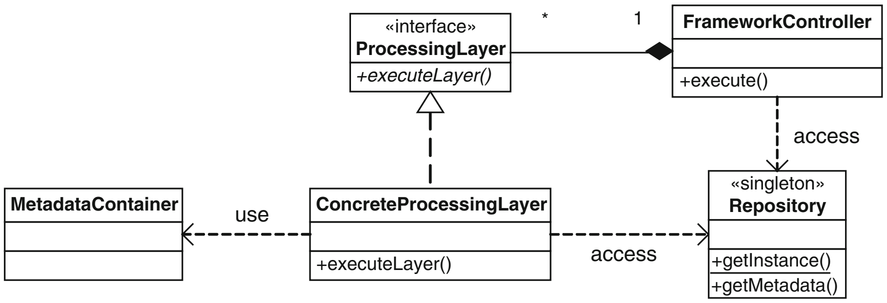

Fig. 1.5 Metadata processing layer structure

should be used to interpret the metadata. To find these custom annotations, the ConcreteMetadataReader should search on all class annotations searching for the ones annotated with @AssociatedDelegateReader. Then, an instance of the associated delegate reader class should be created and used to interpret the annotation.

As a result, the DelegateReader should return an instance responsible to execute the behaviour associated with that piece of metadata, which is called a MetadataProcessor. This MetadataProcessor is added to the MetadataContainer associated to its respective code element. During the metadata processing, part of the framework execution is delegated to the Metadata Processor.

So, based on this solution, an application which needs to extend metadata should define the new metadata type, a metadata reader delegate and a Metadata Processor. The created metadata type should be associated to the metadata reader delegate, which should return the Metadata Processor as the result of the reading.

1.4.4 Metadata Processing Layers

The behaviour extension by defining new metadata types can be appropriate for some scenarios, but in other situations it can be necessary to add application-specific logic on the entire metadata processing. There are also some framework domains in which it is hard to isolate the processing for each piece of metadata. In these cases, it is important to provide an extension point that can interfere with the whole metadata processing.

A solution to this issue found for some frameworks is to divide the processing logic on different layers. Each layer is responsible for part of the framework logic, and the FrameworkController is responsible to coordinate their execution. The solution is represented in the diagram in Fig. 1.5. The interface ProcessingLayer should be implemented by the ConcreteProcessingLayers and represent a framework extension point. Following this structure, new layers with application-specific logic can be easily introduced, and their execution order can also be customized.

1.5 Architectural Scenarios

As presented in the previous section of this chapter, there are several frameworks for cloud architectures which use the metadata-based approach. This section presents some architectural scenarios where the usage of metadata as a hot spot is a suitable solution. Some of the scenarios presented here are based on documented architectural patterns for metadata-based frameworks [3], but contextualized for cloud environments.

The uses presented here are not only based on the existing frameworks designed for cloud architecture but also for other kind of software. Even if the usage of some of these solutions is restricted to the cloud environment, their successful usage in similar scenarios but on other reference architectures can indicate a potential usage in cloud frameworks.

1.5.1 Dependency Injection Indication

Dependency injection [37] is a pattern where the object dependencies are injected externally by some class that creates the instance or manages its life cycle. Common ways of dependency injection are by constructor, by assessor or by an interface method. This practice decouples the class for its dependence, since the concrete instance that is injected is defined and created externally.

Metadata can be used to indicate which fields should be injected in a class. Additionally, metadata can also be used to indicate characteristics of the instance that should be injected. Spring framework [38] uses metadata defined in an XML file to inject the dependencies; however, new versions also support annotation-based injection. Java EE specification [39] also uses annotations to indicate the fields which should be used for injection. An example of the cloud environment is Gaelyk [5], which uses annotations to indicate that a class should receive instances that represent the GAE services.

The decoupling is a consequence of dependency injection that is enhanced by the usage of annotations, which defines which fields should be injected. That strategy can be very useful for cloud frameworks to enable different ways for creating and making available to the application classes the services provided by the cloud, such as for sending emails and connecting to data stores.

As a consequence, the provider can have the same application deployed in different environments and inject in its classes different implementations of the same service. Additionally, the cloud provider can evolve the implementation which is injected in the application class without impacting on it. In brief, this practice allows the cloud framework to use different strategies to create and handle the life cycle of its services instances.

