Functional requirements for a modern application configuration framework

Victor S. Denisov

***Abstract* – this paper describes a set of requirements for a modern application configuration framework for Java-based applications, including requirements for dependency injection support, type-safety and self-documentation.**

***Keywords* — application configuration management, application settings management, framework, Java.**

I. Introduction

As shown in [1], an area of Application Configuration Management had been mostly neglected by Java developers. While approaches, patterns and frameworks for developing Java applications are evolving rapidly – with language features and technologies like annotations, dependency injection, ORM and lambda expressions becoming commonplace (see [2] and [3] for a discussion of adoption of annotations and dependency injection, respectively), configuration libraries stagnated, stuck with providing simplistic key-value mappings and no advanced features expected of any modern Java framework.

This paper proposes a number of functional requirements for an application configuration framework, both common – like being platform-independent and employing a suite of unit/integration tests – and specific to the subject area, like support for complex data structures and type-safety.

II. Common Requirements

Common requirements put forward specifications which should be met by most, if not all, modern production-ready Java applications and libraries. Of particular interest for a potential application configuration framework are the following requirements:

- platform independence;
- suite of unit and integration tests;
- support of dependency injection;

Let's examine each of the above requirements in detail.

A. Platform Independence.

Java applications can be executed by a Java Virtual Machine running on a number of platforms with different computing hardware architecture, different operating systems and otherwise different operational environments (i.e., different types of persistent storage and networking stacks). It follows that configuration framework should not, to the extent possible, rely on any one platform's specific features, and should generally make as few assumptions about its runtime environment as possible. This includes not expecting support for file-based persistent store nor any other platform-specific persistent store, like Windows registry.

B. Unit and integration tests.

As configuration framework is supposed to be employed in a number of different operating environments, including cloud and embedded systems, it is reasonable to expect that it'll have many internal code paths which will not be regularly exercised on any specific platform. Additionally, configuration framework is usually close to the core of the application's functionality – most applications will fail with a fatal error if they won't be able to obtain a set of valid initial configuration options. In other words, services of a configuration framework are critical for the containing application.

As it would be impractical to manually test all changes in a framework in all supported operating environments - and criticality of configuration framework necessitates a rigorous testing regimen – a sophisticated suite of automated unit and integration tests should be employed to maintain an acceptable level of quality of the framework's codebase as it evolves over time. L. Koskela in [4] suggests that a typical level of code coverage should be around 85%, so it seems reasonable to set it as a lower acceptable level of code coverage for a configuration framework.

C. Support of dependency injection

Dependency injection is defined as "a software design pattern that implements inversion of control for resolving dependencies" [5]. This form of IoC is extremely helpful when developing loosely-coupled applications, avoiding strong ties between separate application components while delineating each components' services and dependencies via formally defined APIs (usually in the form of Java interfaces).

Configuration framework can benefit from supporting both injection of its dependencies into the framework and providing its services via dependency injection to other application components.

Injecting different implementations of services consumed by the framework allows it to easily adapt to different environments (including, but not limited to, injecting

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

mock object implementations to test services which would otherwise be impractical to test automatically[1].)

Injecting the framework service object(s) themselves allows the application to be quickly reconfigured, often without changing the source code. It also allows to mock framework's services when needed, or replace it with a different implementation.

III. Specific Requirements

A. Minimal number of dependencies

Configuration framework should, ideally, support projects of all sizes – from just a couple hundreds of LoCs all the way up to the millions. While larger projects can usually afford to manage an arbitrary number of dependencies, often via a dedicated project or dependency management framework[2], dependency management for small projects often becomes a serious issue, especially when transitive dependencies are involved[3].

With that in mind, a potential configuration framework should have a limited number of dependencies, with core functionality, ideally, being available with no dependencies at all. One possible approach would be to split a framework into a core module and a number of optional submodules, each of which can make its services (like access to a persistent storage or a serialization format) available to a core module at runtime. Each submodule would bring its own set of dependencies (i.e., a cloud storage module may depend on an appropriate cloud SDK), but the core module would have no mandatory dependencies at all.

