Annotations-driven configuration framework for Java applications

Victor S. Denisov

***Abstract* – Modern features of Java software platform allow developers to create a new generation of application configuration frameworks with advanced features, such as type-safety, easy extendability and self-documentation. This paper describes an early prototype implementation of such a framework, currently under development as an open-source project.**

***Keywords* — application configuration management, application settings management, framework, Java, open-source, options-util.**

I. Introduction

Runtime configuration is an indispensable feature for most real-world production applications. Over the years, a typical pattern for configuration management had emerged – configuration options are stored in some kind of persistent storage (most often, as one or more files on a host's filesystem) and are read at start-up (and, possibly, later as well – when the configuration file changes, for example) and the options are stored in some sort of internal configuration object(s) for future reference from appropriate application modules.

When a pattern can be detected, a framework (or two) which simplifies application's implementation according to that pattern usually follows. This is the case with configuration management as well – see [1] for an overview of some existing frameworks for configuration management of Java applications.

Unfortunately, frameworks mentioned in [1] hadn't kept pace with recent developments in the Java programming language (such as annotations), other Java-related technologies (such as dependency injection and cloud computing) as well as an overall increase in complexity of typical Java applications (which calls for reliability and readability enhancements such as type-safety, self-documenting code and thorough unit testing). This calls for a new generation of configuration frameworks, requirements for which had been summarized in [2].

This paper presents an overview of a prototype of such a modern configuration framework, named *options-util* which is being developed as an open-source project under the Apache Software License, Version 2.0 [3], available at https://github.com/options-util/options-util.

Victor S. Denisov is with the Lomonosov Moscow State University (e-mail: vdenisov@plukh.org).

II. Development goals

Options-util framework was developed to match functional requirements for a modern application configuration framework. To summarize from [2]:

- Common requirements:
  - ↘ platform independence;
  - ↘ unit and integration tests;
  - ↘ support of dependency injection.
- Specific requirements:
  - ↘ minimal number of dependencies;
  - ↘ annotation-driven configuration;
  - ↘ self-documentation of configuration options;
  - ↘ support of cloud services;
  - ↘ type-safety;
  - ↘ support for different configuration sources/persistent backing stores;
  - ↘ extensibility;
  - ↘ support for complex data structures;
  - ↘ support for value validation and conversion;
  - ↘ runtime configuration changes/change listeners;
  - ↘ support for structured configuration information.

Right now, *options-util* is in an early prototype phase (version 0.1), with only the most important and architecture-defining features implemented. As development goes on, it is expected that this framework will evolve to meet all of the above requirements.

III. Core concepts

The core concept of *options-util* is the so-called options interface, which defines all configuration options in use by the application via appropriate getter and setter methods – which are, in turn, decorated with annotations to provide additional semantic information to the framework at runtime. Here's a very simple example of the options interface:

```
@Persistence(PersistenceType.TRANSIENT)
public interface TestOptions extends Options {
    @Option(key = "int")
    int getInt();
    void setInt(int value);

    @Option(defaultValue = "5")
    int getIntWithDefault();
    void setIntWithDefault(int value);
}
```

Even from this brief example, a couple of important points about the configuration interface can be discovered:

- an interface can be decorated with a *persistence provider* annotation, which attaches the specified provider to the configuration – a class which will provide access to the persistent backing store;
- all interfaces extend the common interface (`org.plukh.options.Options`), which defines some common configuration management methods.
- configuration information is accessed via type-safe method calls;
- `@Option` annotation provides additional information to the framework (in the example above, a value for the key for the property in the backing store for one option and a default value for another).

Another core concept is an `OptionsFactory`, which creates an instance of the interface using a dynamic proxy class. Instantiating the configuration class is trivially easy:

```
MyOptions options = (MyOptions)
    OptionsFactory
        .getOptionsInstance(MyOptions.class)
```

`OptionsFactory` provides the same (thread-safe) options instance for all invocations of `getOptionsInstance` for a certain interface class, thus ensuring that all application components share the same set of configuration properties.

IV. Meeting common functional requirements

In the current release, 2 out of 3 common requirements have been met. The only exception is dependency injection support, which will be coming in future releases.

A. Platform independence

Code for *options-util* does not make any assumptions about the platform it is running on[1]. Specifically:

- it doesn't assume that the platform has a filesystem, unless one of file-based persistence providers is used;
- it doesn't use byte code manipulation;
- it does not log its output anywhere but on the console (and even then, only when there is an error which can't be communicated in any other way);
- it does not create threads or use thread-related functions[2];
- it is extremely light on resources and imposes a very small CPU/memory overhead (less than 2MB of memory is used by the framework itself in a steady state[3], including all the data structures and loaded classes, with the majority of memory used by common Java classes which would probably be loaded by any application, such as classes from `java.lang` and `java.util` packages).

B. Unit and integration tests

Options-util is covered by an extensive suite of unit and integration tests, with current line coverage reaching 74%. Lines not covered by tests are mostly trivial getters/setters and constructors. As issues are discovered within the framework, regression tests are implemented to guard against re-occurrence of the problem.