Figure 1.6 illustrates how the metadata-based dependency injection happens. The cloud service should provide an interface which should be referenced by the components in the application. The fields which should receive the service injection

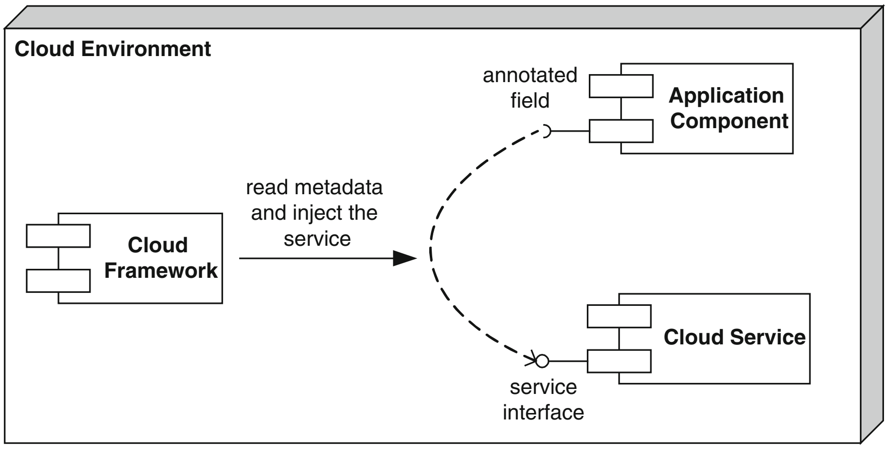

Fig. 1.6 Dependency injection based on metadata

use metadata to indicate that to the framework. In turn, the framework reads the application class metadata, obtains a cloud service instance by the most appropriate way and injects it in the application class instances in their initialization.

1.5.2 Entity Mapping

It is very common for applications to have different representations of the same domain entity. For instance, a business concept represented as a class at runtime can be persisted in a database, presented in a user interface or sent to other applications on web services. The conversion between the different entity representations can generate a very repetitive and error-prone code for the several entities of the system. This practice also couples the application to the other specific representation, for instance, a database schema or an XML format.

Metadata can be associated with one representation of an entity, configuring how it can be mapped to another representation. It should add information about how each characteristic is mapped and converted to the other representation. The most common use is to map between classes and databases, following the pattern metadata mapping [29]. Examples of cloud frameworks which use this kind of mapping are Gaelyk [5] and Objectify [6]. The framework Spring Data [41] also proposes the mapping between interface methods to database queries using code conventions.

It is important to state that this solution is not exclusive for mapping to persistence storages. For instance, when mapping to a web service, a method could be mapped to a service and an entity could be mapped to a parameter.

Cloud providers usually support persistence by using nonrelational databases. The mechanisms of such storages may be different according to different kinds of

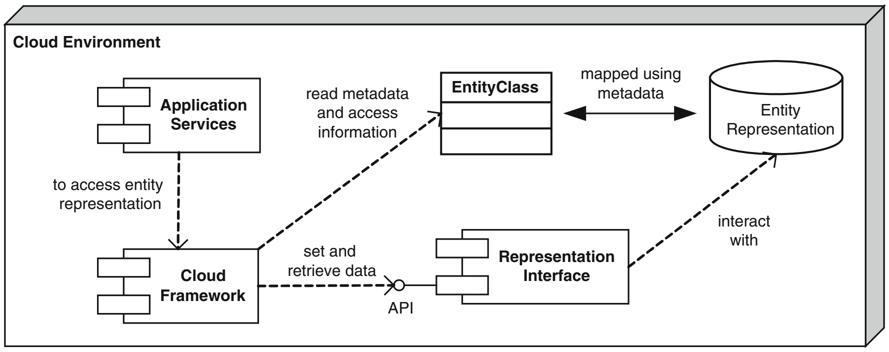

Fig. 1.7 Metadata-based entity mapping

parameters. Metadata can configure the persistent classes with constraints about how it should be persisted, such as which fields are unique identifications, which should not be persisted and even the data format that each one should be stored. That mapping can help in the decoupling between the cloud application and how the data is actually stored. As a consequence, it can improve application portability among cloud providers.

Figure 1.7 depicts the characteristic of an architecture that uses a metadata-based entity mapping. The class that represents an entity in the application should be configured with custom metadata, which maps it to the other representation. When the application components need to access the information from the other representation, they invoke functionality in an API provided by the framework. The EntityClass is used by the application to interact with the framework. After, the framework should access the API from a component which interacts with the other representation, making calls according to the entity metadata to set and retrieve information. For instance, the component called Representation Interface could be a native driver to access a database.