B. Annotation-driven configuration

In Java programming language, annotations allow to add metadata to standard syntactic constructs like classes, methods, fields, etc. Practically, annotations allow one part of the application (or one of its dependencies) to retrieve additional information about syntactic elements of another parts (such as reading annotations on a method of a certain class, or treating classes annotated with a certain annotation differently from other classes).

One important benefit of annotations (as compared to external metadata sources such as XML files) is that they're tightly tied to the code they annotate, and can be moved/copied/modified alongside that code. Another benefit is that they do not depend on the availability of external metadata source – runtime annotations are compiled into Java bytecode and are automatically available (via reflection) to any class within the classpath.

Most recent frameworks, from unit testing [8] to object-relational mapping [9] to serialization [10] include some sort of annotations-based configuration. In context of configuration management framework annotations can supply the framework with information about annotated elements (such as whether a certain configuration property is read-only or read-write, specify mapping of the property's value when persisting the property to a persistent store, etc). Annotations also help with self-documentation of configuration options (see below).

C. Self-documentation of configuration options

When new developers check out a set of configuration options for an established project, they tend to have the same set of questions:

- what options are available to me?
- what options are read-only and what are read-write?
- what options are persisted in a backing store and what are only valid during application's runtime?
- what type does the property have and what's its acceptable values range?
- what backing store(s) are used, and is the store read-only, or read-write?

Having answers to those questions in the options code itself (in other words, having code to self-document itself) would be extremely helpful – both to break in a new developer and to quickly look up information for an experienced one. Also, self-documenting code usually leads to fewer errors and faster/more efficient development.

D. Support of cloud services

In recent years, cloud computing became commonplace, with 93% of companies adopting some form of cloud services [11]. However, efficient use of modern cloud technologies often requires significant changes on the part of the application, including the way it is configured. Some of the limitations imposed by cloud services can include:

- absence of a file-based local storage;
- limited control over computing instance deployment and initialization, custom provisioning technologies;
- extreme variation in workloads, quick turnaround of (virtual) computing instances;
- prevalence of non-traditional storage technologies[4] ([12], [13], [14])

In some cases ([15], [16]), application virtualization may go as far as to deal away with a notion of "computing instance" itself, which makes application configuration management an even more complex issue.

1 This often includes mocking services which require network access (which can be unavailable at the time the tests are run, or which can incur some sort of charge). Services which require a significant time to run (such as backup/restore services) are good candidates for mocking, too.

2 See [6] for an overview of common Java dependency management tools.

3 See [7] for an overview of how complex dependency management is even with a dedicated project management framework like Maven.

4 Traditional storage technologies include stuff like file-based storage and relational databases, while non-traditional can include technologies like NoSQL databases, document stores, REST-based APIs, etc.

A modern application configuration framework must support at least the most common cloud-based persistence services as configuration sources, as well as being able to operate in instance-less deployment scenarios.

E. Type-safety

A common question with simple, file-based configuration frameworks is "what Java type should I use for this option" - for example, is the option's value integer or decimal? or perhaps it can be a character string as well? Or perhaps it's a more complex object serialized as a character string? Unfortunately, existing frameworks rarely provide a developer with intuitive answer, which leads to all sorts of non-obvious runtime issues.

A modern configuration framework must satisfy a requirement of type-safety: it should only be possible to set an option to values of a predetermined type (or a set of types), and it must always return an instance of a predetermined type when being read.

One approach to achieve type-safety is to define application configuration information via a Java interface with methods to get (and set, if required/allowed) each individual option. If configuration information is only accessed via this interface, type safety will be guaranteed by the Java compiler.