C. Support of dependency injection

So far, support for dependency injection had not been implemented. Support for Spring [5] and Guice [6] frameworks is planned for one of the next releases, with support for Java EE CDI [7] being under consideration.

V. Meeting specific requirements

In the current release, 8 out of 11 requirements have been met (at least partially). Support for cloud services will be partially implemented in the next release, bringing the total to 9 out of 11. Remaining requirements (specifically, runtime configuration change listeners and structured configuration information) will be met eventually, as the framework matures.

A. Minimal number of dependencies

Current release has no external runtime dependencies at all. It is expected that it will stay this way for core functionality for the foreseeable future. Features which require external dependencies will be separated into independent optional modules, which will have to be explicitly added to a classpath (manually or via a dependency management framework such as Maven). Such modules will be dynamically discovered at runtime, with *options-util* failing gracefully (to the extent possible) if a certain feature is not available.

B. Annotation-driven configuration

As can be seen from an example in section III, *options-util* uses annotations to provide additional runtime information about various aspects of the application's configuration information management. In the current version, the following annotations have been defined:

- `@Option` – defines properties of a basic scalar option;
- `@CollectionOption` – defines properties of a set of options backed by one of Java's built-in collection interfaces;
- `@Persistence` – defines information about the persistent backing store, including

1 This is true about the framework itself; unit tests do make certain (minimal) assumptions, such as presence of a filesystem with a writable temporary directory.

2 In a future release, instance of the `Timer` class, if available, will be used to periodically refresh information from the persistent backing store; if unavailable, a platform-specific fall back, such as [4], will be used instead.

3 As determined by running a dedicated test exercising framework's functionality, and recording memory allocation data via a YourKit Java Profiler.

information about the type of the store to be used.

In the next release, they will be joined by several new annotations, including some specific to certain operating environments (such as `@AWSTags`, specific for configuration inside an AWS EC2 instance).

C. Self-documentation of configuration options

Let's consider the following options interface:

```
@Persistence(PersistenceType.PROPERTIES_FILE)
public interface TweetCaptureOptions extends Options {
    @Option(key = "jdbc.url")
    String getDatabaseURL();

    @Option(key = "workerthreads", defaultValue = "5")
    int getWorkerThreads();

    @CollectionOption(backingClass =
LinkedBlockingQueue.class, transientOption = true)
    Queue getWorkerQueue();
}
```

As with example in section III, a couple of points are evident from the above, even without knowing all the specifics of *options-util*:

- the interface contains three, and only three available configuration options;
- configuration information is available in a properties file-based storage;
- first option returns a String-valued URL for database connection; it is read-only (as evidenced by a lack of a setter); and its key in the persistent backing store is "jdbc.url";
- second option returns an integer value for a number of worker threads in this application; and its key and default value (which happens to be "5" in this case) can also be determined from the annotation;
- finally, the third option provides a queue, backed by the `LinkedBlockingQueue` implementation from Java Collections Framework; it is set as transient, so it won't be read nor written to the backing store.

Generally, the following information is automatically documented simply by defining it in the options interface:

- type of persistence used to store configuration information;
- all supported configuration options, along with their type and default value;
- keys for all configuration options in the backing store (with defaults available when the key is not specified explicitly);
- backing classes for collection options (with default implementations used when no explicit class is provided);
- state of persistence (writeability) of each option – options without setters will not be persisted to the backing store; also, options can be explicitly marked as read-only, so any changes to their values would be transient.

D. Support of cloud services

This requirement is partially implemented in the current development version: for Amazon Web Services EC2 instances, configuration information can be persisted to a file stored on the AWS Simple Storage Service (S3) [8]. Configuration file(s) can be differentiated by AWS instance id or by an ordered combination of tags specified by an `@AWSTags` annotation.

E. Type-safety

As can be seen in examples in sections III and V.C, options interface is fully type-safe – type checking is guaranteed at compile time. Underlying implementation (including dynamic proxy handler and value converters) can break this contract, which will result in a runtime exception.

Note that generic types are not supported at this time, as, due to type erasure by the Java compiler, getting information about a generic type at runtime is impossible without resorting to various kludges.

F. *Support for different configuration sources/persistent backing stores*

Current *options-util* release supports several backing stores by default, including .properties-based and XML-based files and a JDBC data source. Framework can be easily extended with custom persistence providers, allowing for support for arbitrary backing stores by the application's developer.

In the following releases, support for additional backing stores is planned:

- Core module:
  - ➘ JSON files;
  - ➘ Properties, XML and JSON over HTTP;
  - ➘ Properties, XML and JSON from classpath;
  - ➘ key-value mappings from system properties and environment variables;
  - ➘ JMX support;
- AWS module:
  - ➘ Properties, XML and JSON over AWS S3;
  - ➘ DynamoDB key-value store;
  - ➘ Instance metadata;
- GCP module:
  - ➘ Google App Engine Datastore;
  - ➘ Instance metadata;
- Azure module:
  - ➘ DocumentDB.