1.5.3 Configured Callback Methods

When an application executes embedded on a server, sometimes it needs to know about events that happen on the server. A design pattern appropriate for this scenario is Observer [27], in which an application class needs to realize an interface and receives invocations on its methods when the events happen. That can be a solution to enable application classes to receive notifications from events that happen on the cloud provider side. The problem of this approach is that the application needs to be coupled with the cloud framework interface, and consequently with the parameter types of its methods. That is specially a problem when this interface needs to evolve and when the application needs to be adapted to different kinds of cloud environments.

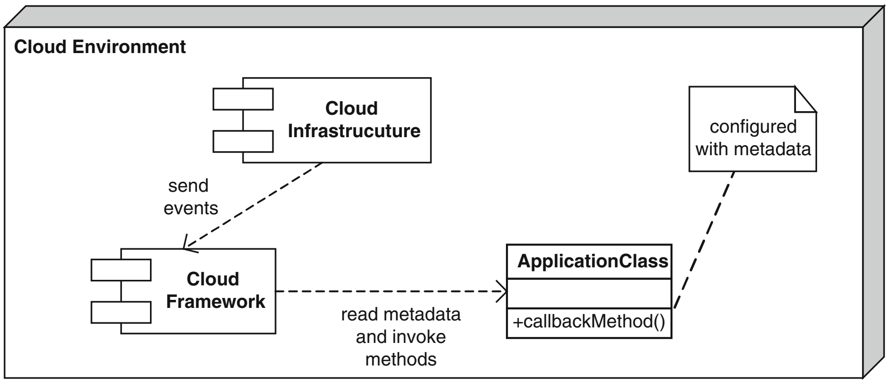

Fig. 1.8 Invoking callback methods configured with metadata

Another common use of metadata is to indicate callback methods in the application classes. By using an annotation or other kind of metadata, the application class indicates which methods should be invoked, instead of implementing an interface. Then the framework looks into the class methods and invoked the configured ones when appropriate.

A very common use of this solution is in application frameworks that handle HTTP requests, such as JSF [42] and VRaptor [33], which were initially designed for regular web applications but can also be used in cloud architectures. This solution is also applied on persistence frameworks, such as Objectify [6], to callback application classes before or after persistence operations, such as saving or loading. Play Framework [7] uses annotations in classes to execute them when the application starts or on scheduled jobs. The scheduling specification is an instance of how metadata can define fine-grained conditions for method execution.

Usually cloud applications abstract the environment in which it is deployed, since it does not have much knowledge about where it is located. For instance, it can use the data storage service available on the cloud provider without actually knowing what the implementation is used. However, sometimes it is important for the application to know when some events happen on the server. For instance, the events can be related to persistence operations, like loading and persisting, or to session migration, like when a session is transferred among servers.

By using a metadata-based approach, the application classes became decoupled from the framework interfaces and only need to handle necessary events. That increases the application portability and even allows the cloud provider to evolve its event model without breaking existing applications. The usage of metadata also allows the addition of constraints, which enables a granular model for event handling. For instance, suppose that a method should be invoked before the migration of a session to another server, the addition of constraints in the metadata can configure for that method to be invoked only when the session attribute “logged” is true.

Figure 1.8 illustrates the structure of this solution. The framework should read the metadata from the application classes and identify which methods should be invoked and in what conditions. After that, the framework should observe the cloud infrastructure events and delegate the execution to the application class method when appropriate.

1.5.4 Other Architectural Uses

The previous sections presented the uses of metadata for frameworks developed specifically for cloud architectures. However, there are other scenarios where metadata can be applied that are found more often on general purpose framework or on the ones design for other kind of architecture. The goal of this section is to describe more briefly these other scenarios, which can potentially be applied in cloud frameworks in the future.

Sometimes, requirements demand that the application iterates through the fields of some classes, executing some logic for each one. In this context, metadata can be applied to allow the creation of a general algorithm, which processes each field according to the metadata found for each one. Examples of this are Hibernate Validator [34], which validates the constraints of each field from an instance, and Esfinge Comparison [43], which compares all the fields from two instances of the same class. Metadata is used respectively in each framework to define each field constraint and to define the comparison criteria.