F. Support for different configuration sources/persistent backing stores

As discussed above, modern Java applications run in a number of environments, from servers and desktops to cloud platforms to mobile devices. Consequently, configuration framework should be able to handle different types of configuration sources:

- traditional configuration files in various formats;
- Java system properties;
- environment variables;
- SQL and NoSQL databases;
- cloud-based document stores;
- web services and other web-based sources;

However, simply being able to interact with different backing stores is not enough – configuration framework should support environment-based preferences for a configuration source, for example:

- on developer's workstation, use a file-based store;
- in an on-premises deployment, use an SQL database;
- in a cloud deployment, use cloud-based document store.

Finally, framework should support chaining of configuration sources, so that configuration information can be retrieved from multiple sources and then combined together (also, see III.K), for example, in an AWS EC2 environment an application can use the following chain:

- read defaults from a classpath resource;
- read common configuration information from an S3-based file;
- read instance-specific configuration from an instance metadata via an HTTP call.

Such chaining should be, to the extent possible, transparent to the application – it should only be concerned with defining the chain and providing necessary credentials, if needed by the underlying service.

G. Extensibility

Despite the requirements of sections II.A and III.F, configuration framework is still inherently environment-dependent (as it should receive and, perhaps, store configuration information from/to *somewhere* outside the application's scope), it is technically impossible to make it absolutely self-contained. Additionally, the format in which configuration information is delivered can be quite different, and often outside of developer's control.

This calls for a requirement of at least two possible extension endpoints: access to the backing persistent store (or other configuration source) and format of the configuration information itself. Framework should define clear extension mechanisms which would allow developers to plug in their custom implementations of persistent store accessors and serialization/deserialization providers.

Additionally, application developers would benefit from the ability to add support for custom data types to be used as part of configuration information – more on this below.

H. Support for complex data structures

Most existing configuration frameworks only support scalar types, such as integers and strings, and simple vector types, like lists and/or arrays, as their value options. However, complex applications would benefit from arbitrarily complex structures (including direct encapsulation of Java objects).

While it can probably be impractical for a framework to implement full support for any conceivable Java object to be read/stored in any supported persistent store in a platform-independent way, it should, at the very least, support the more common (and well-defined) structures, such as Java Beans and collections from Java Collections Framework and other popular collection classes, such as those provided by Google Guava [17].

For other use cases, the framework should define an extension mechanism which would allow developers to add support for custom data types. Most likely, this would be implemented as part of serialization/deserialization API, since platform- and store-independent serialization is probably the most significant obstacle for such support.

I. Support for value validation and conversion

Humans invariably make mistakes, so sooner or later, configuration framework will encounter errors in provided configuration information, either in the structure of the information itself (i.e., a malformed XML document) or in one of the properties' values. Reaction to such errors should be robust and predictable: depending on the severity of the problem and specific application's

requirements, framework should either immediately and unambiguously fail (likely by throwing a checked exception), or (for a recoverable error) substitute a set of reasonable default values for affected properties while alerting the application (via a return value, an application event and/or an error message to a log file or console).

In addition to value validation framework should contain tools to influence value conversion: while for some value types (mostly number-based) conversion rules are reasonably well-established, for many others (like dates/times, boolean values, and enumerated types) conversion is significantly less straightforward. Framework should employ configurable converters for data types where formal format specification is common (i.e., date/time types), as well as support custom value converters for arbitrary data.

J. Runtime configuration changes/change listeners

Configuration information is of interest to many modules and submodules within the application. When this information changes (either externally, like modification of the configuration file on a filesystem – or internally by an application component, like a configuration dialog), framework should propagate notifications about the change to all interested parties. Notifications can be issued via a traditional provider/listener pattern, or a more efficient event bus pattern.