Additionally, next release will add support for persistence providers chaining, which will allow to combine multiple configuration sources together in an ordered fashion, thus eliminating gaps between capabilities of different environments. For example, a typical configuration chain for an application designed to run both on a local development machine, and on an AWS EC2 instance, can look like this:

- read default configuration file from classpath;
- try to read configuration file on a local filesystem, if found → process and abort chain;

- read configuration information from a global S3 object;
- override parts of that information with information from an instance-specific S3 object.

G. Extensibility

Options-util supports several extension points. Applications developer can plug in a custom persistence provider implementation, a persistence chain implementation and an option type handler (value converter) implementation. As the framework evolves, additional extension points will likely be identified and implemented.

H. Support for complex data structures

Options-util supports arbitrary Java objects as entities for configuration information, provided that developers implement a converter which would allow to unambiguously convert human-readable strings to and from object values. Built in converters include:

- all Java primitive types and their object wrappers;
- `java.lang.String`;
- `java.util.Date`;
- any class that extends `java.util.Collection` interface.

Ultimately, it is the task of the converter's developer to create a serialization/deserialization scheme that would be both unambiguous and human-readable enough to allow for editing of configuration information by hand.

Right now, the major limitation of the framework is that value converters are determined by option type (they're known as exactly that – option type handlers – internally), so that, for example, there can't be two different `java.util.Date` converters active at the same time (for example, to handle different date/time formats). It is expected that this limitation will be removed in a future release.

In the future, additional built-in converters will be implemented, including converters for arbitrary Java beans (using a variety of serialization approaches), maps and popular utility classes such as those from Google Guava Collections [9] and Joda Time [10] libraries.

I. Support for value validation and conversion

When handling data prepared and/or edited by human hand, it is extremely important to be lenient to minor errors in value formats, as well as fail gracefully on major errors.

Leniency on conversion is the responsibility of the option type handler. Built-in handlers try to provide leniency whenever possible; for example, built-in `java.util.Date` converter will try to parse dates/times in a number of short, medium and long formats before giving up and throwing an exception.

When handling configuration information retrieval from a persistent store (when human-readable strings are deserialized into Java objects), application can choose one of two strategies: strict and flexible. When strict load policy is enforced, framework will throw an exception on the first fatal conversion issue encountered, and no option values will be changed at all; this policy is well-suited for non-interactive applications, which typically can't continue without retrieving their runtime configuration. When flexible policy is chosen, conversion errors will be detected and stored by the framework (to lately be retrieved by the application), but all option values that converted without errors will be set; this policy is better suited for interactive applications, which can prompt user to. i.e., fix detected configuration issues.

Support for declarative value validation (similar to, or directly based on, JavaBean Validation API [11]) is currently not implemented; each option type handler can, however, enforce their own set of restrictions on values it can accept.

VI. Conclusion

Options-util is a prototype implementation of a modern annotations-driven application configuration framework. Currently on release version 0.1[4], it already is used in production, successfully managing configuration of several proprietary applications running in the AWS computing cloud. With existing and planned features such as type safety, different backing store providers and provider chaining and dependency injection support, it suits both small and large development projects running in the variety of environments, from cloud platforms to desktop computers.

VII. Future work

In the next release, *options-util* framework will receive an implementation of persistence provider chaining, in the form of both a simple annotation-based chaining, as well as a more complex chain builder class. Additionally, support for AWS S3 as a backing store will be fully implemented.

As mentioned elsewhere in this paper, in the subsequent near-term releases, the following features are planned for implementation:

- dependency injection support (Guice, followed by Spring);
- support for other types of persistent providers; split of providers into format providers and storage providers;
- support for serialization/deserialization of arbitrary Java beans.

For longer term, primary focus of development would be on adding additional persistence providers, implementing value validation and on adding support for structured configuration in the form of options sub-interfaces.

References

[1] Denisov, V. (2013). Overview of Java application configuration frameworks. International

4 Following semantic versioning approach, see [12]

Journal of Open Information Technologies, 1(6), 5-9. Available: http://injoit.org/index.php/j1/article/view/33

[2] Denisov, V. (2015). Functional requirements for a modern application configuration framework. International Journal of Open Information Technologies, 3(10), 6-10.

[3] Apache License, Version 2.0 (January 2004) [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.txt

[4] Scheduled Tasks With Cron for Java [Online]. Available: https://cloud.google.com/appengine/docs/java/config/cron

[5] Spring Framework [Online]. Available: http://projects.spring.io/spring-framework/

[6] Google Guice [Online]. Available: https://github.com/google/guice

[7] Contexts & Dependency Injection for Java [Online]. Available: http://www.cdi-spec.org/

[8] Amazon S3 [Online]. Available: https://aws.amazon.com/s3/

[9] Guava: Google Core Libraries for Java [Online]. Available: https://github.com/google/guava

[10] Joda-Time [Online]. Available: http://www.joda.org/joda-time/

[11] Bean Validation [Online]. Available: http://beanvalidation.org/

[12] Semantic Versioning 2.0.0 [Online]. Available: http://semver.org/