# BindForge

Version 0.5

Module Framework

March 26, 2009

# Contents

# 1 What is BindForge?

The mission of BindForge is to ease the OSGi development. BindForge provides sophisticated dependency injection facilities and various abstraction layers for OSGi services and other compendium elements. The configuration is done via a powerful Scala-based DSL. These features are provided non-intrusively so that Java programmers do not need to learn e.g. the Scala compiler or other tools. Simply put a text file in your bundle. It's as simple as that!

BindForge relies on Guice for the dependency injection features. Unlike normally required when using Guice, BindForge users do not need to put special annotations in their classes (e.g. @Inject). Hence the Java classes are full POJOs again. At the same time, BindForge is 100% compatible with Guice. All known features, e.g. Spring beans support, are available. For the OSGi service registry interaction, BindForge uses the Peaberry dynamic service extension. Additionally, BindForge completely abstracts from various OSGi elements. Without introducing code dependencies, users can fully utilise the OSGi service platform.

# 2 Getting BindForge

## 2.1 Direct Download

To use BindForge, you need to install 2 bundles in your OSGi framework:

- bindforge-*version*.jar
- scala-full-bundle-*version*.jar

Both bundles are available via a direct file download or via the BindForge Maven repository. The Scala bundle contains the complete Scala library and the Scala compiler. **Note:** In case you already have a Scala bundle installed, it is recommended to replace the existing one with the BindForge version since the existing bundle may not contain the Scala compiler.

The bundle files are available at the BindForge homepage, download section.

### 2.2 Maven Repository

Besides the direct download links, all required bundles are also available via the BindForge Maven repository. To use the repository, add the following configuration (1) to your pom.xml:

```
1 <repository>
2     <id>bindforge.org</id>
3     <name>BindForge Maven2 Repository</name>
4     <url>http://repository.tuxed.de</url>
5 </repository>
```

**Listing 1:** BindForge Maven Repository

After you configured the repository, you will need to add the artifact dependencies to your pom.xml (2):

```
1 <dependency>
2         <groupId>org.bindforge</groupId>
3         <artifactId>bindforge</artifactId>
4         <version>0.5.0</version>
5 </dependency>
6 <dependency>
7         <groupId>org.scala-lang</groupId>
8         <artifactId>scala-full-bundle</artifactId>
9         <version>2.7.3</version>
10 </dependency>
```

**Listing 2:** BindForge Artifacts

## 3 Creating a BindForge Configuration

### 3.1 Bundle Configuration

To use BindForge, you need to put a configuration file in your bundle. Once your bundle gets started, BindForge will read this file to activate the configuration.[1] A manifest entry in the bundle is used to specify the configuration file. Listing 3 shows an example MANIFEST.MF.

[1] This behavior is called *Extender Pattern* and very common for OSGi frameworks. For example, the *Declarative Services* specification uses the same mechanism.

```
1 Bundle-ManifestVersion: 2
2 Bundle-SymbolicName: org.acme.yourapp
3 BindForge-Config: org.acme.yourapp.Config
```

**Listing 3:** MANIFEST.MF with Configuration

In this example, the header `BindForge-Config` in line 3 specifies the configuration that should be used. As you can see here, the value has the form of `packagename.ClassName`. BindForge configurations are full Java classes written in the Scala programming language[2]. The programmer can choose between 2 options to create these configuration classes:

**Embedded scripts** During the bundle activation, BindForge will compile every `*.scala` file in the folder `/OSGI-INF/bindforge`. Unlike Java, Scala does not impose the restriction on the filename/path that is has to match the name/package of the class it defines. Therefore you can directly put a file, e.g. `config.scala`, in the folder without the need to create several sub-directories, even if you want to "put" the configuration in a package.[3]

This option should be used by Java programmers if they do not want to alter their build system and configuration.

**Compile during build process** If the primary language in the project is Scala, the build system will already be configured to use a Scala compiler. In this case the BindForge configuration can be compiled just like any other Scala source file in the project.

This option can also be used in pure Java projects but requires the additional compilation of Scala files during the build process. The advantage is that the programmer will get a validation of the configuration file before deployment.

Regardless of which option you use, the manifest header only depends on the `packagename` and `ClassName` used for the configuration.

### 3.2 Configuration File

Listing 4 shows a basic BindForge configuration file. The package and class name can be specified by the programmer and only need to match the name used in the bundle manifest header `BindForge-Config`.

[2] http://www.scala-lang.org

[3] It is generally a good idea to use the bundles top-level package as the package for the configuration. We will see later why this is useful.

```
1 package com.acme.app
2
3 class MyConfig extends org.bindforge.Config {
4   // configuration goes here
5 }
```

**Listing 4:** BindForge configuration file

As described earlier, it is generally a good idea to use the same package name for the configuration and for the normal application classes. That is because packages in Scala truly nest. For example, if your configuration is declared in package `com.acme.app` and you want to reference the class `com.acme.app.internal.MyService`, you can directly reference the class with `internal.MyService`. Hence you do not need to repeat the package names if the configuration and referenced class have the same package root.

# 4 Dependency Injection

## 4.1 Introduction

This section describes the dependency injection (DI) DSL.[4]. In BindForge, the configuration is based around the concept of *bindings*. A binding is created for a specific type to make BindForge aware of it. This binding can be used to map e.g. interfaces to specific implementations or to define dependencies that needs to be fulfilled by BindForge.