On server-based applications, it is common for a class to be managed by the server, having its life cycle controlled by it. In these scenarios, usually it is used as a proxy, a decorator [27] or even an aspect to intercept the method invocation and transparently add some kind of functionality. Accordingly, metadata can be used to configure parameters about the functionality that should be executed. For instance, in Java EE standard [36], annotations are used to configure constraints about security and transaction management. For cloud providers, for instance, metadata could configure a method-level cache which should use the memory cache available as a cloud service.

Finally, metadata can also be used in application classes to enable an automatic generation of customized graphical interfaces. Metadata can be used to define constraints about how each field should be represented on screen. This use is appropriate when the service model is “application as a service” and the user is able to edit the domain entities structure. This is often enabled in dynamic languages and on Adaptive Object Model architectural style [44]. SwingBean [45] is an example of a framework that uses metadata to generate tables and forms.

1.6 Final Considerations

One cloud computing service model is known as platform as a service. In this model, it is provided a computing platform and a solution stack where applications should be deployed. The cloud provider should have available tools, libraries and frameworks, which should be used by the applications to access services and resources.

This chapter presents how metadata-based frameworks can be used in the construction of cloud-based applications, helping to decouple the application from cloud provider details. Many examples of cloud framework which use metadata were presented, along with some details about how metadata is consumed and how it is used. A set of practices to develop the internal structure of this kind of framework were also presented, in order to introduce the main kinds of hot spots which can be provided. At last, the chapter presented some architectural scenarios in which the usage of metadata is suitable to.

Despite the metadata approach used in several frameworks designed for the cloud, there are many applications that can still be explored. The goal to make transparent for the application the environment where it is deployed is far from being reached. However, some advances were made, like the some features from Play Framework [7]. The new standard for Java enterprise applications [36], which when this chapter was written was a work in progress, represents another effort to achieve this goal. In this context, to use metadata to define framework hot spots can be a good strategy to achieve the decoupling needed for this portability.

References

1. Guerra, E., Souza, J., Fernandes, C.: A pattern language for metadata-based frameworks. In: Proceedings of the 16th Conference on Patterns Languages of Programs, Chicago, 28–30 August 2009
2. Guerra, E.: A conceptual model for metadata-based frameworks. Ph.D. thesis, Aeronautical Institute of Technology, São José dos Campos (2010)
3. Guerra, E., Fernandes, C., Silveira, F.: Architectural patterns for metadata-based frameworks usage. In: Proceedings of the 17th Conference on Pattern Languages of Programs, Reno, 16–18 October 2010
4. Fernandes, C., Guerra, E., Nakao, E., Ribeiro, D.: XML, annotations and database: a comparative study of metadata definition strategies for frameworks. In: XML: Aplicações e Tecnologias Associadas (XATA 2010), Vila do Conde, 19–20 May 2010
5. Zahariev, A.: Google app engine. In: Seminar on Internetworking, Espoo, 27 April 2009
6. Google: Objectify Framework. http://code.google.com/p/objectify-appengine/ (2012). Accessed 9 June 2012
7. Reelsen, A.: Play Framework Cookbook. Packt Publishing, Birmingham (2011)
8. Venbrabant, R.: Google Guice: Agile Lightweight Dependency Injection Framework. Apress, New York (2008)
9. Johnson, R., Foote, R.: Designing reusable classes. J. Object-Oriented Program **1**(2), 22–35 (1988)
10. Jacobsen, E., Nowack, P.: Frameworks and patterns: architectural abstractions. In: Fayad, M., Schmidt, D., Johnson, R. (eds.) Building Application Frameworks: Object-Oriented Foundations of Frameworks Design. Wiley, New York (1999)
11. Fayad, M., Schmidt, D., Johnson, R.: Application frameworks. In: Fayad, M., Schmidt, D., Johnson, R. (eds.) Building Application Frameworks: Object-Oriented Foundations of Frameworks Design. Wiley, New York (1999)
12. Bosch, J., et al.: Framework problems and experiences. In: Fayad, M., Schmidt, D., Johnson, R. (eds.) Building Application Frameworks: Object-Oriented Foundations of Frameworks Design. Wiley, New York (1999)