K. Support for structured configuration information

For larger applications, the number of distinct configuration options can be quite high, certainly in the hundreds, if not thousands. However, any given application submodule only deals with a handful of properties – so if the framework can provide only a specific subset of configuration information – that of interest to the submodule – it would make the life of a module developer significantly easier. If allowing to pick specific options is impractical (it probably will be without a dedicated dependency injection framework), the framework should, at the very least, support a hierarchical grouping of configuration options, with modules only retrieving specific groups at the specific hierarchy levels.

Additionally, for those persistent backing stores that support structured information (and that would be, in one form or another, most of them – from configuration files to databases to REST services), framework should support both reading and writing configuration information in a reasonably structured way (i.e., using prefixes and/or sections for property files or a proper nesting for XML documents).

IV. Conclusion

This paper presents a set of common and subject area-specific functional requirements for a modern application configuration framework. This includes the following common requirements:

- platform independence;
- extensive coverage by unit and integration tests;
- support for dependency injection frameworks;

and the following specific requirements:

- minimal number of runtime and compile-time dependencies, at least for simpler configurations;
- annotation-driven configuration specification;
- self-documentation of configuration options;
- support of cloud services;
- complete type-safety;
- support for different configuration sources;
- extensibility via plugins and alternative implemenations;
- support for complex object data structures;
- support for value validation and conversion;
- runtime configuration change event propagation;
- support for structured configuration information.

This list is, of course, not in any way exhaustive. However, implementing a framework that matches most of the above requirements would greatly benefit the Java application ecosystem.

V. Future Work

The requirements above were used as guidelines when developing a new open-source configuration library "options-util"[5]. While there is still a lot of work ahead for this library, development-wise, it already provides a solid set of features for accessing configuration information from a variety of configuration sources.

References

[1] Denisov, V. (2013). Overview of Java application configuration frameworks. International Journal of Open Information Technologies, 1(6), 5-9. Available: http://injoit.org/index.php/j1/article/view/33

[2] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. "Mining billions of AST nodes to study actual and potential usage of Java language features" in Proceedings of the 36th International Conference on Software Engineering, pp. 779-790. ACM New York, 2014

[3] H. Y. Yang, E. Tempero, H. Melton. An Empirical Study into Use of Dependency Injection in Java in "19th Australian Conference on Software Engineering, 2008 (ASWEC 2008)", pp. 239 – 247. Perth, WA, 2008

[4] L. Koskela. Test Driven. Practical TDD and Acceptance TDD for Java Developers. Manning, Greenwich, CT, 2008

[5] Dependency injection [Online]. Available: https://en.wikipedia.org/wiki/Dependency_injection

[6] M. Rasmussen. (2013, Nov. 21). Java Build Tools: How Dependency Management Works with Maven, Gradle and Ant + Ivy [Online]. Available: http://zeroturnaround.com/rebellabs/java-build-tools-how-dependency-management-works-with-maven-gradle-and-ant-ivy/

5 https://github.com/options-util/options-util

[7] Introduction to the Dependency Mechanism [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[8] TestNG Documentation – Annotations [Online]. Available: http://testng.org/doc/documentation-main.html#annotations

[9] MyBatis Java API [Online]. Available: https://mybatis.github.io/mybatis-3/java-api.html

[10] Jackson Core Annotations [Online]. Available: https://github.com/FasterXML/jackson-annotations/wiki

[11] RightScale (2015). Cloud Computing Trends: 2015 State of the Cloud Survey [Online].Available: http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2015-state-cloud-survey

[12] Amazon DynamoDB [Online]. Available: https://aws.amazon.com/dynamodb/

[13] Google App Engine Datastore [Online]. Availble: https://cloud.google.com/appengine/features/#datastore

[14] DocumentDB [Online]. Available: https://azure.microsoft.com/en-us/services/documentdb/

[15] AWS Lambda [Online]. Available: https://aws.amazon.com/lambda/

[16] Google App Engine: Platform as a Service [Online]. Available: https://cloud.google.com/appengine/

[17] B. Bejeck. Getting Started with Google Guava. Packt Publishing, Birmingham, UK, 2013.