## 4.2 Basic Binding

Listing 5 shows 2 example interfaces and implementations that will later be configured and wired by BindForge. Here, `BServiceImpl` depends on `AService` and we want to use the method `setAService` to inject an instance of `AService`. We assume that all classes are defined in package `org.example`.

```
1 interface AService {...}
2 class AServiceImpl implements AService {...}
3
4 interface BService {...}
```

[4]Internally, BindForge uses Google Guice to provide the dependency injection features (http://google-guice.googlecode.com).

```
5 class BServiceImpl implements BService {
6     public void setAService(AService aService) {
7         // ...
8     }
9 }
```

**Listing 5:** Example classes

First, we need to bind both services to make BindForge aware of them (listing 6). The `bind` method is used to map the interface to the implementation.

```
1 package org.example
2
3 class MyFirstConfig extends org.bindforge.Config {
4     bind [AService, AServiceImpl]
5     bind [BService, BServiceImpl]
6 }
```

**Listing 6:** Simple bindings

This configuration causes BindForge to create an instance for each service implementation.

### 4.3 Wiring by Type

To define the wiring between the 2 bindings we need to add a *spec block* to the `ServiceB` binding (listing 7).

```
1 package org.example
2
3 class MyFirstConfig extends org.bindforge.Config {
4     bind [AService, AServiceImpl]
5     bind [BService, BServiceImpl] spec {
6         property("aService")
7     }
8 }
```

**Listing 7:** Specify properties

The `property` method is used to define the properties that needs to be injected by BindForge. If the target class is written in Java, BindForge will use the JavaBeans notation to map the property name to the setter method. If the class is written in Scala, BindForge

will use the `propertyname_=` method[5]. The method parameter type is used to determine the required dependency. In our example, the parameter is declared as an instance of `AService`. In our configuration we created a binding for this type that mapped `AService` to `AServiceImpl`. Hence, BindForge will inject an instance of `AServiceImpl`.

**Bindings and Singletons** By default, BindForge will create exactly one instance for each binding. This means that if we would inject `AService` several times, BindForge would always use the same instance. Later versions of BindForge will allow to configure this behavior.

### 4.4 Optional Interface Type

So far we always specified both the interface and implementation type for the bindings. However, if we do not depend on the interface we can leave out the first parameter of the `bind` method. In our example, this applies to the `ServiceBImpl` binding since no other binding depends on it. Listing 8 shows the updated version.

```
1 package org.example
2
3 class MyFirstConfig extends org.bindforge.Config {
4     bind [AService, AServiceImpl]
5     bind [BServiceImpl] spec {
6         property("aService")
7     }
8 }
```

**Listing 8:** Optional interface type

### 4.5 Wiring by ID

The last examples always used a *wiring by type* where the parameter type of the setter method is used to determine the required type. However, if several bindings for a specific type exists, BindForge does not know how to select the instance that should be injected. Therefore the user can assign a unique ID to the bindings. Later, this ID can be used to explicitly refer to a binding. This ID is only required if the bindings are ambiguous and

[5]This method is always created by the Scala compiler if a field is declared as a 'var' member. This method will be used transparently if a value is assigned to the field, e.g. 'obj.field = value'.

the user wants to refer to one specific binding. Listing 9 shows how to use this *wiring by ID*.

```
1 package org.example
2
3 class MyFirstConfig extends org.bindforge.Config {
4     "myID" :: bind [AServiceImpl]
5     bind [BServiceImpl] spec {
6         property("aService") = ref("myID")
7     }
8 }
```

**Listing 9:** Bindings with ID

Since the dependency for binding `BServiceImpl` is now identified by the ID and not by the parameter type of the setter method, we can also leave out the interface type `AService` in the first binding. If we would still specify it we could refer to the `AService` by both the ID and type.

### 4.6 Primitive Types

Listing 10 shows how to inject simple values that are directly specified in the configuration file.

```
1 class MyFirstConfig extends org.bindforge.Config {
2     bind [type] spec {
3         property("timeout") = 1000
4         property("username") = "joe"
5     }
6 }
```

**Listing 10:** Inject primitive types

### 4.7 Collections

TBD

### 4.8 Lifecycle Callbacks

(Description will follow)

```
class MyFirstConfig extends org.bindforge.Config {
    bind [type] spec {
        lifecycle("init", "destroy")
    }
}
```

**Listing 11:** Lifecycle callbacks

# 5 OSGi Service Registry

## 5.1 Registering Services

```
class MyFirstConfig extends org.bindforge.Config {
    bind [type] spec {
        exportService
    }
}
```

**Listing 12:** Registering OSGi services

```
class MyFirstConfig extends org.bindforge.Config {
    bind [type] spec {
        exportService("key1" -> "value1", "key2" -> "value2")
    }
}
```

**Listing 13:** Registering OSGi services with properties

## 5.2 Accessing Services

```
class MyFirstConfig extends org.bindforge.Config {

    bind [LogService] importService

    bind [type] spec {
        property("logService")
    }
}
```

**Listing 14:** Using OSGi services

```
class MyFirstConfig extends org.bindforge.Config {

    bind [LogService] importService ("(key=value)")

    bind [type] spec {
        property("logService")
    }
}
```

**Listing 15:** Using OSGi services and LDAP filters

# 6 OSGi Configuration Admin

```
class MyFirstConfig extends org.bindforge.Config {
    bind [type] spec {
        config("myservice.pid")
    }
}
```

**Listing 16:** Using OSGi ConfigurationAdmin service