13. Pree, W.: Design Patterns for Object-Oriented Software Development. Addison Wesley, Reading (1995)
14. Foote, B., Yoder, J.: Evolution, architecture, and metamorphosis (Chap. 13). In: Vlissides, J., Coplien, J., Kerth, N. (eds.) Pattern Languages of Program Design 2, pp. 295–314. Addison-Wesley Longman, Boston (1996)
15. Damyanov, I., Holmes, N.: Metadata driven code generation using .NET framework. In: International Conference on Computer Systems and Technologies, 5, 2004, Rousse. pp. 1–6 (2004)
16. Quinonez, J., Tschantz, M., Ernest, M.: Inference of reference immutability. In: 22nd European Conference on Object-Oriented Programming, 2008, Paphos. pp. 616–641 (2008)
17. Ernest, M.: Type annotations specification (JSR 308). http://types.cs.washington.edu/jsr308/specification/java-annotation-design.pdf (2011). Accessed 15 May 2012
18. Hel, J., Eichhorn, P., Zwitserloot, R., Grootjans, R., Spilker, R., Koning, S.: Project Lombok. http://projectlombok.org/ (2012). Accessed 15 May 2012
19. Fowler, M.: Using metadata. IEEE Softw. **19**(6), 13–17 (2002)
20. Chen, N.: Convention over configuration. http://softwareengineering.vazexqi.com/files/pattern.html (2006). Accessed 15 May 2012
21. Java Community Process: JavaBeans(TM) specification 1.01 Final release. http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/ (1996). Accessed 15 May 2012
22. Massol, V., Husted, T.: JUnit in Action. Manning, Greenwich (2003)
23. Ruby, S., et al.: Agile Web Development with Rails, 3rd edn. Pragmatic Bookshelf, Raleigh (2009)
24. Java Community Process: JSR 175: a metadata facility for the java programming language. http://www.jcp.org/en/jsr/detail?id=175 (2003). Accessed 15 May 2012
25. Miller, J.: Common Language Infrastructure Annotated Standard. Addison-Wesley, Boston (2003)
26. Schwarz, D.: Peeking inside the box: attribute-oriented programming with Java 1.5. http://missing-manuals.com/pub/a/onjava/2004/06/30/insidebox1.html (2004). Accessed 15 May 2012
27. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
28. O'Brien, L.: Design patterns 15 years later: an interview with Erich Gamma, Richard Helm and Ralph Johnson. InformIT. http://www.informit.com/articles/article.aspx?p=1404056 (2009). Accessed 15 May 2012
29. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Boston (2002)
30. Odersky, M. et al.: An overview of the Scala programming language. Technical report IC/2004/640. EPFL, Lausanne (2004)
31. Heroku: Heroku cloud application platform. http://www.heroku.com/ (2012). Accessed 9 June 2012
32. Java.net: Jersey API. http://jersey.java.net/ (2012). Accessed 9 June 2012
33. Freire, A., Silveira, P.: Vraptor – simple and quick web framework. In: Anais do 5o Workshop sobre Software Livre, Porto Alegre, pp. 39–42 (2004)
34. RedHat: Hibernate validator. http://www.hibernate.org/subprojects/validator.html (2012). Accessed 8 June 2012
35. Java Community Process: JSR 222: JavaTM Architecture for XML Binding (JAXB) 2.0. http://jcp.org/en/jsr/detail?id=222 (2009). Accessed 8 June 2012
36. Java Community Process: JSR 342: JavaTM Platform, Enterprise Edition 7 (Java EE 7) specification. http://jcp.org/en/jsr/detail?id=342 (2012). Accessed 8 June 2012
37. Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html (2004). Accessed 8 June 2012
38. Walls, C., Breidenbach, R.: Spring in Action, 2nd edn. Manning, Greenwich (2007)
39. Java Community Process: JSR 318: Enterprise JavaBeansTM 3.1. http://jcp.org/en/jsr/detail?id=318 (2010). Accessed 8 June 2012
40. Java Community Process: JSR 317: JavaTM Persistence 2.0. http://jcp.org/en/jsr/detail?id=317 (2009). Accessed 8 June 2012
41. Spring Source: Spring projects – Spring data. http://www.springsource.org/spring-data (2012). Accessed 8 June 2012
42. Java Community Process: JSR 344: JavaServerTM Faces 2.2. http://jcp.org/en/jsr/detail?id=344 (2011). Accessed 8 June 2012

43. Esfinge: Esfinge Framework. http://esfinge.sf.net/ (2012). Accessed 15 May 2012
44. Yoder, J., Johnson, R.: The adaptive object-model architectural style. In: Proceedings of the IFIP 17th World Computer Congress – TC2 Stream/3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, Montreal, 25–29 August 2002
45. Sourceforge: SwingBean. http://swingbean.sf.net/ (2012). Accessed 15 May 2012